GNU Octave

A high-level interactive language for numerical computations
Edition 4 for Octave version 4.0.3
March 2015

Free Your Numbers

John W. Eaton
David Bateman
Søren Hauberg
Rik Wehbring

This is the fourth edition of the Octave documentation, and is consistent with version 4.0.3 of Octave.

Portions of this document have been adapted from the `gawk`, `readline`, `gcc`, and C library manuals, published by the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301–1307, USA.

Table of Contents

Preface

Octave was originally intended to be companion software for an undergraduate-level textbook on chemical reactor design being written by James B. Rawlings of the University of Wisconsin-Madison and John G. Ekerdt of the University of Texas.

Clearly, Octave is now much more than just another 'courseware' package with limited utility beyond the classroom. Although our initial goals were somewhat vague, we knew that we wanted to create something that would enable students to solve realistic problems, and that they could use for many things other than chemical reactor design problems. We find that most students pick up the basics of Octave quickly, and are using it confidently in just a few hours.

Although it was originally intended to be used to teach reactor design, it has been used in several other undergraduate and graduate courses in the Chemical Engineering Department at the University of Texas, and the math department at the University of Texas has been using it for teaching differential equations and linear algebra as well. More recently, Octave has been used as the primary computational tool for teaching Stanford's online Machine Learning class (`ml-class.org`) taught by Andrew Ng. Tens of thousands of students participated in the course.

If you find Octave useful, please let us know. We are always interested to find out how Octave is being used.

Virtually everyone thinks that the name Octave has something to do with music, but it is actually the name of one of John W. Eaton's former professors who wrote a famous textbook on chemical reaction engineering, and who was also well known for his ability to do quick 'back of the envelope' calculations. We hope that this software will make it possible for many people to do more ambitious computations just as easily.

Everyone is encouraged to share this software with others under the terms of the GNU General Public License (see Appendix J [Copying], page 917). You are also encouraged to help make Octave more useful by writing and contributing additional functions for it, and by reporting any problems you may have.

Acknowledgements

Many people have contributed to Octave's development. The following people have helped code parts of Octave or aided in various other ways (listed alphabetically).

Ben Abbott	Drew Abbot	Andy Adler
Adam H. Aitkenhead	Giles Anderson	Joel Andersson
Lachlan Andrew	Pedro Angelo	Muthiah Annamalai
Markus Appel	Branden Archer	Willem Atsma
Marco Atzeri	Shai Ayal	Roger Banks
Ben Barrowes	Alexander Barth	David Bateman
Heinz Bauschke	Julien Bect	Stefan Beller
Roman Belov	Markus Bergholz	Karl Berry
Ethan Biery	David Billinghurst	Don Bindner
Jakub Bogusz	Moritz Borgmann	Paul Boven
Richard Bovey	John Bradshaw	Marcus Brinkmann
Max Brister	Remy Bruno	Clemens Buchacher

Ansgar Burchard
Marco Caliari
Daniel Calvelo
John C. Campbell
Juan Pablo Carbajal
Jean-Francois Cardoso
Joao Cardoso
Larrie Carr
David Castelow
Vincent Cautaerts
Clinton Chee
Albert Chin-A-Young
Carsten Clark
Catalin Codreanu
J. D. Cole
Martin Costabel
Michael Creel
Richard Crozier
Jeff Cunningham
Martin Dalecki
Jacob Dawid
Jorge Barros de Abreu
Carlo de Falco
Thomas D. Dean
Philippe Defert
Bill Denney
Fabian Deutsch
Christos Dimitrakakis
Pantxo Diribarne
Vivek Dogra
John Donoghue
David M. Doolin
Carnë Draug
Pascal A. Dupuis
John W. Eaton
Dirk Eddelbuettel
Pieter Eendebak
Paul Eggert
Stephen Eglen
Peter Ekberg
Edmund Grimley Evans
Rolf Fabian
Gunnar Farnebäck
Massimiliano Fasi
Stephen Fegan
Ramon Garcia Fernandez
Torsten Finke
Colin Foster
Jose Daniel Munoz Frias
Brad Froehle
Castor Fu
Eduardo Gallestey
Walter Gautschi
Klaus Gebhardt
Driss Ghaddab
Eugenio Gianniti
Nicolo Giorgetti
Arun Giridhar
Michael D. Godfrey
Michael Goffioul
Glenn Golden
Tomislav Goles
Keith Goodman
Brian Gough
Michael C. Grant
Steffen Groot
Etienne Grossmann
David Grundberg
Kyle Guinn
Vaibhav Gupta
Peter Gustafson
Kai Habel
Patrick Häcker
William P. Y. Hadisoeseno
Jaroslav Hajek
Benjamin Hall
Kim Hansen
Søren Hauberg
Dave Hawthorne
Daniel Heiserer
Martin Helm
Stefan Hepp
Martin Hepperle
Jordi Gutiérrez Hermoso
Yozo Hida
Ryan Hinton
Roman Hodek
A. Scottedward Hodel
Richard Allan Holcombe
Tom Holroyd
David Hoover
Kurt Hornik
Christopher Hulbert
Cyril Humbert
John Hunt
Teemu Ikonen
Alan W. Irwin
Allan Jacobs
Geoff Jacobsen
Vytautas Jančauskas
Nick Jankowski
Mats Jansson
Cai Jianming
Steven G. Johnson
Heikki Junes
Matthias Jüschke
Atsushi Kajita
Jarkko Kaleva
Avinoam Kalma
Mohamed Kamoun
Lute Kamstra
Fotios Kasolis
Thomas Kasper
Joel Keay
Mumit Khan
Paul Kienzle
Aaron A. King
Erik Kjellson
Arno J. Klaassen
Alexander Klein
Geoffrey Knauth
Heine Kolltveit
Ken Kouno
Kacper Kowalik
Daniel Kraft
Nir Krakauer
Aravindh Krishnamoorthy
Oyvind Kristiansen
Artem Krosheninnikov
Piotr Krzyzanowski
Volker Kuhlmann
Ilya Kurdyukov
Tetsuro Kurita
Philipp Kutin
Miroslaw Kwasniak
Rafael Laboissiere
Kai Labusch
Claude Lacoursiere
Walter Landry
Bill Lash
Dirk Laurie

Maurice LeBrun	Friedrich Leisch	Johannes Leuschner
Thorsten Liebig	Jyh-miin Lin	Timo Lindfors
Benjamin Lindner	Ross Lippert	David Livings
Sebastien Loisel	Erik de Castro Lopo	Massimo Lorenzin
Emil Lucretiu	Yi-Hong Lyu	Hoxide Ma
Colin Macdonald	James Macnicol	Jens-Uwe Mager
Stefan Mahr	Rob Mahurin	Alexander Mamonov
Ricardo Marranita	Orestes Mas	Axel Mathéi
Makoto Matsumoto	Tatsuro Matsuoka	Christoph Mayer
Laurent Mazet	G. D. McBain	Ronald van der Meer
Júlio Hoffimann Mendes	Ed Meyer	Thorsten Meyer
Stefan Miereis	Petr Mikulik	Mike Miller
Serviscope Minor	Stefan Monnier	Antoine Moreau
Kai P. Mueller	Armin Müller	Hannes Müller
Victor Munoz	PrasannaKumar Muralidharan	Iain Murray
Carmen Navarrete	Todd Neal	Philip Nienhuis
Al Niessner	Felipe G. Nievinski	Rick Niles
Takuji Nishimura	Kai Noda	Patrick Noffke
Eric Norum	Krzesimir Nowak	Michael O'Brien
Peter O'Gorman	Thorsten Ohl	Kai T. Ohlhus
Arno Onken	Valentin Ortega-Clavero	Luis F. Ortiz
Carl Osterwisch	Janne Olavi Paanajärvi	Scott Pakin
José Luis García Pallero	Jason Alan Palmer	Gabriele Pannocchia
Sylvain Pelissier	Rolando Pereira	Per Persson
Primozz Peterlin	Jim Peterson	Danilo Piazzalunga
Nicholas Piper	Elias Pipping	Robert Platt
Hans Ekkehard Plesser	Sergey Plotnikov	Tom Poage
Nathan Podlich	Orion Poplawski	Ondrej Popp
Jef Poskanzer	Francesco Potortì	Konstantinos Poulios
Jarno Rajahalme	Eduardo Ramos	Pooja Rao
James B. Rawlings	Eric S. Raymond	Balint Reczey
Joshua Redstone	Lukas Reichlin	Michael Reifenberger
Jens Restemeier	Anthony Richardson	Jason Riedy
E. Joshua Rigler	Sander van Rijn	Petter Risholm
Matthew W. Roberts	Dmitry Roshchin	Peter Rosin
Andrew Ross	Fabio Rossi	Mark van Rossum
Joe Rothweiler	David Rörich	Kevin Ruland
Kristian Rumberg	Ryan Rusaw	Olli Saarela
Toni Saarela	Juhani Saastamoinen	Radek Salac
Mike Sander	Ben Sapp	Aleksej Saushev
Alois Schlögl	Michel D. Schmid	Julian Schnidder
Sebastian Schoeps	Nicol N. Schraudolph	Sebastian Schubert
Lasse Schuirmann	Ludwig Schwardt	Thomas L. Scofield
Daniel J. Sebald	Dmitri A. Sergatskov	Vanya Sergeev
Marko Seric	Ahsan Ali Shahid	Baylis Shanks

Andriy Shinkarchuck
Robert T. Short
Joseph P. Skudlarek
John Smith
Julius Smith
Shan G. Smith
Peter L. Sondergaard
Joerg Specht
Quentin H. Spencer
Christoph Spiel
David Spies
Richard Stallman
Russell Standish
Brett Stewart
Doug Stewart
Jonathan Stickel
Judd Storrs
Thomas Stuart
Ivan Sutoris
John Swensen
Daisuke Takago
Ariel Tankus
Falk Tannhäuser
Duncan Temple Lang
Matthew Tenny
Kris Thielemans
Georg Thimm
Corey Thomasson
Olaf Till
Christophe Tournery
Thomas Treichl
Karsten Trulsen
David Turner
Frederick Umminger
Utkarsh Upadhyay
Stefan van der Walt
Peter Van Wieren
James R. Van Zandt
Risto Vanhanen
Gregory Vanuxem
Mihas Varantsou
Ivana Varekova
Sébastien Villemot
Marco Vitetta
Daniel Wagenaar
Thomas Walter
Andreas Weber
Olaf Weber
Thomas Weber
Rik Wehbring
Bob Weigel
Andreas Weingessel
Martin Weiser
Michael Weitzel
David Wells
Joachim Wiesemann
Fook Fah Yap
Sean Young
Johannes Zarl
Michael Zeising
Federico Zenith
Alex Zvoleff

Special thanks to the following people and organizations for supporting the development of Octave:

- The United States Department of Energy, through grant number DE-FG02-04ER25635.
- Ashok Krishnamurthy, David Hudak, Juan Carlos Chaves, and Stanley C. Ahalt of the Ohio Supercomputer Center.
- The National Science Foundation, through grant numbers CTS-0105360, CTS-9708497, CTS-9311420, CTS-8957123, and CNS-0540147.
- The industrial members of the Texas-Wisconsin Modeling and Control Consortium (TWMCC (`http://www.che.utexas.edu/twmcc`)).
- The Paul A. Elfers Endowed Chair in Chemical Engineering at the University of Wisconsin-Madison.
- Digital Equipment Corporation, for an equipment grant as part of their External Research Program.
- Sun Microsystems, Inc., for an Academic Equipment grant.
- International Business Machines, Inc., for providing equipment as part of a grant to the University of Texas College of Engineering.
- Texaco Chemical Company, for providing funding to continue the development of this software.
- The University of Texas College of Engineering, for providing a Challenge for Excellence Research Supplement, and for providing an Academic Development Funds grant.
- The State of Texas, for providing funding through the Texas Advanced Technology Program under Grant No. 003658-078.
- Noel Bell, Senior Engineer, Texaco Chemical Company, Austin Texas.

- John A. Turner, Group Leader, Continuum Dynamics (CCS-2), Los Alamos National Laboratory, for registering the `octave.org` domain name.
- James B. Rawlings, Professor, University of Wisconsin-Madison, Department of Chemical and Biological Engineering.
- Richard Stallman, for writing GNU.

This project would not have been possible without the GNU software used in and to produce Octave.

Citing Octave in Publications

In view of the many contributions made by numerous developers over many years it is common courtesy to cite Octave in publications when it has been used during the course of research or the preparation of figures. The `citation` function can automatically generate a recommended citation text for Octave or any of its packages. See the help text below on how to use `citation`.

`citation` [Command]
`citation` *package* [Command]

Display instructions for citing GNU Octave or its packages in publications.

When called without an argument, display information on how to cite the core GNU Octave system.

When given a package name *package*, display information on citing the specific named package. Note that some packages may not yet have instructions on how to cite them.

The GNU Octave developers and its active community of package authors have invested a lot of time and effort in creating GNU Octave as it is today. Please give credit where credit is due and cite GNU Octave and its packages when you use them.

How You Can Contribute to Octave

There are a number of ways that you can contribute to help make Octave a better system. Perhaps the most important way to contribute is to write high-quality code for solving new problems, and to make your code freely available for others to use. See Appendix D [Contributing Guidelines], page 879, for detailed information on contributing new code.

If you find Octave useful, consider providing additional funding to continue its development. Even a modest amount of additional funding could make a significant difference in the amount of time that is available for development and support.

Donations supporting Octave development may be made on the web at `https://my.fsf.org/donate/working-together/octave`. These donations also help to support the Free Software Foundation

If you'd prefer to pay by check or money order, you can do so by sending a check to the FSF at the following address:

Free Software Foundation
51 Franklin Street, Suite 500
Boston, MA 02110-1335
USA

If you pay by check, please be sure to write "GNU Octave" in the memo field of your check.

If you cannot provide funding or contribute code, you can still help make Octave better and more reliable by reporting any bugs you find and by offering suggestions for ways to improve Octave. See Appendix F [Trouble], page 889, for tips on how to write useful bug reports.

Distribution

Octave is *free* software. This means that everyone is free to use it and free to redistribute it on certain conditions. Octave is not, however, in the public domain. It is copyrighted and there are restrictions on its distribution, but the restrictions are designed to ensure that others will have the same freedom to use and redistribute Octave that you have. The precise conditions can be found in the GNU General Public License that comes with Octave and that also appears in Appendix J [Copying], page 917.

To download a copy of Octave, please visit `http://www.octave.org/download.html`.

1 A Brief Introduction to Octave

GNU Octave is a high-level language, primarily intended for numerical computations. It is typically used for such problems as solving linear and nonlinear equations, numerical linear algebra, statistical analysis, and for performing other numerical experiments. It may also be used as a batch-oriented language for automated data processing.

Until recently GNU Octave provided a command-line interface only with graphical plots displayed in separate windows. However, by default the current version runs with a graphical user interface.

GNU Octave is freely redistributable software. You may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. The GPL is included in this manual, see Appendix J [Copying], page 917.

This manual provides comprehensive documentation on how to install, run, use, and extend GNU Octave. Additional chapters describe how to report bugs and help contribute code.

This document corresponds to Octave version 4.0.3.

1.1 Running Octave

On most systems, Octave is started with the shell command '`octave`'. This starts the graphical user interface (GUI). The central window in the GUI is the Octave command-line interface. In this window Octave displays an initial message and then a prompt indicating it is ready to accept input. If you have chosen the traditional command-line interface then only the command prompt appears in the same window that was running a shell. In any case, you can immediately begin typing Octave commands.

If you get into trouble, you can usually interrupt Octave by typing *`Control-C`* (written *`C-c`* for short). *`C-c`* gets its name from the fact that you type it by holding down `CTRL` and then pressing `c`. Doing this will normally return you to Octave's prompt.

To exit Octave, type *`quit`* or *`exit`* at the Octave prompt.

On systems that support job control, you can suspend Octave by sending it a `SIGTSTP` signal, usually by typing *`C-z`*.

1.2 Simple Examples

The following chapters describe all of Octave's features in detail, but before doing that, it might be helpful to give a sampling of some of its capabilities.

If you are new to Octave, we recommend that you try these examples to begin learning Octave by using it. Lines marked like so, '`octave:13>`', are lines you type, ending each with a carriage return. Octave will respond with an answer, or by displaying a graph.

1.2.1 Elementary Calculations

Octave can easily be used for basic numerical calculations. Octave knows about arithmetic operations (+,-,*,/), exponentiation (^), natural logarithms/exponents (log, exp), and the trigonometric functions (sin, cos, . . .). Moreover, Octave calculations work on real or imaginary numbers (i,j). In addition, some mathematical constants such as the base of the natural logarithm (e) and the ratio of a circle's circumference to its diameter (pi) are pre-defined.

For example, to verify Euler's Identity,

$$e^{\imath\pi} = -1$$

type the following which will evaluate to `-1` within the tolerance of the calculation.

```
octave:1> exp (i*pi)
```

1.2.2 Creating a Matrix

Vectors and matrices are the basic building blocks for numerical analysis. To create a new matrix and store it in a variable so that you can refer to it later, type the command

```
octave:1> A = [ 1, 1, 2; 3, 5, 8; 13, 21, 34 ]
```

Octave will respond by printing the matrix in neatly aligned columns. Octave uses a comma or space to separate entries in a row, and a semicolon or carriage return to separate one row from the next. Ending a command with a semicolon tells Octave not to print the result of the command. For example,

```
octave:2> B = rand (3, 2);
```

will create a 3 row, 2 column matrix with each element set to a random value between zero and one.

To display the value of a variable, simply type the name of the variable at the prompt. For example, to display the value stored in the matrix `B`, type the command

```
octave:3> B
```

1.2.3 Matrix Arithmetic

Octave has a convenient operator notation for performing matrix arithmetic. For example, to multiply the matrix `A` by a scalar value, type the command

```
octave:4> 2 * A
```

To multiply the two matrices `A` and `B`, type the command

```
octave:5> A * B
```

and to form the matrix product $A^T A$, type the command

```
octave:6> A' * A
```

1.2.4 Solving Systems of Linear Equations

Systems of linear equations are ubiquitous in numerical analysis. To solve the set of linear equations `Ax = b`, use the left division operator, '\':

```
x = A \ b
```

This is conceptually equivalent to $A^{-1}b$, but avoids computing the inverse of a matrix directly.

If the coefficient matrix is singular, Octave will print a warning message and compute a minimum norm solution.

A simple example comes from chemistry and the need to obtain balanced chemical equations. Consider the burning of hydrogen and oxygen to produce water.

$$\mathrm{H_2} + \mathrm{O_2} \rightarrow \mathrm{H_2O}$$

The equation above is not accurate. The Law of Conservation of Mass requires that the number of molecules of each type balance on the left- and right-hand sides of the equation. Writing the variable overall reaction with individual equations for hydrogen and oxygen one finds:

$$\begin{aligned} x_1\mathrm{H_2} + x_2\mathrm{O_2} &\rightarrow \mathrm{H_2O} \\ \mathrm{H}: \quad 2x_1 + 0x_2 &\rightarrow 2 \\ \mathrm{O}: \quad 0x_1 + 2x_2 &\rightarrow 1 \end{aligned}$$

The solution in Octave is found in just three steps.

```
octave:1> A = [ 2, 0; 0, 2 ];
octave:2> b = [ 2; 1 ];
octave:3> x = A \ b
```

1.2.5 Integrating Differential Equations

Octave has built-in functions for solving nonlinear differential equations of the form

$$\frac{dx}{dt} = f(x,t), \qquad x(t = t_0) = x_0$$

For Octave to integrate equations of this form, you must first provide a definition of the function $f(x,t)$. This is straightforward, and may be accomplished by entering the function body directly on the command line. For example, the following commands define the right-hand side function for an interesting pair of nonlinear differential equations. Note that while you are entering a function, Octave responds with a different prompt, to indicate that it is waiting for you to complete your input.

```
octave:1> function xdot = f (x, t)
>
>  r = 0.25;
>  k = 1.4;
>  a = 1.5;
>  b = 0.16;
>  c = 0.9;
>  d = 0.8;
>
>  xdot(1) = r*x(1)*(1 - x(1)/k) - a*x(1)*x(2)/(1 + b*x(1));
>  xdot(2) = c*a*x(1)*x(2)/(1 + b*x(1)) - d*x(2);
>
> endfunction
```

Given the initial condition

```
octave:2> x0 = [1; 2];
```

and the set of output times as a column vector (note that the first output time corresponds to the initial condition given above)

```
octave:3> t = linspace (0, 50, 200)';
```

it is easy to integrate the set of differential equations:

```
octave:4> x = lsode ("f", x0, t);
```

The function `lsode` uses the Livermore Solver for Ordinary Differential Equations, described in A. C. Hindmarsh, *ODEPACK, a Systematized Collection of ODE Solvers*, in: Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pages 55–64.

1.2.6 Producing Graphical Output

To display the solution of the previous example graphically, use the command

```
octave:1> plot (t, x)
```

If you are using a graphical user interface, Octave will automatically create a separate window to display the plot.

To save a plot once it has been displayed on the screen, use the print command. For example,

```
print -dpdf foo.pdf
```

will create a file called `foo.pdf` that contains a rendering of the current plot in Portable Document Format. The command

```
help print
```

explains more options for the `print` command and provides a list of additional output file formats.

1.2.7 Editing What You Have Typed

At the Octave prompt, you can recall, edit, and reissue previous commands using Emacs- or vi-style editing commands. The default keybindings use Emacs-style commands. For example, to recall the previous command, press *`Control-p`* (written *`C-p`* for short). Doing this will normally bring back the previous line of input. *`C-n`* will bring up the next line of input, *`C-b`* will move the cursor backward on the line, *`C-f`* will move the cursor forward on the line, etc.

A complete description of the command line editing capability is given in this manual, see Section 2.4 [Command Line Editing], page 25.

1.2.8 Help and Documentation

Octave has an extensive help facility. The same documentation that is available in printed form is also available from the Octave prompt, because both forms of the documentation are created from the same input file.

In order to get good help you first need to know the name of the command that you want to use. The name of this function may not always be obvious, but a good place to start is to type `help --list`. This will show you all the operators, keywords, built-in functions, and loadable functions available in the current session of Octave. An alternative is to search the documentation using the `lookfor` function (described in Section 2.3 [Getting Help], page 20).

Once you know the name of the function you wish to use, you can get more help on the function by simply including the name as an argument to help. For example,

```
help plot
```

will display the help text for the `plot` function.

Octave sends output that is too long to fit on one screen through a pager like `less` or `more`. Type a `RET` to advance one line, a `SPC` to advance one page, and `q` to quit the pager.

The part of Octave's help facility that allows you to read the complete text of the printed manual from within Octave normally uses a separate program called Info. When you invoke Info you will be put into a menu driven program that contains the entire Octave manual. Help for using Info is provided in this manual, see Section 2.3 [Getting Help], page 20.

1.3 Conventions

This section explains the notational conventions that are used in this manual. You may want to skip this section and refer back to it later.

1.3.1 Fonts

Examples of Octave code appear in this font or form: `svd (a)`. Names that represent variables or function arguments appear in this font or form: *first-number*. Commands that you type at the shell prompt appear in this font or form: '`octave --no-init-file`'. Commands that you type at the Octave prompt sometimes appear in this font or form: *`foo --bar --baz`*. Specific keys on your keyboard appear in this font or form: `RET`.

1.3.2 Evaluation Notation

In the examples in this manual, results from expressions that you evaluate are indicated with '⇒'. For example:

```
sqrt (2)
     ⇒ 1.4142
```

You can read this as "`sqrt (2)` evaluates to 1.4142".

In some cases, matrix values that are returned by expressions are displayed like this

```
[1, 2; 3, 4] == [1, 3; 2, 4]
     ⇒ [ 1, 0; 0, 1 ]
```

and in other cases, they are displayed like this

```
eye (3)
     ⇒  1  0  0
        0  1  0
        0  0  1
```

in order to clearly show the structure of the result.

Sometimes to help describe one expression, another expression is shown that produces identical results. The exact equivalence of expressions is indicated with '≡'. For example:

```
rot90 ([1, 2; 3, 4], -1)
≡
rot90 ([1, 2; 3, 4], 3)
≡
rot90 ([1, 2; 3, 4], 7)
```

1.3.3 Printing Notation

Many of the examples in this manual print text when they are evaluated. In this manual the printed text resulting from an example is indicated by '⊣'. The value that is returned by evaluating the expression is displayed with '⇒' (1 in the next example) and follows on a separate line.

```
printf ("foo %s\n", "bar")
     ⊣ foo bar
     ⇒ 1
```

1.3.4 Error Messages

Some examples signal errors. This normally displays an error message on your terminal. Error messages are shown on a line beginning with `error:`.

```
fieldnames ([1, 2; 3, 4])
error: fieldnames: Invalid input argument
```

1.3.5 Format of Descriptions

Functions and commands are described in this manual in a uniform format. The first line of a description contains the name of the item followed by its arguments, if any. The category—function, command, or whatever—is printed next to the right margin. If there are multiple ways to invoke the function then each allowable form is listed.

The description follows on succeeding lines, sometimes with examples.

1.3.5.1 A Sample Function Description

In a function description, the name of the function being described appears first. It is followed on the same line by a list of parameters. The names used for the parameters are also used in the body of the description.

After all of the calling forms have been enumerated, the next line is a concise one-sentence summary of the function.

After the summary there may be documentation on the inputs and outputs, examples of function usage, notes about the algorithm used, and references to related functions.

Here is a description of an imaginary function `foo`:

`foo` (*x*) [Function File]
`foo` (*x*, *y*) [Function File]
`foo` (*x*, *y*, ...) [Function File]

The function `foo` subtracts *x* from *y*, then adds the remaining arguments to the result.

If *y* is not supplied, then the number 19 is used by default.

Example:

```
foo (1, [3, 5], 3, 9)
     ⇒ [ 14, 16 ]
foo (5)
     ⇒ 14
```

More generally,

```
foo (w, x, y, ...)
≡
x - w + y + ...
```

See also: bar

Any parameter whose name contains the name of a type (e.g., *integer* or *matrix*) is expected to be of that type. Parameters named *object* may be of any type. Parameters with other sorts of names (e.g., *new_file*) are discussed specifically in the description of the function. In some sections, features common to parameters of several functions are described at the beginning.

Functions in Octave may be defined in several different ways. The category name for functions may include a tag that indicates the way that the function is defined. These additional tags include

Function File
: The function described is defined using Octave commands stored in a text file. See Section 11.9 [Function Files], page 187.

Built-in Function
: The function described is written in a language like C++, C, or Fortran, and is part of the compiled Octave binary.

Loadable Function
: The function described is written in a language like C++, C, or Fortran. On systems that support dynamic linking of user-supplied functions, it may be automatically linked while Octave is running, but only if it is needed. See Appendix A [External Code Interface], page 817.

Mapping Function
: The function described works element-by-element for matrix and vector arguments.

1.3.5.2 A Sample Command Description

Command descriptions have a format similar to function descriptions, except that the word ‘Function’ is replaced by ‘Command’. Commands are functions that may be called without surrounding their arguments in parentheses. For example, here is the description for Octave’s `diary` command:

`diary` [Command]
`diary` *on* [Command]
`diary` *off* [Command]
`diary` *filename* [Command]

Record a list of all commands *and* the output they produce, mixed together just as they appear on the terminal.

Valid options are:

on
: Start recording a session in a file called `diary` in the current working directory.

off
: Stop recording the session in the diary file.

filename
: Record the session in the file named *filename*.

With no arguments, `diary` toggles the current diary state.

See also: history.

2 Getting Started

This chapter explains some of Octave's basic features, including how to start an Octave session, get help at the command prompt, edit the command line, and write Octave programs that can be executed as commands from your shell.

2.1 Invoking Octave from the Command Line

Normally, Octave is used interactively by running the program '`octave`' without any arguments. Once started, Octave reads commands from the terminal until you tell it to exit.

You can also specify the name of a file on the command line, and Octave will read and execute the commands from the named file and then exit when it is finished.

You can further control how Octave starts by using the command-line options described in the next section, and Octave itself can remind you of the options available. Type '`octave --help`' to display all available options and briefly describe their use ('`octave -h`' is a shorter equivalent).

2.1.1 Command Line Options

Here is a complete list of the command line options that Octave accepts.

`--built-in-docstrings-file` *`filename`*
: Specify the name of the file containing documentation strings for the built-in functions of Octave. This value is normally correct and should only need to specified in extraordinary situations.

`--debug`
`-d`
: Enter parser debugging mode. Using this option will cause Octave's parser to print a lot of information about the commands it reads, and is probably only useful if you are actually trying to debug the parser.

`--debug-jit`
: Enable JIT compiler debugging and tracing.

`--doc-cache-file` *`filename`*
: Specify the name of the doc cache file to use. The value of *filename* specified on the command line will override any value of `OCTAVE_DOC_CACHE_FILE` found in the environment, but not any commands in the system or user startup files that use the `doc_cache_file` function.

`--echo-commands`
`-x`
: Echo commands as they are executed.

`--eval` *`code`*
: Evaluate *code* and exit when finished unless `--persist` is also specified.

`--exec-path` *`path`*
: Specify the path to search for programs to run. The value of *path* specified on the command line will override any value of `OCTAVE_EXEC_PATH` found in the environment, but not any commands in the system or user startup files that set the built-in variable `EXEC_PATH`.

`--force-gui`
: Force the graphical user interface (GUI) to start.

`--help`
`-h`
`-?`
: Print short help message and exit.

`--image-path` *path*
: Add path to the head of the search path for images. The value of *path* specified on the command line will override any value of `OCTAVE_IMAGE_PATH` found in the environment, but not any commands in the system or user startup files that set the built-in variable `IMAGE_PATH`.

`--info-file` *filename*
: Specify the name of the info file to use. The value of *filename* specified on the command line will override any value of `OCTAVE_INFO_FILE` found in the environment, but not any commands in the system or user startup files that use the `info_file` function.

`--info-program` *program*
: Specify the name of the info program to use. The value of *program* specified on the command line will override any value of `OCTAVE_INFO_PROGRAM` found in the environment, but not any commands in the system or user startup files that use the `info_program` function.

`--interactive`
`-i`
: Force interactive behavior. This can be useful for running Octave via a remote shell command or inside an Emacs shell buffer. For another way to run Octave within Emacs, see Appendix H [Emacs Octave Support], page 907.

`--jit-compiler`
: Enable the JIT compiler used for accelerating loops.

`--line-editing`
: Force readline use for command-line editing.

`--no-gui`
: Disable the graphical user interface (GUI) and use the command line interface (CLI) instead.

`--no-history`
`-H`
: Disable recording of command-line history.

`--no-init-file`
: Don't read the initialization files `~/.octaverc` and `.octaverc`.

`--no-init-path`
: Don't initialize the search path for function files to include default locations.

`--no-line-editing`
: Disable command-line editing.

`--no-site-file`
: Don't read the site-wide `octaverc` initialization files.

`--no-window-system`
`-W` Disable use of a windowing system including graphics. This forces a strictly terminal-only environment.

`--norc`
`-f` Don't read any of the system or user initialization files at startup. This is equivalent to using both of the options `--no-init-file` and `--no-site-file`.

`--path` *path*
`-p` *path* Add path to the head of the search path for function files. The value of *path* specified on the command line will override any value of `OCTAVE_PATH` found in the environment, but not any commands in the system or user startup files that set the internal load path through one of the path functions.

`--persist`
Go to interactive mode after `--eval` or reading from a file named on the command line.

`--silent`
`--quiet`
`-q` Don't print the usual greeting and version message at startup.

`--texi-macros-file` *filename*
Specify the name of the file containing Texinfo macros for use by makeinfo.

`--traditional`
`--braindead`
For compatibility with MATLAB, set initial values for user preferences to the following values

```
PS1                             = ">> "
PS2                             = ""
beep_on_error                   = true
confirm_recursive_rmdir         = false
crash_dumps_octave_core         = false
disable_diagonal_matrix         = true
disable_permutation_matrix      = true
disable_range                   = true
fixed_point_format              = true
history_timestamp_format_string = "%%-- %D %I:%M %p --%%"
page_screen_output              = false
print_empty_dimensions          = false
save_default_options            = "-mat-binary"
struct_levels_to_print          = 0
```

and disable the following warnings

```
Octave:abbreviated-property-match
Octave:fopen-file-in-path
Octave:function-name-clash
Octave:load-file-in-path
Octave:possible-matlab-short-circuit-operator
```

Note that this does not enable the `Octave:language-extension` warning, which you might want if you want to be told about writing code that works in Octave but not MATLAB (see [warning], page 212, [warning_ids], page 214).

`--verbose`
`-V` Turn on verbose output.

`--version`
`-v` Print the program version number and exit.

`file` Execute commands from *file*. Exit when done unless `--persist` is also specified.

Octave also includes several functions which return information about the command line, including the number of arguments and all of the options.

`argv` () [Built-in Function]

Return the command line arguments passed to Octave.

For example, if you invoked Octave using the command

```
octave --no-line-editing --silent
```

`argv` would return a cell array of strings with the elements `--no-line-editing` and `--silent`.

If you write an executable Octave script, `argv` will return the list of arguments passed to the script. See Section 2.6 [Executable Octave Programs], page 36, for an example of how to create an executable Octave script.

`program_name` () [Built-in Function]

Return the last component of the value returned by `program_invocation_name`.

See also: [program_invocation_name], page 18.

`program_invocation_name` () [Built-in Function]

Return the name that was typed at the shell prompt to run Octave.

If executing a script from the command line (e.g., `octave foo.m`) or using an executable Octave script, the program name is set to the name of the script. See Section 2.6 [Executable Octave Programs], page 36, for an example of how to create an executable Octave script.

See also: [program_name], page 18.

Here is an example of using these functions to reproduce the command line which invoked Octave.

```
printf ("%s", program_name ());
arg_list = argv ();
for i = 1:nargin
  printf (" %s", arg_list{i});
endfor
printf ("\n");
```

See Section 6.2.3 [Indexing Cell Arrays], page 116, for an explanation of how to retrieve objects from cell arrays, and Section 11.2 [Defining Functions], page 171, for information about the variable `nargin`.

2.1.2 Startup Files

When Octave starts, it looks for commands to execute from the files in the following list. These files may contain any valid Octave commands, including function definitions.

`octave-home/share/octave/site/m/startup/octaverc`
: where *octave-home* is the directory in which Octave is installed (the default is `/usr/local`). This file is provided so that changes to the default Octave environment can be made globally for all users at your site for all versions of Octave you have installed. Care should be taken when making changes to this file since all users of Octave at your site will be affected. The default file may be overridden by the environment variable `OCTAVE_SITE_INITFILE`.

`octave-home/share/octave/version/m/startup/octaverc`
: where *octave-home* is the directory in which Octave is installed (the default is `/usr/local`), and *version* is the version number of Octave. This file is provided so that changes to the default Octave environment can be made globally for all users of a particular version of Octave. Care should be taken when making changes to this file since all users of Octave at your site will be affected. The default file may be overridden by the environment variable `OCTAVE_VERSION_INITFILE`.

`~/.octaverc`
: This file is used to make personal changes to the default Octave environment.

`.octaverc`
: This file can be used to make changes to the default Octave environment for a particular project. Octave searches for this file in the current directory after it reads `~/.octaverc`. Any use of the `cd` command in the `~/.octaverc` file will affect the directory where Octave searches for `.octaverc`.

  If you start Octave in your home directory, commands from the file `~/.octaverc` will only be executed once.

A message will be displayed as each of the startup files is read if you invoke Octave with the `--verbose` option but without the `--silent` option.

2.2 Quitting Octave

Shutdown is initiated with the `exit` or `quit` commands (they are equivalent). Similar to startup, Octave has a shutdown process that can be customized by user script files. During shutdown Octave will search for the script file `finish.m` in the function load path. Commands to save all workspace variables or cleanup temporary files may be placed there. Additional functions to execute on shutdown may be registered with `atexit`.

`exit` [Built-in Function]
`exit (status)` [Built-in Function]
`quit` [Built-in Function]
`quit (status)` [Built-in Function]

Exit the current Octave session.

If the optional integer value *status* is supplied, pass that value to the operating system as Octave's exit status. The default value is zero.

When exiting, Octave will attempt to run the m-file `finish.m` if it exists. User commands to save the workspace or clean up temporary files may be placed in that file. Alternatively, another m-file may be scheduled to run using `atexit`.

See also: [atexit], page 20.

`atexit (`*fcn*`)` [Built-in Function]
`atexit (`*fcn*, *flag*`)` [Built-in Function]

Register a function to be called when Octave exits.

For example,

```
function last_words ()
  disp ("Bye bye");
endfunction
atexit ("last_words");
```

will print the message `"Bye bye"` when Octave exits.

The additional argument *flag* will register or unregister *fcn* from the list of functions to be called when Octave exits. If *flag* is true, the function is registered, and if *flag* is false, it is unregistered. For example, after registering the function `last_words` above,

```
atexit ("last_words", false);
```

will remove the function from the list and Octave will not call `last_words` when it exits.

Note that `atexit` only removes the first occurrence of a function from the list, so if a function was placed in the list multiple times with `atexit`, it must also be removed from the list multiple times.

See also: [quit], page 19.

2.3 Commands for Getting Help

The entire text of this manual is available from the Octave prompt via the command *doc*. In addition, the documentation for individual user-written functions and variables is also available via the *help* command. This section describes the commands used for reading the manual and the documentation strings for user-supplied functions and variables. See Section 11.9 [Function Files], page 187, for more information about how to document the functions you write.

`help` *name* [Command]
`help --list` [Command]
`help .` [Command]
`help` [Command]

Display the help text for *name*.

For example, the command *help help* prints a short message describing the `help` command.

Given the single argument `--list`, list all operators, keywords, built-in functions, and loadable functions available in the current session of Octave.

Given the single argument `.`, list all operators available in the current session of Octave.

If invoked without any arguments, `help` display instructions on how to access help from the command line.

The help command can provide information about most operators, for example `help +`, but not the comma and semicolon characters which are used by the Octave interpreter as command separators. For help on either of these type *`help comma`* or *`help semicolon`*.

See also: [doc], page 21, [lookfor], page 21, [which], page 132, [info], page 22.

`doc` *`function_name`* [Command]
`doc` [Command]

Display documentation for the function *function_name* directly from an online version of the printed manual, using the GNU Info browser.

If invoked without an argument, the manual is shown from the beginning.

For example, the command *`doc rand`* starts the GNU Info browser at the `rand` node in the online version of the manual.

Once the GNU Info browser is running, help for using it is available using the command *`C-h`*.

See also: [help], page 20.

`lookfor` *`str`* [Command]
`lookfor` *-all* *`str`* [Command]
[*`fcn`*, *`help1str`*] = `lookfor` (*`str`*) [Function File]
[*`fcn`*, *`help1str`*] = `lookfor` ("*-all*", *`str`*) [Function File]

Search for the string *str* in the documentation of all functions in the current function search path.

By default, `lookfor` looks for *str* in just the first sentence of the help string for each function found. The entire help text of each function can be searched by using the `"-all"` argument. All searches are case insensitive.

When called with no output arguments, `lookfor` prints the list of matching functions to the terminal. Otherwise, the output argument *fcns* contains the function names and *help1str* contains the first sentence from the help string of each function.

Programming Note: The ability of `lookfor` to correctly identify the first sentence of the help text is dependent on the format of the function's help. All Octave core functions are correctly formatted, but the same can not be guaranteed for external packages and user-supplied functions. Therefore, the use of the `"-all"` argument may be necessary to find related functions that are not a part of Octave.

The speed of lookup is greatly enhanced by having a cached documentation file. See `doc_cache_create` for more information.

See also: [help], page 20, [doc], page 21, [which], page 132, [path], page 190, [doc_cache_create], page 24.

To see what is new in the current release of Octave, use the `news` function.

`news` [Command]
`news` *`package`* [Command]

Display the current NEWS file for Octave or an installed package.

When called without an argument, display the NEWS file for Octave.

When given a package name *package*, display the current NEWS file for that package.

See also: [ver], page 790, [pkg], page 807.

`info ()` [Function File]

Display contact information for the GNU Octave community.

`warranty ()` [Built-in Function]

Describe the conditions for copying and distributing Octave.

The following functions can be used to change which programs are used for displaying the documentation, and where the documentation can be found.

`val` `= info_file ()` [Built-in Function]
`old_val` `= info_file (`*`new_val`*`)` [Built-in Function]
`info_file (`*`new_val`*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that specifies the name of the Octave info file.

The default value is *`octave-home`*`/info/octave.info`, in which *octave-home* is the root directory of the Octave installation. The default value may be overridden by the environment variable `OCTAVE_INFO_FILE`, or the command line argument `--info-file FNAME`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [info_program], page 22, [doc], page 21, [help], page 20, [makeinfo_program], page 23.

`val` `= info_program ()` [Built-in Function]
`old_val` `= info_program (`*`new_val`*`)` [Built-in Function]
`info_program (`*`new_val`*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that specifies the name of the info program to run.

The default value is *`octave-home`*`/libexec/octave/`*`version`*`/exec/`*`arch`*`/info` in which *octave-home* is the root directory of the Octave installation, *version* is the Octave version number, and *arch* is the system type (for example, `i686-pc-linux-gnu`). The default value may be overridden by the environment variable `OCTAVE_INFO_PROGRAM`, or the command line argument `--info-program NAME`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [info_file], page 22, [doc], page 21, [help], page 20, [makeinfo_program], page 23.

val = `makeinfo_program` () [Built-in Function]
old_val = `makeinfo_program` (*new_val*) [Built-in Function]
`makeinfo_program` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the name of the program that Octave runs to format help text containing Texinfo markup commands.

The default value is `makeinfo`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [texi_macros_file], page 23, [info_file], page 22, [info_program], page 22, [doc], page 21, [help], page 20.

val = `texi_macros_file` () [Built-in Function]
old_val = `texi_macros_file` (*new_val*) [Built-in Function]
`texi_macros_file` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the name of the file containing Texinfo macros that are prepended to documentation strings before they are passed to makeinfo.

The default value is *octave-home*`/share/octave/`*version*`/etc/macros.texi`, in which *octave-home* is the root directory of the Octave installation, and *version* is the Octave version number. The default value may be overridden by the environment variable `OCTAVE_TEXI_MACROS_FILE`, or the command line argument `--texi-macros-file FNAME`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [makeinfo_program], page 23.

val = `doc_cache_file` () [Built-in Function]
old_val = `doc_cache_file` (*new_val*) [Built-in Function]
`doc_cache_file` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the name of the Octave documentation cache file.

A cache file significantly improves the performance of the `lookfor` command. The default value is *octave-home*`/share/octave/`*version*`/etc/doc-cache`, in which *octave-home* is the root directory of the Octave installation, and *version* is the Octave version number. The default value may be overridden by the environment variable `OCTAVE_DOC_CACHE_FILE`, or the command line argument `--doc-cache-file FNAME`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [doc_cache_create], page 24, [lookfor], page 21, [info_program], page 22, [doc], page 21, [help], page 20, [makeinfo_program], page 23.

See also: [lookfor], page 21.

val = built_in_docstrings_file () [Built-in Function]
old_val = built_in_docstrings_file (*new_val*) [Built-in Function]
built_in_docstrings_file (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the name of the file containing docstrings for built-in Octave functions.

The default value is *octave-home*/share/octave/*version*/etc/built-in-docstrings, in which *octave-home* is the root directory of the Octave installation, and *version* is the Octave version number. The default value may be overridden by the environment variable OCTAVE_BUILT_IN_DOCSTRINGS_FILE, or the command line argument --built-in-docstrings-file FNAME.

Note: This variable is only used when Octave is initializing itself. Modifying it during a running session of Octave will have no effect.

val = suppress_verbose_help_message () [Built-in Function]
old_val = suppress_verbose_help_message (*new_val*) [Built-in Function]
suppress_verbose_help_message (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether Octave will add additional help information to the end of the output from the help command and usage messages for built-in commands.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

The following functions are principally used internally by Octave for generating the documentation. They are documented here for completeness and because they may occasionally be useful for users.

doc_cache_create (*out_file*, *directory*) [Function File]
doc_cache_create (*out_file*) [Function File]
doc_cache_create () [Function File]

Generate documentation cache for all functions in *directory*.

A documentation cache is generated for all functions in *directory* which may be a single string or a cell array of strings. The cache is used to speed up the function lookfor.

The cache is saved in the file *out_file* which defaults to the value doc-cache if not given.

If no directory is given (or it is the empty matrix), a cache for built-in operators, etc. is generated.

See also: [doc_cache_file], page 23, [lookfor], page 21, [path], page 190.

[*text*, *format*] = get_help_text (*name*) [Built-in Function]

Return the raw help text of function *name*.

The raw help text is returned in *text* and the format in *format* The format is a string which is one of "texinfo", "html", or "plain text".

See also: [get_help_text_from_file], page 25.

`[text, format] = get_help_text_from_file (fname)` [Built-in Function]

Return the raw help text from the file *fname*.

The raw help text is returned in *text* and the format in *format* The format is a string which is one of `"texinfo"`, `"html"`, or `"plain text"`.

See also: [get_help_text], page 24.

`text = get_first_help_sentence (name)` [Function File]
`text = get_first_help_sentence (name, max_len)` [Function File]
`[text, status] = get_first_help_sentence (...)` [Function File]

Return the first sentence of a function's help text.

The first sentence is defined as the text after the function declaration until either the first period (`"."`) or the first appearance of two consecutive newlines (`"\n\n"`). The text is truncated to a maximum length of *max_len*, which defaults to 80.

The optional output argument *status* returns the status reported by `makeinfo`. If only one output argument is requested, and *status* is nonzero, a warning is displayed.

As an example, the first sentence of this help text is

```
get_first_help_sentence ("get_first_help_sentence")
⊣ ans = Return the first sentence of a function's help text.
```

2.4 Command Line Editing

Octave uses the GNU Readline library to provide an extensive set of command-line editing and history features. Only the most common features are described in this manual. In addition, all of the editing functions can be bound to different key strokes at the user's discretion. This manual assumes no changes from the default Emacs bindings. See the GNU Readline Library manual for more information on customizing Readline and for a complete feature list.

To insert printing characters (letters, digits, symbols, etc.), simply type the character. Octave will insert the character at the cursor and advance the cursor forward.

Many of the command-line editing functions operate using control characters. For example, the character `Control-a` moves the cursor to the beginning of the line. To type `C-a`, hold down `CTRL` and then press `a`. In the following sections, control characters such as `Control-a` are written as `C-a`.

Another set of command-line editing functions use Meta characters. To type `M-u`, hold down the `META` key and press `u`. Depending on the keyboard, the `META` key may be labeled `ALT` or even `WINDOWS`. If your terminal does not have a `META` key, you can still type Meta characters using two-character sequences starting with `ESC`. Thus, to enter `M-u`, you would type `ESC u`. The `ESC` character sequences are also allowed on terminals with real Meta keys. In the following sections, Meta characters such as `Meta-u` are written as `M-u`.

2.4.1 Cursor Motion

The following commands allow you to position the cursor.

`C-b` Move back one character.

`C-f` Move forward one character.

`BACKSPACE`
: Delete the character to the left of the cursor.

`DEL` Delete the character underneath the cursor.

`C-d` Delete the character underneath the cursor.

`M-f` Move forward a word.

`M-b` Move backward a word.

`C-a` Move to the start of the line.

`C-e` Move to the end of the line.

`C-l` Clear the screen, reprinting the current line at the top.

`C-_`
`C-/` Undo the last action. You can undo all the way back to an empty line.

`M-r` Undo all changes made to this line. This is like typing the 'undo' command enough times to get back to the beginning.

The above table describes the most basic possible keystrokes that you need in order to do editing of the input line. On most terminals, you can also use the left and right arrow keys in place of *`C-f`* and *`C-b`* to move forward and backward.

Notice how *`C-f`* moves forward a character, while *`M-f`* moves forward a word. It is a loose convention that control keystrokes operate on characters while meta keystrokes operate on words.

The function `clc` will allow you to clear the screen from within Octave programs.

`clc ()` [Built-in Function]
`home ()` [Built-in Function]
Clear the terminal screen and move the cursor to the upper left corner.

2.4.2 Killing and Yanking

Killing text means to delete the text from the line, but to save it away for later use, usually by *yanking* it back into the line. If the description for a command says that it 'kills' text, then you can be sure that you can get the text back in a different (or the same) place later.

Here is the list of commands for killing text.

`C-k` Kill the text from the current cursor position to the end of the line.

`M-d` Kill from the cursor to the end of the current word, or if between words, to the end of the next word.

`M-DEL` Kill from the cursor to the start of the previous word, or if between words, to the start of the previous word.

`C-w` Kill from the cursor to the previous whitespace. This is different than *`M-DEL`* because the word boundaries differ.

And, here is how to *yank* the text back into the line. Yanking means to copy the most-recently-killed text from the kill buffer.

`C-y` Yank the most recently killed text back into the buffer at the cursor.

M-y Rotate the kill-ring, and yank the new top. You can only do this if the prior command is *C-y* or *M-y*.

When you use a kill command, the text is saved in a *kill-ring*. Any number of consecutive kills save all of the killed text together, so that when you yank it back, you get it in one clean sweep. The kill ring is not line specific; the text that you killed on a previously typed line is available to be yanked back later, when you are typing another line.

2.4.3 Commands For Changing Text

The following commands can be used for entering characters that would otherwise have a special meaning (e.g., `TAB`, *C-q*, etc.), or for quickly correcting typing mistakes.

C-q
C-v Add the next character that you type to the line verbatim. This is how to insert things like *C-q* for example.

M-TAB Insert a tab character.

C-t Drag the character before the cursor forward over the character at the cursor, also moving the cursor forward. If the cursor is at the end of the line, then transpose the two characters before it.

M-t Drag the word behind the cursor past the word in front of the cursor moving the cursor over that word as well.

M-u Uppercase the characters following the cursor to the end of the current (or following) word, moving the cursor to the end of the word.

M-l Lowercase the characters following the cursor to the end of the current (or following) word, moving the cursor to the end of the word.

M-c Uppercase the character following the cursor (or the beginning of the next word if the cursor is between words), moving the cursor to the end of the word.

2.4.4 Letting Readline Type For You

The following commands allow Octave to complete command and variable names for you.

`TAB` Attempt to do completion on the text before the cursor. Octave can complete the names of commands and variables.

M-? List the possible completions of the text before the cursor.

val = `completion_append_char ()` [Built-in Function]
old_val = `completion_append_char (`*new_val*`)` [Built-in Function]
`completion_append_char (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal character variable that is appended to successful command-line completion attempts.

The default value is `" "` (a single space).

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

`completion_matches (`*`hint`*`)` [Built-in Function]

Generate possible completions given *hint*.

This function is provided for the benefit of programs like Emacs which might be controlling Octave and handling user input. The current command number is not incremented when this function is called. This is a feature, not a bug.

2.4.5 Commands For Manipulating The History

Octave normally keeps track of the commands you type so that you can recall previous commands to edit or execute them again. When you exit Octave, the most recent commands you have typed, up to the number specified by the variable `history_size`, are saved in a file. When Octave starts, it loads an initial list of commands from the file named by the variable `history_file`.

Here are the commands for simple browsing and searching the history list.

`LFD`
`RET` Accept the current line regardless of where the cursor is. If the line is non-empty, add it to the history list. If the line was a history line, then restore the history line to its original state.

`C-p` Move 'up' through the history list.

`C-n` Move 'down' through the history list.

`M-<` Move to the first line in the history.

`M->` Move to the end of the input history, i.e., the line you are entering!

`C-r` Search backward starting at the current line and moving 'up' through the history as necessary. This is an incremental search.

`C-s` Search forward starting at the current line and moving 'down' through the history as necessary.

On most terminals, you can also use the up and down arrow keys in place of *`C-p`* and *`C-n`* to move through the history list.

In addition to the keyboard commands for moving through the history list, Octave provides three functions for viewing, editing, and re-running chunks of commands from the history list.

`history` [Command]
`history` *`opt1`* `...` [Command]
`h` `= history ()` [Built-in Function]
`h` `= history (`*`opt1`*`, ...)` [Built-in Function]

If invoked with no arguments, `history` displays a list of commands that you have executed.

Valid options are:

`n`
`-`*`n`* Display only the most recent *n* lines of history.

`-c` Clear the history list.

`-q` Don't number the displayed lines of history. This is useful for cutting and pasting commands using the X Window System.

`-r` *file* Read the file *file*, appending its contents to the current history list. If the name is omitted, use the default history file (normally `~/.octave_hist`).

`-w` *file* Write the current history to the file *file*. If the name is omitted, use the default history file (normally `~/.octave_hist`).

For example, to display the five most recent commands that you have typed without displaying line numbers, use the command `history -q 5`.

If invoked with a single output argument, the history will be saved to that argument as a cell string and will not be output to screen.

See also: [edit_history], page 29, [run_history], page 29.

`edit_history` [Command]
`edit_history` *cmd_number* [Command]
`edit_history` *first last* [Command]

Edit the history list using the editor named by the variable `EDITOR`.

The commands to be edited are first copied to a temporary file. When you exit the editor, Octave executes the commands that remain in the file. It is often more convenient to use `edit_history` to define functions rather than attempting to enter them directly on the command line. The block of commands is executed as soon as you exit the editor. To avoid executing any commands, simply delete all the lines from the buffer before leaving the editor.

When invoked with no arguments, edit the previously executed command; With one argument, edit the specified command *cmd_number*; With two arguments, edit the list of commands between *first* and *last*. Command number specifiers may also be negative where -1 refers to the most recently executed command. The following are equivalent and edit the most recently executed command.

```
edit_history
edit_history -1
```

When using ranges, specifying a larger number for the first command than the last command reverses the list of commands before they are placed in the buffer to be edited.

See also: [run_history], page 29, [history], page 28.

`run_history` [Command]
`run_history` *cmd_number* [Command]
`run_history` *first last* [Command]

Run commands from the history list.

When invoked with no arguments, run the previously executed command;

With one argument, run the specified command *cmd_number*;

With two arguments, run the list of commands between *first* and *last*. Command number specifiers may also be negative where -1 refers to the most recently executed command. For example, the command

```
run_history
     OR
run_history -1
```

executes the most recent command again. The command

```
run_history 13 169
```

executes commands 13 through 169.

Specifying a larger number for the first command than the last command reverses the list of commands before executing them. For example:

```
disp (1)
disp (2)
run_history -1 -2
⇒
 2
 1
```

See also: [edit_history], page 29, [history], page 28.

Octave also allows you customize the details of when, where, and how history is saved.

val = `history_save` () [Built-in Function]
old_val = `history_save` (*new_val*) [Built-in Function]
`history_save` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether commands entered on the command line are saved in the history file.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [history_control], page 30, [history_file], page 31, [history_size], page 31, [history_timestamp_format_string], page 31.

val = `history_control` () [Built-in Function]
old_val = `history_control` (*new_val*) [Built-in Function]

Query or set the internal variable that specifies how commands are saved to the history list.

The default value is an empty character string, but may be overridden by the environment variable `OCTAVE_HISTCONTROL`.

The value of `history_control` is a colon-separated list of values controlling how commands are saved on the history list. If the list of values includes `ignorespace`, lines which begin with a space character are not saved in the history list. A value of `ignoredups` causes lines matching the previous history entry to not be saved. A value of `ignoreboth` is shorthand for `ignorespace` and `ignoredups`. A value of `erasedups` causes all previous lines matching the current line to be removed from the history list before that line is saved. Any value not in the above list is ignored. If `history_control` is the empty string, all commands are saved on the history list, subject to the value of `history_save`.

See also: [history_file], page 31, [history_size], page 31, [history_timestamp_format_string], page 31, [history_save], page 30.

val = `history_file ()` [Built-in Function]
old_val = `history_file (`*new_val*`)` [Built-in Function]

Query or set the internal variable that specifies the name of the file used to store command history.

The default value is `~/.octave_hist`, but may be overridden by the environment variable `OCTAVE_HISTFILE`.

See also: [history_size], page 31, [history_save], page 30, [history_timestamp_format_string], page 31.

val = `history_size ()` [Built-in Function]
old_val = `history_size (`*new_val*`)` [Built-in Function]

Query or set the internal variable that specifies how many entries to store in the history file.

The default value is `1000`, but may be overridden by the environment variable `OCTAVE_HISTSIZE`.

See also: [history_file], page 31, [history_timestamp_format_string], page 31, [history_save], page 30.

val = `history_timestamp_format_string ()` [Built-in Function]
old_val = `history_timestamp_format_string (`*new_val*`)` [Built-in Function]
`history_timestamp_format_string (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that specifies the format string for the comment line that is written to the history file when Octave exits.

The format string is passed to `strftime`. The default value is

```
"# Octave VERSION, %a %b %d %H:%M:%S %Y %Z <USER@HOST>"
```

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [strftime], page 751, [history_file], page 31, [history_size], page 31, [history_save], page 30.

val = `EDITOR ()` [Built-in Function]
old_val = `EDITOR (`*new_val*`)` [Built-in Function]
`EDITOR (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that specifies the default text editor.

The default value is taken from the environment variable `EDITOR` when Octave starts. If the environment variable is not initialized, `EDITOR` will be set to `"emacs"`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [edit], page 187, [edit_history], page 29.

2.4.6 Customizing `readline`

Octave uses the GNU Readline library for command-line editing and history features. Readline is very flexible and can be modified through a configuration file of commands (See the GNU Readline library for the exact command syntax). The default configuration file is normally `~/.inputrc`.

Octave provides two commands for initializing Readline and thereby changing the command line behavior.

`readline_read_init_file (`*`file`*`)` [Built-in Function]

Read the readline library initialization file *file*.

If *file* is omitted, read the default initialization file (normally `~/.inputrc`).

See Section "Readline Init File" in *GNU Readline Library*, for details.

See also: [readline_re_read_init_file], page 32.

`readline_re_read_init_file ()` [Built-in Function]

Re-read the last readline library initialization file that was read.

See Section "Readline Init File" in *GNU Readline Library*, for details.

See also: [readline_read_init_file], page 32.

2.4.7 Customizing the Prompt

The following variables are available for customizing the appearance of the command-line prompts. Octave allows the prompt to be customized by inserting a number of backslash-escaped special characters that are decoded as follows:

'`\t`' The time.

'`\d`' The date.

'`\n`' Begins a new line by printing the equivalent of a carriage return followed by a line feed.

'`\s`' The name of the program (usually just '`octave`').

'`\w`' The current working directory.

'`\W`' The basename of the current working directory.

'`\u`' The username of the current user.

'`\h`' The hostname, up to the first '.'.

'`\H`' The hostname.

'`\#`' The command number of this command, counting from when Octave starts.

'`\!`' The history number of this command. This differs from '`\#`' by the number of commands in the history list when Octave starts.

'`\$`' If the effective UID is 0, a '`#`', otherwise a '`$`'.

'`\nnn`' The character whose character code in octal is *nnn*.

'`\\`' A backslash.

val = PS1 () [Built-in Function]
old_val = PS1 (*new_val*) [Built-in Function]
PS1 (*new_val*, "*local*") [Built-in Function]

Query or set the primary prompt string.

When executing interactively, Octave displays the primary prompt when it is ready to read a command.

The default value of the primary prompt string is `"octave:\#> "`. To change it, use a command like

```
PS1 ("\\u@\\H> ")
```

which will result in the prompt '`boris@kremvax> `' for the user '`boris`' logged in on the host '`kremvax.kgb.su`'. Note that two backslashes are required to enter a backslash into a double-quoted character string. See Chapter 5 [Strings], page 67.

You can also use ANSI escape sequences if your terminal supports them. This can be useful for coloring the prompt. For example,

```
PS1 ("\\[\\033[01;31m\\]\\s:\\#> \\[\\033[0m\\]")
```

will give the default Octave prompt a red coloring.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [PS2], page 33, [PS4], page 33.

val = PS2 () [Built-in Function]
old_val = PS2 (*new_val*) [Built-in Function]
PS2 (*new_val*, "*local*") [Built-in Function]

Query or set the secondary prompt string.

The secondary prompt is printed when Octave is expecting additional input to complete a command. For example, if you are typing a `for` loop that spans several lines, Octave will print the secondary prompt at the beginning of each line after the first. The default value of the secondary prompt string is `"> "`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [PS1], page 32, [PS4], page 33.

val = PS4 () [Built-in Function]
old_val = PS4 (*new_val*) [Built-in Function]
PS4 (*new_val*, "*local*") [Built-in Function]

Query or set the character string used to prefix output produced when echoing commands is enabled.

The default value is `"+ "`. See Section 2.4.8 [Diary and Echo Commands], page 34, for a description of echoing commands.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [echo], page 34, [echo_executing_commands], page 34, [PS1], page 32, [PS2], page 33.

2.4.8 Diary and Echo Commands

Octave's diary feature allows you to keep a log of all or part of an interactive session by recording the input you type and the output that Octave produces in a separate file.

`diary` [Command]
`diary` *on* [Command]
`diary` *off* [Command]
`diary` *`filename`* [Command]

Record a list of all commands *and* the output they produce, mixed together just as they appear on the terminal.

Valid options are:

on — Start recording a session in a file called `diary` in the current working directory.

off — Stop recording the session in the diary file.

filename — Record the session in the file named *filename*.

With no arguments, `diary` toggles the current diary state.

See also: [history], page 28.

Sometimes it is useful to see the commands in a function or script as they are being evaluated. This can be especially helpful for debugging some kinds of problems.

`echo` [Command]
`echo` *on* [Command]
`echo` *off* [Command]
`echo` *on all* [Command]
`echo` *off all* [Command]

Control whether commands are displayed as they are executed.

Valid options are:

`on` — Enable echoing of commands as they are executed in script files.

`off` — Disable echoing of commands as they are executed in script files.

`on all` — Enable echoing of commands as they are executed in script files and functions.

`off all` — Disable echoing of commands as they are executed in script files and functions.

With no arguments, `echo` toggles the current echo state.

`val` `= echo_executing_commands ()` [Built-in Function]
`old_val` `= echo_executing_commands (`*`new_val`*`)` [Built-in Function]
`echo_executing_commands (`*`new_val`*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that controls the echo state.

It may be the sum of the following values:

1 Echo commands read from script files.

2 Echo commands from functions.

4 Echo commands read from command line.

More than one state can be active at once. For example, a value of 3 is equivalent to the command *echo on all*.

The value of `echo_executing_commands` may be set by the *echo* command or the command line option `--echo-commands`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

2.5 How Octave Reports Errors

Octave reports two kinds of errors for invalid programs.

A *parse error* occurs if Octave cannot understand something you have typed. For example, if you misspell a keyword,

```
octave:13> function y = f (x) y = x***2; endfunction
```

Octave will respond immediately with a message like this:

```
parse error:

  syntax error

>>> function y = f (x) y = x***2; endfunction
                                ^
```

For most parse errors, Octave uses a caret ('^') to mark the point on the line where it was unable to make sense of your input. In this case, Octave generated an error message because the keyword for exponentiation (`**`) was misspelled. It marked the error at the third '`*`' because the code leading up to this was correct but the final '`*`' was not understood.

Another class of error message occurs at evaluation time. These errors are called *run-time errors*, or sometimes *evaluation errors*, because they occur when your program is being *run*, or *evaluated*. For example, if after correcting the mistake in the previous function definition, you type

```
octave:13> f ()
```

Octave will respond with

```
error: 'x' undefined near line 1 column 24
error: called from:
error:   f at line 1, column 22
```

This error message has several parts, and gives quite a bit of information to help you locate the source of the error. The messages are generated from the point of the innermost error, and provide a traceback of enclosing expressions and function calls.

In the example above, the first line indicates that a variable named '`x`' was found to be undefined near line 1 and column 24 of some function or expression. For errors occurring within functions, lines are counted from the beginning of the file containing the function definition. For errors occurring outside of an enclosing function, the line number indicates the input line number, which is usually displayed in the primary prompt string.

The second and third lines of the error message indicate that the error occurred within the function `f`. If the function `f` had been called from within another function, for example, `g`, the list of errors would have ended with one more line:

```
error:   g at line 1, column 17
```

These lists of function calls make it fairly easy to trace the path your program took before the error occurred, and to correct the error before trying again.

2.6 Executable Octave Programs

Once you have learned Octave, you may want to write self-contained Octave scripts, using the '`#!`' script mechanism. You can do this on GNU systems and on many Unix systems[1].

Self-contained Octave scripts are useful when you want to write a program which users can invoke without knowing that the program is written in the Octave language. Octave scripts are also used for batch processing of data files. Once an algorithm has been developed and tested in the interactive portion of Octave, it can be committed to an executable script and used again and again on new data files.

As a trivial example of an executable Octave script, you might create a text file named `hello`, containing the following lines:

```
#! octave-interpreter-name -qf
# a sample Octave program
printf ("Hello, world!\n");
```

(where *octave-interpreter-name* should be replaced with the full path and name of your Octave binary). Note that this will only work if '`#!`' appears at the very beginning of the file. After making the file executable (with the `chmod` command on Unix systems), you can simply type:

```
hello
```

at the shell, and the system will arrange to run Octave as if you had typed:

```
octave hello
```

The line beginning with '`#!`' lists the full path and filename of an interpreter to be run, and an optional initial command line argument to pass to that interpreter. The operating system then runs the interpreter with the given argument and the full argument list of the executed program. The first argument in the list is the full file name of the Octave executable. The rest of the argument list will either be options to Octave, or data files, or both. The '`-qf`' options are usually specified in stand-alone Octave programs to prevent them from printing the normal startup message, and to keep them from behaving differently depending on the contents of a particular user's `~/.octaverc` file. See Section 2.1 [Invoking Octave from the Command Line], page 15.

Note that some operating systems may place a limit on the number of characters that are recognized after '`#!`'. Also, the arguments appearing in a '`#!`' line are parsed differently

[1] The '`#!`' mechanism works on Unix systems derived from Berkeley Unix, System V Release 4, and some System V Release 3 systems.

by various shells/systems. The majority of them group all the arguments together in one string and pass it to the interpreter as a single argument. In this case, the following script:

```
#! octave-interpreter-name -q -f # comment
```

is equivalent to typing at the command line:

```
octave "-q -f # comment"
```

which will produce an error message. Unfortunately, it is not possible for Octave to determine whether it has been called from the command line or from a '#!' script, so some care is needed when using the '#!' mechanism.

Note that when Octave is started from an executable script, the built-in function `argv` returns a cell array containing the command line arguments passed to the executable Octave script, not the arguments passed to the Octave interpreter on the '#!' line of the script. For example, the following program will reproduce the command line that was used to execute the script, not '-qf'.

```
#! /bin/octave -qf
printf ("%s", program_name ());
arg_list = argv ();
for i = 1:nargin
  printf (" %s", arg_list{i});
endfor
printf ("\n");
```

2.7 Comments in Octave Programs

A *comment* is some text that is included in a program for the sake of human readers, and which is NOT an executable part of the program. Comments can explain what the program does, and how it works. Nearly all programming languages have provisions for comments, because programs are typically hard to understand without them.

2.7.1 Single Line Comments

In the Octave language, a comment starts with either the sharp sign character, '#', or the percent symbol '%' and continues to the end of the line. Any text following the sharp sign or percent symbol is ignored by the Octave interpreter and not executed. The following example shows whole line and partial line comments.

```
function countdown
  # Count down for main rocket engines
  disp (3);
  disp (2);
  disp (1);
  disp ("Blast Off!");  # Rocket leaves pad
endfunction
```

2.7.2 Block Comments

Entire blocks of code can be commented by enclosing the code between matching '#{' and '#}' or '%{' and '%}' markers. For example,

```
function quick_countdown
  # Count down for main rocket engines
  disp (3);
 #{
  disp (2);
  disp (1);
 #}
  disp ("Blast Off!");  # Rocket leaves pad
endfunction
```

will produce a very quick countdown from '3' to "Blast Off" as the lines "`disp (2);`" and "`disp (1);`" won't be executed.

The block comment markers must appear alone as the only characters on a line (excepting whitespace) in order to be parsed correctly.

2.7.3 Comments and the Help System

The `help` command (see Section 2.3 [Getting Help], page 20) is able to find the first block of comments in a function and return those as a documentation string. This means that the same commands used to get help on built-in functions are available for properly formatted user-defined functions. For example, after defining the function `f` below,

```
function xdot = f (x, t)

# usage: f (x, t)
#
# This function defines the right-hand
# side functions for a set of nonlinear
# differential equations.

  r = 0.25;
  ...
endfunction
```

the command `help f` produces the output

```
 usage: f (x, t)

 This function defines the right-hand
 side functions for a set of nonlinear
 differential equations.
```

Although it is possible to put comment lines into keyboard-composed, throw-away Octave programs, it usually isn't very useful because the purpose of a comment is to help you or another person understand the program at a later time.

The `help` parser currently only recognizes single line comments (see Section 2.7.1 [Single Line Comments], page 37) and not block comments for the initial help text.

3 Data Types

All versions of Octave include a number of built-in data types, including real and complex scalars and matrices, character strings, a data structure type, and an array that can contain all data types.

It is also possible to define new specialized data types by writing a small amount of C++ code. On some systems, new data types can be loaded dynamically while Octave is running, so it is not necessary to recompile all of Octave just to add a new type. See Appendix A [External Code Interface], page 817, for more information about Octave's dynamic linking capabilities. Section 3.2 [User-defined Data Types], page 44, describes what you must do to define a new data type for Octave.

`typeinfo ()` [Built-in Function]
`typeinfo (`*`expr`*`)` [Built-in Function]

Return the type of the expression *expr*, as a string.

If *expr* is omitted, return a cell array of strings containing all the currently installed data types.

See also: [class], page 39, [isa], page 39.

3.1 Built-in Data Types

The standard built-in data types are real and complex scalars and matrices, ranges, character strings, a data structure type, and cell arrays. Additional built-in data types may be added in future versions. If you need a specialized data type that is not currently provided as a built-in type, you are encouraged to write your own user-defined data type and contribute it for distribution in a future release of Octave.

The data type of a variable can be determined and changed through the use of the following functions.

`classname`` = class (`*`obj`*`)` [Function File]
`class (`*`s`*`, `*`id`*`)` [Function File]
`class (`*`s`*`, `*`id`*`, `*`p`*`, ...)` [Function File]

Return the class of the object *obj*, or create a class with fields from structure *s* and name (string) *id*.

Additional arguments name a list of parent classes from which the new class is derived.

See also: [typeinfo], page 39, [isa], page 39.

`isa (`*`obj`*`, `*`classname`*`)` [Function File]

Return true if *obj* is an object from the class *classname*.

classname may also be one of the following class categories:

`"float"` Floating point value comprising classes `"double"` and `"single"`.

`"integer"`
Integer value comprising classes (u)int8, (u)int16, (u)int32, (u)int64.

`"numeric"`
Numeric value comprising either a floating point or integer value.

If *classname* is a cell array of string, a logical array of the same size is returned, containing true for each class to which *obj* belongs to.

See also: [class], page 39, [typeinfo], page 39.

`cast (val, "type")` [Function File]

Convert *val* to data type *type*.

val must be one of the numeric classes:

```
"double"
"single"
"logical"
"char"
"int8"
"int16"
"int32"
"int64"
"uint8"
"uint16"
"uint32"
"uint64"
```

The value *val* may be modified to fit within the range of the new type.

Examples:

```
cast (-5, "uint8")
   ⇒ 0
cast (300, "int8")
   ⇒ 127
```

See also: [typecast], page 40, [int8], page 54, [uint8], page 55, [int16], page 55, [uint16], page 55, [int32], page 55, [uint32], page 55, [int64], page 55, [uint64], page 55, [double], page 47, [single], page 53, [logical], page 60, [char], page 71, [class], page 39, [typeinfo], page 39.

`y = typecast (x, "class")` [Built-in Function]

Return a new array *y* resulting from interpreting the data of *x* in memory as data of the numeric class *class*.

Both the class of *x* and *class* must be one of the built-in numeric classes:

```
"logical"
"char"
"int8"
"int16"
"int32"
"int64"
"uint8"
"uint16"
"uint32"
"uint64"
"double"
"single"
"double complex"
"single complex"
```

the last two are only used with *class*; they indicate that a complex-valued result is requested. Complex arrays are stored in memory as consecutive pairs of real numbers. The sizes of integer types are given by their bit counts. Both logical and char are typically one byte wide; however, this is not guaranteed by C++. If your system is IEEE conformant, single and double will be 4 bytes and 8 bytes wide, respectively. `"logical"` is not allowed for *class*.

If the input is a row vector, the return value is a row vector, otherwise it is a column vector.

If the bit length of *x* is not divisible by that of *class*, an error occurs.

An example of the use of typecast on a little-endian machine is

```
x = uint16 ([1, 65535]);
typecast (x, "uint8")
⇒ [   1,   0, 255, 255]
```

See also: [cast], page 40, [bitpack], page 41, [bitunpack], page 42, [swapbytes], page 41.

`swapbytes` (*x*) [Function File]

Swap the byte order on values, converting from little endian to big endian and vice versa.

For example:

```
swapbytes (uint16 (1:4))
⇒ [   256   512   768  1024]
```

See also: [typecast], page 40, [cast], page 40.

y = `bitpack` (*x*, *class*) [Built-in Function]

Return a new array *y* resulting from interpreting the logical array *x* as raw bit patterns for data of the numeric class *class*.

class must be one of the built-in numeric classes:

```
"double"
"single"
"double complex"
"single complex"
"char"
"int8"
"int16"
"int32"
"int64"
"uint8"
"uint16"
"uint32"
"uint64"
```

The number of elements of *x* should be divisible by the bit length of *class*. If it is not, excess bits are discarded. Bits come in increasing order of significance, i.e., `x(1)` is bit 0, `x(2)` is bit 1, etc.

The result is a row vector if *x* is a row vector, otherwise it is a column vector.

See also: [bitunpack], page 42, [typecast], page 40.

y = `bitunpack` (*x*) [Built-in Function]

Return a logical array *y* corresponding to the raw bit patterns of *x*.

x must belong to one of the built-in numeric classes:

```
"double"
"single"
"char"
"int8"
"int16"
"int32"
"int64"
"uint8"
"uint16"
"uint32"
"uint64"
```

The result is a row vector if *x* is a row vector; otherwise, it is a column vector.

See also: [bitpack], page 41, [typecast], page 40.

3.1.1 Numeric Objects

Octave's built-in numeric objects include real, complex, and integer scalars and matrices. All built-in floating point numeric data is currently stored as double precision numbers. On systems that use the IEEE floating point format, values in the range of approximately 2.2251×10^{-308} to 1.7977×10^{308} can be stored, and the relative precision is approximately 2.2204×10^{-16}. The exact values are given by the variables `realmin`, `realmax`, and `eps`, respectively.

Matrix objects can be of any size, and can be dynamically reshaped and resized. It is easy to extract individual rows, columns, or submatrices using a variety of powerful indexing features. See Section 8.1 [Index Expressions], page 135.

See Chapter 4 [Numeric Data Types], page 47, for more information.

3.1.2 Missing Data

It is possible to represent missing data explicitly in Octave using `NA` (short for "Not Available"). Missing data can only be represented when data is represented as floating point numbers. In this case missing data is represented as a special case of the representation of `NaN`.

`NA` [Built-in Function]
`NA` (*n*) [Built-in Function]
`NA` (*n*, *m*) [Built-in Function]
`NA` (*n*, *m*, *k*, ...) [Built-in Function]
`NA` (..., *class*) [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the special constant used to designate missing values.

Note that NA always compares not equal to NA (NA != NA). To find NA values, use the `isna` function.

When called with no arguments, return a scalar with the value '`NA`'.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [isna], page 43.

`isna` (*x*) [Mapping Function]

Return a logical array which is true where the elements of *x* are NA (missing) values and false where they are not.

For example:

```
isna ([13, Inf, NA, NaN])
     ⇒ [ 0, 0, 1, 0 ]
```

See also: [isnan], page 406, [isinf], page 406, [isfinite], page 407.

3.1.3 String Objects

A character string in Octave consists of a sequence of characters enclosed in either double-quote or single-quote marks. Internally, Octave currently stores strings as matrices of characters. All the indexing operations that work for matrix objects also work for strings.

See Chapter 5 [Strings], page 67, for more information.

3.1.4 Data Structure Objects

Octave's data structure type can help you to organize related objects of different types. The current implementation uses an associative array with indices limited to strings, but the syntax is more like C-style structures.

See Section 6.1 [Structures], page 99, for more information.

3.1.5 Cell Array Objects

A Cell Array in Octave is general array that can hold any number of different data types.

See Section 6.2 [Cell Arrays], page 112, for more information.

3.2 User-defined Data Types

Someday I hope to expand this to include a complete description of Octave's mechanism for managing user-defined data types. Until this feature is documented here, you will have to make do by reading the code in the `ov.h`, `ops.h`, and related files from Octave's `src` directory.

3.3 Object Sizes

The following functions allow you to determine the size of a variable or expression. These functions are defined for all objects. They return −1 when the operation doesn't make sense. For example, Octave's data structure type doesn't have rows or columns, so the `rows` and `columns` functions return −1 for structure arguments.

`ndims (a)` [Built-in Function]

Return the number of dimensions of *a*.

For any array, the result will always be greater than or equal to 2. Trailing singleton dimensions are not counted.

```
ndims (ones (4, 1, 2, 1))
    ⇒ 3
```

See also: [size], page 45.

`columns (a)` [Built-in Function]

Return the number of columns of *a*.

See also: [rows], page 44, [size], page 45, [length], page 45, [numel], page 44, [isscalar], page 63, [isvector], page 63, [ismatrix], page 63.

`rows (a)` [Built-in Function]

Return the number of rows of *a*.

See also: [columns], page 44, [size], page 45, [length], page 45, [numel], page 44, [isscalar], page 63, [isvector], page 63, [ismatrix], page 63.

`numel (a)` [Built-in Function]
`numel (a, idx1, idx2, ...)` [Built-in Function]

Return the number of elements in the object *a*.

Optionally, if indices *idx1*, *idx2*, . . . are supplied, return the number of elements that would result from the indexing

```
a(idx1, idx2, ...)
```

Note that the indices do not have to be scalar numbers. For example,

```
a = 1;
b = ones (2, 3);
numel (a, b)
```

will return 6, as this is the number of ways to index with *b*. Or the index could be the string `":"` which represents the colon operator. For example,

```
a = ones (5, 3);
numel (a, 2, ":")
```

will return 3 as the second row has three column entries.

This method is also called when an object appears as lvalue with cs-list indexing, i.e., `object{...}` or `object(...).field`.

See also: [size], page 45.

`length (a)` [Built-in Function]

Return the length of the object *a*.

The length is 0 for empty objects, 1 for scalars, and the number of elements for vectors. For matrix or N-dimensional objects, the length is the number of elements along the largest dimension (equivalent to `max (size (a))`).

See also: [numel], page 44, [size], page 45.

`size (a)` [Built-in Function]
`size (a, dim)` [Built-in Function]

Return the number of rows and columns of *a*.

With one input argument and one output argument, the result is returned in a row vector. If there are multiple output arguments, the number of rows is assigned to the first, and the number of columns to the second, etc. For example:

```
size ([1, 2; 3, 4; 5, 6])
   ⇒ [ 3, 2 ]

[nr, nc] = size ([1, 2; 3, 4; 5, 6])
    ⇒ nr = 3
    ⇒ nc = 2
```

If given a second argument, `size` will return the size of the corresponding dimension. For example,

```
size ([1, 2; 3, 4; 5, 6], 2)
    ⇒ 2
```

returns the number of columns in the given matrix.

See also: [numel], page 44, [ndims], page 44, [length], page 45, [rows], page 44, [columns], page 44.

`isempty (a)` [Built-in Function]

Return true if *a* is an empty matrix (any one of its dimensions is zero).

See also: [isnull], page 45, [isa], page 39.

`isnull (x)` [Built-in Function]

Return true if *x* is a special null matrix, string, or single quoted string.

Indexed assignment with such a value on the right-hand side should delete array elements. This function should be used when overloading indexed assignment for user-defined classes instead of `isempty`, to distinguish the cases:

`A(I) = []` This should delete elements if `I` is nonempty.

`X = []; A(I) = X`
This should give an error if `I` is nonempty.

See also: [isempty], page 45, [isindex], page 138.

`sizeof (`*val*`)` [Built-in Function]
Return the size of *val* in bytes.

See also: [whos], page 128.

`size_equal (`*a*, *b*, `...)` [Built-in Function]
Return true if the dimensions of all arguments agree.

Trailing singleton dimensions are ignored. When called with a single or no argument `size_equal` returns true.

See also: [size], page 45, [numel], page 44, [ndims], page 44.

`squeeze (`*x*`)` [Built-in Function]
Remove singleton dimensions from *x* and return the result.

Note that for compatibility with MATLAB, all objects have a minimum of two dimensions and row vectors are left unchanged.

See also: [reshape], page 412.

4 Numeric Data Types

A *numeric constant* may be a scalar, a vector, or a matrix, and it may contain complex values.

The simplest form of a numeric constant, a scalar, is a single number that can be an integer, a decimal fraction, a number in scientific (exponential) notation, or a complex number. Note that by default numeric constants are represented within Octave in double-precision floating point format (complex constants are stored as pairs of double-precision floating point values). It is, however, possible to represent real integers as described in Section 4.4 [Integer Data Types], page 54. Here are some examples of real-valued numeric constants, which all have the same value:

```
105
1.05e+2
1050e-1
```

To specify complex constants, you can write an expression of the form

```
3 + 4i
3.0 + 4.0i
0.3e1 + 40e-1i
```

all of which are equivalent. The letter '`i`' in the previous example stands for the pure imaginary constant, defined as $\sqrt{-1}$.

For Octave to recognize a value as the imaginary part of a complex constant, a space must not appear between the number and the '`i`'. If it does, Octave will print an error message, like this:

```
octave:13> 3 + 4 i

parse error:

  syntax error

>>> 3 + 4 i
          ^
```

You may also use '`j`', '`I`', or '`J`' in place of the '`i`' above. All four forms are equivalent.

`double` (*x*) [Built-in Function]

Convert *x* to double precision type.

See also: [single], page 53.

`complex` (*x*) [Built-in Function]
`complex` (*re*, *im*) [Built-in Function]

Return a complex value from real arguments.

With 1 real argument *x*, return the complex result `x + 0i`.

With 2 real arguments, return the complex result `re + im`. `complex` can often be more convenient than expressions such as `a + i*b`. For example:

```
complex ([1, 2], [3, 4])
  ⇒ [ 1 + 3i   2 + 4i ]
```

See also: [real], page 438, [imag], page 437, [iscomplex], page 63, [abs], page 437, [arg], page 437.

4.1 Matrices

It is easy to define a matrix of values in Octave. The size of the matrix is determined automatically, so it is not necessary to explicitly state the dimensions. The expression

```
a = [1, 2; 3, 4]
```

results in the matrix

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Elements of a matrix may be arbitrary expressions, provided that the dimensions all make sense when combining the various pieces. For example, given the above matrix, the expression

```
[ a, a ]
```

produces the matrix

```
ans =

  1  2  1  2
  3  4  3  4
```

but the expression

```
[ a, 1 ]
```

produces the error

```
error: number of rows must match (1 != 2) near line 13, column 6
```

(assuming that this expression was entered as the first thing on line 13, of course).

Inside the square brackets that delimit a matrix expression, Octave looks at the surrounding context to determine whether spaces and newline characters should be converted into element and row separators, or simply ignored, so an expression like

```
a = [ 1 2
      3 4 ]
```

will work. However, some possible sources of confusion remain. For example, in the expression

```
[ 1 - 1 ]
```

the '`-`' is treated as a binary operator and the result is the scalar 0, but in the expression

```
[ 1 -1 ]
```

the '`-`' is treated as a unary operator and the result is the vector `[ 1, -1 ]`. Similarly, the expression

```
[ sin (pi) ]
```

will be parsed as

```
[ sin, (pi) ]
```

and will result in an error since the `sin` function will be called with no arguments. To get around this, you must omit the space between `sin` and the opening parenthesis, or enclose the expression in a set of parentheses:

```
[ (sin (pi)) ]
```

Whitespace surrounding the single quote character (‘'’, used as a transpose operator and for delimiting character strings) can also cause confusion. Given `a = 1`, the expression

```
[ 1 a' ]
```

results in the single quote character being treated as a transpose operator and the result is the vector `[ 1, 1 ]`, but the expression

```
[ 1 a ' ]
```

produces the error message

```
parse error:

  syntax error

>>> [ 1 a ' ]
              ^
```

because not doing so would cause trouble when parsing the valid expression

```
[ a 'foo' ]
```

For clarity, it is probably best to always use commas and semicolons to separate matrix elements and rows.

The maximum number of elements in a matrix is fixed when Octave is compiled. The allowable number can be queried with the function `sizemax`. Note that other factors, such as the amount of memory available on your machine, may limit the maximum size of matrices to something smaller.

`sizemax ()` [Built-in Function]

Return the largest value allowed for the size of an array.

If Octave is compiled with 64-bit indexing, the result is of class int64, otherwise it is of class int32. The maximum array size is slightly smaller than the maximum value allowable for the relevant class as reported by `intmax`.

See also: [intmax], page 55.

When you type a matrix or the name of a variable whose value is a matrix, Octave responds by printing the matrix in with neatly aligned rows and columns. If the rows of the matrix are too large to fit on the screen, Octave splits the matrix and displays a header before each section to indicate which columns are being displayed. You can use the following variables to control the format of the output.

val = `output_max_field_width ()` [Built-in Function]
old_val = `output_max_field_width (`*new_val*`)` [Built-in Function]
`output_max_field_width (`*new_val*, `"`*local*`"`)` [Built-in Function]

Query or set the internal variable that specifies the maximum width of a numeric output field.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [format], page 232, [fixed_point_format], page 51, [output_precision], page 50.

`val = output_precision ()` [Built-in Function]
`old_val = output_precision (new_val)` [Built-in Function]
`output_precision (new_val, "local")` [Built-in Function]

Query or set the internal variable that specifies the minimum number of significant figures to display for numeric output.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [format], page 232, [fixed_point_format], page 51, [output_max_field_width], page 49.

It is possible to achieve a wide range of output styles by using different values of `output_precision` and `output_max_field_width`. Reasonable combinations can be set using the `format` function. See Section 14.1 [Basic Input and Output], page 231.

`val = split_long_rows ()` [Built-in Function]
`old_val = split_long_rows (new_val)` [Built-in Function]
`split_long_rows (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether rows of a matrix may be split when displayed to a terminal window.

If the rows are split, Octave will display the matrix in a series of smaller pieces, each of which can fit within the limits of your terminal width and each set of rows is labeled so that you can easily see which columns are currently being displayed. For example:

```
octave:13> rand (2,10)
ans =

 Columns 1 through 6:

  0.75883  0.93290  0.40064  0.43818  0.94958  0.16467
  0.75697  0.51942  0.40031  0.61784  0.92309  0.40201

 Columns 7 through 10:

  0.90174  0.11854  0.72313  0.73326
  0.44672  0.94303  0.56564  0.82150
```

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [format], page 232.

Octave automatically switches to scientific notation when values become very large or very small. This guarantees that you will see several significant figures for every value in a matrix. If you would prefer to see all values in a matrix printed in a fixed point format, you can set the built-in variable `fixed_point_format` to a nonzero value. But doing so is not recommended, because it can produce output that can easily be misinterpreted.

val = `fixed_point_format` () [Built-in Function]
old_val = `fixed_point_format` (*new_val*) [Built-in Function]
`fixed_point_format` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether Octave will use a scaled format to print matrix values.

The scaled format prints a scaling factor on the first line of output chosen such that the largest matrix element can be written with a single leading digit. For example:

```
logspace (1, 7, 5)'
ans =

  1.0e+07  *

  0.00000
  0.00003
  0.00100
  0.03162
  1.00000
```

Notice that the first value appears to be 0 when it is actually 1. Because of the possibility for confusion you should be careful about enabling `fixed_point_format`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [format], page 232, [output_max_field_width], page 49, [output_precision], page 50.

4.1.1 Empty Matrices

A matrix may have one or both dimensions zero, and operations on empty matrices are handled as described by Carl de Boor in *An Empty Exercise*, SIGNUM, Volume 25, pages 2-6, 1990 and C. N. Nett and W. M. Haddad, in *A System-Theoretic Appropriate Realization of the Empty Matrix Concept*, IEEE Transactions on Automatic Control, Volume 38, Number 5, May 1993. Briefly, given a scalar s, an $m \times n$ matrix $M_{m\times n}$, and an $m \times n$ empty matrix $[\,]_{m\times n}$ (with either one or both dimensions equal to zero), the following are true:

$$\begin{aligned} s \cdot [\,]_{m\times n} = [\,]_{m\times n} \cdot s &= [\,]_{m\times n} \\ [\,]_{m\times n} + [\,]_{m\times n} &= [\,]_{m\times n} \\ [\,]_{0\times m} \cdot M_{m\times n} &= [\,]_{0\times n} \\ M_{m\times n} \cdot [\,]_{n\times 0} &= [\,]_{m\times 0} \\ [\,]_{m\times 0} \cdot [\,]_{0\times n} &= 0_{m\times n} \end{aligned}$$

By default, dimensions of the empty matrix are printed along with the empty matrix symbol, '`[]`'. The built-in variable `print_empty_dimensions` controls this behavior.

val = `print_empty_dimensions ()` [Built-in Function]
old_val = `print_empty_dimensions (`*new_val*`)` [Built-in Function]
`print_empty_dimensions (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that controls whether the dimensions of empty matrices are printed along with the empty matrix symbol, '`[]`'.

For example, the expression

```
zeros (3, 0)
```

will print

```
ans = [](3x0)
```

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [format], page 232.

Empty matrices may also be used in assignment statements as a convenient way to delete rows or columns of matrices. See Section 8.6 [Assignment Expressions], page 149.

When Octave parses a matrix expression, it examines the elements of the list to determine whether they are all constants. If they are, it replaces the list with a single matrix constant.

4.2 Ranges

A *range* is a convenient way to write a row vector with evenly spaced elements. A range expression is defined by the value of the first element in the range, an optional value for the increment between elements, and a maximum value which the elements of the range will not exceed. The base, increment, and limit are separated by colons (the '`:`' character) and may contain any arithmetic expressions and function calls. If the increment is omitted, it is assumed to be 1. For example, the range

```
1 : 5
```

defines the set of values '`[ 1, 2, 3, 4, 5 ]`', and the range

```
1 : 3 : 5
```

defines the set of values '`[ 1, 4 ]`'.

Although a range constant specifies a row vector, Octave does *not* normally convert range constants to vectors unless it is necessary to do so. This allows you to write a constant like '`1 : 10000`' without using 80,000 bytes of storage on a typical 32-bit workstation.

A common example of when it does become necessary to convert ranges into vectors occurs when they appear within a vector (i.e., inside square brackets). For instance, whereas

```
x = 0 : 0.1 : 1;
```

defines *x* to be a variable of type `range` and occupies 24 bytes of memory, the expression

```
y = [ 0 : 0.1 : 1];
```

defines *y* to be of type `matrix` and occupies 88 bytes of memory.

This space saving optimization may be disabled using the function *disable_range*.

val = disable_range () [Built-in Function]
old_val = disable_range (*new_val*) [Built-in Function]
disable_range (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether ranges are stored in a special space-efficient format.

The default value is true. If this option is disabled Octave will store ranges as full matrices.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [disable_diagonal_matrix], page 511, [disable_permutation_matrix], page 511.

Note that the upper (or lower, if the increment is negative) bound on the range is not always included in the set of values, and that ranges defined by floating point values can produce surprising results because Octave uses floating point arithmetic to compute the values in the range. If it is important to include the endpoints of a range and the number of elements is known, you should use the linspace function instead (see Section 16.3 [Special Utility Matrices], page 418).

When adding a scalar to a range, subtracting a scalar from it (or subtracting a range from a scalar) and multiplying by scalar, Octave will attempt to avoid unpacking the range and keep the result as a range, too, if it can determine that it is safe to do so. For instance, doing

```
a = 2*(1:1e7) - 1;
```

will produce the same result as '1:2:2e7-1', but without ever forming a vector with ten million elements.

Using zero as an increment in the colon notation, as '1:0:1' is not allowed, because a division by zero would occur in determining the number of range elements. However, ranges with zero increment (i.e., all elements equal) are useful, especially in indexing, and Octave allows them to be constructed using the built-in function *ones*. Note that because a range must be a row vector, 'ones (1, 10)' produces a range, while 'ones (10, 1)' does not.

When Octave parses a range expression, it examines the elements of the expression to determine whether they are all constants. If they are, it replaces the range expression with a single range constant.

4.3 Single Precision Data Types

Octave includes support for single precision data types, and most of the functions in Octave accept single precision values and return single precision answers. A single precision variable is created with the single function.

single (*x*) [Built-in Function]

Convert x to single precision type.

See also: [double], page 47.

for example:

```
sngl = single (rand (2, 2))
     ⇒ sngl =
        0.37569   0.92982
        0.11962   0.50876
class (sngl)
    ⇒ single
```

Many functions can also return single precision values directly. For example

```
ones (2, 2, "single")
zeros (2, 2, "single")
eye (2, 2,  "single")
rand (2, 2, "single")
NaN (2, 2, "single")
NA (2, 2, "single")
Inf (2, 2, "single")
```

will all return single precision matrices.

4.4 Integer Data Types

Octave supports integer matrices as an alternative to using double precision. It is possible to use both signed and unsigned integers represented by 8, 16, 32, or 64 bits. It should be noted that most computations require floating point data, meaning that integers will often change type when involved in numeric computations. For this reason integers are most often used to store data, and not for calculations.

In general most integer matrices are created by casting existing matrices to integers. The following example shows how to cast a matrix into 32 bit integers.

```
float = rand (2, 2)
     ⇒ float = 0.37569   0.92982
               0.11962   0.50876
integer = int32 (float)
     ⇒ integer = 0  1
                 0  1
```

As can be seen, floating point values are rounded to the nearest integer when converted.

`isinteger` (*x*) [Built-in Function]

Return true if *x* is an integer object (int8, uint8, int16, etc.).

Note that `isinteger (14)` is false because numeric constants in Octave are double precision floating point values.

See also: [isfloat], page 62, [ischar], page 68, [islogical], page 62, [isnumeric], page 62, [isa], page 39.

`int8` (*x*) [Built-in Function]

Convert *x* to 8-bit integer type.

See also: [uint8], page 55, [int16], page 55, [uint16], page 55, [int32], page 55, [uint32], page 55, [int64], page 55, [uint64], page 55.

`uint8` (*x*) [Built-in Function]

Convert *x* to unsigned 8-bit integer type.

See also: [int8], page 54, [int16], page 55, [uint16], page 55, [int32], page 55, [uint32], page 55, [int64], page 55, [uint64], page 55.

`int16` (*x*) [Built-in Function]

Convert *x* to 16-bit integer type.

See also: [int8], page 54, [uint8], page 55, [uint16], page 55, [int32], page 55, [uint32], page 55, [int64], page 55, [uint64], page 55.

`uint16` (*x*) [Built-in Function]

Convert *x* to unsigned 16-bit integer type.

See also: [int8], page 54, [uint8], page 55, [int16], page 55, [int32], page 55, [uint32], page 55, [int64], page 55, [uint64], page 55.

`int32` (*x*) [Built-in Function]

Convert *x* to 32-bit integer type.

See also: [int8], page 54, [uint8], page 55, [int16], page 55, [uint16], page 55, [uint32], page 55, [int64], page 55, [uint64], page 55.

`uint32` (*x*) [Built-in Function]

Convert *x* to unsigned 32-bit integer type.

See also: [int8], page 54, [uint8], page 55, [int16], page 55, [uint16], page 55, [int32], page 55, [int64], page 55, [uint64], page 55.

`int64` (*x*) [Built-in Function]

Convert *x* to 64-bit integer type.

See also: [int8], page 54, [uint8], page 55, [int16], page 55, [uint16], page 55, [int32], page 55, [uint32], page 55, [uint64], page 55.

`uint64` (*x*) [Built-in Function]

Convert *x* to unsigned 64-bit integer type.

See also: [int8], page 54, [uint8], page 55, [int16], page 55, [uint16], page 55, [int32], page 55, [uint32], page 55, [int64], page 55.

`intmax` (*type*) [Built-in Function]

Return the largest integer that can be represented in an integer type.

The variable *type* can be

`int8` signed 8-bit integer.

`int16` signed 16-bit integer.

`int32` signed 32-bit integer.

`int64` signed 64-bit integer.

`uint8` unsigned 8-bit integer.

`uint16` unsigned 16-bit integer.

`uint32` unsigned 32-bit integer.

`uint64` unsigned 64-bit integer.

The default for *type* is `int32`.

See also: [intmin], page 56, [flintmax], page 56, [bitmax], page 58.

`intmin (type)` [Built-in Function]

Return the smallest integer that can be represented in an integer type.

The variable *type* can be

`int8` signed 8-bit integer.

`int16` signed 16-bit integer.

`int32` signed 32-bit integer.

`int64` signed 64-bit integer.

`uint8` unsigned 8-bit integer.

`uint16` unsigned 16-bit integer.

`uint32` unsigned 32-bit integer.

`uint64` unsigned 64-bit integer.

The default for *type* is `int32`.

See also: [intmax], page 55, [flintmax], page 56, [bitmax], page 58.

`flintmax ()` [Built-in Function]
`flintmax ("double")` [Built-in Function]
`flintmax ("single")` [Built-in Function]

Return the largest integer that can be represented consecutively in a floating point value.

The default class is `"double"`, but `"single"` is a valid option. On IEEE 754 compatible systems, `flintmax` is 2^{53} for `"double"` and 2^{24} for `"single"`.

See also: [bitmax], page 58, [intmax], page 55, [realmax], page 463, [realmin], page 463.

4.4.1 Integer Arithmetic

While many numerical computations can't be carried out in integers, Octave does support basic operations like addition and multiplication on integers. The operators `+`, `-`, `.*`, and `./` work on integers of the same type. So, it is possible to add two 32 bit integers, but not to add a 32 bit integer and a 16 bit integer.

When doing integer arithmetic one should consider the possibility of underflow and overflow. This happens when the result of the computation can't be represented using the chosen integer type. As an example it is not possible to represent the result of $10 - 20$ when using unsigned integers. Octave makes sure that the result of integer computations is the integer that is closest to the true result. So, the result of $10 - 20$ when using unsigned integers is zero.

When doing integer division Octave will round the result to the nearest integer. This is different from most programming languages, where the result is often floored to the nearest integer. So, the result of `int32 (5) ./ int32 (8)` is `1`.

`idivide (x, y, op)` [Function File]

Integer division with different rounding rules.

The standard behavior of integer division such as `a ./ b` is to round the result to the nearest integer. This is not always the desired behavior and `idivide` permits integer element-by-element division to be performed with different treatment for the fractional part of the division as determined by the *op* flag. *op* is a string with one of the values:

`"fix"` Calculate `a ./ b` with the fractional part rounded towards zero.

`"round"` Calculate `a ./ b` with the fractional part rounded towards the nearest integer.

`"floor"` Calculate `a ./ b` with the fractional part rounded towards negative infinity.

`"ceil"` Calculate `a ./ b` with the fractional part rounded towards positive infinity.

If *op* is not given it defaults to `"fix"`. An example demonstrating these rounding rules is

```
idivide (int8 ([-3, 3]), int8 (4), "fix")
  ⇒ int8 ([0, 0])
idivide (int8 ([-3, 3]), int8 (4), "round")
  ⇒ int8 ([-1, 1])
idivide (int8 ([-3, 3]), int8 (4), "floor")
  ⇒ int8 ([-1, 0])
idivide (int8 ([-3, 3]), int8 (4), "ceil")
  ⇒ int8 ([0, 1])
```

See also: [ldivide], page 143, [rdivide], page 144.

4.5 Bit Manipulations

Octave provides a number of functions for the manipulation of numeric values on a bit by bit basis. The basic functions to set and obtain the values of individual bits are `bitset` and `bitget`.

`C = bitset (A, n)` [Function File]
`C = bitset (A, n, val)` [Function File]

Set or reset bit(s) n of the unsigned integers in A.

val $= 0$ resets and *val* $= 1$ sets the bits. The least significant bit is $n = 1$. All variables must be the same size or scalars.

```
dec2bin (bitset (10, 1))
  ⇒ 1011
```

See also: [bitand], page 58, [bitor], page 58, [bitxor], page 59, [bitget], page 57, [bitcmp], page 59, [bitshift], page 59, [bitmax], page 58.

`c = bitget (A, n)` [Function File]

Return the status of bit(s) n of the unsigned integers in A.

The least significant bit is $n = 1$.

```
bitget (100, 8:-1:1)
⇒ 0  1  1  0  0  1  0  0
```

See also: [bitand], page 58, [bitor], page 58, [bitxor], page 59, [bitset], page 57, [bitcmp], page 59, [bitshift], page 59, [bitmax], page 58.

The arguments to all of Octave's bitwise operations can be scalar or arrays, except for `bitcmp`, whose *k* argument must a scalar. In the case where more than one argument is an array, then all arguments must have the same shape, and the bitwise operator is applied to each of the elements of the argument individually. If at least one argument is a scalar and one an array, then the scalar argument is duplicated. Therefore

```
bitget (100, 8:-1:1)
```

is the same as

```
bitget (100 * ones (1, 8), 8:-1:1)
```

It should be noted that all values passed to the bit manipulation functions of Octave are treated as integers. Therefore, even though the example for `bitset` above passes the floating point value `10`, it is treated as the bits `[1, 0, 1, 0]` rather than the bits of the native floating point format representation of `10`.

As the maximum value that can be represented by a number is important for bit manipulation, particularly when forming masks, Octave supplies the function `bitmax`.

`bitmax ()` [Built-in Function]
`bitmax ("`*double*`")` [Built-in Function]
`bitmax ("`*single*`")` [Built-in Function]

Return the largest integer that can be represented within a floating point value.

The default class is `"double"`, but `"single"` is a valid option. On IEEE 754 compatible systems, `bitmax` is $2^{53} - 1$ for `"double"` and $2^{24} - 1$ for `"single"`.

See also: [flintmax], page 56, [intmax], page 55, [realmax], page 463, [realmin], page 463.

This is the double precision version of the function `intmax`, previously discussed.

Octave also includes the basic bitwise 'and', 'or', and 'exclusive or' operators.

`bitand (`*x*, *y*`)` [Built-in Function]

Return the bitwise AND of non-negative integers.

x, *y* must be in the range [0,bitmax]

See also: [bitor], page 58, [bitxor], page 59, [bitset], page 57, [bitget], page 57, [bitcmp], page 59, [bitshift], page 59, [bitmax], page 58.

`bitor (`*x*, *y*`)` [Built-in Function]

Return the bitwise OR of non-negative integers.

x, *y* must be in the range [0,bitmax]

See also: [bitor], page 58, [bitxor], page 59, [bitset], page 57, [bitget], page 57, [bitcmp], page 59, [bitshift], page 59, [bitmax], page 58.

`bitxor (x, y)` [Built-in Function]

Return the bitwise XOR of non-negative integers.

x, *y* must be in the range [0,bitmax]

See also: [bitand], page 58, [bitor], page 58, [bitset], page 57, [bitget], page 57, [bitcmp], page 59, [bitshift], page 59, [bitmax], page 58.

The bitwise 'not' operator is a unary operator that performs a logical negation of each of the bits of the value. For this to make sense, the mask against which the value is negated must be defined. Octave's bitwise 'not' operator is `bitcmp`.

`bitcmp (A, k)` [Function File]

Return the *k*-bit complement of integers in *A*.

If *k* is omitted `k = log2 (bitmax) + 1` is assumed.

```
bitcmp (7,4)
  ⇒ 8
dec2bin (11)
  ⇒ 1011
dec2bin (bitcmp (11, 6))
  ⇒ 110100
```

See also: [bitand], page 58, [bitor], page 58, [bitxor], page 59, [bitset], page 57, [bitget], page 57, [bitcmp], page 59, [bitshift], page 59, [bitmax], page 58.

Octave also includes the ability to left-shift and right-shift values bitwise.

`bitshift (a, k)` [Built-in Function]
`bitshift (a, k, n)` [Built-in Function]

Return a *k* bit shift of *n*-digit unsigned integers in *a*.

A positive *k* leads to a left shift; A negative value to a right shift.

If *n* is omitted it defaults to log2(bitmax)+1. *n* must be in the range [1,log2(bitmax)+1] usually [1,33].

```
bitshift (eye (3), 1)
⇒
2 0 0
0 2 0
0 0 2

bitshift (10, [-2, -1, 0, 1, 2])
⇒ 2   5  10  20  40
```

See also: [bitand], page 58, [bitor], page 58, [bitxor], page 59, [bitset], page 57, [bitget], page 57, [bitcmp], page 59, [bitmax], page 58.

Bits that are shifted out of either end of the value are lost. Octave also uses arithmetic shifts, where the sign bit of the value is kept during a right shift. For example:

```
bitshift (-10, -1)
⇒ -5
bitshift (int8 (-1), -1)
⇒ -1
```

Note that `bitshift (int8 (-1), -1)` is `-1` since the bit representation of `-1` in the `int8` data type is `[1, 1, 1, 1, 1, 1, 1, 1]`.

4.6 Logical Values

Octave has built-in support for logical values, i.e., variables that are either `true` or `false`. When comparing two variables, the result will be a logical value whose value depends on whether or not the comparison is true.

The basic logical operations are `&`, `|`, and `!`, which correspond to "Logical And", "Logical Or", and "Logical Negation". These operations all follow the usual rules of logic.

It is also possible to use logical values as part of standard numerical calculations. In this case `true` is converted to `1`, and `false` to 0, both represented using double precision floating point numbers. So, the result of `true*22 - false/6` is `22`.

Logical values can also be used to index matrices and cell arrays. When indexing with a logical array the result will be a vector containing the values corresponding to `true` parts of the logical array. The following example illustrates this.

```
data = [ 1, 2; 3, 4 ];
idx = (data <= 2);
data(idx)
     ⇒ ans = [ 1; 2 ]
```

Instead of creating the `idx` array it is possible to replace `data(idx)` with `data( data <= 2 )` in the above code.

Logical values can also be constructed by casting numeric objects to logical values, or by using the `true` or `false` functions.

`logical` (*x*) [Built-in Function]

Convert the numeric object *x* to logical type.

Any nonzero values will be converted to true (1) while zero values will be converted to false (0). The non-numeric value NaN cannot be converted and will produce an error.

Compatibility Note: Octave accepts complex values as input, whereas MATLAB issues an error.

See also: [double], page 47, [single], page 53, [char], page 71.

`true` (*x*) [Built-in Function]
`true` (*n*, *m*) [Built-in Function]
`true` (*n*, *m*, *k*, . . .) [Built-in Function]

Return a matrix or N-dimensional array whose elements are all logical 1.

If invoked with a single scalar integer argument, return a square matrix of the specified size.

If invoked with two or more scalar integer arguments, or a vector of integer values, return an array with given dimensions.

See also: [false], page 61.

`false (x)` [Built-in Function]
`false (n, m)` [Built-in Function]
`false (n, m, k, ...)` [Built-in Function]

Return a matrix or N-dimensional array whose elements are all logical 0.

If invoked with a single scalar integer argument, return a square matrix of the specified size.

If invoked with two or more scalar integer arguments, or a vector of integer values, return an array with given dimensions.

See also: [true], page 60.

4.7 Promotion and Demotion of Data Types

Many operators and functions can work with mixed data types. For example,

```
uint8 (1) + 1
    ⇒ 2
```

where the above operator works with an 8-bit integer and a double precision value and returns an 8-bit integer value. Note that the type is demoted to an 8-bit integer, rather than promoted to a double precision value as might be expected. The reason is that if Octave promoted values in expressions like the above with all numerical constants would need to be explicitly cast to the appropriate data type like

```
uint8 (1) + uint8 (1)
    ⇒ 2
```

which becomes difficult for the user to apply uniformly and might allow hard to find bugs to be introduced. The same applies to single precision values where a mixed operation such as

```
single (1) + 1
    ⇒ 2
```

returns a single precision value. The mixed operations that are valid and their returned data types are

Mixed Operation	**Result**
double OP single	single
double OP integer	integer
double OP char	double
double OP logical	double
single OP integer	integer
single OP char	single
single OP logical	single

The same logic applies to functions with mixed arguments such as

```
min (single (1), 0)
  ⇒ 0
```

where the returned value is single precision.

In the case of mixed type indexed assignments, the type is not changed. For example,

```
x = ones (2, 2);
x(1, 1) = single (2)
   ⇒ x = 2   1
          1   1
```

where x remains of the double precision type.

4.8 Predicates for Numeric Objects

Since the type of a variable may change during the execution of a program, it can be necessary to do type checking at run-time. Doing this also allows you to change the behavior of a function depending on the type of the input. As an example, this naive implementation of abs returns the absolute value of the input if it is a real number, and the length of the input if it is a complex number.

```
function a = abs (x)
  if (isreal (x))
    a = sign (x) .* x;
  elseif (iscomplex (x))
    a = sqrt (real(x).^2 + imag(x).^2);
  endif
endfunction
```

The following functions are available for determining the type of a variable.

isnumeric (*x*) [Built-in Function]

Return true if *x* is a numeric object, i.e., an integer, real, or complex array.

Logical and character arrays are not considered to be numeric.

See also: [isinteger], page 54, [isfloat], page 62, [isreal], page 62, [iscomplex], page 63, [islogical], page 62, [ischar], page 68, [iscell], page 113, [isstruct], page 107, [isa], page 39.

islogical (*x*) [Built-in Function]
isbool (*x*) [Built-in Function]

Return true if *x* is a logical object.

See also: [isfloat], page 62, [isinteger], page 54, [ischar], page 68, [isnumeric], page 62, [isa], page 39.

isfloat (*x*) [Built-in Function]

Return true if *x* is a floating-point numeric object.

Objects of class double or single are floating-point objects.

See also: [isinteger], page 54, [ischar], page 68, [islogical], page 62, [isnumeric], page 62, [isa], page 39.

isreal (*x*) [Built-in Function]

Return true if *x* is a non-complex matrix or scalar.

For compatibility with MATLAB, this includes logical and character matrices.

See also: [iscomplex], page 63, [isnumeric], page 62, [isa], page 39.

`iscomplex (x)` [Built-in Function]

Return true if *x* is a complex-valued numeric object.

See also: [isreal], page 62, [isnumeric], page 62, [islogical], page 62, [ischar], page 68, [isfloat], page 62, [isa], page 39.

`ismatrix (a)` [Built-in Function]

Return true if *a* is a 2-D array.

See also: [isscalar], page 63, [isvector], page 63, [iscell], page 113, [isstruct], page 107, [issparse], page 526, [isa], page 39.

`isvector (x)` [Function File]

Return true if *x* is a vector.

A vector is a 2-D array where one of the dimensions is equal to 1. As a consequence a 1x1 array, or scalar, is also a vector.

See also: [isscalar], page 63, [ismatrix], page 63, [size], page 45, [rows], page 44, [columns], page 44, [length], page 45.

`isrow (x)` [Function File]

Return true if *x* is a row vector 1xN with non-negative N.

See also: [iscolumn], page 63, [isscalar], page 63, [isvector], page 63, [ismatrix], page 63.

`iscolumn (x)` [Function File]

Return true if *x* is a column vector Nx1 with non-negative N.

See also: [isrow], page 63, [isscalar], page 63, [isvector], page 63, [ismatrix], page 63.

`isscalar (x)` [Built-in Function]

Return true if *x* is a scalar.

See also: [isvector], page 63, [ismatrix], page 63.

`issquare (x)` [Function File]

Return true if *x* is a square matrix.

See also: [isscalar], page 63, [isvector], page 63, [ismatrix], page 63, [size], page 45.

`issymmetric (A)` [Function File]
`issymmetric (A, tol)` [Function File]

Return true if *A* is a symmetric matrix within the tolerance specified by *tol*.

The default tolerance is zero (uses faster code).

Matrix *A* is considered symmetric if `norm (A - A.', Inf) / norm (A, Inf) < tol`.

See also: [ishermitian], page 63, [isdefinite], page 64.

`ishermitian (A)` [Function File]
`ishermitian (A, tol)` [Function File]

Return true if *A* is Hermitian within the tolerance specified by *tol*.

The default tolerance is zero (uses faster code).

Matrix *A* is considered symmetric if `norm (A - A', Inf) / norm (A, Inf) < tol`.

See also: [issymmetric], page 63, [isdefinite], page 64.

`isdefinite (A)` [Function File]
`isdefinite (A, tol)` [Function File]

Return 1 if A is symmetric positive definite within the tolerance specified by *tol* or 0 if A is symmetric positive semidefinite. Otherwise, return -1.

If *tol* is omitted, use a tolerance of `100 * eps * norm (A, "fro")`

See also: [issymmetric], page 63, [ishermitian], page 63.

`isbanded (A, lower, upper)` [Function File]

Return true if A is a matrix with entries confined between *lower* diagonals below the main diagonal and *upper* diagonals above the main diagonal.

lower and *upper* must be non-negative integers.

See also: [isdiag], page 64, [istril], page 64, [istriu], page 64, [bandwidth], page 466.

`isdiag (A)` [Function File]

Return true if A is a diagonal matrix.

See also: [isbanded], page 64, [istril], page 64, [istriu], page 64, [diag], page 418, [bandwidth], page 466.

`istril (A)` [Function File]

Return true if A is a lower triangular matrix.

A lower triangular matrix has nonzero entries only on the main diagonal and below.

See also: [istriu], page 64, [isbanded], page 64, [isdiag], page 64, [tril], page 416, [bandwidth], page 466.

`istriu (A)` [Function File]

Return true if A is an upper triangular matrix.

An upper triangular matrix has nonzero entries only on the main diagonal and above.

See also: [isdiag], page 64, [isbanded], page 64, [istril], page 64, [triu], page 416, [bandwidth], page 466.

`isprime (x)` [Function File]

Return a logical array which is true where the elements of x are prime numbers and false where they are not.

A prime number is conventionally defined as a positive integer greater than 1 (e.g., 2, 3, . . .) which is divisible only by itself and 1. Octave extends this definition to include both negative integers and complex values. A negative integer is prime if its positive counterpart is prime. This is equivalent to `isprime (abs (x))`.

If `class (x)` is complex, then primality is tested in the domain of Gaussian integers (`http://en.wikipedia.org/wiki/Gaussian_integer`). Some non-complex integers are prime in the ordinary sense, but not in the domain of Gaussian integers. For example, $5 = (1 + 2i) * (1 - 2i)$ shows that 5 is not prime because it has a factor other than itself and 1. Exercise caution when testing complex and real values together in the same matrix.

Examples:

```
isprime (1:6)
    ⇒ [0, 1, 1, 0, 1, 0]
isprime ([i, 2, 3, 5])
    ⇒ [0, 0, 1, 0]
```

Programming Note: `isprime` is appropriate if the maximum value in *x* is not too large (< 1e15). For larger values special purpose factorization code should be used.

Compatibility Note: *matlab* does not extend the definition of prime numbers and will produce an error if given negative or complex inputs.

See also: [primes], page 450, [factor], page 449, [gcd], page 449, [lcm], page 449.

If instead of knowing properties of variables, you wish to know which variables are defined and to gather other information about the workspace itself, see Section 7.3 [Status of Variables], page 127.

5 Strings

A *string constant* consists of a sequence of characters enclosed in either double-quote or single-quote marks. For example, both of the following expressions

```
"parrot"
'parrot'
```

represent the string whose contents are '`parrot`'. Strings in Octave can be of any length.

Since the single-quote mark is also used for the transpose operator (see Section 8.3 [Arithmetic Ops], page 141) but double-quote marks have no other purpose in Octave, it is best to use double-quote marks to denote strings.

Strings can be concatenated using the notation for defining matrices. For example, the expression

```
[ "foo" , "bar" , "baz" ]
```

produces the string whose contents are '`foobarbaz`'. See Chapter 4 [Numeric Data Types], page 47, for more information about creating matrices.

5.1 Escape Sequences in String Constants

In double-quoted strings, the backslash character is used to introduce *escape sequences* that represent other characters. For example, '`\n`' embeds a newline character in a double-quoted string and '`\"`' embeds a double quote character. In single-quoted strings, backslash is not a special character. Here is an example showing the difference:

```
toascii ("\n")
    ⇒ 10
toascii ('\n')
    ⇒ [ 92 110 ]
```

Here is a table of all the escape sequences used in Octave (within double quoted strings). They are the same as those used in the C programming language.

`\\` Represents a literal backslash, '`\`'.

`\"` Represents a literal double-quote character, '`"`'.

`\'` Represents a literal single-quote character, '`'`'.

`\0` Represents the null character, control-@, ASCII code 0.

`\a` Represents the "alert" character, control-g, ASCII code 7.

`\b` Represents a backspace, control-h, ASCII code 8.

`\f` Represents a formfeed, control-l, ASCII code 12.

`\n` Represents a newline, control-j, ASCII code 10.

`\r` Represents a carriage return, control-m, ASCII code 13.

`\t` Represents a horizontal tab, control-i, ASCII code 9.

`\v` Represents a vertical tab, control-k, ASCII code 11.

`\`*nnn* Represents the octal value *nnn*, where *nnn* are one to three digits between 0 and 7. For example, the code for the ASCII ESC (escape) character is '`\033`'.

`\xhh...` Represents the hexadecimal value *hh*, where *hh* are hexadecimal digits ('0' through '9' and either 'A' through 'F' or 'a' through 'f'). Like the same construct in ANSI C, the escape sequence continues until the first non-hexadecimal digit is seen. However, using more than two hexadecimal digits produces undefined results.

In a single-quoted string there is only one escape sequence: you may insert a single quote character using two single quote characters in succession. For example,

```
'I can''t escape'
    ⇒ I can't escape
```

In scripts the two different string types can be distinguished if necessary by using `is_dq_string` and `is_sq_string`.

`is_dq_string (x)` [Built-in Function]

Return true if *x* is a double-quoted character string.

See also: [is_sq_string], page 68, [ischar], page 68.

`is_sq_string (x)` [Built-in Function]

Return true if *x* is a single-quoted character string.

See also: [is_dq_string], page 68, [ischar], page 68.

5.2 Character Arrays

The string representation used by Octave is an array of characters, so internally the string `"dddddddddd"` is actually a row vector of length 10 containing the value 100 in all places (100 is the ASCII code of `"d"`). This lends itself to the obvious generalization to character matrices. Using a matrix of characters, it is possible to represent a collection of same-length strings in one variable. The convention used in Octave is that each row in a character matrix is a separate string, but letting each column represent a string is equally possible.

The easiest way to create a character matrix is to put several strings together into a matrix.

```
collection = [ "String #1"; "String #2" ];
```

This creates a 2-by-9 character matrix.

The function `ischar` can be used to test if an object is a character matrix.

`ischar (x)` [Built-in Function]

Return true if *x* is a character array.

See also: [isfloat], page 62, [isinteger], page 54, [islogical], page 62, [isnumeric], page 62, [iscellstr], page 119, [isa], page 39.

To test if an object is a string (i.e., a character vector and not a character matrix) you can use the `ischar` function in combination with the `isvector` function as in the following example:

```
ischar (collection)
     ⇒ 1

ischar (collection) && isvector (collection)
     ⇒ 0

ischar ("my string") && isvector ("my string")
     ⇒ 1
```

One relevant question is, what happens when a character matrix is created from strings of different length. The answer is that Octave puts blank characters at the end of strings shorter than the longest string. It is possible to use a different character than the blank character using the `string_fill_char` function.

val = `string_fill_char ()` [Built-in Function]
old_val = `string_fill_char (`*new_val*`)` [Built-in Function]
`string_fill_char (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable used to pad all rows of a character matrix to the same length.

The value must be a single character and the default is `" "` (a single space). For example:

```
string_fill_char ("X");
[ "these"; "are"; "strings" ]
     ⇒  "theseXX"
        "areXXXX"
        "strings"
```

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

This shows a problem with character matrices. It simply isn't possible to represent strings of different lengths. The solution is to use a cell array of strings, which is described in Section 6.2.4 [Cell Arrays of Strings], page 118.

5.3 Creating Strings

The easiest way to create a string is, as illustrated in the introduction, to enclose a text in double-quotes or single-quotes. It is however possible to create a string without actually writing a text. The function `blanks` creates a string of a given length consisting only of blank characters (ASCII code 32).

`blanks (`*n*`)` [Function File]

Return a string of *n* blanks.

For example:

```
blanks (10);
whos ans
     ⇒
      Attr Name        Size                     Bytes  Class
      ==== ====        ====                     =====  =====
           ans         1x10                        10  char
```

See also: [repmat], page 420.

5.3.1 Concatenating Strings

Strings can be concatenated using matrix notation (see Chapter 5 [Strings], page 67, Section 5.2 [Character Arrays], page 68) which is often the most natural method. For example:

```
fullname = [fname ".txt"];
email = ["<" user "@" domain ">"];
```

In each case it is easy to see what the final string will look like. This method is also the most efficient. When using matrix concatenation the parser immediately begins joining the strings without having to process the overhead of a function call and the input validation of the associated function.

Nevertheless, there are several other functions for concatenating string objects which can be useful in specific circumstances: `char`, `strvcat`, `strcat`, and `cstrcat`. Finally, the general purpose concatenation functions can be used: see [cat], page 411, [horzcat], page 411, and [vertcat], page 411.

- All string concatenation functions except `cstrcat` convert numerical input into character data by taking the corresponding ASCII character for each element, as in the following example:

  ```
  char ([98, 97, 110, 97, 110, 97])
     ⇒ banana
  ```

- `char` and `strvcat` concatenate vertically, while `strcat` and `cstrcat` concatenate horizontally. For example:

  ```
  char ("an apple", "two pears")
      ⇒ an apple
         two pears

  strcat ("oc", "tave", " is", " good", " for you")
       ⇒ octave is good for you
  ```

- `char` generates an empty row in the output for each empty string in the input. `strvcat`, on the other hand, eliminates empty strings.

  ```
  char ("orange", "green", "", "red")
      ⇒ orange
         green

         red
  ```

```
strvcat ("orange", "green", "", "red")
     ⇒ orange
       green
       red
```

- All string concatenation functions except `cstrcat` also accept cell array data (see Section 6.2 [Cell Arrays], page 112). `char` and `strvcat` convert cell arrays into character arrays, while `strcat` concatenates within the cells of the cell arrays:

```
char ({"red", "green", "", "blue"})
     ⇒ red
       green

       blue

strcat ({"abc"; "ghi"}, {"def"; "jkl"})
     ⇒
       {
         [1,1] = abcdef
         [2,1] = ghijkl
       }
```

- `strcat` removes trailing white space in the arguments (except within cell arrays), while `cstrcat` leaves white space untouched. Both kinds of behavior can be useful as can be seen in the examples:

```
strcat (["dir1";"directory2"], ["/";"/"], ["file1";"file2"])
     ⇒ dir1/file1
       directory2/file2

cstrcat (["thirteen apples"; "a banana"], [" 5$";" 1$"])
      ⇒ thirteen apples 5$
        a banana        1$
```

  Note that in the above example for `cstrcat`, the white space originates from the internal representation of the strings in a string array (see Section 5.2 [Character Arrays], page 68).

char (*x*) [Built-in Function]
char (*x*, ...) [Built-in Function]
char (*s1*, *s2*, ...) [Built-in Function]
char (*cell_array*) [Built-in Function]

Create a string array from one or more numeric matrices, character matrices, or cell arrays.

Arguments are concatenated vertically. The returned values are padded with blanks as needed to make each row of the string array have the same length. Empty input strings are significant and will concatenated in the output.

For numerical input, each element is converted to the corresponding ASCII character. A range error results if an input is outside the ASCII range (0-255).

For cell arrays, each element is concatenated separately. Cell arrays converted through `char` can mostly be converted back with `cellstr`. For example:

```
char ([97, 98, 99], "", {"98", "99", 100}, "str1", ["ha", "lf"])
   ⇒ ["abc    "
       "       "
       "98     "
       "99     "
       "d      "
       "str1   "
       "half   "]
```

See also: [strvcat], page 72, [cellstr], page 119.

`strvcat` (*x*) [Built-in Function]
`strvcat` (*x*, ...) [Built-in Function]
`strvcat` (*s1*, *s2*, ...) [Built-in Function]
`strvcat` (*cell_array*) [Built-in Function]

Create a character array from one or more numeric matrices, character matrices, or cell arrays.

Arguments are concatenated vertically. The returned values are padded with blanks as needed to make each row of the string array have the same length. Unlike `char`, empty strings are removed and will not appear in the output.

For numerical input, each element is converted to the corresponding ASCII character. A range error results if an input is outside the ASCII range (0-255).

For cell arrays, each element is concatenated separately. Cell arrays converted through `strvcat` can mostly be converted back with `cellstr`. For example:

```
strvcat ([97, 98, 99], "", {"98", "99", 100}, "str1", ["ha", "lf"])
      ⇒ ["abc    "
          "98     "
          "99     "
          "d      "
          "str1   "
          "half   "]
```

See also: [char], page 71, [strcat], page 72, [cstrcat], page 73.

`strcat` (*s1*, *s2*, ...) [Function File]

Return a string containing all the arguments concatenated horizontally.

If the arguments are cell strings, `strcat` returns a cell string with the individual cells concatenated. For numerical input, each element is converted to the corresponding ASCII character. Trailing white space for any character string input is eliminated before the strings are concatenated. Note that cell string values do **not** have whitespace trimmed.

For example:

```
strcat ("|", " leading space is preserved", "|")
    ⇒ | leading space is preserved|
strcat ("|", "trailing space is eliminated ", "|")
    ⇒ |trailing space is eliminated|
```

```
strcat ("homogeneous space |", "  ", "| is also eliminated")
    ⇒ homogeneous space || is also eliminated
s = [ "ab"; "cde" ];
strcat (s, s, s)
    ⇒
        "ababab   "
        "cdecdecde"
s = { "ab"; "cd " };
strcat (s, s, s)
    ⇒
        {
          [1,1] = ababab
          [2,1] = cd cd cd
        }
```

See also: [cstrcat], page 73, [char], page 71, [strvcat], page 72.

`cstrcat` (*s1*, *s2*, ...) [Function File]

Return a string containing all the arguments concatenated horizontally with trailing white space preserved.

For example:

```
cstrcat ("ab   ", "cd")
      ⇒ "ab   cd"
s = [ "ab"; "cde" ];
cstrcat (s, s, s)
      ⇒ "ab ab ab "
         "cdecdecde"
```

See also: [strcat], page 72, [char], page 71, [strvcat], page 72.

5.3.2 Converting Numerical Data to Strings

Apart from the string concatenation functions (see Section 5.3.1 [Concatenating Strings], page 70) which cast numerical data to the corresponding ASCII characters, there are several functions that format numerical data as strings. `mat2str` and `num2str` convert real or complex matrices, while `int2str` converts integer matrices. `int2str` takes the real part of complex values and round fractional values to integer. A more flexible way to format numerical data as strings is the `sprintf` function (see Section 14.2.4 [Formatted Output], page 253, [sprintf], page 254).

s = `mat2str` (*x*, *n*) [Function File]
s = `mat2str` (*x*, *n*, "*class*") [Function File]

Format real, complex, and logical matrices as strings.

The returned string may be used to reconstruct the original matrix by using the `eval` function.

The precision of the values is given by *n*. If *n* is a scalar then both real and imaginary parts of the matrix are printed to the same precision. Otherwise `n(1)` defines the precision of the real part and *n*(2) defines the precision of the imaginary part. The default for *n* is 15.

If the argument `"class"` is given then the class of *x* is included in the string in such a way that `eval` will result in the construction of a matrix of the same class.

```
mat2str ([ -1/3 + i/7; 1/3 - i/7 ], [4 2])
     ⇒ "[-0.3333+0.14i;0.3333-0.14i]"

mat2str ([ -1/3 +i/7; 1/3 -i/7 ], [4 2])
     ⇒ "[-0.3333+0i 0+0.14i;0.3333+0i -0-0.14i]"

mat2str (int16 ([1 -1]), "class")
     ⇒ "int16([1 -1])"

mat2str (logical (eye (2)))
     ⇒ "[true false;false true]"

isequal (x, eval (mat2str (x)))
     ⇒ 1
```

See also: [sprintf], page 254, [num2str], page 74, [int2str], page 75.

`num2str` (*x*) [Function File]
`num2str` (*x*, *precision*) [Function File]
`num2str` (*x*, *format*) [Function File]

Convert a number (or array) to a string (or a character array).

The optional second argument may either give the number of significant digits (*precision*) to be used in the output or a format template string (*format*) as in `sprintf` (see Section 14.2.4 [Formatted Output], page 253). `num2str` can also process complex numbers.

Examples:

```
num2str (123.456)
     ⇒ "123.46"

num2str (123.456, 4)
     ⇒ "123.5"

s = num2str ([1, 1.34; 3, 3.56], "%5.1f")
     ⇒ s =
        1.0  1.3
        3.0  3.6
whos s
     ⇒
      Attr Name        Size                     Bytes  Class
      ==== ====        ====                     =====  =====
           s           2x8                         16  char

num2str (1.234 + 27.3i)
     ⇒ "1.234+27.3i"
```

Notes:

For MATLAB compatibility, leading spaces are stripped before returning the string.

The `num2str` function is not very flexible. For better control over the results, use `sprintf` (see Section 14.2.4 [Formatted Output], page 253).

For complex *x*, the format string may only contain one output conversion specification and nothing else. Otherwise, results will be unpredictable.

See also: [sprintf], page 254, [int2str], page 75, [mat2str], page 73.

`int2str` (*n*) [Function File]

Convert an integer (or array of integers) to a string (or a character array).

```
int2str (123)
     ⇒ "123"

s = int2str ([1, 2, 3; 4, 5, 6])
     ⇒ s =
        1  2  3
        4  5  6

whos s
     ⇒
      Attr Name        Size                     Bytes  Class
      ==== ====        ====                     =====  =====
           s           2x7                         14  char
```

This function is not very flexible. For better control over the results, use `sprintf` (see Section 14.2.4 [Formatted Output], page 253).

See also: [sprintf], page 254, [num2str], page 74, [mat2str], page 73.

5.4 Comparing Strings

Since a string is a character array, comparisons between strings work element by element as the following example shows:

```
GNU = "GNU's Not UNIX";
spaces = (GNU == " ")
     ⇒ spaces =
       0   0   0   0   0   1   0   0   0   1   0   0   0   0
```

To determine if two strings are identical it is necessary to use the `strcmp` function. It compares complete strings and is case sensitive. `strncmp` compares only the first `N` characters (with `N` given as a parameter). `strcmpi` and `strncmpi` are the corresponding functions for case-insensitive comparison.

`strcmp` (*s1*, *s2*) [Built-in Function]

Return 1 if the character strings *s1* and *s2* are the same, and 0 otherwise.

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Caution: For compatibility with MATLAB, Octave's strcmp function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

See also: [strcmpi], page 76, [strncmp], page 76, [strncmpi], page 77.

`strncmp` (*s1*, *s2*, *n*) [Built-in Function]

Return 1 if the first *n* characters of strings *s1* and *s2* are the same, and 0 otherwise.

```
strncmp ("abce", "abcd", 3)
      ⇒ 1
```

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

```
strncmp ("abce", {"abcd", "bca", "abc"}, 3)
     ⇒ [1, 0, 1]
```

Caution: For compatibility with MATLAB, Octave's strncmp function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

See also: [strncmpi], page 77, [strcmp], page 76, [strcmpi], page 76.

`strcmpi` (*s1*, *s2*) [Built-in Function]

Return 1 if the character strings *s1* and *s2* are the same, disregarding case of alphabetic characters, and 0 otherwise.

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Caution: For compatibility with MATLAB, Octave's strcmp function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

Caution: National alphabets are not supported.

See also: [strcmp], page 76, [strncmp], page 76, [strncmpi], page 77.

`strncmpi` (*s1*, *s2*, *n*) [Built-in Function]

Return 1 if the first *n* character of *s1* and *s2* are the same, disregarding case of alphabetic characters, and 0 otherwise.

If either *s1* or *s2* is a cell array of strings, then an array of the same size is returned, containing the values described above for every member of the cell array. The other argument may also be a cell array of strings (of the same size or with only one element), char matrix or character string.

Caution: For compatibility with MATLAB, Octave's strncmpi function returns 1 if the character strings are equal, and 0 otherwise. This is just the opposite of the corresponding C library function.

Caution: National alphabets are not supported.

See also: [strncmp], page 76, [strcmp], page 76, [strcmpi], page 76.

5.5 Manipulating Strings

Octave supports a wide range of functions for manipulating strings. Since a string is just a matrix, simple manipulations can be accomplished using standard operators. The following example shows how to replace all blank characters with underscores.

```
quote = ...
  "First things first, but not necessarily in that order";
quote( quote == " " ) = "_"
⇒ quote =
    First_things_first,_but_not_necessarily_in_that_order
```

For more complex manipulations, such as searching, replacing, and general regular expressions, the following functions come with Octave.

`deblank` (*s*) [Function File]

Remove trailing whitespace and nulls from *s*.

If *s* is a matrix, *deblank* trims each row to the length of longest string. If *s* is a cell array of strings, operate recursively on each string element.

Examples:

```
deblank ("    abc  ")
     ⇒  "    abc"

deblank ([" abc   "; "   def   "])
     ⇒  [" abc  " ; "   def"]
```

See also: [strtrim], page 78.

`strtrim (s)` [Function File]

Remove leading and trailing whitespace from *s*.

If *s* is a matrix, *strtrim* trims each row to the length of longest string. If *s* is a cell array of strings, operate recursively on each string element.

For example:

```
strtrim ("    abc  ")
     ⇒  "abc"

strtrim ([" abc   "; "   def   "])
     ⇒  ["abc  "  ; "  def"]
```

See also: [deblank], page 77.

`strtrunc (s, n)` [Function File]

Truncate the character string *s* to length *n*.

If *s* is a character matrix, then the number of columns is adjusted.

If *s* is a cell array of strings, then the operation is performed on each cell element and the new cell array is returned.

`findstr (s, t)` [Function File]
`findstr (s, t, overlap)` [Function File]

Return the vector of all positions in the longer of the two strings *s* and *t* where an occurrence of the shorter of the two starts.

If the optional argument *overlap* is true (default), the returned vector can include overlapping positions. For example:

```
findstr ("ababab", "a")
     ⇒ [1, 3, 5];
findstr ("abababa", "aba", 0)
     ⇒ [1, 5]
```

Caution: `findstr` is scheduled for deprecation. Use `strfind` in all new code.

See also: [strfind], page 79, [strmatch], page 80, [strcmp], page 76, [strncmp], page 76, [strcmpi], page 76, [strncmpi], page 77, [find], page 407.

`idx = strchr (str, chars)` [Function File]
`idx = strchr (str, chars, n)` [Function File]
`idx = strchr (str, chars, n, direction)` [Function File]
`[i, j] = strchr (...)` [Function File]

Search for the string *str* for occurrences of characters from the set *chars*.

The return value(s), as well as the *n* and *direction* arguments behave identically as in `find`.

This will be faster than using regexp in most cases.

See also: [find], page 407.

`index (s, t)` [Function File]
`index (s, t, direction)` [Function File]

Return the position of the first occurrence of the string *t* in the string *s*, or 0 if no occurrence is found.

s may also be a string array or cell array of strings.

For example:

```
index ("Teststring", "t")
    ⇒ 4
```

If *direction* is `"first"`, return the first element found. If *direction* is `"last"`, return the last element found.

See also: [find], page 407, [rindex], page 79.

`rindex (`*s*, *t*`)` [Function File]

Return the position of the last occurrence of the character string *t* in the character string *s*, or 0 if no occurrence is found.

s may also be a string array or cell array of strings.

For example:

```
rindex ("Teststring", "t")
     ⇒ 6
```

The `rindex` function is equivalent to `index` with *direction* set to `"last"`.

See also: [find], page 407, [index], page 78.

idx `= strfind (`*str*, *pattern*`)` [Built-in Function]
idx `= strfind (`*cellstr*, *pattern*`)` [Built-in Function]
idx `= strfind (`. . . , `"`*overlaps*`"`, *val*`)` [Built-in Function]

Search for *pattern* in the string *str* and return the starting index of every such occurrence in the vector *idx*.

If there is no such occurrence, or if *pattern* is longer than *str*, or if *pattern* itself is empty, then *idx* is the empty array `[]`.

The optional argument `"overlaps"` determines whether the pattern can match at every position in *str* (true), or only for unique occurrences of the complete pattern (false). The default is true.

If a cell array of strings *cellstr* is specified then *idx* is a cell array of vectors, as specified above.

Examples:

```
strfind ("abababa", "aba")
     ⇒ [1, 3, 5]

strfind ("abababa", "aba", "overlaps", false)
     ⇒ [1, 5]

strfind ({"abababa", "bebebe", "ab"}, "aba")
     ⇒
        {
          [1,1] =

             1   3   5

          [1,2] = [](1x0)
          [1,3] = [](1x0)
        }
```

See also: [findstr], page 78, [strmatch], page 80, [regexp], page 87, [regexpi], page 89, [find], page 407.

str = strjoin (*cstr*) [Function File]
str = strjoin (*cstr*, *delimiter*) [Function File]

Join the elements of the cell string array, *cstr*, into a single string.

If no *delimiter* is specified, the elements of *cstr* are separated by a space.

If *delimiter* is specified as a string, the cell string array is joined using the string. Escape sequences are supported.

If *delimiter* is a cell string array whose length is one less than *cstr*, then the elements of *cstr* are joined by interleaving the cell string elements of *delimiter*. Escape sequences are not supported.

```
strjoin ({'Octave','Scilab','Lush','Yorick'}, '*')
      ⇒ 'Octave*Scilab*Lush*Yorick'
```

See also: [strsplit], page 81.

strmatch (*s*, *A*) [Function File]
strmatch (*s*, *A*, "*exact*") [Function File]

Return indices of entries of *A* which begin with the string *s*.

The second argument *A* must be a string, character matrix, or a cell array of strings.

If the third argument `"exact"` is not given, then *s* only needs to match *A* up to the length of *s*. Trailing spaces and nulls in *s* and *A* are ignored when matching.

For example:

```
strmatch ("apple", "apple juice")
     ⇒ 1

strmatch ("apple", ["apple  "; "apple juice"; "an apple"])
     ⇒ [1; 2]

strmatch ("apple", ["apple  "; "apple juice"; "an apple"], "exact")
     ⇒ [1]
```

Caution: `strmatch` is scheduled for deprecation. Use `strncmp` (normal case), or `strcmp` (`"exact"` case), or `regexp` in all new code.

See also: [strfind], page 79, [findstr], page 78, [strcmp], page 76, [strncmp], page 76, [strcmpi], page 76, [strncmpi], page 77, [find], page 407.

`[`*tok*`,` *rem*`] = strtok (`*str*`)` [Function File]
`[`*tok*`,` *rem*`] = strtok (`*str*`,` *delim*`)` [Function File]

Find all characters in the string *str* up to, but not including, the first character which is in the string *delim*.

str may also be a cell array of strings in which case the function executes on every individual string and returns a cell array of tokens and remainders.

Leading delimiters are ignored. If *delim* is not specified, whitespace is assumed.

If *rem* is requested, it contains the remainder of the string, starting at the first delimiter.

Examples:

```
strtok ("this is the life")
     ⇒ "this"

[tok, rem] = strtok ("14*27+31", "+-*/")
     ⇒
        tok = 14
        rem = *27+31
```

See also: [index], page 78, [strsplit], page 81, [strchr], page 78, [isspace], page 97.

`[`*cstr*`] = strsplit (`*str*`)` [Function File]
`[`*cstr*`] = strsplit (`*str*`,` *del*`)` [Function File]
`[`*cstr*`] = strsplit (...,` *name*`,` *value*`)` [Function File]
`[`*cstr*`,` *matches*`] = strsplit (...)` [Function File]

Split the string *str* using the delimiters specified by *del* and return a cell string array of substrings.

If a delimiter is not specified the string is split at whitespace `{" ", "\f", "\n", "\r", "\t", "\v"}`. Otherwise, the delimiter, *del* must be a string or cell array of strings. By default, consecutive delimiters in the input string *s* are collapsed into one resulting in a single split.

Supported *name*/*value* pair arguments are:

- *collapsedelimiters* which may take the value of `true` (default) or `false`.

- *delimitertype* which may take the value of `"simple"` (default) or `"regularexpression"`. A simple delimiter matches the text exactly as written. Otherwise, the syntax for regular expressions outlined in `regexp` is used.

The optional second output, *matches*, returns the delimiters which were matched in the original string.

Examples with simple delimiters:

```
strsplit ("a b c")
      ⇒
          {
            [1,1] = a
            [1,2] = b
            [1,3] = c
          }

strsplit ("a,b,c", ",")
      ⇒
          {
            [1,1] = a
            [1,2] = b
            [1,3] = c
          }

strsplit ("a foo b,bar c", {" ", ",", "foo", "bar"})
      ⇒
          {
            [1,1] = a
            [1,2] = b
            [1,3] = c
          }

strsplit ("a,,b, c", {",", " "}, "collapsedelimiters", false)
      ⇒
          {
            [1,1] = a
            [1,2] =
            [1,3] = b
            [1,4] =
            [1,5] = c
          }
```

Examples with regularexpression delimiters:

```
strsplit ("a foo b,bar c", ',|\s|foo|bar', "delimitertype", "regularexpression")
⇒
{
          [1,1] = a
          [1,2] = b
          [1,3] = c
```

```
}

strsplit ("a,,b, c", '[, ]', "collapsedelimiters", false, "delimitertype", "regularexpression")
⇒
{
          [1,1] = a
          [1,2] =
          [1,3] = b
          [1,4] =
          [1,5] = c
}

strsplit ("a,\t,b, c", {',', '\s'}, "delimitertype", "regularexpression")
⇒
{
          [1,1] = a
          [1,2] = b
          [1,3] = c
}

strsplit ("a,\t,b, c", {',', ' ', '\t'}, "collapsedelimiters", false)
⇒
{
          [1,1] = a
          [1,2] =
          [1,3] =
          [1,4] = b
          [1,5] =
          [1,6] = c
}
```

See also: [ostrsplit], page 83, [strjoin], page 80, [strtok], page 81, [regexp], page 87.

`[`*cstr*`] = ostrsplit (`*s*, *sep*`)` [Function File]
`[`*cstr*`] = ostrsplit (`*s*, *sep*, *strip_empty*`)` [Function File]

Split the string *s* using one or more separators *sep* and return a cell array of strings.

Consecutive separators and separators at boundaries result in empty strings, unless *strip_empty* is true. The default value of *strip_empty* is false.

2-D character arrays are split at separators and at the original column boundaries.

Example:

```
ostrsplit ("a,b,c", ",")
      ⇒
          {
            [1,1] = a
            [1,2] = b
            [1,3] = c
          }

ostrsplit (["a,b" ; "cde"], ",")
      ⇒
          {
            [1,1] = a
            [1,2] = b
            [1,3] = cde
          }
```

See also: [strsplit], page 81, [strtok], page 81.

[a, ...] = strread (*str*) [Function File]
[a, ...] = strread (*str*, *format*) [Function File]
[a, ...] = strread (*str*, *format*, *format_repeat*) [Function File]
[a, ...] = strread (*str*, *format*, *prop1*, *value1*, ...) [Function File]
[a, ...] = strread (*str*, *format*, *format_repeat*, *prop1*, *value1*, ...) [Function File]

Read data from a string.

The string *str* is split into words that are repeatedly matched to the specifiers in *format*. The first word is matched to the first specifier, the second to the second specifier and so forth. If there are more words than specifiers, the process is repeated until all words have been processed.

The string *format* describes how the words in *str* should be parsed. It may contain any combination of the following specifiers:

`%s` The word is parsed as a string.

`%f`
`%n` The word is parsed as a number and converted to double.

`%d`
`%u` The word is parsed as a number and converted to int32.

`%*', '%*f', '%*s`
The word is skipped.

For %s and %d, %f, %n, %u and the associated %*s ... specifiers an optional width can be specified as %Ns, etc. where N is an integer > 1. For %f, format specifiers like %N.Mf are allowed.

`literals` In addition the format may contain literal character strings; these will be skipped during reading.

Parsed word corresponding to the first specifier are returned in the first output argument and likewise for the rest of the specifiers.

By default, *format* is "%f", meaning that numbers are read from *str*. This will do if *str* contains only numeric fields.

For example, the string

```
str = "\
Bunny Bugs   5.5\n\
Duck Daffy  -7.5e-5\n\
Penguin Tux   6"
```

can be read using

```
[a, b, c] = strread (str, "%s %s %f");
```

Optional numeric argument *format_repeat* can be used for limiting the number of items read:

-1 (default) read all of the string until the end.

N Read N times *nargout* items. 0 (zero) is an acceptable value for *format_repeat*.

The behavior of `strread` can be changed via property-value pairs. The following properties are recognized:

`"commentstyle"`
Parts of *str* are considered comments and will be skipped. *value* is the comment style and can be any of the following.

- `"shell"` Everything from `#` characters to the nearest end-of-line is skipped.
- `"c"` Everything between `/*` and `*/` is skipped.
- `"c++"` Everything from `//` characters to the nearest end-of-line is skipped.
- `"matlab"` Everything from `%` characters to the nearest end-of-line is skipped.
- user-supplied. Two options: (1) One string, or 1x1 cell string: Skip everything to the right of it; (2) 2x1 cell string array: Everything between the left and right strings is skipped.

`"delimiter"`
Any character in *value* will be used to split *str* into words (default value = any whitespace). Note that whitespace is implicitly added to the set of delimiter characters unless a `"%s"` format conversion specifier is supplied; see `"whitespace"` parameter below. The set of delimiter characters cannot be empty; if needed Octave substitutes a space as delimiter.

`"emptyvalue"`
Value to return for empty numeric values in non-whitespace delimited data. The default is NaN. When the data type does not support NaN (int32 for example), then default is zero.

`"multipledelimsasone"`
Treat a series of consecutive delimiters, without whitespace in between, as a single delimiter. Consecutive delimiter series need not be vertically `"aligned"`.

`"treatasempty"`
: Treat single occurrences (surrounded by delimiters or whitespace) of the string(s) in *value* as missing values.

`"returnonerror"`
: If *value* true (1, default), ignore read errors and return normally. If false (0), return an error.

`"whitespace"`
: Any character in *value* will be interpreted as whitespace and trimmed; the string defining whitespace must be enclosed in double quotes for proper processing of special characters like `"\t"`. In each data field, multiple consecutive whitespace characters are collapsed into one space and leading and trailing whitespace is removed. The default value for whitespace is `" \b\r\n\t"` (note the space). Whitespace is always added to the set of delimiter characters unless at least one `"%s"` format conversion specifier is supplied; in that case only whitespace explicitly specified in `"delimiter"` is retained as delimiter and removed from the set of whitespace characters. If whitespace characters are to be kept as-is (in e.g., strings), specify an empty value (i.e., `""`) for `"whitespace"`; obviously, whitespace cannot be a delimiter then.

When the number of words in *str* doesn't match an exact multiple of the number of format conversion specifiers, strread's behavior depends on the last character of *str*:

last character = `"\n"`
: Data columns are padded with empty fields or Nan so that all columns have equal length

last character is not `"\n"`
: Data columns are not padded; strread returns columns of unequal length

See also: [textscan], page 246, [textread], page 244, [load], page 241, [dlmread], page 244, [fscanf], page 259.

newstr = `strrep` (*str*, *ptn*, *rep*) [Built-in Function]
newstr = `strrep` (*cellstr*, *ptn*, *rep*) [Built-in Function]
newstr = `strrep` (. . . , "*overlaps*", *val*) [Built-in Function]

Replace all occurrences of the pattern *ptn* in the string *str* with the string *rep* and return the result.

The optional argument `"overlaps"` determines whether the pattern can match at every position in *str* (true), or only for unique occurrences of the complete pattern (false). The default is true.

s may also be a cell array of strings, in which case the replacement is done for each element and a cell array is returned.

Example:

```
strrep ("This is a test string", "is", "&%$")
    ⇒  "Th&%$ &%$ a test string"
```

See also: [regexprep], page 90, [strfind], page 79, [findstr], page 78.

`substr` (*s*, *offset*) [Function File]
`substr` (*s*, *offset*, *len*) [Function File]

Return the substring of *s* which starts at character number *offset* and is *len* characters long.

Position numbering for offsets begins with 1. If *offset* is negative, extraction starts that far from the end of the string.

If *len* is omitted, the substring extends to the end of *s*. A negative value for *len* extracts to within *len* characters of the end of the string

Examples:

```
substr ("This is a test string", 6, 9)
     ⇒ "is a test"
substr ("This is a test string", -11)
     ⇒ "test string"
substr ("This is a test string", -11, -7)
     ⇒ "test"
```

This function is patterned after the equivalent function in Perl.

[*s*, *e*, *te*, *m*, *t*, *nm*, *sp*] = `regexp` (*str*, *pat*) [Built-in Function]
[...] = `regexp` (*str*, *pat*, "*opt1*", ...) [Built-in Function]

Regular expression string matching.

Search for *pat* in *str* and return the positions and substrings of any matches, or empty values if there are none.

The matched pattern *pat* can include any of the standard regex operators, including:

`.` Match any character

`* + ? {}` Repetition operators, representing

- `*` Match zero or more times
- `+` Match one or more times
- `?` Match zero or one times
- `{n}` Match exactly *n* times
- `{n,}` Match *n* or more times
- `{m,n}` Match between *m* and *n* times

`[...] [^...]`
List operators. The pattern will match any character listed between "[" and "]". If the first character is "^" then the pattern is inverted and any character except those listed between brackets will match.

Escape sequences defined below can also be used inside list operators. For example, a template for a floating point number might be `[-+.\d]+`.

`() (?:)` Grouping operator. The first form, parentheses only, also creates a token.

`|` Alternation operator. Match one of a choice of regular expressions. The alternatives must be delimited by the grouping operator `()` above.

`^ $` Anchoring operators. Requires pattern to occur at the start (`^`) or end (`$`) of the string.

In addition, the following escaped characters have special meaning.

`\d` Match any digit

`\D` Match any non-digit

`\s` Match any whitespace character

`\S` Match any non-whitespace character

`\w` Match any word character

`\W` Match any non-word character

`\<` Match the beginning of a word

`\>` Match the end of a word

`\B` Match within a word

Implementation Note: For compatibility with MATLAB, escape sequences in *pat* (e.g., `"\n"` => newline) are expanded even when *pat* has been defined with single quotes. To disable expansion use a second backslash before the escape sequence (e.g., `"\\n"`) or use the `regexptranslate` function.

The outputs of `regexp` default to the order given below

s The start indices of each matching substring

e The end indices of each matching substring

te The extents of each matched token surrounded by `(...)` in *pat*

m A cell array of the text of each match

t A cell array of the text of each token matched

nm A structure containing the text of each matched named token, with the name being used as the fieldname. A named token is denoted by `(?<name>...)`.

sp A cell array of the text not returned by match, i.e., what remains if you split the string based on *pat*.

Particular output arguments, or the order of the output arguments, can be selected by additional *opt* arguments. These are strings and the correspondence between the output arguments and the optional argument are

`'start'`	*s*
`'end'`	*e*
`'tokenExtents'`	*te*
`'match'`	*m*
`'tokens'`	*t*
`'names'`	*nm*
`'split'`	*sp*

Additional arguments are summarized below.

'`once`' Return only the first occurrence of the pattern.

'matchcase'
: Make the matching case sensitive. (default)

  Alternatively, use (?-i) in the pattern.

'ignorecase'
: Ignore case when matching the pattern to the string.

  Alternatively, use (?i) in the pattern.

'stringanchors'
: Match the anchor characters at the beginning and end of the string. (default)

  Alternatively, use (?-m) in the pattern.

'lineanchors'
: Match the anchor characters at the beginning and end of the line.

  Alternatively, use (?m) in the pattern.

'dotall'
: The pattern `.` matches all characters including the newline character. (default)

  Alternatively, use (?s) in the pattern.

'dotexceptnewline'
: The pattern `.` matches all characters except the newline character.

  Alternatively, use (?-s) in the pattern.

'literalspacing'
: All characters in the pattern, including whitespace, are significant and are used in pattern matching. (default)

  Alternatively, use (?-x) in the pattern.

'freespacing'
: The pattern may include arbitrary whitespace and also comments beginning with the character '#'.

  Alternatively, use (?x) in the pattern.

'noemptymatch'
: Zero-length matches are not returned. (default)

'emptymatch'
: Return zero-length matches.

  `regexp ('a', 'b*', 'emptymatch')` returns `[1 2]` because there are zero or more `'b'` characters at positions 1 and end-of-string.

See also: [regexpi], page 89, [strfind], page 79, [regexprep], page 90.

`[`*s*`,` *e*`,` *te*`,` *m*`,` *t*`,` *nm*`,` *sp*`] = regexpi (`*str*`,` *pat*`)` [Built-in Function]
`[...] = regexpi (`*str*`,` *pat*`, "`*opt1*`", ...)` [Built-in Function]

Case insensitive regular expression string matching.

Search for *pat* in *str* and return the positions and substrings of any matches, or empty values if there are none. See [regexp], page 87, for details on the syntax of the search pattern.

See also: [regexp], page 87.

outstr = `regexprep` (*string*, *pat*, *repstr*) [Built-in Function]
outstr = `regexprep` (*string*, *pat*, *repstr*, "*opt1*", ...) [Built-in Function]

Replace occurrences of pattern *pat* in *string* with *repstr*.

The pattern is a regular expression as documented for `regexp`. See [regexp], page 87.

The replacement string may contain `$i`, which substitutes for the ith set of parentheses in the match string. For example,

```
regexprep ("Bill Dunn", '(\w+) (\w+)', '$2, $1')
```

returns "Dunn, Bill"

Options in addition to those of `regexp` are

'`once`' Replace only the first occurrence of *pat* in the result.

'`warnings`'
This option is present for compatibility but is ignored.

Implementation Note: For compatibility with MATLAB, escape sequences in *pat* (e.g., `"\n"` => newline) are expanded even when *pat* has been defined with single quotes. To disable expansion use a second backslash before the escape sequence (e.g., `"\\n"`) or use the `regexptranslate` function.

See also: [regexp], page 87, [regexpi], page 89, [strrep], page 86.

`regexptranslate` (*op*, *s*) [Function File]

Translate a string for use in a regular expression.

This may include either wildcard replacement or special character escaping.

The behavior is controlled by *op* which can take the following values

`"wildcard"`
The wildcard characters `.`, `*`, and `?` are replaced with wildcards that are appropriate for a regular expression. For example:

```
regexptranslate ("wildcard", "*.m")
     ⇒ ".*\.m"
```

`"escape"` The characters `$.?[]`, that have special meaning for regular expressions are escaped so that they are treated literally. For example:

```
regexptranslate ("escape", "12.5")
     ⇒ "12\.5"
```

See also: [regexp], page 87, [regexpi], page 89, [regexprep], page 90.

`untabify` (*t*) [Function File]
`untabify` (*t*, *tw*) [Function File]
`untabify` (*t*, *tw*, *deblank*) [Function File]

Replace TAB characters in *t* with spaces.

The input, *t*, may be either a 2-D character array, or a cell array of character strings. The output is the same class as the input.

The tab width is specified by *tw*, and defaults to eight.

If the optional argument *deblank* is true, then the spaces will be removed from the end of the character data.

The following example reads a file and writes an untabified version of the same file with trailing spaces stripped.

```
fid = fopen ("tabbed_script.m");
text = char (fread (fid, "uchar")');
fclose (fid);
fid = fopen ("untabified_script.m", "w");
text = untabify (strsplit (text, "\n"), 8, true);
fprintf (fid, "%s\n", text{:});
fclose (fid);
```

See also: [strjust], page 94, [strsplit], page 81, [deblank], page 77.

5.6 String Conversions

Octave supports various kinds of conversions between strings and numbers. As an example, it is possible to convert a string containing a hexadecimal number to a floating point number.

```
hex2dec ("FF")
     ⇒ 255
```

`bin2dec (s)` [Function File]

Return the decimal number corresponding to the binary number represented by the string *s*.

For example:

```
bin2dec ("1110")
     ⇒ 14
```

Spaces are ignored during conversion and may be used to make the binary number more readable.

```
bin2dec ("1000 0001")
     ⇒ 129
```

If *s* is a string matrix, return a column vector with one converted number per row of *s*; Invalid rows evaluate to NaN.

If *s* is a cell array of strings, return a column vector with one converted number per cell element in *s*.

See also: [dec2bin], page 91, [base2dec], page 92, [hex2dec], page 92.

`dec2bin (d, len)` [Function File]

Return a binary number corresponding to the non-negative integer *d*, as a string of ones and zeros.

For example:

```
dec2bin (14)
     ⇒ "1110"
```

If *d* is a matrix or cell array, return a string matrix with one row per element in *d*, padded with leading zeros to the width of the largest value.

The optional second argument, *len*, specifies the minimum number of digits in the result.

See also: [bin2dec], page 91, [dec2base], page 92, [dec2hex], page 92.

`dec2hex (d, len)` [Function File]

Return the hexadecimal string corresponding to the non-negative integer *d*.

For example:

```
dec2hex (2748)
     ⇒ "ABC"
```

If *d* is a matrix or cell array, return a string matrix with one row per element in *d*, padded with leading zeros to the width of the largest value.

The optional second argument, *len*, specifies the minimum number of digits in the result.

See also: [hex2dec], page 92, [dec2base], page 92, [dec2bin], page 91.

`hex2dec (s)` [Function File]

Return the integer corresponding to the hexadecimal number represented by the string *s*.

For example:

```
hex2dec ("12B")
      ⇒ 299
hex2dec ("12b")
      ⇒ 299
```

If *s* is a string matrix, return a column vector with one converted number per row of *s*; Invalid rows evaluate to NaN.

If *s* is a cell array of strings, return a column vector with one converted number per cell element in *s*.

See also: [dec2hex], page 92, [base2dec], page 92, [bin2dec], page 91.

`dec2base (d, base)` [Function File]
`dec2base (d, base, len)` [Function File]

Return a string of symbols in base *base* corresponding to the non-negative integer *d*.

```
dec2base (123, 3)
   ⇒ "11120"
```

If *d* is a matrix or cell array, return a string matrix with one row per element in *d*, padded with leading zeros to the width of the largest value.

If *base* is a string then the characters of *base* are used as the symbols for the digits of *d*. Space (' ') may not be used as a symbol.

```
dec2base (123, "aei")
   ⇒ "eeeia"
```

The optional third argument, *len*, specifies the minimum number of digits in the result.

See also: [base2dec], page 92, [dec2bin], page 91, [dec2hex], page 92.

`base2dec (s, base)` [Function File]

Convert *s* from a string of digits in base *base* to a decimal integer (base 10).

```
base2dec ("11120", 3)
   ⇒ 123
```

If *s* is a string matrix, return a column vector with one value per row of *s*. If a row contains invalid symbols then the corresponding value will be NaN.

If *s* is a cell array of strings, return a column vector with one value per cell element in *s*.

If *base* is a string, the characters of *base* are used as the symbols for the digits of *s*. Space (' ') may not be used as a symbol.

```
base2dec ("yyyzx", "xyz")
   ⇒ 123
```

See also: [dec2base], page 92, [bin2dec], page 91, [hex2dec], page 92.

s = `num2hex` (*n*) [Built-in Function]

Typecast a double or single precision number or vector to a 8 or 16 character hexadecimal string of the IEEE 754 representation of the number.

For example:

```
num2hex ([-1, 1, e, Inf])
⇒ "bff0000000000000
    3ff0000000000000
    4005bf0a8b145769
    7ff0000000000000"
```

If the argument *n* is a single precision number or vector, the returned string has a length of 8. For example:

```
num2hex (single ([-1, 1, e, Inf]))
⇒ "bf800000
    3f800000
    402df854
    7f800000"
```

See also: [hex2num], page 93, [hex2dec], page 92, [dec2hex], page 92.

n = `hex2num` (*s*) [Built-in Function]
n = `hex2num` (*s*, *class*) [Built-in Function]

Typecast the 16 character hexadecimal character string to an IEEE 754 double precision number.

If fewer than 16 characters are given the strings are right padded with `'0'` characters.

Given a string matrix, `hex2num` treats each row as a separate number.

```
hex2num (["4005bf0a8b145769"; "4024000000000000"])
   ⇒ [2.7183; 10.000]
```

The optional argument *class* can be passed as the string `"single"` to specify that the given string should be interpreted as a single precision number. In this case, *s* should be an 8 character hexadecimal string. For example:

```
hex2num (["402df854"; "41200000"], "single")
   ⇒ [2.7183; 10.000]
```

See also: [num2hex], page 93, [hex2dec], page 92, [dec2hex], page 92.

`str2double` (*s*) [Built-in Function]

Convert a string to a real or complex number.

The string must be in one of the following formats where a and b are real numbers and the complex unit is `'i'` or `'j'`:

- a + bi
- a + b*i
- a + i*b
- bi + a
- b*i + a
- i*b + a

If present, a and/or b are of the form [+-]d[,.]d[[eE][+-]d] where the brackets indicate optional arguments and `'d'` indicates zero or more digits. The special input values `Inf`, `NaN`, and `NA` are also accepted.

s may be a character string, character matrix, or cell array. For character arrays the conversion is repeated for every row, and a double or complex array is returned. Empty rows in *s* are deleted and not returned in the numeric array. For cell arrays each character string element is processed and a double or complex array of the same dimensions as *s* is returned.

For unconvertible scalar or character string input `str2double` returns a NaN. Similarly, for character array input `str2double` returns a NaN for any row of *s* that could not be converted. For a cell array, `str2double` returns a NaN for any element of *s* for which conversion fails. Note that numeric elements in a mixed string/numeric cell array are not strings and the conversion will fail for these elements and return NaN.

`str2double` can replace `str2num`, and it avoids the security risk of using `eval` on unknown data.

See also: [str2num], page 95.

`strjust` (*s*) [Function File]
`strjust` (*s*, *pos*) [Function File]

Return the text, *s*, justified according to *pos*, which may be `"left"`, `"center"`, or `"right"`.

If *pos* is omitted it defaults to `"right"`.

Null characters are replaced by spaces. All other character data are treated as non-white space.

Example:

```
strjust (["a"; "ab"; "abc"; "abcd"])
     ⇒
        "   a"
        "  ab"
        " abc"
        "abcd"
```

See also: [deblank], page 77, [strrep], page 86, [strtrim], page 78, [untabify], page 90.

`x = str2num (s)` [Function File]
`[x, state] = str2num (s)` [Function File]

Convert the string (or character array) *s* to a number (or an array).

Examples:

```
str2num ("3.141596")
     ⇒ 3.141596

str2num (["1, 2, 3"; "4, 5, 6"])
     ⇒ 1  2  3
        4  5  6
```

The optional second output, *state*, is logically true when the conversion is successful. If the conversion fails the numeric output, *x*, is empty and *state* is false.

Caution: As `str2num` uses the `eval` function to do the conversion, `str2num` will execute any code contained in the string *s*. Use `str2double` for a safer and faster conversion.

For cell array of strings use `str2double`.

See also: [str2double], page 94, [eval], page 155.

`toascii (s)` [Mapping Function]

Return ASCII representation of *s* in a matrix.

For example:

```
toascii ("ASCII")
    ⇒ [ 65, 83, 67, 73, 73 ]
```

See also: [char], page 71.

`tolower (s)` [Mapping Function]
`lower (s)` [Mapping Function]

Return a copy of the string or cell string *s*, with each uppercase character replaced by the corresponding lowercase one; non-alphabetic characters are left unchanged.

For example:

```
tolower ("MiXeD cAsE 123")
     ⇒ "mixed case 123"
```

See also: [toupper], page 95.

`toupper (s)` [Mapping Function]
`upper (s)` [Mapping Function]

Return a copy of the string or cell string *s*, with each lowercase character replaced by the corresponding uppercase one; non-alphabetic characters are left unchanged.

For example:

```
toupper ("MiXeD cAsE 123")
     ⇒ "MIXED CASE 123"
```

See also: [tolower], page 95.

`do_string_escapes` (*string*) [Built-in Function]

Convert escape sequences in *string* to the characters they represent.

Escape sequences begin with a leading backslash ('\') followed by 1–3 characters (.e.g., `"\n"` => newline).

See also: [undo_string_escapes], page 96.

`undo_string_escapes` (*s*) [Built-in Function]

Convert special characters in strings back to their escaped forms.

For example, the expression

```
bell = "\a";
```

assigns the value of the alert character (control-g, ASCII code 7) to the string variable `bell`. If this string is printed, the system will ring the terminal bell (if it is possible). This is normally the desired outcome. However, sometimes it is useful to be able to print the original representation of the string, with the special characters replaced by their escape sequences. For example,

```
octave:13> undo_string_escapes (bell)
ans = \a
```

replaces the unprintable alert character with its printable representation.

See also: [do_string_escapes], page 96.

5.7 Character Class Functions

Octave also provides the following character class test functions patterned after the functions in the standard C library. They all operate on string arrays and return matrices of zeros and ones. Elements that are nonzero indicate that the condition was true for the corresponding character in the string array. For example:

```
isalpha ("!Q@WERT^Y&")
     ⇒ [ 0, 1, 0, 1, 1, 1, 1, 0, 1, 0 ]
```

`isalnum` (*s*) [Mapping Function]

Return a logical array which is true where the elements of *s* are letters or digits and false where they are not.

This is equivalent to (`isalpha (s) | isdigit (s)`).

See also: [isalpha], page 96, [isdigit], page 97, [ispunct], page 97, [isspace], page 97, [iscntrl], page 97.

`isalpha` (*s*) [Mapping Function]

Return a logical array which is true where the elements of *s* are letters and false where they are not.

This is equivalent to (`islower (s) | isupper (s)`).

See also: [isdigit], page 97, [ispunct], page 97, [isspace], page 97, [iscntrl], page 97, [isalnum], page 96, [islower], page 97, [isupper], page 97.

`isletter (s)` [Function File]

Return a logical array which is true where the elements of *s* are letters and false where they are not.

This is an alias for the `isalpha` function.

See also: [isalpha], page 96, [isdigit], page 97, [ispunct], page 97, [isspace], page 97, [iscntrl], page 97, [isalnum], page 96.

`islower (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are lowercase letters and false where they are not.

See also: [isupper], page 97, [isalpha], page 96, [isletter], page 97, [isalnum], page 96.

`isupper (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are uppercase letters and false where they are not.

See also: [islower], page 97, [isalpha], page 96, [isletter], page 97, [isalnum], page 96.

`isdigit (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are decimal digits (0-9) and false where they are not.

See also: [isxdigit], page 97, [isalpha], page 96, [isletter], page 97, [ispunct], page 97, [isspace], page 97, [iscntrl], page 97.

`isxdigit (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are hexadecimal digits (0-9 and a-fA-F).

See also: [isdigit], page 97.

`ispunct (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are punctuation characters and false where they are not.

See also: [isalpha], page 96, [isdigit], page 97, [isspace], page 97, [iscntrl], page 97.

`isspace (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are whitespace characters (space, formfeed, newline, carriage return, tab, and vertical tab) and false where they are not.

See also: [iscntrl], page 97, [ispunct], page 97, [isalpha], page 96, [isdigit], page 97.

`iscntrl (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are control characters and false where they are not.

See also: [ispunct], page 97, [isspace], page 97, [isalpha], page 96, [isdigit], page 97.

`isgraph (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are printable characters (but not the space character) and false where they are not.

See also: [isprint], page 98.

`isprint (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are printable characters (including the space character) and false where they are not.

See also: [isgraph], page 97.

`isascii (s)` [Mapping Function]

Return a logical array which is true where the elements of *s* are ASCII characters (in the range 0 to 127 decimal) and false where they are not.

`isstrprop (str, prop)` [Function File]

Test character string properties.

For example:

```
isstrprop ("abc123", "alpha")
⇒ [1, 1, 1, 0, 0, 0]
```

If *str* is a cell array, `isstrpop` is applied recursively to each element of the cell array.

Numeric arrays are converted to character strings.

The second argument *prop* must be one of

`"alpha"` True for characters that are alphabetic (letters).

`"alnum"`
`"alphanum"`
True for characters that are alphabetic or digits.

`"lower"` True for lowercase letters.

`"upper"` True for uppercase letters.

`"digit"` True for decimal digits (0-9).

`"xdigit"` True for hexadecimal digits (a-fA-F0-9).

`"space"`
`"wspace"` True for whitespace characters (space, formfeed, newline, carriage return, tab, vertical tab).

`"punct"` True for punctuation characters (printing characters except space or letter or digit).

`"cntrl"` True for control characters.

`"graph"`
`"graphic"`
True for printing characters except space.

`"print"` True for printing characters including space.

`"ascii"` True for characters that are in the range of ASCII encoding.

See also: [isalpha], page 96, [isalnum], page 96, [islower], page 97, [isupper], page 97, [isdigit], page 97, [isxdigit], page 97, [isspace], page 97, [ispunct], page 97, [iscntrl], page 97, [isgraph], page 97, [isprint], page 98, [isascii], page 98.

6 Data Containers

Octave includes support for two different mechanisms to contain arbitrary data types in the same variable. Structures, which are C-like, and are indexed with named fields, and cell arrays, where each element of the array can have a different data type and or shape. Multiple input arguments and return values of functions are organized as another data container, the comma separated list.

6.1 Structures

Octave includes support for organizing data in structures. The current implementation uses an associative array with indices limited to strings, but the syntax is more like C-style structures.

6.1.1 Basic Usage and Examples

Here are some examples of using data structures in Octave.

Elements of structures can be of any value type. For example, the three expressions

```
x.a = 1;
x.b = [1, 2; 3, 4];
x.c = "string";
```

create a structure with three elements. The '`.`' character separates the structure name from the field name and indicates to Octave that this variable is a structure. To print the value of the structure you can type its name, just as for any other variable:

```
x
     ⇒ x =
        {
          a = 1
          b =

            1  2
            3  4

          c = string
        }
```

Note that Octave may print the elements in any order.

Structures may be copied just like any other variable:

```
y = x
     ⇒ y =
        {
          a = 1
          b =

            1  2
            3  4

          c = string
        }
```

Since structures are themselves values, structure elements may reference other structures. The following statements change the value of the element `b` of the structure `x` to be a data structure containing the single element `d`, which has a value of 3.

```
x.b.d = 3;
x.b
     ⇒ ans =
        {
          d = 3
        }

x
     ⇒ x =
        {
          a = 1
          b =
          {
            d = 3
          }

          c = string
        }
```

Note that when Octave prints the value of a structure that contains other structures, only a few levels are displayed. For example:

```
a.b.c.d.e = 1;
a
     ⇒ a =
        {
          b =
          {
            c =
            {
              1x1 struct array containing the fields:

              d: 1x1 struct
            }
          }
        }
```

This prevents long and confusing output from large deeply nested structures. The number of levels to print for nested structures may be set with the function `struct_levels_to_print`, and the function `print_struct_array_contents` may be used to enable printing of the contents of structure arrays.

val = `struct_levels_to_print ()` [Built-in Function]
old_val = `struct_levels_to_print` (*new_val*) [Built-in Function]
`struct_levels_to_print` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the number of structure levels to display.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [print_struct_array_contents], page 101.

val = `print_struct_array_contents ()` [Built-in Function]
old_val = `print_struct_array_contents` (*new_val*) [Built-in Function]
`print_struct_array_contents` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies whether to print struct array contents.

If true, values of struct array elements are printed. This variable does not affect scalar structures whose elements are always printed. In both cases, however, printing will be limited to the number of levels specified by *struct_levels_to_print*.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [struct_levels_to_print], page 101.

Functions can return structures. For example, the following function separates the real and complex parts of a matrix and stores them in two elements of the same structure variable.

```
function y = f (x)
  y.re = real (x);
  y.im = imag (x);
endfunction
```

When called with a complex-valued argument, `f` returns the data structure containing the real and imaginary parts of the original function argument.

```
f (rand (2) + rand (2) * I)
     ⇒ ans =
        {
          im =

            0.26475  0.14828
            0.18436  0.83669

          re =

            0.040239  0.242160
            0.238081  0.402523

        }
```

Function return lists can include structure elements, and they may be indexed like any other variable. For example:

```
[ x.u, x.s(2:3,2:3), x.v ] = svd ([1, 2; 3, 4]);
x
     ⇒ x =
        {
          u =

            -0.40455  -0.91451
            -0.91451   0.40455

          s =

             0.00000   0.00000   0.00000
             0.00000   5.46499   0.00000
             0.00000   0.00000   0.36597

          v =

            -0.57605   0.81742
            -0.81742  -0.57605

        }
```

It is also possible to cycle through all the elements of a structure in a loop, using a special form of the `for` statement (see Section 10.5.1 [Looping Over Structure Elements], page 165).

6.1.2 Structure Arrays

A structure array is a particular instance of a structure, where each of the fields of the structure is represented by a cell array. Each of these cell arrays has the same dimensions. Conceptually, a structure array can also be seen as an array of structures with identical fields. An example of the creation of a structure array is

```
x(1).a = "string1";
x(2).a = "string2";
x(1).b = 1;
x(2).b = 2;
```

which creates a 2-by-1 structure array with two fields. Another way to create a structure array is with the `struct` function (see Section 6.1.3 [Creating Structures], page 104). As previously, to print the value of the structure array, you can type its name:

```
x
     ⇒ x =
        {
          1x2 struct array containing the fields:

            a
            b
        }
```

Individual elements of the structure array can be returned by indexing the variable like *x*(1), which returns a structure with two fields:

```
x(1)
     ⇒ ans =
        {
          a = string1
          b =  1
        }
```

Furthermore, the structure array can return a comma separated list of field values (see Section 6.3 [Comma Separated Lists], page 120), if indexed by one of its own field names. For example:

```
x.a
     ⇒
        ans = string1
        ans = string2
```

Here is another example, using this comma separated list on the left-hand side of an assignment:

```
[x.a] = deal ("new string1", "new string2");
 x(1).a
     ⇒ ans = new string1
 x(2).a
     ⇒ ans = new string2
```

Just as for numerical arrays, it is possible to use vectors as indices (see Section 8.1 [Index Expressions], page 135):

```
x(3:4) = x(1:2);
[x([1,3]).a] = deal ("other string1", "other string2");
x.a
     ⇒
        ans = other string1
        ans = new string2
        ans = other string2
        ans = new string2
```

The function `size` will return the size of the structure. For the example above

```
size (x)
     ⇒ ans =

          1   4
```

Elements can be deleted from a structure array in a similar manner to a numerical array, by assigning the elements to an empty matrix. For example

```
in = struct ("call1", {x, Inf, "last"},
             "call2", {x, Inf, "first"})
     ⇒ in =
        {
          1x3 struct array containing the fields:

            call1
            call2
        }

in(1) = [];
in.call1
     ⇒
       ans = Inf
       ans = last
```

6.1.3 Creating Structures

Besides the index operator ".", Octave can use dynamic naming "(var)" or the `struct` function to create structures. Dynamic naming uses the string value of a variable as the field name. For example:

```
a = "field2";
x.a = 1;
x.(a) = 2;
x
     ⇒ x =
        {
          a =  1
          field2 =  2
        }
```

Dynamic indexing also allows you to use arbitrary strings, not merely valid Octave identifiers (note that this does not work on MATLAB):

```
a = "long field with spaces (and funny char$)";
x.a = 1;
x.(a) = 2;
x
     ⇒ x =
        {
          a =  1
          long field with spaces (and funny char$) =  2
        }
```

The warning id `Octave:language-extension` can be enabled to warn about this usage. See [warning_ids], page 214.

More realistically, all of the functions that operate on strings can be used to build the correct field name before it is entered into the data structure.

```
names = ["Bill"; "Mary"; "John"];
ages  = [37; 26; 31];
for i = 1:rows (names)
  database.(names(i,:)) = ages(i);
endfor
database
     ⇒ database =
        {
          Bill =  37
          Mary =  26
          John =  31
        }
```

The third way to create structures is the `struct` command. `struct` takes pairs of arguments, where the first argument in the pair is the fieldname to include in the structure and the second is a scalar or cell array, representing the values to include in the structure or structure array. For example:

```
struct ("field1", 1, "field2", 2)
⇒ ans =
      {
        field1 =  1
        field2 =  2
      }
```

If the values passed to `struct` are a mix of scalar and cell arrays, then the scalar arguments are expanded to create a structure array with a consistent dimension. For example:

```
s = struct ("field1", {1, "one"}, "field2", {2, "two"},
        "field3", 3);
s.field1
     ⇒
        ans =  1
        ans = one

s.field2
     ⇒
        ans =  2
        ans = two

s.field3
     ⇒
        ans =  3
        ans =  3
```

If you want to create a struct which contains a cell array as an individual field, you must wrap it in another cell array as shown in the following example:

```
struct ("field1", {{1, "one"}}, "field2", 2)
     ⇒ ans =
        {
          field1 =

        {
          [1,1] =  1
          [1,2] = one
        }

          field2 =  2
        }
```

s = `struct` () [Built-in Function]
s = `struct` (*field1*, *value1*, *field2*, *value2*, ...) [Built-in Function]
s = `struct` (*obj*) [Built-in Function]

Create a scalar or array structure and initialize its values.

The *field1*, *field2*, ... variables are strings specifying the names of the fields and the *value1*, *value2*, ... variables can be of any type.

If the values are cell arrays, create a structure array and initialize its values. The dimensions of each cell array of values must match. Singleton cells and non-cell values are repeated so that they fill the entire array. If the cells are empty, create an empty structure array with the specified field names.

If the argument is an object, return the underlying struct.

Observe that the syntax is optimized for struct **arrays**. Consider the following examples:

```
struct ("foo", 1)
  ⇒ scalar structure containing the fields:
    foo =  1

struct ("foo", {})
  ⇒ 0x0 struct array containing the fields:
    foo

struct ("foo", { {} })
  ⇒ scalar structure containing the fields:
    foo = {}(0x0)

struct ("foo", {1, 2, 3})
  ⇒ 1x3 struct array containing the fields:
    foo
```

The first case is an ordinary scalar struct—one field, one value. The second produces an empty struct array with one field and no values, since being passed an empty cell array of struct array values. When the value is a cell array containing a single entry, this becomes a scalar struct with that single entry as the value of the field. That single entry happens to be an empty cell array.

Finally, if the value is a non-scalar cell array, then `struct` produces a struct **array**.

See also: [cell2struct], page 120, [fieldnames], page 107, [getfield], page 109, [setfield], page 108, [rmfield], page 109, [isfield], page 108, [orderfields], page 109, [isstruct], page 107, [structfun], page 498.

The function `isstruct` can be used to test if an object is a structure or a structure array.

`isstruct` (*x*) [Built-in Function]

Return true if *x* is a structure or a structure array.

See also: [ismatrix], page 63, [iscell], page 113, [isa], page 39.

6.1.4 Manipulating Structures

Other functions that can manipulate the fields of a structure are given below.

`numfields` (*s*) [Built-in Function]

Return the number of fields of the structure *s*.

See also: [fieldnames], page 107.

names = `fieldnames` (*struct*) [Function File]
names = `fieldnames` (*obj*) [Function File]
names = `fieldnames` (*javaobj*) [Function File]
names = `fieldnames` ("*jclassname*") [Function File]

Return a cell array of strings with the names of the fields in the specified input.

When the input is a structure *struct*, the names are the elements of the structure.

When the input is an Octave object *obj*, the names are the public properties of the object.

When the input is a Java object *javaobj* or Java classname *jclassname* the name are the public data elements of the object or class.

See also: [numfields], page 107, [isfield], page 108, [orderfields], page 109, [struct], page 106, [methods], page 722.

`isfield (x, "name")` [Built-in Function]
`isfield (x, name)` [Built-in Function]

Return true if the *x* is a structure and it includes an element named *name*.

If *name* is a cell array of strings then a logical array of equal dimension is returned.

See also: [fieldnames], page 107.

`sout = setfield (s, field, val)` [Function File]
`sout = setfield (s, sidx1, field1, fidx1, sidx2, field2, fidx2, ..., val)` [Function File]

Return a *copy* of the structure *s* with the field member *field* set to the value *val*.

For example:

```
s = struct ();
s = setfield (s, "foo bar", 42);
```

This is equivalent to

```
s.("foo bar") = 42;
```

Note that ordinary structure syntax `s.foo bar = 42` cannot be used here, as the field name is not a valid Octave identifier because of the space character. Using arbitrary strings for field names is incompatible with MATLAB, and this usage will emit a warning if the warning ID `Octave:language-extension` is enabled. See [XREFwarning_ids], page 214.

With the second calling form, set a field of a structure array. The input *sidx* selects an element of the structure array, *field* specifies the field name of the selected element, and *fidx* selects which element of the field (in the case of an array or cell array). The *sidx*, *field*, and *fidx* inputs can be repeated to address nested structure array elements. The structure array index and field element index must be cell arrays while the field name must be a string.

For example:

```
s = struct ("baz", 42);
setfield (s, {1}, "foo", {1}, "bar", 54)
⇒
  ans =
    scalar structure containing the fields:
      baz =  42
      foo =
        scalar structure containing the fields:
          bar =  54
```

The example begins with an ordinary scalar structure to which a nested scalar structure is added. In all cases, if the structure index *sidx* is not specified it defaults to

1 (scalar structure). Thus, the example above could be written more concisely as `setfield (s, "foo", "bar", 54)`

Finally, an example with nested structure arrays:

```
sa.foo = 1;
sa = setfield (sa, {2}, "bar", {3}, "baz", {1, 4}, 5);
sa(2).bar(3)
⇒
  ans =
    scalar structure containing the fields:
      baz =  0   0   0   5
```

Here *sa* is a structure array whose field at elements 1 and 2 is in turn another structure array whose third element is a simple scalar structure. The terminal scalar structure has a field which contains a matrix value.

Note that the same result as in the above example could be achieved by:

```
sa.foo = 1;
sa(2).bar(3).baz(1,4) = 5
```

See also: [getfield], page 109, [rmfield], page 109, [orderfields], page 109, [isfield], page 108, [fieldnames], page 107, [isstruct], page 107, [struct], page 106.

val = **getfield** (*s*, *field*) [Function File]
val = **getfield** (*s*, *sidx1*, *field1*, *fidx1*, ...) [Function File]

Get the value of the field named *field* from a structure or nested structure *s*.

If *s* is a structure array then *sidx* selects an element of the structure array, *field* specifies the field name of the selected element, and *fidx* selects which element of the field (in the case of an array or cell array). See `setfield` for a more complete description of the syntax.

See also: [setfield], page 108, [rmfield], page 109, [orderfields], page 109, [isfield], page 108, [fieldnames], page 107, [isstruct], page 107, [struct], page 106.

sout = **rmfield** (*s*, "*f*") [Built-in Function]
sout = **rmfield** (*s*, *f*) [Built-in Function]

Return a *copy* of the structure (array) *s* with the field *f* removed.

If *f* is a cell array of strings or a character array, remove each of the named fields.

See also: [orderfields], page 109, [fieldnames], page 107, [isfield], page 108.

sout = **orderfields** (*s1*) [Function File]
sout = **orderfields** (*s1*, *s2*) [Function File]
sout = **orderfields** (*s1*, {*cellstr*}) [Function File]
sout = **orderfields** (*s1*, *p*) [Function File]
[*sout*, *p*] = **orderfields** (...) [Function File]

Return a *copy* of *s1* with fields arranged alphabetically, or as specified by the second input.

Given one input struct *s1*, arrange field names alphabetically.

If a second struct argument is given, arrange field names in *s1* as they appear in *s2*. The second argument may also specify the order in a cell array of strings *cellstr*. The second argument may also be a permutation vector.

The optional second output argument *p* is the permutation vector which converts the original name order to the new name order.

Examples:

```
s = struct ("d", 4, "b", 2, "a", 1, "c", 3);
t1 = orderfields (s)
     ⇒ t1 =
        {
          a =  1
          b =  2
          c =  3
          d =  4
        }
t = struct ("d", {}, "c", {}, "b", {}, "a", {});
t2 = orderfields (s, t)
     ⇒ t2 =
        {
          d =  4
          c =  3
          b =  2
          a =  1
        }
t3 = orderfields (s, [3, 2, 4, 1])
     ⇒ t3 =
        {
          a =  1
          b =  2
          c =  3
          d =  4
        }
[t4, p] = orderfields (s, {"d", "c", "b", "a"})
     ⇒ t4 =
        {
          d =  4
          c =  3
          b =  2
          a =  1
        }
        p =
           1
           4
           2
           3
```

See also: [fieldnames], page 107, [getfield], page 109, [setfield], page 108, [rmfield], page 109, [isfield], page 108, [isstruct], page 107, [struct], page 106.

substruct (*type*, *subs*, . . .) [Function File]

Create a subscript structure for use with `subsref` or `subsasgn`.

For example:

```
idx = substruct ("()", {3, ":"})
     ⇒
       idx =
       {
         type = ()
         subs =
         {
           [1,1] =  3
           [1,2] = :
         }
       }
x = [1, 2, 3;
     4, 5, 6;
     7, 8, 9];
subsref (x, idx)
   ⇒ 7  8  9
```

See also: [subsref], page 726, [subsasgn], page 728.

6.1.5 Processing Data in Structures

The simplest way to process data in a structure is within a `for` loop (see Section 10.5.1 [Looping Over Structure Elements], page 165). A similar effect can be achieved with the `structfun` function, where a user defined function is applied to each field of the structure. See [structfun], page 498.

Alternatively, to process the data in a structure, the structure might be converted to another type of container before being treated.

c = struct2cell (*s*) [Built-in Function]

Create a new cell array from the objects stored in the struct object.

If *f* is the number of fields in the structure, the resulting cell array will have a dimension vector corresponding to `[f size(s)]`. For example:

```
s = struct ("name", {"Peter", "Hannah", "Robert"},
           "age", {23, 16, 3});
c = struct2cell (s)
   ⇒ c = {2x1x3 Cell Array}
c(1,1,:)(:)
   ⇒
      {
        [1,1] = Peter
        [2,1] = Hannah
        [3,1] = Robert
      }
c(2,1,:)(:)
   ⇒
      {
        [1,1] = 23
        [2,1] = 16
        [3,1] = 3
      }
```

See also: [cell2struct], page 120, [fieldnames], page 107.

6.2 Cell Arrays

It can be both necessary and convenient to store several variables of different size or type in one variable. A cell array is a container class able to do just that. In general cell arrays work just like N-dimensional arrays with the exception of the use of '`{`' and '`}`' as allocation and indexing operators.

6.2.1 Basic Usage of Cell Arrays

As an example, the following code creates a cell array containing a string and a 2-by-2 random matrix

```
c = {"a string", rand(2, 2)};
```

To access the elements of a cell array, it can be indexed with the { and } operators. Thus, the variable created in the previous example can be indexed like this:

```
c{1}
     ⇒ ans = a string
```

As with numerical arrays several elements of a cell array can be extracted by indexing with a vector of indexes

```
c{1:2}
     ⇒ ans = a string
     ⇒ ans =

                 0.593993   0.627732
                 0.377037   0.033643
```

The indexing operators can also be used to insert or overwrite elements of a cell array. The following code inserts the scalar 3 on the third place of the previously created cell array

```
c{3} = 3
     ⇒ c =

         {
           [1,1] = a string
           [1,2] =

              0.593993   0.627732
              0.377037   0.033643

           [1,3] =  3
         }
```

Details on indexing cell arrays are explained in Section 6.2.3 [Indexing Cell Arrays], page 116.

In general nested cell arrays are displayed hierarchically as in the previous example. In some circumstances it makes sense to reference them by their index, and this can be performed by the `celldisp` function.

`celldisp` (*c*) [Function File]
`celldisp` (*c*, *name*) [Function File]

Recursively display the contents of a cell array.

By default the values are displayed with the name of the variable *c*. However, this name can be replaced with the variable *name*. For example:

```
c = {1, 2, {31, 32}};
celldisp (c, "b")
   ⇒
      b{1} =
       1
      b{2} =
       2
      b{3}{1} =
       31
      b{3}{2} =
       32
```

See also: [disp], page 231.

To test if an object is a cell array, use the `iscell` function. For example:

```
iscell (c)
     ⇒ ans = 1

iscell (3)
     ⇒ ans = 0
```

`iscell` (*x*) [Built-in Function]

Return true if *x* is a cell array object.

See also: [ismatrix], page 63, [isstruct], page 107, [iscellstr], page 119, [isa], page 39.

6.2.2 Creating Cell Arrays

The introductory example (see Section 6.2.1 [Basic Usage of Cell Arrays], page 112) showed how to create a cell array containing currently available variables. In many situations, however, it is useful to create a cell array and then fill it with data.

The `cell` function returns a cell array of a given size, containing empty matrices. This function is similar to the `zeros` function for creating new numerical arrays. The following example creates a 2-by-2 cell array containing empty matrices

```
c = cell (2,2)
     ⇒ c =

         {
           [1,1] = [](0x0)
           [2,1] = [](0x0)
           [1,2] = [](0x0)
           [2,2] = [](0x0)
         }
```

Just like numerical arrays, cell arrays can be multi-dimensional. The `cell` function accepts any number of positive integers to describe the size of the returned cell array. It is also possible to set the size of the cell array through a vector of positive integers. In the following example two cell arrays of equal size are created, and the size of the first one is displayed

```
c1 = cell (3, 4, 5);
c2 = cell ( [3, 4, 5] );
size (c1)
     ⇒ ans =
         3   4   5
```

As can be seen, the [size], page 45, function also works for cell arrays. As do other functions describing the size of an object, such as [length], page 45, [numel], page 44, [rows], page 44, and [columns], page 44.

`cell` (*n*) [Built-in Function]
`cell` (*m*, *n*) [Built-in Function]
`cell` (*m*, *n*, *k*, . . .) [Built-in Function]
`cell` ([*m* *n* . . .]) [Built-in Function]

Create a new cell array object.

If invoked with a single scalar integer argument, return a square NxN cell array. If invoked with two or more scalar integer arguments, or a vector of integer values, return an array with the given dimensions.

See also: [cellstr], page 119, [mat2cell], page 115, [num2cell], page 114, [struct2cell], page 111.

As an alternative to creating empty cell arrays, and then filling them, it is possible to convert numerical arrays into cell arrays using the `num2cell`, `mat2cell` and `cellslices` functions.

C = `num2cell` (*A*) [Built-in Function]
C = `num2cell` (*A*, *dim*) [Built-in Function]

Convert the numeric matrix A to a cell array.

If *dim* is defined, the value C is of dimension 1 in this dimension and the elements of A are placed into C in slices. For example:

```
num2cell ([1,2;3,4])
   ⇒
      {
        [1,1] =  1
        [2,1] =  3
        [1,2] =  2
        [2,2] =  4
      }
num2cell ([1,2;3,4],1)
   ⇒
      {
        [1,1] =
           1
           3
        [1,2] =
           2
           4
      }
```

See also: [mat2cell], page 115.

C = `mat2cell` (*A*, *m*, *n*) [Built-in Function]
C = `mat2cell` (*A*, *d1*, *d2*, ...) [Built-in Function]
C = `mat2cell` (*A*, *r*) [Built-in Function]

Convert the matrix A to a cell array.

If A is 2-D, then it is required that `sum (m) == size (A, 1)` and `sum (n) == size (A, 2)`. Similarly, if A is multi-dimensional and the number of dimensional arguments is equal to the dimensions of A, then it is required that `sum (di) == size (A, i)`.

Given a single dimensional argument r, the other dimensional arguments are assumed to equal `size (A,i)`.

An example of the use of mat2cell is

```
mat2cell (reshape (1:16,4,4), [3,1], [3,1])
⇒
{
   [1,1] =

      1   5   9
      2   6  10
      3   7  11

   [2,1] =
```

```
          4    8   12

      [1,2] =

        13
        14
        15

      [2,2] = 16
    }
```

See also: [num2cell], page 114, [cell2mat], page 119.

sl = `cellslices` (*x*, *lb*, *ub*, *dim*) [Built-in Function]

Given an array *x*, this function produces a cell array of slices from the array determined by the index vectors *lb*, *ub*, for lower and upper bounds, respectively.

In other words, it is equivalent to the following code:

```
n = length (lb);
sl = cell (1, n);
for i = 1:length (lb)
  sl{i} = x(:,...,lb(i):ub(i),...,:);
endfor
```

The position of the index is determined by *dim*. If not specified, slicing is done along the first non-singleton dimension.

See also: [cell2mat], page 119, [cellindexmat], page 118, [cellfun], page 496.

6.2.3 Indexing Cell Arrays

As shown in see Section 6.2.1 [Basic Usage of Cell Arrays], page 112, elements can be extracted from cell arrays using the '`{`' and '`}`' operators. If you want to extract or access subarrays which are still cell arrays, you need to use the '`(`' and '`)`' operators. The following example illustrates the difference:

```
c = {"1", "2", "3"; "x", "y", "z"; "4", "5", "6"};
c{2,3}
     ⇒ ans = z

c(2,3)
     ⇒ ans =
        {
          [1,1] = z
        }
```

So with '`{}`' you access elements of a cell array, while with '`()`' you access a sub array of a cell array.

Using the '`(`' and '`)`' operators, indexing works for cell arrays like for multi-dimensional arrays. As an example, all the rows of the first and third column of a cell array can be set to `0` with the following command:

```
c(:, [1, 3]) = {0}
     ⇒  =
        {
          [1,1] = 0
          [2,1] = 0
          [3,1] = 0
          [1,2] = 2
          [2,2] =  10
          [3,2] =  20
          [1,3] = 0
          [2,3] = 0
          [3,3] = 0
        }
```

Note, that the above can also be achieved like this:

```
c(:, [1, 3]) = 0;
```

Here, the scalar '0' is automatically promoted to cell array '{0}' and then assigned to the subarray of c.

To give another example for indexing cell arrays with '()', you can exchange the first and the second row of a cell array as in the following command:

```
c = {1, 2, 3; 4, 5, 6};
c([1, 2], :) = c([2, 1], :)
     ⇒ =
        {
          [1,1] =  4
          [2,1] =  1
          [1,2] =  5
          [2,2] =  2
          [1,3] =  6
          [2,3] =  3
        }
```

Accessing multiple elements of a cell array with the '{' and '}' operators will result in a comma-separated list of all the requested elements (see Section 6.3 [Comma Separated Lists], page 120). Using the '{' and '}' operators the first two rows in the above example can be swapped back like this:

```
[c{[1,2], :}] = deal (c{[2, 1], :})
     ⇒ =
        {
          [1,1] =  1
          [2,1] =  4
          [1,2] =  2
          [2,2] =  5
          [1,3] =  3
          [2,3] =  6
        }
```

As for struct arrays and numerical arrays, the empty matrix '`[]`' can be used to delete elements from a cell array:

```
x = {"1", "2"; "3", "4"};
x(1, :) = []
     ⇒ x =
        {
          [1,1] = 3
          [1,2] = 4
        }
```

The following example shows how to just remove the contents of cell array elements but not delete the space for them:

```
x = {"1", "2"; "3", "4"};
x(1, :) = {[]}
⇒ x =
      {
        [1,1] = [](0x0)
        [2,1] = 3
        [1,2] = [](0x0)
        [2,2] = 4
      }
```

The indexing operations operate on the cell array and not on the objects within the cell array. By contrast, `cellindexmat` applies matrix indexing to the objects within each cell array entry and returns the requested values.

y = `cellindexmat` (*x*, *varargin*) [Built-in Function]

Perform indexing of matrices in a cell array.

Given a cell array of matrices *x*, this function computes

```
Y = cell (size (X));
for i = 1:numel (X)
  Y{i} = X{i}(varargin{:});
endfor
```

See also: [cellslices], page 116, [cellfun], page 496.

6.2.4 Cell Arrays of Strings

One common use of cell arrays is to store multiple strings in the same variable. It is also possible to store multiple strings in a character matrix by letting each row be a string. This, however, introduces the problem that all strings must be of equal length. Therefore, it is recommended to use cell arrays to store multiple strings. For cases, where the character matrix representation is required for an operation, there are several functions that convert a cell array of strings to a character array and back. `char` and `strvcat` convert cell arrays to a character array (see Section 5.3.1 [Concatenating Strings], page 70), while the function `cellstr` converts a character array to a cell array of strings:

```
a = ["hello"; "world"];
c = cellstr (a)
     ⇒ c =
         {
           [1,1] = hello
           [2,1] = world
         }
```

cstr = cellstr (*strmat*) [Built-in Function]

Create a new cell array object from the elements of the string array *strmat*.

Each row of *strmat* becomes an element of *cstr*. Any trailing spaces in a row are deleted before conversion.

To convert back from a cellstr to a character array use `char`.

See also: [cell], page 114, [char], page 71.

One further advantage of using cell arrays to store multiple strings is that most functions for string manipulations included with Octave support this representation. As an example, it is possible to compare one string with many others using the `strcmp` function. If one of the arguments to this function is a string and the other is a cell array of strings, each element of the cell array will be compared to the string argument:

```
c = {"hello", "world"};
strcmp ("hello", c)
     ⇒ ans =
        1   0
```

The following string functions support cell arrays of strings: `char`, `strvcat`, `strcat` (see Section 5.3.1 [Concatenating Strings], page 70), `strcmp`, `strncmp`, `strcmpi`, `strncmpi` (see Section 5.4 [Comparing Strings], page 76), `str2double`, `deblank`, `strtrim`, `strtrunc`, `strfind`, `strmatch`, , `regexp`, `regexpi` (see Section 5.5 [Manipulating Strings], page 77) and `str2double` (see Section 5.6 [String Conversions], page 91).

The function `iscellstr` can be used to test if an object is a cell array of strings.

iscellstr (*cell*) [Built-in Function]

Return true if every element of the cell array *cell* is a character string.

See also: [ischar], page 68.

6.2.5 Processing Data in Cell Arrays

Data that is stored in a cell array can be processed in several ways depending on the actual data. The simplest way to process that data is to iterate through it using one or more `for` loops. The same idea can be implemented more easily through the use of the `cellfun` function that calls a user-specified function on all elements of a cell array. See [cellfun], page 496.

An alternative is to convert the data to a different container, such as a matrix or a data structure. Depending on the data this is possible using the `cell2mat` and `cell2struct` functions.

`m = cell2mat (c)` [Function File]

Convert the cell array *c* into a matrix by concatenating all elements of *c* into a hyperrectangle.

Elements of *c* must be numeric, logical, or char matrices; or cell arrays; or structs; and `cat` must be able to concatenate them together.

See also: [mat2cell], page 115, [num2cell], page 114.

`cell2struct (cell, fields)` [Built-in Function]
`cell2struct (cell, fields, dim)` [Built-in Function]

Convert *cell* to a structure.

The number of fields in *fields* must match the number of elements in *cell* along dimension *dim*, that is `numel (fields) == size (cell, dim)`. If *dim* is omitted, a value of 1 is assumed.

```
A = cell2struct ({"Peter", "Hannah", "Robert";
                   185, 170, 168},
                 {"Name","Height"}, 1);
A(1)
   ⇒
      {
        Name   = Peter
        Height = 185
      }
```

See also: [struct2cell], page 111, [cell2mat], page 119, [struct], page 106.

6.3 Comma Separated Lists

Comma separated lists[1] are the basic argument type to all Octave functions - both for input and return arguments. In the example

```
max (a, b)
```

'`a, b`' is a comma separated list. Comma separated lists can appear on both the right and left hand side of an assignment. For example

```
x = [1 0 1 0 0 1 1; 0 0 0 0 0 0 7];
[i, j] = find (x, 2, "last");
```

Here, '`x, 2, "last"`' is a comma separated list constituting the input arguments of `find`. `find` returns a comma separated list of output arguments which is assigned element by element to the comma separated list '`i, j`'.

Another example of where comma separated lists are used is in the creation of a new array with `[]` (see Section 4.1 [Matrices], page 48) or the creation of a cell array with `{}` (see Section 6.2.1 [Basic Usage of Cell Arrays], page 112). In the expressions

```
a = [1, 2, 3, 4];
c = {4, 5, 6, 7};
```

both '`1, 2, 3, 4`' and '`4, 5, 6, 7`' are comma separated lists.

[1] Comma-separated lists are also sometimes informally referred to as *cs-lists*.

Comma separated lists cannot be directly manipulated by the user. However, both structure arrays and cell arrays can be converted into comma separated lists, and thus used in place of explicitly written comma separated lists. This feature is useful in many ways, as will be shown in the following subsections.

6.3.1 Comma Separated Lists Generated from Cell Arrays

As has been mentioned above (see Section 6.2.3 [Indexing Cell Arrays], page 116), elements of a cell array can be extracted into a comma separated list with the `{` and `}` operators. By surrounding this list with `[` and `]`, it can be concatenated into an array. For example:

```
a = {1, [2, 3], 4, 5, 6};
b = [a{1:4}]
     ⇒ b =
          1   2   3   4   5
```

Similarly, it is possible to create a new cell array containing cell elements selected with `{}`. By surrounding the list with '`{`' and '`}`' a new cell array will be created, as the following example illustrates:

```
a = {1, rand(2, 2), "three"};
b = { a{ [1, 3] } }
     ⇒ b =
          {
            [1,1] =  1
            [1,2] = three
          }
```

Furthermore, cell elements (accessed by `{}`) can be passed directly to a function. The list of elements from the cell array will be passed as an argument list to a given function as if it is called with the elements as individual arguments. The two calls to `printf` in the following example are identical but the latter is simpler and can handle cell arrays of an arbitrary size:

```
c = {"GNU", "Octave", "is", "Free", "Software"};
printf ("%s ", c{1}, c{2}, c{3}, c{4}, c{5});
     ⊣ GNU Octave is Free Software
printf ("%s ", c{:});
     ⊣ GNU Octave is Free Software
```

If used on the left-hand side of an assignment, a comma separated list generated with `{}` can be assigned to. An example is

```
in{1} = [10, 20, 30, 40, 50, 60, 70, 80, 90];
in{2} = inf;
in{3} = "last";
in{4} = "first";
out = cell (4, 1);
[out{1:3}] = find (in{1 : 3});
[out{4:6}] = find (in{[1, 2, 4]})
     ⇒ out =
        {
          [1,1] = 1
          [2,1] = 9
          [3,1] = 90
          [4,1] = 1
          [3,1] = 1
          [4,1] = 10
        }
```

6.3.2 Comma Separated Lists Generated from Structure Arrays

Structure arrays can equally be used to create comma separated lists. This is done by addressing one of the fields of a structure array. For example:

```
x = ceil (randn (10, 1));
in = struct ("call1", {x, 3, "last"},
             "call2", {x, inf, "first"});
out = struct ("call1", cell (2, 1), "call2", cell (2, 1));
[out.call1] = find (in.call1);
[out.call2] = find (in.call2);
```

7 Variables

Variables let you give names to values and refer to them later. You have already seen variables in many of the examples. The name of a variable must be a sequence of letters, digits and underscores, but it may not begin with a digit. Octave does not enforce a limit on the length of variable names, but it is seldom useful to have variables with names longer than about 30 characters. The following are all valid variable names

```
x
x15
__foo_bar_baz__
fucnrdthsucngtagdjb
```

However, names like `__foo_bar_baz__` that begin and end with two underscores are understood to be reserved for internal use by Octave. You should not use them in code you write, except to access Octave's documented internal variables and built-in symbolic constants.

Case is significant in variable names. The symbols `a` and `A` are distinct variables.

A variable name is a valid expression by itself. It represents the variable's current value. Variables are given new values with *assignment operators* and *increment operators*. See Section 8.6 [Assignment Expressions], page 149.

There is one built-in variable with a special meaning. The `ans` variable always contains the result of the last computation, where the output wasn't assigned to any variable. The code `a = cos (pi)` will assign the value -1 to the variable `a`, but will not change the value of `ans`. However, the code `cos (pi)` will set the value of `ans` to -1.

Variables in Octave do not have fixed types, so it is possible to first store a numeric value in a variable and then to later use the same name to hold a string value in the same program. Variables may not be used before they have been given a value. Doing so results in an error.

`ans` [Automatic Variable]

The most recently computed result that was not explicitly assigned to a variable.

For example, after the expression

```
3^2 + 4^2
```

is evaluated, the value returned by `ans` is 25.

`isvarname (`*name*`)` [Built-in Function]

Return true if *name* is a valid variable name.

See also: [iskeyword], page 915, [exist], page 130, [who], page 128.

varname `= genvarname (`*str*`)` [Function File]
varname `= genvarname (`*str*`, `*exclusions*`)` [Function File]

Create valid unique variable name(s) from *str*.

If *str* is a cellstr, then a unique variable is created for each cell in *str*.

```
genvarname ({"foo", "foo"})
  ⇒
     {
       [1,1] = foo
       [1,2] = foo1
     }
```

If *exclusions* is given, then the variable(s) will be unique to each other and to *exclusions* (*exclusions* may be either a string or a cellstr).

```
x = 3.141;
genvarname ("x", who ())
  ⇒ x1
```

Note that the result is a char array or cell array of strings, not the variables themselves. To define a variable, `eval()` can be used. The following trivial example sets `x` to `42`.

```
name = genvarname ("x");
eval ([name " = 42"]);
  ⇒ x =  42
```

This can be useful for creating unique struct field names.

```
x = struct ();
for i = 1:3
  x.(genvarname ("a", fieldnames (x))) = i;
endfor
  ⇒ x =
     {
       a =  1
       a1 =  2
       a2 =  3
     }
```

Since variable names may only contain letters, digits, and underscores, `genvarname` will replace any sequence of disallowed characters with an underscore. Also, variables may not begin with a digit; in this case an '`x`' is added before the variable name.

Variable names beginning and ending with two underscores `"__"` are valid, but they are used internally by Octave and should generally be avoided; therefore, `genvarname` will not generate such names.

`genvarname` will also ensure that returned names do not clash with keywords such as `"for"` and `"if"`. A number will be appended if necessary. Note, however, that this does **not** include function names such as `"sin"`. Such names should be included in *exclusions* if necessary.

See also: [isvarname], page 123, [iskeyword], page 915, [exist], page 130, [who], page 128, [tempname], page 266, [eval], page 155.

`namelengthmax ()` [Function File]

Return the MATLAB compatible maximum variable name length.

Octave is capable of storing strings up to $2^{31} - 1$ in length. However for MATLAB compatibility all variable, function, and structure field names should be shorter than the length returned by `namelengthmax`. In particular, variables stored to a MATLAB file format (`*.mat`) will have their names truncated to this length.

7.1 Global Variables

A variable that has been declared *global* may be accessed from within a function body without having to pass it as a formal parameter.

A variable may be declared global using a `global` declaration statement. The following statements are all global declarations.

```
global a
global a b
global c = 2
global d = 3 e f = 5
```

A global variable may only be initialized once in a `global` statement. For example, after executing the following code

```
global gvar = 1
global gvar = 2
```

the value of the global variable `gvar` is 1, not 2. Issuing a '`clear gvar`' command does not change the above behavior, but '`clear all`' does.

It is necessary declare a variable as global within a function body in order to access it. For example,

```
global x
function f ()
  x = 1;
endfunction
f ()
```

does *not* set the value of the global variable `x` to 1. In order to change the value of the global variable `x`, you must also declare it to be global within the function body, like this

```
function f ()
  global x;
  x = 1;
endfunction
```

Passing a global variable in a function parameter list will make a local copy and not modify the global value. For example, given the function

```
function f (x)
  x = 0
endfunction
```

and the definition of `x` as a global variable at the top level,

```
global x = 13
```

the expression

```
f (x)
```

will display the value of `x` from inside the function as 0, but the value of `x` at the top level remains unchanged, because the function works with a *copy* of its argument.

`isglobal (`*`name`*`)` [Built-in Function]

Return true if *name* is a globally visible variable.

For example:

```
global x
isglobal ("x")
   ⇒ 1
```

See also: [isvarname], page 123, [exist], page 130.

7.2 Persistent Variables

A variable that has been declared *persistent* within a function will retain its contents in memory between subsequent calls to the same function. The difference between persistent variables and global variables is that persistent variables are local in scope to a particular function and are not visible elsewhere.

The following example uses a persistent variable to create a function that prints the number of times it has been called.

```
function count_calls ()
  persistent calls = 0;
  printf ("'count_calls' has been called %d times\n",
          ++calls);
endfunction

for i = 1:3
  count_calls ();
endfor

⊣ 'count_calls' has been called 1 times
⊣ 'count_calls' has been called 2 times
⊣ 'count_calls' has been called 3 times
```

As the example shows, a variable may be declared persistent using a `persistent` declaration statement. The following statements are all persistent declarations.

```
persistent a
persistent a b
persistent c = 2
persistent d = 3 e f = 5
```

The behavior of persistent variables is equivalent to the behavior of static variables in C.

Like global variables, a persistent variable may only be initialized once. For example, after executing the following code

```
persistent pvar = 1
persistent pvar = 2
```

the value of the persistent variable `pvar` is 1, not 2.

If a persistent variable is declared but not initialized to a specific value, it will contain an empty matrix. So, it is also possible to initialize a persistent variable by checking whether it is empty, as the following example illustrates.

```
function count_calls ()
  persistent calls;
  if (isempty (calls))
    calls = 0;
  endif
  printf ("'count_calls' has been called %d times\n",
          ++calls);
endfunction
```

This implementation behaves in exactly the same way as the previous implementation of `count_calls`.

The value of a persistent variable is kept in memory until it is explicitly cleared. Assuming that the implementation of `count_calls` is saved on disk, we get the following behavior.

```
for i = 1:2
  count_calls ();
endfor
⊣ 'count_calls' has been called 1 times
⊣ 'count_calls' has been called 2 times

clear
for i = 1:2
  count_calls ();
endfor
⊣ 'count_calls' has been called 3 times
⊣ 'count_calls' has been called 4 times

clear all
for i = 1:2
  count_calls ();
endfor
⊣ 'count_calls' has been called 1 times
⊣ 'count_calls' has been called 2 times

clear count_calls
for i = 1:2
  count_calls ();
endfor
⊣ 'count_calls' has been called 1 times
⊣ 'count_calls' has been called 2 times
```

That is, the persistent variable is only removed from memory when the function containing the variable is removed. Note that if the function definition is typed directly into the Octave prompt, the persistent variable will be cleared by a simple `clear` command as the entire function definition will be removed from memory. If you do not want a persistent variable to be removed from memory even if the function is cleared, you should use the `mlock` function (see Section 11.9.6 [Function Locking], page 196).

7.3 Status of Variables

When creating simple one-shot programs it can be very convenient to see which variables are available at the prompt. The function `who` and its siblings `whos` and `whos_line_format` will show different information about what is in memory, as the following shows.

```
str = "A random string";
who -variables
     ⊣ *** local user variables:
     ⊣
     ⊣ __nargin__  str
```

`who` [Command]
`who` *pattern ...* [Command]
`who` *option pattern ...* [Command]
`C = who ("`*pattern*`", ...)` [Command]

List currently defined variables matching the given patterns.

Valid pattern syntax is the same as described for the `clear` command. If no patterns are supplied, all variables are listed.

By default, only variables visible in the local scope are displayed.

The following are valid options, but may not be combined.

`global` List variables in the global scope rather than the current scope.

`-regexp` The patterns are considered to be regular expressions when matching the variables to display. The same pattern syntax accepted by the `regexp` function is used.

`-file` The next argument is treated as a filename. All variables found within the specified file are listed. No patterns are accepted when reading variables from a file.

If called as a function, return a cell array of defined variable names matching the given patterns.

See also: [whos], page 128, [isglobal], page 125, [isvarname], page 123, [exist], page 130, [regexp], page 87.

`whos` [Command]
`whos` *pattern ...* [Command]
`whos` *option pattern ...* [Command]
`S = whos ("`*pattern*`", ...)` [Built-in Function]

Provide detailed information on currently defined variables matching the given patterns.

Options and pattern syntax are the same as for the `who` command.

Extended information about each variable is summarized in a table with the following default entries.

Attr Attributes of the listed variable. Possible attributes are:

blank Variable in local scope

`a` Automatic variable. An automatic variable is one created by the interpreter, for example `argn`.

`c` Variable of complex type.

`f` Formal parameter (function argument).

`g` Variable with global scope.

`p` Persistent variable.

Name The name of the variable.

Size The logical size of the variable. A scalar is 1x1, a vector is 1xN or Nx1, a 2-D matrix is MxN.

Bytes The amount of memory currently used to store the variable.

Class The class of the variable. Examples include double, single, char, uint16, cell, and struct.

The table can be customized to display more or less information through the function `whos_line_format`.

If `whos` is called as a function, return a struct array of defined variable names matching the given patterns. Fields in the structure describing each variable are: name, size, bytes, class, global, sparse, complex, nesting, persistent.

See also: [who], page 128, [whos_line_format], page 129.

val = whos_line_format () [Built-in Function]
old_val = whos_line_format (*new_val*) [Built-in Function]
whos_line_format (*new_val*, "*local*") [Built-in Function]

Query or set the format string used by the command `whos`.

A full format string is:

```
%[modifier]<command>[:width[:left-min[:balance]]];
```

The following command sequences are available:

`%a` Prints attributes of variables (g=global, p=persistent, f=formal parameter, a=automatic variable).

`%b` Prints number of bytes occupied by variables.

`%c` Prints class names of variables.

`%e` Prints elements held by variables.

`%n` Prints variable names.

`%s` Prints dimensions of variables.

`%t` Prints type names of variables.

Every command may also have an alignment modifier:

`l` Left alignment.

`r` Right alignment (default).

`c` Column-aligned (only applicable to command %s).

The `width` parameter is a positive integer specifying the minimum number of columns used for printing. No maximum is needed as the field will auto-expand as required.

The parameters `left-min` and `balance` are only available when the column-aligned modifier is used with the command '`%s`'. `balance` specifies the column number within the field width which will be aligned between entries. Numbering starts from 0 which indicates the leftmost column. `left-min` specifies the minimum field width to the left of the specified balance column.

The default format is:

`" %a:4; %ln:6; %cs:16:6:1; %rb:12; %lc:-1;\n"`

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [whos], page 128.

Instead of displaying which variables are in memory, it is possible to determine if a given variable is available. That way it is possible to alter the behavior of a program depending on the existence of a variable. The following example illustrates this.

```
if (! exist ("meaning", "var"))
  disp ("The program has no 'meaning'");
endif
```

c = `exist` (*name*) [Built-in Function]
c = `exist` (*name*, *type*) [Built-in Function]

Check for the existence of *name* as a variable, function, file, directory, or class.

The return code *c* is one of

1 *name* is a variable.

2 *name* is an absolute file name, an ordinary file in Octave's `path`, or (after appending '`.m`') a function file in Octave's `path`.

3 *name* is a '`.oct`' or '`.mex`' file in Octave's `path`.

5 *name* is a built-in function.

7 *name* is a directory.

103 *name* is a function not associated with a file (entered on the command line).

0 *name* does not exist.

If the optional argument *type* is supplied, check only for symbols of the specified type. Valid types are

`"var"` Check only for variables.

`"builtin"`
Check only for built-in functions.

`"dir"` Check only for directories.

`"file"` Check only for files and directories.

`"class"` Check only for classes. (Note: This option is accepted, but not currently implemented)

If no type is given, and there are multiple possible matches for name, `exist` will return a code according to the following priority list: variable, built-in function, oct-file, directory, file, class.

`exist` returns 2 if a regular file called *name* is present in Octave's search path. If you want information about other types of files not on the search path you should use some combination of the functions `file_in_path` and `stat` instead.

See also: [file_in_loadpath], page 191, [file_in_path], page 766, [dir_in_loadpath], page 192, [stat], page 763.

Usually Octave will manage the memory, but sometimes it can be practical to remove variables from memory manually. This is usually needed when working with large variables that fill a substantial part of the memory. On a computer that uses the IEEE floating point format, the following program allocates a matrix that requires around 128 MB memory.

```
large_matrix = zeros (4000, 4000);
```

Since having this variable in memory might slow down other computations, it can be necessary to remove it manually from memory. The `clear` function allows this.

`clear` [*options*] *pattern* . . . [Command]

Delete the names matching the given patterns from the symbol table.

The pattern may contain the following special characters:

`?` Match any single character.

`*` Match zero or more characters.

`[ list ]` Match the list of characters specified by *list*. If the first character is `!` or `^`, match all characters except those specified by *list*. For example, the pattern '`[a-zA-Z]`' will match all lowercase and uppercase alphabetic characters.

For example, the command

```
clear foo b*r
```

clears the name `foo` and all names that begin with the letter `b` and end with the letter `r`.

If `clear` is called without any arguments, all user-defined variables (local and global) are cleared from the symbol table.

If `clear` is called with at least one argument, only the visible names matching the arguments are cleared. For example, suppose you have defined a function `foo`, and then hidden it by performing the assignment `foo = 2`. Executing the command *`clear foo`* once will clear the variable definition and restore the definition of `foo` as a function. Executing *`clear foo`* a second time will clear the function definition.

The following options are available in both long and short form

`-all, -a` Clear all local and global user-defined variables and all functions from the symbol table.

`-exclusive, -x`
Clear the variables that don't match the following pattern.

`-functions, -f`
: Clear the function names and the built-in symbols names.

`-global, -g`
: Clear global symbol names.

`-variables, -v`
: Clear local variable names.

`-classes, -c`
: Clears the class structure table and clears all objects.

`-regexp, -r`
: The arguments are treated as regular expressions as any variables that match will be cleared.

With the exception of `exclusive`, all long options can be used without the dash as well.

See also: [who], page 128, [whos], page 128, [exist], page 130.

`pack ()` [Function File]

Consolidate workspace memory in MATLAB.

This function is provided for compatibility, but does nothing in Octave.

See also: [clear], page 131.

Information about a function or variable such as its location in the file system can also be acquired from within Octave. This is usually only useful during development of programs, and not within a program.

`type` *name* ... [Command]
`type` *-q name* ... [Command]
`text = type ("`*name*`", ...)` [Function File]

Display the contents of *name* which may be a file, function (m-file), variable, operator, or keyword.

`type` normally prepends a header line describing the category of *name* such as function or variable; The `-q` option suppresses this behavior.

If no output variable is used the contents are displayed on screen. Otherwise, a cell array of strings is returned, where each element corresponds to the contents of each requested function.

`which` *name* ... [Command]

Display the type of each *name*.

If *name* is defined from a function file, the full name of the file is also displayed.

See also: [help], page 20, [lookfor], page 21.

`what` [Command]
`what` *dir* [Command]
`w = what (`*dir*`)` [Function File]

List the Octave specific files in directory *dir*.

If *dir* is not specified then the current directory is used.

If a return argument is requested, the files found are returned in the structure *w*. The structure contains the following fields:

path Full path to directory *dir*

m Cell array of m-files

mat Cell array of mat files

mex Cell array of mex files

oct Cell array of oct files

mdl Cell array of mdl files

slx Cell array of slx files

p Cell array of p-files

classes Cell array of class directories (`@classname/`)

packages Cell array of package directories (`+pkgname/`)

Compatibility Note: Octave does not support mdl, slx, and p files; nor does it support package directories. `what` will always return an empty list for these categories.

See also: [which], page 132, [ls], page 785, [exist], page 130.

8 Expressions

Expressions are the basic building block of statements in Octave. An expression evaluates to a value, which you can print, test, store in a variable, pass to a function, or assign a new value to a variable with an assignment operator.

An expression can serve as a statement on its own. Most other kinds of statements contain one or more expressions which specify data to be operated on. As in other languages, expressions in Octave include variables, array references, constants, and function calls, as well as combinations of these with various operators.

8.1 Index Expressions

An *index expression* allows you to reference or extract selected elements of a matrix or vector.

Indices may be scalars, vectors, ranges, or the special operator '`:`', which may be used to select entire rows or columns.

Vectors are indexed using a single index expression. Matrices (2-D) and higher multi-dimensional arrays are indexed using either one index or N indices where N is the dimension of the array. When using a single index expression to index 2-D or higher data the elements of the array are taken in column-first order (like Fortran).

The output from indexing assumes the dimensions of the index expression. For example:

```
a(2)       # result is a scalar
a(1:2)     # result is a row vector
a([1; 2])  # result is a column vector
```

As a special case, when a colon is used as a single index, the output is a column vector containing all the elements of the vector or matrix. For example:

```
a(:)       # result is a column vector
a(:)'      # result is a row vector
```

The above two code idioms are often used in place of `reshape` when a simple vector, rather than an arbitrarily sized array, is needed.

Given the matrix

```
a = [1, 2; 3, 4]
```

all of the following expressions are equivalent and select the first row of the matrix.

```
a(1, [1, 2])  # row 1, columns 1 and 2
a(1, 1:2)     # row 1, columns in range 1-2
a(1, :)       # row 1, all columns
```

In index expressions the keyword `end` automatically refers to the last entry for a particular dimension. This magic index can also be used in ranges and typically eliminates the needs to call `size` or `length` to gather array bounds before indexing. For example:

```
a = [1, 2, 3, 4];

a(1:end/2)        # first half of a => [1, 2]
a(end + 1) = 5;   # append element
a(end) = [];      # delete element
a(1:2:end)        # odd elements of a => [1, 3]
a(2:2:end)        # even elements of a => [2, 4]
a(end:-1:1)       # reversal of a => [4, 3, 2 , 1]
```

8.1.1 Advanced Indexing

An array with 'n' dimensions can be indexed using 'm' indices. More generally, the set of index tuples determining the result is formed by the Cartesian product of the index vectors (or ranges or scalars).

For the ordinary and most common case, `m == n`, and each index corresponds to its respective dimension. If `m < n` and every index is less than the size of the array in the i^{th} dimension, `m(i) < n(i)`, then the index expression is padded with trailing singleton dimensions (`[ones (m-n, 1)]`). If `m < n` but one of the indices `m(i)` is outside the size of the current array, then the last `n-m+1` dimensions are folded into a single dimension with an extent equal to the product of extents of the original dimensions. This is easiest to understand with an example.

```
a = reshape (1:8, 2, 2, 2)  # Create 3-D array
a =

ans(:,:,1) =

   1   3
   2   4

ans(:,:,2) =

   5   7
   6   8

a(2,1,2);   # Case (m == n): ans = 6
a(2,1);     # Case (m < n), idx within array:
            # equivalent to a(2,1,1), ans = 2
a(2,4);     # Case (m < n), idx outside array:
            # Dimension 2 & 3 folded into new dimension of size 2x2 = 4
            # Select 2nd row, 4th element of [2, 4, 6, 8], ans = 8
```

One advanced use of indexing is to create arrays filled with a single value. This can be done by using an index of ones on a scalar value. The result is an object with the dimensions of the index expression and every element equal to the original scalar. For example, the following statements

```
a = 13;
a(ones (1, 4))
```

produce a vector whose four elements are all equal to 13.

Similarly, by indexing a scalar with two vectors of ones it is possible to create a matrix. The following statements

```
a = 13;
a(ones (1, 2), ones (1, 3))
```

create a 2x3 matrix with all elements equal to 13.

The last example could also be written as

```
13(ones (2, 3))
```

It is more efficient to use indexing rather than the code construction `scalar * ones (N, M, ...)` because it avoids the unnecessary multiplication operation. Moreover, multiplication may not be defined for the object to be replicated whereas indexing an array is always defined. The following code shows how to create a 2x3 cell array from a base unit which is not itself a scalar.

```
{"Hello"}(ones (2, 3))
```

It should be, noted that `ones (1, n)` (a row vector of ones) results in a range (with zero increment). A range is stored internally as a starting value, increment, end value, and total number of values; hence, it is more efficient for storage than a vector or matrix of ones whenever the number of elements is greater than 4. In particular, when '`r`' is a row vector, the expressions

```
r(ones (1, n), :)
r(ones (n, 1), :)
```

will produce identical results, but the first one will be significantly faster, at least for '`r`' and '`n`' large enough. In the first case the index is held in compressed form as a range which allows Octave to choose a more efficient algorithm to handle the expression.

A general recommendation, for a user unaware of these subtleties, is to use the function `repmat` for replicating smaller arrays into bigger ones.

A second use of indexing is to speed up code. Indexing is a fast operation and judicious use of it can reduce the requirement for looping over individual array elements which is a slow operation.

Consider the following example which creates a 10-element row vector a containing the values $a_i = \sqrt{i}$.

```
for i = 1:10
  a(i) = sqrt (i);
endfor
```

It is quite inefficient to create a vector using a loop like this. In this case, it would have been much more efficient to use the expression

```
a = sqrt (1:10);
```

which avoids the loop entirely.

In cases where a loop cannot be avoided, or a number of values must be combined to form a larger matrix, it is generally faster to set the size of the matrix first (pre-allocate storage), and then insert elements using indexing commands. For example, given a matrix `a`,

```
[nr, nc] = size (a);
x = zeros (nr, n * nc);
for i = 1:n
  x(:,(i-1)*nc+1:i*nc) = a;
endfor
```

is considerably faster than

```
x = a;
for i = 1:n-1
  x = [x, a];
endfor
```

because Octave does not have to repeatedly resize the intermediate result.

ind = sub2ind (*dims*, *i*, *j*) [Function File]
ind = sub2ind (*dims*, *s1*, *s2*, ..., *sN*) [Function File]

Convert subscripts to a linear index.

The following example shows how to convert the two-dimensional index (2,3) of a 3-by-3 matrix to a linear index. The matrix is linearly indexed moving from one column to next, filling up all rows in each column.

```
linear_index = sub2ind ([3, 3], 2, 3)
⇒ 8
```

See also: [ind2sub], page 138.

[*s1*, *s2*, ..., *sN*] = ind2sub (*dims*, *ind*) [Function File]

Convert a linear index to subscripts.

The following example shows how to convert the linear index 8 in a 3-by-3 matrix into a subscript. The matrix is linearly indexed moving from one column to next, filling up all rows in each column.

```
[r, c] = ind2sub ([3, 3], 8)
    ⇒ r =  2
    ⇒ c =  3
```

See also: [sub2ind], page 138.

isindex (*ind*) [Built-in Function]
isindex (*ind*, *n*) [Built-in Function]

Return true if *ind* is a valid index.

Valid indices are either positive integers (although possibly of real data type), or logical arrays.

If present, *n* specifies the maximum extent of the dimension to be indexed. When possible the internal result is cached so that subsequent indexing using *ind* will not perform the check again.

Implementation Note: Strings are first converted to double values before the checks for valid indices are made. Unless a string contains the NULL character "\0", it will always be a valid index.

val = allow_noninteger_range_as_index () [Built-in Function]
old_val = allow_noninteger_range_as_index (*new_val*) [Built-in Function]
allow_noninteger_range_as_index (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether non-integer ranges are allowed as indices.

This might be useful for MATLAB compatibility; however, it is still not entirely compatible because MATLAB treats the range expression differently in different contexts.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

8.2 Calling Functions

A *function* is a name for a particular calculation. Because it has a name, you can ask for it by name at any point in the program. For example, the function sqrt computes the square root of a number.

A fixed set of functions are *built-in*, which means they are available in every Octave program. The sqrt function is one of these. In addition, you can define your own functions. See Chapter 11 [Functions and Scripts], page 171, for information about how to do this.

The way to use a function is with a *function call* expression, which consists of the function name followed by a list of *arguments* in parentheses. The arguments are expressions which give the raw materials for the calculation that the function will do. When there is more than one argument, they are separated by commas. If there are no arguments, you can omit the parentheses, but it is a good idea to include them anyway, to clearly indicate that a function call was intended. Here are some examples:

```
sqrt (x^2 + y^2)      # One argument
ones (n, m)           # Two arguments
rand ()               # No arguments
```

Each function expects a particular number of arguments. For example, the sqrt function must be called with a single argument, the number to take the square root of:

```
sqrt (argument)
```

Some of the built-in functions take a variable number of arguments, depending on the particular usage, and their behavior is different depending on the number of arguments supplied.

Like every other expression, the function call has a value, which is computed by the function based on the arguments you give it. In this example, the value of sqrt (*argument*) is the square root of the argument. A function can also have side effects, such as assigning the values of certain variables or doing input or output operations.

Unlike most languages, functions in Octave may return multiple values. For example, the following statement

```
[u, s, v] = svd (a)
```

computes the singular value decomposition of the matrix a and assigns the three result matrices to u, s, and v.

The left side of a multiple assignment expression is itself a list of expressions, and is allowed to be a list of variable names or index expressions. See also Section 8.1 [Index Expressions], page 135, and Section 8.6 [Assignment Ops], page 149.

8.2.1 Call by Value

In Octave, unlike Fortran, function arguments are passed by value, which means that each argument in a function call is evaluated and assigned to a temporary location in memory before being passed to the function. There is currently no way to specify that a function parameter should be passed by reference instead of by value. This means that it is impossible to directly alter the value of a function parameter in the calling function. It can only change the local copy within the function body. For example, the function

```
function f (x, n)
  while (n-- > 0)
    disp (x);
  endwhile
endfunction
```

displays the value of the first argument *n* times. In this function, the variable *n* is used as a temporary variable without having to worry that its value might also change in the calling function. Call by value is also useful because it is always possible to pass constants for any function parameter without first having to determine that the function will not attempt to modify the parameter.

The caller may use a variable as the expression for the argument, but the called function does not know this: it only knows what value the argument had. For example, given a function called as

```
foo = "bar";
fcn (foo)
```

you should not think of the argument as being "the variable `foo`." Instead, think of the argument as the string value, `"bar"`.

Even though Octave uses pass-by-value semantics for function arguments, values are not copied unnecessarily. For example,

```
x = rand (1000);
f (x);
```

does not actually force two 1000 by 1000 element matrices to exist *unless* the function `f` modifies the value of its argument. Then Octave must create a copy to avoid changing the value outside the scope of the function `f`, or attempting (and probably failing!) to modify the value of a constant or the value of a temporary result.

8.2.2 Recursion

With some restrictions[1], recursive function calls are allowed. A *recursive function* is one which calls itself, either directly or indirectly. For example, here is an inefficient[2] way to compute the factorial of a given integer:

```
function retval = fact (n)
  if (n > 0)
    retval = n * fact (n-1);
  else
    retval = 1;
  endif
endfunction
```

This function is recursive because it calls itself directly. It eventually terminates because each time it calls itself, it uses an argument that is one less than was used for the previous call. Once the argument is no longer greater than zero, it does not call itself, and the recursion ends.

The built-in variable `max_recursion_depth` specifies a limit to the recursion depth and prevents Octave from recursing infinitely.

val = `max_recursion_depth ()` [Built-in Function]
old_val = `max_recursion_depth (`*new_val*`)` [Built-in Function]
`max_recursion_depth (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal limit on the number of times a function may be called recursively.

If the limit is exceeded, an error message is printed and control returns to the top level.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

8.3 Arithmetic Operators

The following arithmetic operators are available, and work on scalars and matrices. The element-by-element operators and functions broadcast (see Section 19.2 [Broadcasting], page 491).

x `+` *y* Addition. If both operands are matrices, the number of rows and columns must both agree, or they must be broadcastable to the same shape.

x `.+` *y* Element-by-element addition. This operator is equivalent to `+`.

x `-` *y* Subtraction. If both operands are matrices, the number of rows and columns of both must agree, or they must be broadcastable to the same shape.

---

[1] Some of Octave's functions are implemented in terms of functions that cannot be called recursively. For example, the ODE solver `lsode` is ultimately implemented in a Fortran subroutine that cannot be called recursively, so `lsode` should not be called either directly or indirectly from within the user-supplied function that `lsode` requires. Doing so will result in an error.

[2] It would be much better to use `prod (1:n)`, or `gamma (n+1)` instead, after first checking to ensure that the value `n` is actually a positive integer.

x .- *y*
Element-by-element subtraction. This operator is equivalent to `-`.

x * *y*
Matrix multiplication. The number of columns of *x* must agree with the number of rows of *y*, or they must be broadcastable to the same shape.

x .* *y*
Element-by-element multiplication. If both operands are matrices, the number of rows and columns must both agree, or they must be broadcastable to the same shape.

x / *y*
Right division. This is conceptually equivalent to the expression

```
(inverse (y') * x')'
```

but it is computed without forming the inverse of *y'*.

If the system is not square, or if the coefficient matrix is singular, a minimum norm solution is computed.

x ./ *y*
Element-by-element right division.

x \ *y*
Left division. This is conceptually equivalent to the expression

```
inverse (x) * y
```

but it is computed without forming the inverse of *x*.

If the system is not square, or if the coefficient matrix is singular, a minimum norm solution is computed.

x .\ *y*
Element-by-element left division. Each element of *y* is divided by each corresponding element of *x*.

x ^ *y*
x ** *y*
Power operator. If *x* and *y* are both scalars, this operator returns *x* raised to the power *y*. If *x* is a scalar and *y* is a square matrix, the result is computed using an eigenvalue expansion. If *x* is a square matrix, the result is computed by repeated multiplication if *y* is an integer, and by an eigenvalue expansion if *y* is not an integer. An error results if both *x* and *y* are matrices.

The implementation of this operator needs to be improved.

x .^ *y*
x .** *y*
Element-by-element power operator. If both operands are matrices, the number of rows and columns must both agree, or they must be broadcastable to the same shape. If several complex results are possible, the one with smallest non-negative argument (angle) is taken. This rule may return a complex root even when a real root is also possible. Use `realpow`, `realsqrt`, `cbrt`, or `nthroot` if a real result is preferred.

-*x*
Negation.

+*x*
Unary plus. This operator has no effect on the operand.

x'
Complex conjugate transpose. For real arguments, this operator is the same as the transpose operator. For complex arguments, this operator is equivalent to the expression

```
conj (x.')
```

x.'
Transpose.

Note that because Octave's element-by-element operators begin with a '`.`', there is a possible ambiguity for statements like

```
1./m
```

because the period could be interpreted either as part of the constant or as part of the operator. To resolve this conflict, Octave treats the expression as if you had typed

```
(1) ./ m
```

and not

```
(1.) / m
```

Although this is inconsistent with the normal behavior of Octave's lexer, which usually prefers to break the input into tokens by preferring the longest possible match at any given point, it is more useful in this case.

`ctranspose` (*x*) [Built-in Function]

Return the complex conjugate transpose of *x*.

This function and `x'` are equivalent.

See also: [transpose], page 144.

`ldivide` (*x*, *y*) [Built-in Function]

Return the element-by-element left division of *x* and *y*.

This function and `x .\ y` are equivalent.

See also: [rdivide], page 144, [mldivide], page 143, [times], page 144, [plus], page 144.

`minus` (*x*, *y*) [Built-in Function]

This function and `x - y` are equivalent.

See also: [plus], page 144, [uminus], page 145.

`mldivide` (*x*, *y*) [Built-in Function]

Return the matrix left division of *x* and *y*.

This function and `x \ y` are equivalent.

See also: [mrdivide], page 143, [ldivide], page 143, [rdivide], page 144.

`mpower` (*x*, *y*) [Built-in Function]

Return the matrix power operation of *x* raised to the *y* power.

This function and `x ^ y` are equivalent.

See also: [power], page 144, [mtimes], page 144, [plus], page 144, [minus], page 143.

`mrdivide` (*x*, *y*) [Built-in Function]

Return the matrix right division of *x* and *y*.

This function and `x / y` are equivalent.

See also: [mldivide], page 143, [rdivide], page 144, [plus], page 144, [minus], page 143.

`mtimes` (*x*, *y*) [Built-in Function]
`mtimes` (*x1*, *x2*, ...) [Built-in Function]

Return the matrix multiplication product of inputs.

This function and `x * y` are equivalent. If more arguments are given, the multiplication is applied cumulatively from left to right:

```
(...((x1 * x2) * x3) * ...)
```

At least one argument is required.

See also: [times], page 144, [plus], page 144, [minus], page 143, [rdivide], page 144, [mrdivide], page 143, [mldivide], page 143, [mpower], page 143.

`plus` (*x*, *y*) [Built-in Function]
`plus` (*x1*, *x2*, ...) [Built-in Function]

This function and `x + y` are equivalent.

If more arguments are given, the summation is applied cumulatively from left to right:

```
(...((x1 + x2) + x3) + ...)
```

At least one argument is required.

See also: [minus], page 143, [uplus], page 145.

`power` (*x*, *y*) [Built-in Function]

Return the element-by-element operation of *x* raised to the *y* power.

This function and `x .^ y` are equivalent.

If several complex results are possible, returns the one with smallest non-negative argument (angle). Use `realpow`, `realsqrt`, `cbrt`, or `nthroot` if a real result is preferred.

See also: [mpower], page 143, [realpow], page 436, [realsqrt], page 436, [cbrt], page 436, [nthroot], page 436.

`rdivide` (*x*, *y*) [Built-in Function]

Return the element-by-element right division of *x* and *y*.

This function and `x ./ y` are equivalent.

See also: [ldivide], page 143, [mrdivide], page 143, [times], page 144, [plus], page 144.

`times` (*x*, *y*) [Built-in Function]
`times` (*x1*, *x2*, ...) [Built-in Function]

Return the element-by-element multiplication product of inputs.

This function and `x .* y` are equivalent. If more arguments are given, the multiplication is applied cumulatively from left to right:

```
(...((x1 .* x2) .* x3) .* ...)
```

At least one argument is required.

See also: [mtimes], page 144, [rdivide], page 144.

`transpose` (*x*) [Built-in Function]

Return the transpose of *x*.

This function and `x.'` are equivalent.

See also: [ctranspose], page 143.

`uminus (x)` [Built-in Function]

This function and `- x` are equivalent.

See also: [uplus], page 145, [minus], page 143.

`uplus (x)` [Built-in Function]

This function and `+ x` are equivalent.

See also: [uminus], page 145, [plus], page 144, [minus], page 143.

8.4 Comparison Operators

Comparison operators compare numeric values for relationships such as equality. They are written using *relational operators*.

All of Octave's comparison operators return a value of 1 if the comparison is true, or 0 if it is false. For matrix values, they all work on an element-by-element basis. Broadcasting rules apply. See Section 19.2 [Broadcasting], page 491. For example:

```
[1, 2; 3, 4] == [1, 3; 2, 4]
     ⇒  1  0
         0  1
```

According to broadcasting rules, if one operand is a scalar and the other is a matrix, the scalar is compared to each element of the matrix in turn, and the result is the same size as the matrix.

`x < y` True if *x* is less than *y*.

`x <= y` True if *x* is less than or equal to *y*.

`x == y` True if *x* is equal to *y*.

`x >= y` True if *x* is greater than or equal to *y*.

`x > y` True if *x* is greater than *y*.

`x != y`
`x ~= y` True if *x* is not equal to *y*.

For complex numbers, the following ordering is defined: `z1 < z2` if and only if

```
abs (z1) < abs (z2)
|| (abs (z1) == abs (z2) && arg (z1) < arg (z2))
```

This is consistent with the ordering used by *max*, *min* and *sort*, but is not consistent with MATLAB, which only compares the real parts.

String comparisons may also be performed with the `strcmp` function, not with the comparison operators listed above. See Chapter 5 [Strings], page 67.

`eq (x, y)` [Built-in Function]

Return true if the two inputs are equal.

This function is equivalent to `x == y`.

See also: [ne], page 146, [isequal], page 146, [le], page 146, [ge], page 146, [gt], page 146, [ne], page 146, [lt], page 146.

`ge (x, y)` [Built-in Function]

This function is equivalent to `x >= y`.

See also: [le], page 146, [eq], page 145, [gt], page 146, [ne], page 146, [lt], page 146.

`gt (x, y)` [Built-in Function]

This function is equivalent to `x > y`.

See also: [le], page 146, [eq], page 145, [ge], page 146, [ne], page 146, [lt], page 146.

`isequal (x1, x2, ...)` [Function File]

Return true if all of *x1*, *x2*, ... are equal.

See also: [isequaln], page 146.

`isequaln (x1, x2, ...)` [Function File]

Return true if all of *x1*, *x2*, ... are equal under the additional assumption that NaN == NaN (no comparison of NaN placeholders in dataset).

See also: [isequal], page 146.

`le (x, y)` [Built-in Function]

This function is equivalent to `x <= y`.

See also: [eq], page 145, [ge], page 146, [gt], page 146, [ne], page 146, [lt], page 146.

`lt (x, y)` [Built-in Function]

This function is equivalent to `x < y`.

See also: [le], page 146, [eq], page 145, [ge], page 146, [gt], page 146, [ne], page 146.

`ne (x, y)` [Built-in Function]

Return true if the two inputs are not equal.

This function is equivalent to `x != y`.

See also: [eq], page 145, [isequal], page 146, [le], page 146, [ge], page 146, [lt], page 146.

8.5 Boolean Expressions

8.5.1 Element-by-element Boolean Operators

An *element-by-element boolean expression* is a combination of comparison expressions using the boolean operators "or" ('`|`'), "and" ('`&`'), and "not" ('`!`'), along with parentheses to control nesting. The truth of the boolean expression is computed by combining the truth values of the corresponding elements of the component expressions. A value is considered to be false if it is zero, and true otherwise.

Element-by-element boolean expressions can be used wherever comparison expressions can be used. They can be used in `if` and `while` statements. However, a matrix value used as the condition in an `if` or `while` statement is only true if *all* of its elements are nonzero.

Like comparison operations, each element of an element-by-element boolean expression also has a numeric value (1 if true, 0 if false) that comes into play if the result of the boolean expression is stored in a variable, or used in arithmetic.

Here are descriptions of the three element-by-element boolean operators.

`boolean1 & boolean2`
: Elements of the result are true if both corresponding elements of *boolean1* and *boolean2* are true.

`boolean1 | boolean2`
: Elements of the result are true if either of the corresponding elements of *boolean1* or *boolean2* is true.

`! boolean`
`~ boolean`
: Each element of the result is true if the corresponding element of *boolean* is false.

These operators work on an element-by-element basis. For example, the expression

```
[1, 0; 0, 1] & [1, 0; 2, 3]
```

returns a two by two identity matrix.

For the binary operators, broadcasting rules apply. See Section 19.2 [Broadcasting], page 491. In particular, if one of the operands is a scalar and the other a matrix, the operator is applied to the scalar and each element of the matrix.

For the binary element-by-element boolean operators, both subexpressions *boolean1* and *boolean2* are evaluated before computing the result. This can make a difference when the expressions have side effects. For example, in the expression

```
a & b++
```

the value of the variable *b* is incremented even if the variable *a* is zero.

This behavior is necessary for the boolean operators to work as described for matrix-valued operands.

z = `and` (*x*, *y*) [Built-in Function]
z = `and` (*x1*, *x2*, ...) [Built-in Function]

Return the logical AND of *x* and *y*.

This function is equivalent to the operator syntax `x & y`. If more than two arguments are given, the logical AND is applied cumulatively from left to right:

```
(...((x1 & x2) & x3) & ...)
```

At least one argument is required.

See also: [or], page 147, [not], page 147, [xor], page 405.

z = `not` (*x*) [Built-in Function]

Return the logical NOT of *x*.

This function is equivalent to the operator syntax `! x`.

See also: [and], page 147, [or], page 147, [xor], page 405.

z = `or` (*x*, *y*) [Built-in Function]
z = `or` (*x1*, *x2*, ...) [Built-in Function]

Return the logical OR of *x* and *y*.

This function is equivalent to the operator syntax `x | y`. If more than two arguments are given, the logical OR is applied cumulatively from left to right:

```
(...((x1 | x2) | x3) | ...)
```

At least one argument is required.

See also: [and], page 147, [not], page 147, [xor], page 405.

8.5.2 Short-circuit Boolean Operators

Combined with the implicit conversion to scalar values in `if` and `while` conditions, Octave's element-by-element boolean operators are often sufficient for performing most logical operations. However, it is sometimes desirable to stop evaluating a boolean expression as soon as the overall truth value can be determined. Octave's *short-circuit* boolean operators work this way.

boolean1 `&&` *boolean2*

The expression *boolean1* is evaluated and converted to a scalar using the equivalent of the operation `all (boolean1(:))`. If it is false, the result of the overall expression is 0. If it is true, the expression *boolean2* is evaluated and converted to a scalar using the equivalent of the operation `all (boolean2(:))`. If it is true, the result of the overall expression is 1. Otherwise, the result of the overall expression is 0.

Warning: there is one exception to the rule of evaluating `all (boolean1(:))`, which is when `boolean1` is the empty matrix. The truth value of an empty matrix is always `false` so `[] && true` evaluates to `false` even though `all ([])` is `true`.

boolean1 `||` *boolean2*

The expression *boolean1* is evaluated and converted to a scalar using the equivalent of the operation `all (boolean1(:))`. If it is true, the result of the overall expression is 1. If it is false, the expression *boolean2* is evaluated and converted to a scalar using the equivalent of the operation `all (boolean2(:))`. If it is true, the result of the overall expression is 1. Otherwise, the result of the overall expression is 0.

Warning: the truth value of an empty matrix is always `false`, see the previous list item for details.

The fact that both operands may not be evaluated before determining the overall truth value of the expression can be important. For example, in the expression

```
a && b++
```

the value of the variable *b* is only incremented if the variable *a* is nonzero.

This can be used to write somewhat more concise code. For example, it is possible write

```
function f (a, b, c)
  if (nargin > 2 && ischar (c))
    ...
```

instead of having to use two `if` statements to avoid attempting to evaluate an argument that doesn't exist. For example, without the short-circuit feature, it would be necessary to write

```
function f (a, b, c)
  if (nargin > 2)
    if (ischar (c))
      ...
```

Writing

```
function f (a, b, c)
  if (nargin > 2 & ischar (c))
    ...
```

would result in an error if `f` were called with one or two arguments because Octave would be forced to try to evaluate both of the operands for the operator '`&`'.

MATLAB has special behavior that allows the operators '`&`' and '`|`' to short-circuit when used in the truth expression for `if` and `while` statements. Octave also behaves the same way by default, though the use of the '`&`' and '`|`' operators in this way is strongly discouraged. Instead, you should use the '`&&`' and '`||`' operators that always have short-circuit behavior.

Finally, the ternary operator (?:) is not supported in Octave. If short-circuiting is not important, it can be replaced by the `ifelse` function.

`merge (`*`mask`*`, `*`tval`*`, `*`fval`*`)` [Built-in Function]
`ifelse (`*`mask`*`, `*`tval`*`, `*`fval`*`)` [Built-in Function]

Merge elements of *true_val* and *false_val*, depending on the value of *mask*.

If *mask* is a logical scalar, the other two arguments can be arbitrary values. Otherwise, *mask* must be a logical array, and *tval*, *fval* should be arrays of matching class, or cell arrays. In the scalar mask case, *tval* is returned if *mask* is true, otherwise *fval* is returned.

In the array mask case, both *tval* and *fval* must be either scalars or arrays with dimensions equal to *mask*. The result is constructed as follows:

```
result(mask) = tval(mask);
result(! mask) = fval(! mask);
```

mask can also be arbitrary numeric type, in which case it is first converted to logical.

See also: [logical], page 60, [diff], page 406.

8.6 Assignment Expressions

An *assignment* is an expression that stores a new value into a variable. For example, the following expression assigns the value 1 to the variable `z`:

```
z = 1
```

After this expression is executed, the variable `z` has the value 1. Whatever old value `z` had before the assignment is forgotten. The '`=`' sign is called an *assignment operator*.

Assignments can store string values also. For example, the following expression would store the value `"this food is good"` in the variable `message`:

```
thing = "food"
predicate = "good"
message = [ "this " , thing , " is " , predicate ]
```

(This also illustrates concatenation of strings.)

Most operators (addition, concatenation, and so on) have no effect except to compute a value. If you ignore the value, you might as well not use the operator. An assignment operator is different. It does produce a value, but even if you ignore the value, the assignment still makes itself felt through the alteration of the variable. We call this a *side effect*.

The left-hand operand of an assignment need not be a variable (see Chapter 7 [Variables], page 123). It can also be an element of a matrix (see Section 8.1 [Index Expressions], page 135) or a list of return values (see Section 8.2 [Calling Functions], page 139). These are all called *lvalues*, which means they can appear on the left-hand side of an assignment operator. The right-hand operand may be any expression. It produces the new value which the assignment stores in the specified variable, matrix element, or list of return values.

It is important to note that variables do *not* have permanent types. The type of a variable is simply the type of whatever value it happens to hold at the moment. In the following program fragment, the variable `foo` has a numeric value at first, and a string value later on:

```
octave:13> foo = 1
foo = 1
octave:13> foo = "bar"
foo = bar
```

When the second assignment gives `foo` a string value, the fact that it previously had a numeric value is forgotten.

Assignment of a scalar to an indexed matrix sets all of the elements that are referenced by the indices to the scalar value. For example, if `a` is a matrix with at least two columns,

```
a(:, 2) = 5
```

sets all the elements in the second column of `a` to 5.

Assigning an empty matrix '`[]`' works in most cases to allow you to delete rows or columns of matrices and vectors. See Section 4.1.1 [Empty Matrices], page 51. For example, given a 4 by 5 matrix A, the assignment

```
A (3, :) = []
```

deletes the third row of A, and the assignment

```
A (:, 1:2:5) = []
```

deletes the first, third, and fifth columns.

An assignment is an expression, so it has a value. Thus, `z = 1` as an expression has the value 1. One consequence of this is that you can write multiple assignments together:

```
x = y = z = 0
```

stores the value 0 in all three variables. It does this because the value of `z = 0`, which is 0, is stored into `y`, and then the value of `y = z = 0`, which is 0, is stored into `x`.

This is also true of assignments to lists of values, so the following is a valid expression

```
[a, b, c] = [u, s, v] = svd (a)
```

that is exactly equivalent to

```
[u, s, v] = svd (a)
a = u
b = s
c = v
```

In expressions like this, the number of values in each part of the expression need not match. For example, the expression

```
[a, b] = [u, s, v] = svd (a)
```

is equivalent to

```
[u, s, v] = svd (a)
a = u
b = s
```

The number of values on the left side of the expression can, however, not exceed the number of values on the right side. For example, the following will produce an error.

```
[a, b, c, d] = [u, s, v] = svd (a);
 ⊣ error: element number 4 undefined in return list
```

The symbol `~` may be used as a placeholder in the list of lvalues, indicating that the corresponding return value should be ignored and not stored anywhere:

```
[~, s, v] = svd (a);
```

This is cleaner and more memory efficient than using a dummy variable. The `nargout` value for the right-hand side expression is not affected. If the assignment is used as an expression, the return value is a comma-separated list with the ignored values dropped.

A very common programming pattern is to increment an existing variable with a given value, like this

```
a = a + 2;
```

This can be written in a clearer and more condensed form using the `+=` operator

```
a += 2;
```

Similar operators also exist for subtraction (`-=`), multiplication (`*=`), and division (`/=`). An expression of the form

```
expr1 op= expr2
```

is evaluated as

```
expr1 = (expr1) op (expr2)
```

where *op* can be either `+`, `-`, `*`, or `/`, as long as *expr2* is a simple expression with no side effects. If *expr2* also contains an assignment operator, then this expression is evaluated as

```
temp = expr2
expr1 = (expr1) op temp
```

where *temp* is a placeholder temporary value storing the computed result of evaluating *expr2*. So, the expression

```
a *= b+1
```

is evaluated as

```
a = a * (b+1)
```

and *not*

```
a = a * b + 1
```

You can use an assignment anywhere an expression is called for. For example, it is valid to write `x != (y = 1)` to set `y` to 1 and then test whether `x` equals 1. But this style tends to make programs hard to read. Except in a one-shot program, you should rewrite it to get rid of such nesting of assignments. This is never very hard.

8.7 Increment Operators

Increment operators increase or decrease the value of a variable by 1. The operator to increment a variable is written as '`++`'. It may be used to increment a variable either before or after taking its value.

For example, to pre-increment the variable *x*, you would write `++x`. This would add one to *x* and then return the new value of *x* as the result of the expression. It is exactly the same as the expression `x = x + 1`.

To post-increment a variable *x*, you would write `x++`. This adds one to the variable *x*, but returns the value that *x* had prior to incrementing it. For example, if *x* is equal to 2, the result of the expression `x++` is 2, and the new value of *x* is 3.

For matrix and vector arguments, the increment and decrement operators work on each element of the operand.

Here is a list of all the increment and decrement expressions.

`++x`
: This expression increments the variable *x*. The value of the expression is the *new* value of *x*. It is equivalent to the expression `x = x + 1`.

`--x`
: This expression decrements the variable *x*. The value of the expression is the *new* value of *x*. It is equivalent to the expression `x = x - 1`.

`x++`
: This expression causes the variable *x* to be incremented. The value of the expression is the *old* value of *x*.

`x--`
: This expression causes the variable *x* to be decremented. The value of the expression is the *old* value of *x*.

8.8 Operator Precedence

Operator precedence determines how operators are grouped, when different operators appear close by in one expression. For example, '`*`' has higher precedence than '`+`'. Thus, the expression `a + b * c` means to multiply `b` and `c`, and then add `a` to the product (i.e., `a + (b * c)`).

You can overrule the precedence of the operators by using parentheses. You can think of the precedence rules as saying where the parentheses are assumed if you do not write parentheses yourself. In fact, it is wise to use parentheses whenever you have an unusual combination of operators, because other people who read the program may not remember what the precedence is in this case. You might forget as well, and then you too could make a mistake. Explicit parentheses will help prevent any such mistake.

When operators of equal precedence are used together, the leftmost operator groups first, except for the assignment operators, which group in the opposite order. Thus, the expression `a - b + c` groups as `(a - b) + c`, but the expression `a = b = c` groups as `a = (b = c)`.

The precedence of prefix unary operators is important when another operator follows the operand. For example, `-x^2` means `-(x^2)`, because '`-`' has lower precedence than '`^`'.

Here is a table of the operators in Octave, in order of decreasing precedence. Unless noted, all operators group left to right.

function call and array indexing, cell array indexing, and structure element indexing
‘()’ ‘{}’ ‘.’

postfix increment, and postfix decrement
‘++’ ‘--’

These operators group right to left.

transpose and exponentiation
‘'’ ‘.'’ ‘^’ ‘**’ ‘.^’ ‘.**’

unary plus, unary minus, prefix increment, prefix decrement, and logical "not"
‘+’ ‘-’ ‘++’ ‘--’ ‘~’ ‘!’

multiply and divide
‘*’ ‘/’ ‘\’ ‘.\’ ‘.*’ ‘./’

add, subtract
‘+’ ‘-’

colon ‘:’

relational
‘<’ ‘<=’ ‘==’ ‘>=’ ‘>’ ‘!=’ ‘~=’

element-wise "and"
‘&’

element-wise "or"
‘|’

logical "and"
‘&&’

logical "or"
‘||’

assignment
‘=’ ‘+=’ ‘-=’ ‘*=’ ‘/=’ ‘\=’ ‘^=’ ‘.*=’ ‘./=’ ‘.\=’ ‘.^=’ ‘|=’ ‘&=’

These operators group right to left.

9 Evaluation

Normally, you evaluate expressions simply by typing them at the Octave prompt, or by asking Octave to interpret commands that you have saved in a file.

Sometimes, you may find it necessary to evaluate an expression that has been computed and stored in a string, which is exactly what the `eval` function lets you do.

`eval (`*`try`*`)` [Built-in Function]
`eval (`*`try`*`, `*`catch`*`)` [Built-in Function]

Parse the string *try* and evaluate it as if it were an Octave program.

If execution fails, evaluate the optional string *catch*.

The string *try* is evaluated in the current context, so any results remain available after `eval` returns.

The following example creates the variable A with the approximate value of 3.1416 in the current workspace.

```
eval ("A = acos(-1);");
```

If an error occurs during the evaluation of *try* then the *catch* string is evaluated, as the following example shows:

```
eval ('error ("This is a bad example");',
      'printf ("This error occurred:\n%s\n", lasterr ());');
     ⊣ This error occurred:
       This is a bad example
```

Programming Note: if you are only using `eval` as an error-capturing mechanism, rather than for the execution of arbitrary code strings, Consider using try/catch blocks or unwind_protect/unwind_protect_cleanup blocks instead. These techniques have higher performance and don't introduce the security considerations that the evaluation of arbitrary code does.

See also: [evalin], page 158.

9.1 Calling a Function by its Name

The `feval` function allows you to call a function from a string containing its name. This is useful when writing a function that needs to call user-supplied functions. The `feval` function takes the name of the function to call as its first argument, and the remaining arguments are given to the function.

The following example is a simple-minded function using `feval` that finds the root of a user-supplied function of one variable using Newton's method.

```
function result = newtroot (fname, x)

# usage: newtroot (fname, x)
#
#   fname : a string naming a function f(x).
#   x     : initial guess

  delta = tol = sqrt (eps);
```

```
    maxit = 200;
    fx = feval (fname, x);
    for i = 1:maxit
      if (abs (fx) < tol)
        result = x;
        return;
      else
        fx_new = feval (fname, x + delta);
        deriv = (fx_new - fx) / delta;
        x = x - fx / deriv;
        fx = fx_new;
      endif
    endfor

    result = x;

  endfunction
```

Note that this is only meant to be an example of calling user-supplied functions and should not be taken too seriously. In addition to using a more robust algorithm, any serious code would check the number and type of all the arguments, ensure that the supplied function really was a function, etc. See Section 4.8 [Predicates for Numeric Objects], page 62, for a list of predicates for numeric objects, and see Section 7.3 [Status of Variables], page 127, for a description of the `exist` function.

`feval` (*name*, ...) [Built-in Function]

Evaluate the function named *name*.

Any arguments after the first are passed as inputs to the named function. For example,

```
feval ("acos", -1)
     ⇒ 3.1416
```

calls the function `acos` with the argument '`-1`'.

The function `feval` can also be used with function handles of any sort (see Section 11.11.1 [Function Handles], page 199). Historically, `feval` was the only way to call user-supplied functions in strings, but function handles are now preferred due to the cleaner syntax they offer. For example,

```
f = @exp;
feval (f, 1)
    ⇒ 2.7183
f (1)
    ⇒ 2.7183
```

are equivalent ways to call the function referred to by *f*. If it cannot be predicted beforehand whether *f* is a function handle, function name in a string, or inline function then `feval` can be used instead.

A similar function `run` exists for calling user script files, that are not necessarily on the user path

`run` *script* [Command]
`run ("`*script*`")` [Function File]

Run *script* in the current workspace.

Scripts which reside in directories specified in Octave's load path, and which end with the extension `".m"`, can be run simply by typing their name. For scripts not located on the load path, use `run`.

The file name *script* can be a bare, fully qualified, or relative filename and with or without a file extension. If no extension is specified, Octave will first search for a script with the `".m"` extension before falling back to the script name without an extension.

Implementation Note: If *script* includes a path component, then `run` first changes the working directory to the directory where *script* is found. Next, the script is executed. Finally, `run` returns to the original working directory unless `script` has specifically changed directories.

See also: [path], page 190, [addpath], page 189, [source], page 199.

9.2 Evaluation in a Different Context

Before you evaluate an expression you need to substitute the values of the variables used in the expression. These are stored in the symbol table. Whenever the interpreter starts a new function it saves the current symbol table and creates a new one, initializing it with the list of function parameters and a couple of predefined variables such as `nargin`. Expressions inside the function use the new symbol table.

Sometimes you want to write a function so that when you call it, it modifies variables in your own context. This allows you to use a pass-by-name style of function, which is similar to using a pointer in programming languages such as C.

Consider how you might write `save` and `load` as m-files. For example:

```
function create_data
  x = linspace (0, 10, 10);
  y = sin (x);
  save mydata x y
endfunction
```

With `evalin`, you could write `save` as follows:

```
function save (file, name1, name2)
  f = open_save_file (file);
  save_var (f, name1, evalin ("caller", name1));
  save_var (f, name2, evalin ("caller", name2));
endfunction
```

Here, '`caller`' is the `create_data` function and `name1` is the string `"x"`, which evaluates simply as the value of `x`.

You later want to load the values back from `mydata` in a different context:

```
function process_data
  load mydata
  ... do work ...
endfunction
```

With `assignin`, you could write `load` as follows:

```
function load (file)
  f = open_load_file (file);
  [name, val] = load_var (f);
  assignin ("caller", name, val);
  [name, val] = load_var (f);
  assignin ("caller", name, val);
endfunction
```

Here, '`caller`' is the `process_data` function.

You can set and use variables at the command prompt using the context '`base`' rather than '`caller`'.

These functions are rarely used in practice. One example is the `fail ('code', 'pattern')` function which evaluates '`code`' in the caller's context and checks that the error message it produces matches the given pattern. Other examples such as `save` and `load` are written in C++ where all Octave variables are in the '`caller`' context and `evalin` is not needed.

`evalin` (*context*, *try*) [Built-in Function]
`evalin` (*context*, *try*, *catch*) [Built-in Function]

Like `eval`, except that the expressions are evaluated in the context *context*, which may be either `"caller"` or `"base"`.

See also: [eval], page 155, [assignin], page 158.

`assignin` (*context*, *varname*, *value*) [Built-in Function]

Assign *value* to *varname* in context *context*, which may be either `"base"` or `"caller"`.

See also: [evalin], page 158.

10 Statements

Statements may be a simple constant expression or a complicated list of nested loops and conditional statements.

Control statements such as `if`, `while`, and so on control the flow of execution in Octave programs. All the control statements start with special keywords such as `if` and `while`, to distinguish them from simple expressions. Many control statements contain other statements; for example, the `if` statement contains another statement which may or may not be executed.

Each control statement has a corresponding *end* statement that marks the end of the control statement. For example, the keyword `endif` marks the end of an `if` statement, and `endwhile` marks the end of a `while` statement. You can use the keyword `end` anywhere a more specific end keyword is expected, but using the more specific keywords is preferred because if you use them, Octave is able to provide better diagnostics for mismatched or missing end tokens.

The list of statements contained between keywords like `if` or `while` and the corresponding end statement is called the *body* of a control statement.

10.1 The if Statement

The `if` statement is Octave's decision-making statement. There are three basic forms of an `if` statement. In its simplest form, it looks like this:

```
if (condition)
  then-body
endif
```

condition is an expression that controls what the rest of the statement will do. The *then-body* is executed only if *condition* is true.

The condition in an `if` statement is considered true if its value is nonzero, and false if its value is zero. If the value of the conditional expression in an `if` statement is a vector or a matrix, it is considered true only if it is non-empty and *all* of the elements are nonzero. The conceptually equivalent code when *condition* is a matrix is shown below.

```
if (matrix) ≡ if (all (matrix(:)))
```

The second form of an if statement looks like this:

```
if (condition)
  then-body
else
  else-body
endif
```

If *condition* is true, *then-body* is executed; otherwise, *else-body* is executed.

Here is an example:

```
if (rem (x, 2) == 0)
  printf ("x is even\n");
else
  printf ("x is odd\n");
endif
```

In this example, if the expression `rem (x, 2) == 0` is true (that is, the value of `x` is divisible by 2), then the first `printf` statement is evaluated, otherwise the second `printf` statement is evaluated.

The third and most general form of the `if` statement allows multiple decisions to be combined in a single statement. It looks like this:

```
if (condition)
  then-body
elseif (condition)
  elseif-body
else
  else-body
endif
```

Any number of `elseif` clauses may appear. Each condition is tested in turn, and if one is found to be true, its corresponding *body* is executed. If none of the conditions are true and the `else` clause is present, its body is executed. Only one `else` clause may appear, and it must be the last part of the statement.

In the following example, if the first condition is true (that is, the value of `x` is divisible by 2), then the first `printf` statement is executed. If it is false, then the second condition is tested, and if it is true (that is, the value of `x` is divisible by 3), then the second `printf` statement is executed. Otherwise, the third `printf` statement is performed.

```
if (rem (x, 2) == 0)
  printf ("x is even\n");
elseif (rem (x, 3) == 0)
  printf ("x is odd and divisible by 3\n");
else
  printf ("x is odd\n");
endif
```

Note that the `elseif` keyword must not be spelled `else if`, as is allowed in Fortran. If it is, the space between the `else` and `if` will tell Octave to treat this as a new `if` statement within another `if` statement's `else` clause. For example, if you write

```
if (c1)
  body-1
else if (c2)
  body-2
endif
```

Octave will expect additional input to complete the first `if` statement. If you are using Octave interactively, it will continue to prompt you for additional input. If Octave is reading this input from a file, it may complain about missing or mismatched `end` statements, or, if you have not used the more specific `end` statements (`endif`, `endfor`, etc.), it may simply produce incorrect results, without producing any warning messages.

It is much easier to see the error if we rewrite the statements above like this,

```
if (c1)
  body-1
else
  if (c2)
    body-2
  endif
```

using the indentation to show how Octave groups the statements. See Chapter 11 [Functions and Scripts], page 171.

10.2 The switch Statement

It is very common to take different actions depending on the value of one variable. This is possible using the `if` statement in the following way

```
if (X == 1)
  do_something ();
elseif (X == 2)
  do_something_else ();
else
  do_something_completely_different ();
endif
```

This kind of code can however be very cumbersome to both write and maintain. To overcome this problem Octave supports the `switch` statement. Using this statement, the above example becomes

```
switch (X)
  case 1
    do_something ();
  case 2
    do_something_else ();
  otherwise
    do_something_completely_different ();
endswitch
```

This code makes the repetitive structure of the problem more explicit, making the code easier to read, and hence maintain. Also, if the variable `X` should change its name, only one line would need changing compared to one line per case when `if` statements are used.

The general form of the `switch` statement is

```
switch (expression)
  case label
    command_list
  case label
    command_list
  ...

  otherwise
    command_list
endswitch
```

where *label* can be any expression. However, duplicate *label* values are not detected, and only the *command_list* corresponding to the first match will be executed. For the `switch` statement to be meaningful at least one `case label command_list` clause must be present, while the `otherwise command_list` clause is optional.

If *label* is a cell array the corresponding *command_list* is executed if *any* of the elements of the cell array match *expression*. As an example, the following program will print '`Variable is either 6 or 7`'.

```
A = 7;
switch (A)
  case { 6, 7 }
    printf ("variable is either 6 or 7\n");
  otherwise
    printf ("variable is neither 6 nor 7\n");
endswitch
```

As with all other specific `end` keywords, `endswitch` may be replaced by `end`, but you can get better diagnostics if you use the specific forms.

One advantage of using the `switch` statement compared to using `if` statements is that the *label*s can be strings. If an `if` statement is used it is *not* possible to write

```
if (X == "a string") # This is NOT valid
```

since a character-to-character comparison between `X` and the string will be made instead of evaluating if the strings are equal. This special-case is handled by the `switch` statement, and it is possible to write programs that look like this

```
switch (X)
  case "a string"
    do_something
  ...
endswitch
```

10.2.1 Notes for the C Programmer

The `switch` statement is also available in the widely used C programming language. There are, however, some differences between the statement in Octave and C

- Cases are exclusive, so they don't 'fall through' as do the cases in the `switch` statement of the C language.
- The *command_list* elements are not optional. Making the list optional would have meant requiring a separator between the label and the command list. Otherwise, things like

  ```
  switch (foo)
    case (1) -2
    ...
  ```

  would produce surprising results, as would

  ```
  switch (foo)
    case (1)
    case (2)
      doit ();
    ...
  ```

particularly for C programmers. If `doit()` should be executed if *foo* is either `1` or `2`, the above code should be written with a cell array like this

```
switch (foo)
  case { 1, 2 }
    doit ();
  ...
```

10.3 The while Statement

In programming, a *loop* means a part of a program that is (or at least can be) executed two or more times in succession.

The `while` statement is the simplest looping statement in Octave. It repeatedly executes a statement as long as a condition is true. As with the condition in an `if` statement, the condition in a `while` statement is considered true if its value is nonzero, and false if its value is zero. If the value of the conditional expression in a `while` statement is a vector or a matrix, it is considered true only if it is non-empty and *all* of the elements are nonzero.

Octave's `while` statement looks like this:

```
while (condition)
  body
endwhile
```

Here *body* is a statement or list of statements that we call the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running.

The first thing the `while` statement does is test *condition*. If *condition* is true, it executes the statement *body*. After *body* has been executed, *condition* is tested again, and if it is still true, *body* is executed again. This process repeats until *condition* is no longer true. If *condition* is initially false, the body of the loop is never executed.

This example creates a variable `fib` that contains the first ten elements of the Fibonacci sequence.

```
fib = ones (1, 10);
i = 3;
while (i <= 10)
  fib (i) = fib (i-1) + fib (i-2);
  i++;
endwhile
```

Here the body of the loop contains two statements.

The loop works like this: first, the value of `i` is set to 3. Then, the `while` tests whether `i` is less than or equal to 10. This is the case when `i` equals 3, so the value of the `i`-th element of `fib` is set to the sum of the previous two values in the sequence. Then the `i++` increments the value of `i` and the loop repeats. The loop terminates when `i` reaches 11.

A newline is not required between the condition and the body; but using one makes the program clearer unless the body is very simple.

10.4 The do-until Statement

The `do-until` statement is similar to the `while` statement, except that it repeatedly executes a statement until a condition becomes true, and the test of the condition is at the end of the loop, so the body of the loop is always executed at least once. As with the condition in an `if` statement, the condition in a `do-until` statement is considered true if its value is nonzero, and false if its value is zero. If the value of the conditional expression in a `do-until` statement is a vector or a matrix, it is considered true only if it is non-empty and *all* of the elements are nonzero.

Octave's `do-until` statement looks like this:

```
do
  body
until (condition)
```

Here *body* is a statement or list of statements that we call the *body* of the loop, and *condition* is an expression that controls how long the loop keeps running.

This example creates a variable `fib` that contains the first ten elements of the Fibonacci sequence.

```
fib = ones (1, 10);
i = 2;
do
  i++;
  fib (i) = fib (i-1) + fib (i-2);
until (i == 10)
```

A newline is not required between the `do` keyword and the body; but using one makes the program clearer unless the body is very simple.

10.5 The for Statement

The `for` statement makes it more convenient to count iterations of a loop. The general form of the `for` statement looks like this:

```
for var = expression
  body
endfor
```

where *body* stands for any statement or list of statements, *expression* is any valid expression, and *var* may take several forms. Usually it is a simple variable name or an indexed variable. If the value of *expression* is a structure, *var* may also be a vector with two elements. See Section 10.5.1 [Looping Over Structure Elements], page 165, below.

The assignment expression in the `for` statement works a bit differently than Octave's normal assignment statement. Instead of assigning the complete result of the expression, it assigns each column of the expression to *var* in turn. If *expression* is a range, a row vector, or a scalar, the value of *var* will be a scalar each time the loop body is executed. If *var* is a column vector or a matrix, *var* will be a column vector each time the loop body is executed.

The following example shows another way to create a vector containing the first ten elements of the Fibonacci sequence, this time using the `for` statement:

```
fib = ones (1, 10);
for i = 3:10
  fib(i) = fib(i-1) + fib(i-2);
endfor
```

This code works by first evaluating the expression `3:10`, to produce a range of values from 3 to 10 inclusive. Then the variable `i` is assigned the first element of the range and the body of the loop is executed once. When the end of the loop body is reached, the next value in the range is assigned to the variable `i`, and the loop body is executed again. This process continues until there are no more elements to assign.

Within Octave is it also possible to iterate over matrices or cell arrays using the `for` statement. For example consider

```
disp ("Loop over a matrix")
for i = [1,3;2,4]
  i
endfor
disp ("Loop over a cell array")
for i = {1,"two";"three",4}
  i
endfor
```

In this case the variable `i` takes on the value of the columns of the matrix or cell matrix. So the first loop iterates twice, producing two column vectors `[1;2]`, followed by `[3;4]`, and likewise for the loop over the cell array. This can be extended to loops over multi-dimensional arrays. For example:

```
a = [1,3;2,4]; c = cat (3, a, 2*a);
for i = c
  i
endfor
```

In the above case, the multi-dimensional matrix *c* is reshaped to a two-dimensional matrix as `reshape (c, rows (c), prod (size (c)(2:end)))` and then the same behavior as a loop over a two dimensional matrix is produced.

Although it is possible to rewrite all `for` loops as `while` loops, the Octave language has both statements because often a `for` loop is both less work to type and more natural to think of. Counting the number of iterations is very common in loops and it can be easier to think of this counting as part of looping rather than as something to do inside the loop.

10.5.1 Looping Over Structure Elements

A special form of the `for` statement allows you to loop over all the elements of a structure:

```
for [ val, key ] = expression
  body
endfor
```

In this form of the `for` statement, the value of *expression* must be a structure. If it is, *key* and *val* are set to the name of the element and the corresponding value in turn, until there are no more elements. For example:

```
x.a = 1
x.b = [1, 2; 3, 4]
x.c = "string"
for [val, key] = x
  key
  val
endfor

     ⊣ key = a
     ⊣ val = 1
     ⊣ key = b
     ⊣ val =
     ⊣
     ⊣   1  2
     ⊣   3  4
     ⊣
     ⊣ key = c
     ⊣ val = string
```

The elements are not accessed in any particular order. If you need to cycle through the list in a particular way, you will have to use the function `fieldnames` and sort the list yourself.

10.6 The break Statement

The `break` statement jumps out of the innermost `while`, `do-until`, or `for` loop that encloses it. The `break` statement may only be used within the body of a loop. The following example finds the smallest divisor of a given integer, and also identifies prime numbers:

```
num = 103;
div = 2;
while (div*div <= num)
  if (rem (num, div) == 0)
    break;
  endif
  div++;
endwhile
if (rem (num, div) == 0)
  printf ("Smallest divisor of %d is %d\n", num, div)
else
  printf ("%d is prime\n", num);
endif
```

When the remainder is zero in the first `while` statement, Octave immediately *breaks out* of the loop. This means that Octave proceeds immediately to the statement following the loop and continues processing. (This is very different from the `exit` statement which stops the entire Octave program.)

Here is another program equivalent to the previous one. It illustrates how the *condition* of a `while` statement could just as well be replaced with a `break` inside an `if`:

```
num = 103;
div = 2;
while (1)
  if (rem (num, div) == 0)
    printf ("Smallest divisor of %d is %d\n", num, div);
    break;
  endif
  div++;
  if (div*div > num)
    printf ("%d is prime\n", num);
    break;
  endif
endwhile
```

10.7 The continue Statement

The `continue` statement, like `break`, is used only inside `while`, `do-until`, or `for` loops. It skips over the rest of the loop body, causing the next cycle around the loop to begin immediately. Contrast this with `break`, which jumps out of the loop altogether. Here is an example:

```
# print elements of a vector of random
# integers that are even.

# first, create a row vector of 10 random
# integers with values between 0 and 100:

vec = round (rand (1, 10) * 100);

# print what we're interested in:

for x = vec
  if (rem (x, 2) != 0)
    continue;
  endif
  printf ("%d\n", x);
endfor
```

If one of the elements of *vec* is an odd number, this example skips the print statement for that element, and continues back to the first statement in the loop.

This is not a practical example of the `continue` statement, but it should give you a clear understanding of how it works. Normally, one would probably write the loop like this:

```
for x = vec
  if (rem (x, 2) == 0)
    printf ("%d\n", x);
  endif
endfor
```

10.8 The unwind_protect Statement

Octave supports a limited form of exception handling modeled after the unwind-protect form of Lisp.

The general form of an `unwind_protect` block looks like this:

```
unwind_protect
  body
unwind_protect_cleanup
  cleanup
end_unwind_protect
```

where *body* and *cleanup* are both optional and may contain any Octave expressions or commands. The statements in *cleanup* are guaranteed to be executed regardless of how control exits *body*.

This is useful to protect temporary changes to global variables from possible errors. For example, the following code will always restore the original value of the global variable `frobnosticate` even if an error occurs in the first part of the `unwind_protect` block.

```
save_frobnosticate = frobnosticate;
unwind_protect
  frobnosticate = true;
  ...
unwind_protect_cleanup
  frobnosticate = save_frobnosticate;
end_unwind_protect
```

Without `unwind_protect`, the value of *frobnosticate* would not be restored if an error occurs while evaluating the first part of the `unwind_protect` block because evaluation would stop at the point of the error and the statement to restore the value would not be executed.

In addition to unwind_protect, Octave supports another form of exception handling, the `try` block.

10.9 The try Statement

The original form of a `try` block looks like this:

```
try
  body
catch
  cleanup
end_try_catch
```

where *body* and *cleanup* are both optional and may contain any Octave expressions or commands. The statements in *cleanup* are only executed if an error occurs in *body*.

No warnings or error messages are printed while *body* is executing. If an error does occur during the execution of *body*, *cleanup* can use the functions `lasterr` or `lasterror` to access the text of the message that would have been printed, as well as its identifier. The alternative form,

```
try
  body
catch err
  cleanup
end_try_catch
```

will automatically store the output of `lasterror` in the structure *err*. See Chapter 12 [Errors and Warnings], page 205, for more information about the `lasterr` and `lasterror` functions.

10.10 Continuation Lines

In the Octave language, most statements end with a newline character and you must tell Octave to ignore the newline character in order to continue a statement from one line to the next. Lines that end with the characters `...` are joined with the following line before they are divided into tokens by Octave's parser. For example, the lines

```
x = long_variable_name ...
    + longer_variable_name ...
    - 42
```

form a single statement.

Any text between the continuation marker and the newline character is ignored. For example, the statement

```
x = long_variable_name ...    # comment one
    + longer_variable_name ...comment two
    - 42                      # last comment
```

is equivalent to the one shown above.

Inside double-quoted string constants, the character `\` has to be used as continuation marker. The `\` must appear at the end of the line just before the newline character:

```
s = "This text starts in the first line \
and is continued in the second line."
```

Input that occurs inside parentheses can be continued to the next line without having to use a continuation marker. For example, it is possible to write statements like

```
if (fine_dining_destination == on_a_boat
    || fine_dining_destination == on_a_train)
  seuss (i, will, not, eat, them, sam, i, am, i,
         will, not, eat, green, eggs, and, ham);
endif
```

without having to add to the clutter with continuation markers.

11 Functions and Scripts

Complicated Octave programs can often be simplified by defining functions. Functions can be defined directly on the command line during interactive Octave sessions, or in external files, and can be called just like built-in functions.

11.1 Introduction to Function and Script Files

There are seven different things covered in this section.

1. Typing in a function at the command prompt.
2. Storing a group of commands in a file — called a script file.
3. Storing a function in a file—called a function file.
4. Subfunctions in function files.
5. Multiple functions in one script file.
6. Private functions.
7. Nested functions.

Both function files and script files end with an extension of .m, for MATLAB compatibility. If you want more than one independent functions in a file, it must be a script file (see Section 11.10 [Script Files], page 198), and to use these functions you must execute the script file before you can use the functions that are in the script file.

11.2 Defining Functions

In its simplest form, the definition of a function named *name* looks like this:

```
function name
  body
endfunction
```

A valid function name is like a valid variable name: a sequence of letters, digits and underscores, not starting with a digit. Functions share the same pool of names as variables.

The function *body* consists of Octave statements. It is the most important part of the definition, because it says what the function should actually *do*.

For example, here is a function that, when executed, will ring the bell on your terminal (assuming that it is possible to do so):

```
function wakeup
  printf ("\a");
endfunction
```

The `printf` statement (see Chapter 14 [Input and Output], page 231) simply tells Octave to print the string `"\a"`. The special character '`\a`' stands for the alert character (ASCII 7). See Chapter 5 [Strings], page 67.

Once this function is defined, you can ask Octave to evaluate it by typing the name of the function.

Normally, you will want to pass some information to the functions you define. The syntax for passing parameters to a function in Octave is

```
function name (arg-list)
  body
endfunction
```

where *arg-list* is a comma-separated list of the function's arguments. When the function is called, the argument names are used to hold the argument values given in the call. The list of arguments may be empty, in which case this form is equivalent to the one shown above.

To print a message along with ringing the bell, you might modify the `wakeup` to look like this:

```
function wakeup (message)
  printf ("\a%s\n", message);
endfunction
```

Calling this function using a statement like this

```
wakeup ("Rise and shine!");
```

will cause Octave to ring your terminal's bell and print the message '`Rise and shine!`', followed by a newline character (the '`\n`' in the first argument to the `printf` statement).

In most cases, you will also want to get some information back from the functions you define. Here is the syntax for writing a function that returns a single value:

```
function ret-var = name (arg-list)
  body
endfunction
```

The symbol *ret-var* is the name of the variable that will hold the value to be returned by the function. This variable must be defined before the end of the function body in order for the function to return a value.

Variables used in the body of a function are local to the function. Variables named in *arg-list* and *ret-var* are also local to the function. See Section 7.1 [Global Variables], page 124, for information about how to access global variables inside a function.

For example, here is a function that computes the average of the elements of a vector:

```
function retval = avg (v)
  retval = sum (v) / length (v);
endfunction
```

If we had written `avg` like this instead,

```
function retval = avg (v)
  if (isvector (v))
    retval = sum (v) / length (v);
  endif
endfunction
```

and then called the function with a matrix instead of a vector as the argument, Octave would have printed an error message like this:

```
error: value on right hand side of assignment is undefined
```

because the body of the `if` statement was never executed, and `retval` was never defined. To prevent obscure errors like this, it is a good idea to always make sure that the return variables will always have values, and to produce meaningful error messages when problems are encountered. For example, `avg` could have been written like this:

```
function retval = avg (v)
  retval = 0;
  if (isvector (v))
    retval = sum (v) / length (v);
  else
    error ("avg: expecting vector argument");
  endif
endfunction
```

There is still one additional problem with this function. What if it is called without an argument? Without additional error checking, Octave will probably print an error message that won't really help you track down the source of the error. To allow you to catch errors like this, Octave provides each function with an automatic variable called `nargin`. Each time a function is called, `nargin` is automatically initialized to the number of arguments that have actually been passed to the function. For example, we might rewrite the `avg` function like this:

```
function retval = avg (v)
  retval = 0;
  if (nargin != 1)
    usage ("avg (vector)");
  endif
  if (isvector (v))
    retval = sum (v) / length (v);
  else
    error ("avg: expecting vector argument");
  endif
endfunction
```

Although Octave does not automatically report an error if you call a function with more arguments than expected, doing so probably indicates that something is wrong. Octave also does not automatically report an error if a function is called with too few arguments, but any attempt to use a variable that has not been given a value will result in an error. To avoid such problems and to provide useful messages, we check for both possibilities and issue our own error message.

nargin () [Built-in Function]
nargin (*fcn*) [Built-in Function]

Report the number of input arguments to a function.

Called from within a function, return the number of arguments passed to the function. At the top level, return the number of command line arguments passed to Octave.

If called with the optional argument *fcn*—a function name or handle— return the declared number of arguments that the function can accept.

If the last argument to *fcn* is *varargin* the returned value is negative. For example, the function `union` for sets is declared as

```
function [y, ia, ib] = union (a, b, varargin)

and

nargin ("union")
⇒ -3
```

Programming Note: `nargin` does not work on built-in functions.

See also: [nargout], page 176, [narginchk], page 177, [varargin], page 182, [inputname], page 174.

`inputname (`*n*`)` [Function File]

Return the name of the *n*-th argument to the calling function.

If the argument is not a simple variable name, return an empty string. `inputname` may only be used within a function body, not at the command line.

See also: [nargin], page 173, [nthargout], page 175.

val `= silent_functions ()` [Built-in Function]
old_val `= silent_functions (`*new_val*`)` [Built-in Function]
`silent_functions (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that controls whether internal output from a function is suppressed.

If this option is disabled, Octave will display the results produced by evaluating expressions within a function body that are not terminated with a semicolon.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

11.3 Multiple Return Values

Unlike many other computer languages, Octave allows you to define functions that return more than one value. The syntax for defining functions that return multiple values is

```
function [ret-list] = name (arg-list)
  body
endfunction
```

where *name*, *arg-list*, and *body* have the same meaning as before, and *ret-list* is a comma-separated list of variable names that will hold the values returned from the function. The list of return values must have at least one element. If *ret-list* has only one element, this form of the `function` statement is equivalent to the form described in the previous section.

Here is an example of a function that returns two values, the maximum element of a vector and the index of its first occurrence in the vector.

```
function [max, idx] = vmax (v)
  idx = 1;
  max = v (idx);
  for i = 2:length (v)
    if (v (i) > max)
      max = v (i);
      idx = i;
    endif
  endfor
endfunction
```

In this particular case, the two values could have been returned as elements of a single array, but that is not always possible or convenient. The values to be returned may not have compatible dimensions, and it is often desirable to give the individual return values distinct names.

It is possible to use the `nthargout` function to obtain only some of the return values or several at once in a cell array. See Section 3.1.5 [Cell Array Objects], page 44.

`nthargout (`*`n`*`, `*`func`*`, ...)` [Function File]
`nthargout (`*`n`*`, `*`ntot`*`, `*`func`*`, ...)` [Function File]

Return the *n*th output argument of the function specified by the function handle or string *func*.

Any additional arguments are passed directly to *func*. The total number of arguments to call *func* with can be passed in *ntot*; by default *ntot* is *n*. The input *n* can also be a vector of indices of the output, in which case the output will be a cell array of the requested output arguments.

The intended use `nthargout` is to avoid intermediate variables. For example, when finding the indices of the maximum entry of a matrix, the following two compositions of nthargout

```
m = magic (5);
cell2mat (nthargout ([1, 2], @ind2sub, size (m),
                     nthargout (2, @max, m(:))))
⇒ 5   3
```

are completely equivalent to the following lines:

```
m = magic (5);
[~, idx] = max (M(:));
[i, j] = ind2sub (size (m), idx);
[i, j]
⇒ 5   3
```

It can also be helpful to have all output arguments in a single cell in the following manner:

```
USV = nthargout ([1:3], @svd, hilb (5));
```

See also: [nargin], page 173, [nargout], page 176, [varargin], page 182, [varargout], page 182, [isargout], page 184.

In addition to setting `nargin` each time a function is called, Octave also automatically initializes `nargout` to the number of values that are expected to be returned. This allows you to write functions that behave differently depending on the number of values that the user of the function has requested. The implicit assignment to the built-in variable `ans` does not figure in the count of output arguments, so the value of `nargout` may be zero.

The `svd` and `lu` functions are examples of built-in functions that behave differently depending on the value of `nargout`.

It is possible to write functions that only set some return values. For example, calling the function

```
function [x, y, z] = f ()
  x = 1;
  z = 2;
endfunction
```

as

```
[a, b, c] = f ()
```

produces:

```
a = 1

b = [](0x0)

c = 2
```

along with a warning.

`nargout ()` [Built-in Function]
`nargout (`*fcn*`)` [Built-in Function]

Report the number of output arguments from a function.

Called from within a function, return the number of values the caller expects to receive. At the top level, `nargout` with no argument is undefined and will produce an error.

If called with the optional argument *fcn*—a function name or handle—return the number of declared output values that the function can produce.

If the final output argument is *varargout* the returned value is negative.

For example,

```
f ()
```

will cause `nargout` to return 0 inside the function `f` and

```
[s, t] = f ()
```

will cause `nargout` to return 2 inside the function `f`.

In the second usage,

```
nargout (@histc) % or nargout ("histc")
```

will return 2, because `histc` has two outputs, whereas

```
nargout (@imread)
```

will return -2, because `imread` has two outputs and the second is *varargout*.

Programming Note. `nargout` does not work for built-in functions and returns -1 for all anonymous functions.

See also: [nargin], page 173, [varargout], page 182, [isargout], page 184, [nthargout], page 175.

It is good practice at the head of a function to verify that it has been called correctly. In Octave the following idiom is seen frequently

```
if (nargin < min_#_inputs || nargin > max_#_inputs)
  print_usage ();
endif
```

which stops the function execution and prints a message about the correct way to call the function whenever the number of inputs is wrong.

For compatibility with MATLAB, `narginchk` and `nargoutchk` are available which provide similar error checking.

`narginchk` (*minargs*, *maxargs*) [Function File]

Check for correct number of input arguments.

Generate an error message if the number of arguments in the calling function is outside the range *minargs* and *maxargs*. Otherwise, do nothing.

Both *minargs* and *maxargs* must be scalar numeric values. Zero, Inf, and negative values are all allowed, and *minargs* and *maxargs* may be equal.

Note that this function evaluates `nargin` on the caller.

See also: [nargoutchk], page 177, [error], page 205, [nargout], page 176, [nargin], page 173.

`nargoutchk` (*minargs*, *maxargs*) [Function File]
msgstr = `nargoutchk` (*minargs*, *maxargs*, *nargs*) [Function File]
msgstr = `nargoutchk` (*minargs*, *maxargs*, *nargs*, "*string*") [Function File]
msgstruct = `nargoutchk` (*minargs*, *maxargs*, *nargs*, "*struct*") [Function File]

Check for correct number of output arguments.

In the first form, return an error if the number of arguments is not between *minargs* and *maxargs*. Otherwise, do nothing. Note that this function evaluates the value of `nargout` on the caller so its value must have not been tampered with.

Both *minargs* and *maxargs* must be numeric scalars. Zero, Inf, and negative are all valid, and they can have the same value.

For backwards compatibility, the other forms return an appropriate error message string (or structure) if the number of outputs requested is invalid.

This is useful for checking to that the number of output arguments supplied to a function is within an acceptable range.

See also: [narginchk], page 177, [error], page 205, [nargout], page 176, [nargin], page 173.

Besides the number of arguments, inputs can be checked for various properties. `validatestring` is used for string arguments and `validateattributes` for numeric arguments.

validstr = validatestring (*str*, *strarray*) [Function File]
validstr = validatestring (*str*, *strarray*, *funcname*) [Function File]
validstr = validatestring (*str*, *strarray*, *funcname*, *varname*) [Function File]
validstr = validatestring (..., *position*) [Function File]

Verify that *str* is an element, or substring of an element, in *strarray*.

When *str* is a character string to be tested, and *strarray* is a cellstr of valid values, then *validstr* will be the validated form of *str* where validation is defined as *str* being a member or substring of *validstr*. This is useful for both verifying and expanding short options, such as "r", to their longer forms, such as "red". If *str* is a substring of *validstr*, and there are multiple matches, the shortest match will be returned if all matches are substrings of each other. Otherwise, an error will be raised because the expansion of *str* is ambiguous. All comparisons are case insensitive.

The additional inputs *funcname*, *varname*, and *position* are optional and will make any generated validation error message more specific.

Examples:

```
validatestring ("r", {"red", "green", "blue"})
⇒ "red"

validatestring ("b", {"red", "green", "blue", "black"})
⇒ error: validatestring: multiple unique matches were found for 'b':
   blue, black
```

See also: [strcmp], page 76, [strcmpi], page 76, [validateattributes], page 178, [inputParser], page 180.

validateattributes (*A*, *classes*, *attributes*) [Function File]
validateattributes (*A*, *classes*, *attributes*, *arg_idx*) [Function File]
validateattributes (*A*, *classes*, *attributes*, *func_name*) [Function File]
validateattributes (*A*, *classes*, *attributes*, *func_name*, *arg_name*) [Function File]
validateattributes (*A*, *classes*, *attributes*, *func_name*, *arg_name*, *arg_idx*) [Function File]

Check validity of input argument.

Confirms that the argument *A* is valid by belonging to one of *classes*, and holding all of the *attributes*. If it does not, an error is thrown, with a message formatted accordingly. The error message can be made further complete by the function name *fun_name*, the argument name *arg_name*, and its position in the input *arg_idx*.

classes must be a cell array of strings (an empty cell array is allowed) with the name of classes (remember that a class name is case sensitive). In addition to the class name, the following categories names are also valid:

"float" Floating point value comprising classes "double" and "single".

"integer"
Integer value comprising classes (u)int8, (u)int16, (u)int32, (u)int64.

"numeric"
Numeric value comprising either a floating point or integer value.

attributes must be a cell array with names of checks for *A*. Some of them require an additional value to be supplied right after the name (see details for each below).

`"<="` All values are less than or equal to the following value in *attributes*.

`"<"` All values are less than the following value in *attributes*.

`">="` All values are greater than or equal to the following value in *attributes*.

`">"` All values are greater than the following value in *attributes*.

`"2d"` A 2-dimensional matrix. Note that vectors and empty matrices have 2 dimensions, one of them being of length 1, or both length 0.

`"3d"` Has no more than 3 dimensions. A 2-dimensional matrix is a 3-D matrix whose 3rd dimension is of length 1.

`"binary"` All values are either 1 or 0.

`"column"` Values are arranged in a single column.

`"decreasing"`
: No value is *NaN*, and each is less than the preceding one.

`"even"` All values are even numbers.

`"finite"` All values are finite.

`"increasing"`
: No value is *NaN*, and each is greater than the preceding one.

`"integer"`
: All values are integer. This is different than using `isinteger` which only checks its an integer type. This checks that each value in *A* is an integer value, i.e., it has no decimal part.

`"ncols"` Has exactly as many columns as the next value in *attributes*.

`"ndims"` Has exactly as many dimensions as the next value in *attributes*.

`"nondecreasing"`
: No value is *NaN*, and each is greater than or equal to the preceding one.

`"nonempty"`
: It is not empty.

`"nonincreasing"`
: No value is *NaN*, and each is less than or equal to the preceding one.

`"nonnan"` No value is a `NaN`.

`"nonnegative"`
: All values are non negative.

`"nonsparse"`
: It is not a sparse matrix.

`"nonzero"`
: No value is zero.

`"nrows"` Has exactly as many rows as the next value in *attributes*.

`"numel"` Has exactly as many elements as the next value in *attributes*.

`"odd"` All values are odd numbers.

`"positive"`
All values are positive.

`"real"` It is a non-complex matrix.

`"row"` Values are arranged in a single row.

`"scalar"` It is a scalar.

`"size"` Its size has length equal to the values of the next in *attributes*. The next value must is an array with the length for each dimension. To ignore the check for a certain dimension, the value of `NaN` can be used.

`"square"` Is a square matrix.

`"vector"` Values are arranged in a single vector (column or vector).

See also: [isa], page 39, [validatestring], page 177, [inputParser], page 180.

If none of the preceding functions is sufficient there is also the class `inputParser` which can perform extremely complex input checking for functions.

p = **inputParser** () [Function File]

Create object *p* of the inputParser class.

This class is designed to allow easy parsing of function arguments. The class supports four types of arguments:

1. mandatory (see `addRequired`);
2. optional (see `addOptional`);
3. named (see `addParamValue`);
4. switch (see `addSwitch`).

After defining the function API with these methods, the supplied arguments can be parsed with the `parse` method and the parsing results accessed with the `Results` accessor.

inputParser.Parameters [Accessor method]

Return list of parameter names already defined.

inputParser.Results [Accessor method]

Return structure with argument names as fieldnames and corresponding values.

inputParser.Unmatched [Accessor method]

Return structure similar to `Results`, but for unmatched parameters. See the `KeepUnmatched` property.

inputParser.UsingDefaults [Accessor method]

Return cell array with the names of arguments that are using default values.

`inputParser.CaseSensitive` = *boolean* [Class property]

Set whether matching of argument names should be case sensitive. Defaults to false.

`inputParser.FunctionName` = *name* [Class property]

Set function name to be used in error messages; Defaults to empty string.

`inputParser.KeepUnmatched` = *boolean* [Class property]

Set whether an error should be given for non-defined arguments. Defaults to false. If set to true, the extra arguments can be accessed through `Unmatched` after the `parse` method. Note that since `Switch` and `ParamValue` arguments can be mixed, it is not possible to know the unmatched type. If argument is found unmatched it is assumed to be of the `ParamValue` type and it is expected to be followed by a value.

`inputParser.StructExpand` = *boolean* [Class property]

Set whether a structure can be passed to the function instead of parameter/value pairs. Defaults to true. Not implemented yet.

The following example shows how to use this class:

```
function check (varargin)
  p = inputParser ();                      # create object
  p.FunctionName = "check";                # set function name
  p.addRequired ("pack", @ischar);         # mandatory argument
  p.addOptional ("path", pwd(), @ischar);  # optional argument

  ## create a function handle to anonymous functions for validators
  val_mat = @(x) isvector (x) && all (x <= 1) && all (x >= 0);
  p.addOptional ("mat", [0 0], val_mat);

  ## create two arguments of type "ParamValue"
  val_type = @(x) any (strcmp (x, {"linear", "quadratic"}));
  p.addParamValue ("type", "linear", val_type);
  val_verb = @(x) any (strcmp (x, {"low", "medium", "high"}));
  p.addParamValue ("tolerance", "low", val_verb);

  ## create a switch type of argument
  p.addSwitch ("verbose");

  p.parse (varargin{:});  # Run created parser on inputs

  ## the rest of the function can access inputs by using p.Results.
  ## for example, get the tolerance input with p.Results.tolerance
endfunction

check ("mech");           # valid, use defaults for other arguments
check ();                 # error, one argument is mandatory
check (1);                # error, since ! ischar
check ("mech", "~/dev");  # valid, use defaults for other arguments
```

```
check ("mech", "~/dev", [0 1 0 0], "type", "linear");  # valid

## following is also valid.  Note how the Switch argument type can
## be mixed into or before the ParamValue argument type (but it
## must still appear after any Optional argument).
check ("mech", "~/dev", [0 1 0 0], "verbose", "tolerance", "high");

## following returns an error since not all optional arguments,
## ‘path’ and ‘mat’, were given before the named argument ‘type’.
check ("mech", "~/dev", "type", "linear");
```

Note 1: A function can have any mixture of the four API types but they must appear in a specific order. `Required` arguments must be first and can be followed by any `Optional` arguments. Only the `ParamValue` and `Switch` arguments may be mixed together and they must appear at the end.

Note 2: If both `Optional` and `ParamValue` arguments are mixed in a function API then once a string Optional argument fails to validate it will be considered the end of the `Optional` arguments. The remaining arguments will be compared against any `ParamValue` or `Switch` arguments.

See also: [nargin], page 173, [validateattributes], page 178, [validatestring], page 177, [varargin], page 182.

11.4 Variable-length Argument Lists

Sometimes the number of input arguments is not known when the function is defined. As an example think of a function that returns the smallest of all its input arguments. For example:

```
a = smallest (1, 2, 3);
b = smallest (1, 2, 3, 4);
```

In this example both `a` and `b` would be 1. One way to write the `smallest` function is

```
function val = smallest (arg1, arg2, arg3, arg4, arg5)
  body
endfunction
```

and then use the value of `nargin` to determine which of the input arguments should be considered. The problem with this approach is that it can only handle a limited number of input arguments.

If the special parameter name `varargin` appears at the end of a function parameter list it indicates that the function takes a variable number of input arguments. Using `varargin` the function looks like this

```
function val = smallest (varargin)
  body
endfunction
```

In the function body the input arguments can be accessed through the variable `varargin`. This variable is a cell array containing all the input arguments. See Section 6.2 [Cell Arrays], page 112, for details on working with cell arrays. The `smallest` function can now be defined like this

```
function val = smallest (varargin)
  val = min ([varargin{:}]);
endfunction
```

This implementation handles any number of input arguments, but it's also a very simple solution to the problem.

A slightly more complex example of `varargin` is a function `print_arguments` that prints all input arguments. Such a function can be defined like this

```
function print_arguments (varargin)
  for i = 1:length (varargin)
    printf ("Input argument %d: ", i);
    disp (varargin{i});
  endfor
endfunction
```

This function produces output like this

```
print_arguments (1, "two", 3);
     ⊣ Input argument 1:  1
     ⊣ Input argument 2: two
     ⊣ Input argument 3:  3
```

[*reg*, *prop*] = parseparams (*params*) [Function File]
[*reg*, *var1*, ...] = parseparams (*params*, *name1*, *default1*, ...) [Function File]

Return in *reg* the cell elements of *param* up to the first string element and in *prop* all remaining elements beginning with the first string element.

For example:

```
[reg, prop] = parseparams ({1, 2, "linewidth", 10})
reg =
{
  [1,1] = 1
  [1,2] = 2
}
prop =
{
  [1,1] = linewidth
  [1,2] = 10
}
```

The parseparams function may be used to separate regular numeric arguments from additional arguments given as property/value pairs of the *varargin* cell array.

In the second form of the call, available options are specified directly with their default values given as name-value pairs. If *params* do not form name-value pairs, or if an option occurs that does not match any of the available options, an error occurs.

When called from an m-file function, the error is prefixed with the name of the caller function.

The matching of options is case-insensitive.

See also: [varargin], page 182, [inputParser], page 180.

11.5 Ignoring Arguments

In the formal argument list, it is possible to use the dummy placeholder ~ instead of a name. This indicates that the corresponding argument value should be ignored and not stored to any variable.

```
function val = pick2nd (~, arg2)
  val = arg2;
endfunction
```

The value of `nargin` is not affected by using this declaration.

Return arguments can also be ignored using the same syntax. Functions may take advantage of ignored outputs to reduce the number of calculations performed. To do so, use the `isargout` function to query whether the output argument is wanted. For example:

```
function [out1, out2] = long_function (x, y, z)
  if (isargout (1))
    ## Long calculation
    ...
    out1 = result;
  endif
  ...
endfunction
```

`isargout` (*k*) [Built-in Function]

Within a function, return a logical value indicating whether the argument *k* will be assigned to a variable on output.

If the result is false, the argument has been ignored during the function call through the use of the tilde (~) special output argument. Functions can use `isargout` to avoid performing unnecessary calculations for outputs which are unwanted.

If *k* is outside the range `1:max (nargout)`, the function returns false. *k* can also be an array, in which case the function works element-by-element and a logical array is returned. At the top level, `isargout` returns an error.

See also: [nargout], page 176, [varargout], page 182, [nthargout], page 175.

11.6 Variable-length Return Lists

It is possible to return a variable number of output arguments from a function using a syntax that's similar to the one used with the special `varargin` parameter name. To let a function return a variable number of output arguments the special output parameter name `varargout` is used. As with `varargin`, `varargout` is a cell array that will contain the requested output arguments.

As an example the following function sets the first output argument to 1, the second to 2, and so on.

```
function varargout = one_to_n ()
  for i = 1:nargout
    varargout{i} = i;
  endfor
endfunction
```

When called this function returns values like this

```
[a, b, c] = one_to_n ()
     ⇒ a =  1
     ⇒ b =  2
     ⇒ c =  3
```

If `varargin` (`varargout`) does not appear as the last element of the input (output) parameter list, then it is not special, and is handled the same as any other parameter name.

`[r1, r2, ..., rn] = deal (a)` [Function File]
`[r1, r2, ..., rn] = deal (a1, a2, ..., an)` [Function File]

Copy the input parameters into the corresponding output parameters.

If only a single input parameter is supplied, its value is copied to each of the outputs.

For example,

```
[a, b, c] = deal (x, y, z);
```

is equivalent to

```
a = x;
b = y;
c = z;
```

and

```
[a, b, c] = deal (x);
```

is equivalent to

```
a = b = c = x;
```

Programming Note: `deal` is often used with comma separated lists derived from cell arrays or structures. This is unnecessary as the interpreter can perform the same action without the overhead of a function call. For example:

```
c = {[1 2], "Three", 4};
[x, y, z ] = c{:}
⇒
   x =

      1   2

   y = Three
   z =  4
```

See also: [cell2struct], page 120, [struct2cell], page 111, [repmat], page 420.

11.7 Returning from a Function

The body of a user-defined function can contain a `return` statement. This statement returns control to the rest of the Octave program. It looks like this:

```
return
```

Unlike the `return` statement in C, Octave's `return` statement cannot be used to return a value from a function. Instead, you must assign values to the list of return variables that are part of the `function` statement. The `return` statement simply makes it easier to exit a function from a deeply nested loop or conditional statement.

Here is an example of a function that checks to see if any elements of a vector are nonzero.

```
function retval = any_nonzero (v)
  retval = 0;
  for i = 1:length (v)
    if (v (i) != 0)
      retval = 1;
      return;
    endif
  endfor
  printf ("no nonzero elements found\n");
endfunction
```

Note that this function could not have been written using the `break` statement to exit the loop once a nonzero value is found without adding extra logic to avoid printing the message if the vector does contain a nonzero element.

`return` [Keyword]

When Octave encounters the keyword `return` inside a function or script, it returns control to the caller immediately. At the top level, the return statement is ignored. A `return` statement is assumed at the end of every function definition.

11.8 Default Arguments

Since Octave supports variable number of input arguments, it is very useful to assign default values to some input arguments. When an input argument is declared in the argument list it is possible to assign a default value to the argument like this

```
function name (arg1 = val1, ...)
  body
endfunction
```

If no value is assigned to *arg1* by the user, it will have the value *val1*.

As an example, the following function implements a variant of the classic "Hello, World" program.

```
function hello (who = "World")
  printf ("Hello, %s!\n", who);
endfunction
```

When called without an input argument the function prints the following

```
hello ();
     ⊣ Hello, World!
```

and when it's called with an input argument it prints the following

```
hello ("Beautiful World of Free Software");
     ⊣ Hello, Beautiful World of Free Software!
```

Sometimes it is useful to explicitly tell Octave to use the default value of an input argument. This can be done writing a '`:`' as the value of the input argument when calling the function.

```
hello (:);
     ⊣ Hello, World!
```

11.9 Function Files

Except for simple one-shot programs, it is not practical to have to define all the functions you need each time you need them. Instead, you will normally want to save them in a file so that you can easily edit them, and save them for use at a later time.

Octave does not require you to load function definitions from files before using them. You simply need to put the function definitions in a place where Octave can find them.

When Octave encounters an identifier that is undefined, it first looks for variables or functions that are already compiled and currently listed in its symbol table. If it fails to find a definition there, it searches a list of directories (the *path*) for files ending in `.m` that have the same base name as the undefined identifier.[1] Once Octave finds a file with a name that matches, the contents of the file are read. If it defines a *single* function, it is compiled and executed. See Section 11.10 [Script Files], page 198, for more information about how you can define more than one function in a single file.

When Octave defines a function from a function file, it saves the full name of the file it read and the time stamp on the file. If the time stamp on the file changes, Octave may reload the file. When Octave is running interactively, time stamp checking normally happens at most once each time Octave prints the prompt. Searching for new function definitions also occurs if the current working directory changes.

Checking the time stamp allows you to edit the definition of a function while Octave is running, and automatically use the new function definition without having to restart your Octave session.

To avoid degrading performance unnecessarily by checking the time stamps on functions that are not likely to change, Octave assumes that function files in the directory tree *octave-home*`/share/octave/`*version*`/m` will not change, so it doesn't have to check their time stamps every time the functions defined in those files are used. This is normally a very good assumption and provides a significant improvement in performance for the function files that are distributed with Octave.

If you know that your own function files will not change while you are running Octave, you can improve performance by calling `ignore_function_time_stamp ("all")`, so that Octave will ignore the time stamps for all function files. Passing `"system"` to this function resets the default behavior.

`edit` *name* [Command]
`edit` *field value* [Command]
value `= edit` *get field* [Command]

Edit the named function, or change editor settings.

If `edit` is called with the name of a file or function as its argument it will be opened in the text editor defined by `EDITOR`.

- If the function *name* is available in a file on your path and that file is modifiable, then it will be edited in place. If it is a system function, then it will first be

[1] The '`.m`' suffix was chosen for compatibility with MATLAB.

copied to the directory `HOME` (see below) and then edited. If no file is found, then the m-file variant, ending with `".m"`, will be considered. If still no file is found, then variants with a leading `"@"` and then with both a leading `"@"` and trailing `".m"` will be considered.

- If *name* is the name of a function defined in the interpreter but not in an m-file, then an m-file will be created in `HOME` to contain that function along with its current definition.
- If *name*`.cc` is specified, then it will search for *name*`.cc` in the path and try to modify it, otherwise it will create a new `.cc` file in the current directory. If *name* happens to be an m-file or interpreter defined function, then the text of that function will be inserted into the .cc file as a comment.
- If *name*`.ext` is on your path then it will be edited, otherwise the editor will be started with *name*`.ext` in the current directory as the filename. If *name*`.ext` is not modifiable, it will be copied to `HOME` before editing.

  Warning: You may need to clear *name* before the new definition is available. If you are editing a .cc file, you will need to execute `mkoctfile` *name*`.cc` before the definition will be available.

If `edit` is called with *field* and *value* variables, the value of the control field *field* will be set to *value*. If an output argument is requested and the first input argument is `get` then `edit` will return the value of the control field *field*. If the control field does not exist, edit will return a structure containing all fields and values. Thus, `edit get all` returns a complete control structure.

The following control fields are used:

'`home`' This is the location of user local m-files. Be sure it is in your path. The default is `~/octave`.

'`author`' This is the name to put after the "## Author:" field of new functions. By default it guesses from the `gecos` field of the password database.

'`email`' This is the e-mail address to list after the name in the author field. By default it guesses `<$LOGNAME@$HOSTNAME>`, and if `$HOSTNAME` is not defined it uses `uname -n`. You probably want to override this. Be sure to use the format `user@host`.

'`license`'

'`gpl`' GNU General Public License (default).

'`bsd`' BSD-style license without advertising clause.

'`pd`' Public domain.

'`"text"`' Your own default copyright and license.

Unless you specify '`pd`', edit will prepend the copyright statement with "Copyright (C) yyyy Function Author".

'`mode`' This value determines whether the editor should be started in async mode (editor is started in the background and Octave continues) or sync mode (Octave waits until the editor exits). Set it to `"sync"` to start the editor in sync mode. The default is `"async"` (see [system], page 775).

'`editinplace`'
: Determines whether files should be edited in place, without regard to whether they are modifiable or not. The default is `false`.

`mfilename ()` [Built-in Function]
`mfilename ("`*fullpath*`")` [Built-in Function]
`mfilename ("`*fullpathext*`")` [Built-in Function]

Return the name of the currently executing file.

When called from outside an m-file return the empty string.

Given the argument `"fullpath"`, include the directory part of the file name, but not the extension.

Given the argument `"fullpathext"`, include the directory part of the file name and the extension.

val `= ignore_function_time_stamp ()` [Built-in Function]
old_val `= ignore_function_time_stamp (`*new_val*`)` [Built-in Function]

Query or set the internal variable that controls whether Octave checks the time stamp on files each time it looks up functions defined in function files.

If the internal variable is set to `"system"`, Octave will not automatically recompile function files in subdirectories of *octave-home*`/lib/`*version* if they have changed since they were last compiled, but will recompile other function files in the search path if they change.

If set to `"all"`, Octave will not recompile any function files unless their definitions are removed with `clear`.

If set to `"none"`, Octave will always check time stamps on files to determine whether functions defined in function files need to recompiled.

11.9.1 Manipulating the Load Path

When a function is called, Octave searches a list of directories for a file that contains the function declaration. This list of directories is known as the load path. By default the load path contains a list of directories distributed with Octave plus the current working directory. To see your current load path call the `path` function without any input or output arguments.

It is possible to add or remove directories to or from the load path using `addpath` and `rmpath`. As an example, the following code adds '`~/Octave`' to the load path.

```
addpath ("~/Octave")
```

After this the directory '`~/Octave`' will be searched for functions.

`addpath (`*dir1*`, ...)` [Built-in Function]
`addpath (`*dir1*`, ..., `*option*`)` [Built-in Function]

Add named directories to the function search path.

If *option* is `"-begin"` or 0 (the default), prepend the directory name to the current path. If *option* is `"-end"` or 1, append the directory name to the current path. Directories added to the path must exist.

In addition to accepting individual directory arguments, lists of directory names separated by `pathsep` are also accepted. For example:

```
addpath ("dir1:/dir2:~/dir3")
```

See also: [path], page 190, [rmpath], page 190, [genpath], page 190, [pathdef], page 191, [savepath], page 190, [pathsep], page 191.

`genpath (`*`dir`*`)` [Built-in Function]
`genpath (`*`dir`*`, `*`skip`*`, ...)` [Built-in Function]

Return a path constructed from *dir* and all its subdirectories.

If additional string parameters are given, the resulting path will exclude directories with those names.

`rmpath (`*`dir1`*`, ...)` [Built-in Function]

Remove *dir1*, ... from the current function search path.

In addition to accepting individual directory arguments, lists of directory names separated by `pathsep` are also accepted. For example:

```
rmpath ("dir1:/dir2:~/dir3")
```

See also: [path], page 190, [addpath], page 189, [genpath], page 190, [pathdef], page 191, [savepath], page 190, [pathsep], page 191.

`savepath ()` [Function File]
`savepath (`*`file`*`)` [Function File]
`status` `= savepath (...)` [Function File]

Save the unique portion of the current function search path that is not set during Octave's initialization process to *file*.

If *file* is omitted, Octave looks in the current directory for a project-specific `.octaverc` file in which to save the path information. If no such file is present then the user's configuration file `~/.octaverc` is used.

If successful, `savepath` returns 0.

The `savepath` function makes it simple to customize a user's configuration file to restore the working paths necessary for a particular instance of Octave. Assuming no filename is specified, Octave will automatically restore the saved directory paths from the appropriate `.octaverc` file when starting up. If a filename has been specified then the paths may be restored manually by calling `source` *`file`*.

See also: [path], page 190, [addpath], page 189, [rmpath], page 190, [genpath], page 190, [pathdef], page 191.

`path (...)` [Built-in Function]

Modify or display Octave's load path.

If *nargin* and *nargout* are zero, display the elements of Octave's load path in an easy to read format.

If *nargin* is zero and nargout is greater than zero, return the current load path.

If *nargin* is greater than zero, concatenate the arguments, separating them with `pathsep`. Set the internal search path to the result and return it.

No checks are made for duplicate elements.

See also: [addpath], page 189, [rmpath], page 190, [genpath], page 190, [pathdef], page 191, [savepath], page 190, [pathsep], page 191.

val = `pathdef ()` [Function File]

Return the default path for Octave.

The path information is extracted from one of four sources. The possible sources, in order of preference, are:

1. `.octaverc`
2. `~/.octaverc`
3. `<OCTAVE_HOME>/.../<version>/m/startup/octaverc`
4. Octave's path prior to changes by any octaverc file.

See also: [path], page 190, [addpath], page 189, [rmpath], page 190, [genpath], page 190, [savepath], page 190.

val = `pathsep ()` [Built-in Function]
old_val = `pathsep (`*new_val*`)` [Built-in Function]

Query or set the character used to separate directories in a path.

See also: [filesep], page 767.

`rehash ()` [Built-in Function]

Reinitialize Octave's load path directory cache.

`file_in_loadpath (`*file*`)` [Built-in Function]
`file_in_loadpath (`*file*`, "`*all*`")` [Built-in Function]

Return the absolute name of *file* if it can be found in the list of directories specified by `path`.

If no file is found, return an empty character string.

If the first argument is a cell array of strings, search each directory of the loadpath for element of the cell array and return the first that matches.

If the second optional argument `"all"` is supplied, return a cell array containing the list of all files that have the same name in the path. If no files are found, return an empty cell array.

See also: [file_in_path], page 766, [dir_in_loadpath], page 192, [path], page 190.

`restoredefaultpath (...)` [Built-in Function]

Restore Octave's path to its initial state at startup.

See also: [path], page 190, [addpath], page 189, [rmpath], page 190, [genpath], page 190, [pathdef], page 191, [savepath], page 190, [pathsep], page 191.

`command_line_path (...)` [Built-in Function]

Return the command line path variable.

See also: [path], page 190, [addpath], page 189, [rmpath], page 190, [genpath], page 190, [pathdef], page 191, [savepath], page 190, [pathsep], page 191.

`dir_in_loadpath (dir)` [Built-in Function]
`dir_in_loadpath (dir, "all")` [Built-in Function]

Return the full name of the path element matching *dir*.

The match is performed at the end of each path element. For example, if *dir* is `"foo/bar"`, it matches the path element `"/some/dir/foo/bar"`, but not `"/some/dir/foo/bar/baz"` `"/some/dir/allfoo/bar"`.

If the optional second argument is supplied, return a cell array containing all name matches rather than just the first.

See also: [file_in_path], page 766, [file_in_loadpath], page 191, [path], page 190.

11.9.2 Subfunctions

A function file may contain secondary functions called *subfunctions*. These secondary functions are only visible to the other functions in the same function file. For example, a file `f.m` containing

```
function f ()
  printf ("in f, calling g\n");
  g ()
endfunction
function g ()
  printf ("in g, calling h\n");
  h ()
endfunction
function h ()
  printf ("in h\n")
endfunction
```

defines a main function `f` and two subfunctions. The subfunctions `g` and `h` may only be called from the main function `f` or from the other subfunctions, but not from outside the file `f.m`.

11.9.3 Private Functions

In many cases one function needs to access one or more helper functions. If the helper function is limited to the scope of a single function, then subfunctions as discussed above might be used. However, if a single helper function is used by more than one function, then this is no longer possible. In this case the helper functions might be placed in a subdirectory, called `"private"`, of the directory in which the functions needing access to this helper function are found.

As a simple example, consider a function `func1`, that calls a helper function `func2` to do much of the work. For example:

```
function y = func1 (x)
  y = func2 (x);
endfunction
```

Then if the path to `func1` is `<directory>/func1.m`, and if `func2` is found in the directory `<directory>/private/func2.m`, then `func2` is only available for use of the functions, like `func1`, that are found in `<directory>`.

11.9.4 Nested Functions

Nested functions are similar to subfunctions in that only the main function is visible outside the file. However, they also allow for child functions to access the local variables in their parent function. This shared access mimics using a global variable to share information — but a global variable which is not visible to the rest of Octave. As a programming strategy, sharing data this way can create code which is difficult to maintain. It is recommended to use subfunctions in place of nested functions when possible.

As a simple example, consider a parent function `foo`, that calls a nested child function `bar`, with a shared variable *x*.

```
function y = foo ()
  x = 10;
  bar ();
  y = x;

  function bar ()
    x = 20;
  endfunction
endfunction

foo ()
 ⇒ 20
```

Notice that there is no special syntax for sharing *x*. This can lead to problems with accidental variable sharing between a parent function and its child. While normally variables are inherited, child function parameters and return values are local to the child function.

Now consider the function `foobar` that uses variables *x* and *y*. `foobar` calls a nested function `foo` which takes *x* as a parameter and returns *y*. `foo` then calls `bat` which does some computation.

```
function z = foobar ()
  x = 0;
  y = 0;
  z = foo (5);
  z += x + y;

  function y = foo (x)
    y = x + bat ();

    function z = bat ()
      z = x;
    endfunction
  endfunction
endfunction

foobar ()
    ⇒ 10
```

It is important to note that the *x* and *y* in `foobar` remain zero, as in `foo` they are a return value and parameter respectively. The *x* in `bat` refers to the *x* in `foo`.

Variable inheritance leads to a problem for `eval` and scripts. If a new variable is created in a parent function, it is not clear what should happen in nested child functions. For example, consider a parent function `foo` with a nested child function `bar`:

```
function y = foo (to_eval)
  bar ();
  eval (to_eval);

  function bar ()
    eval ("x = 100;");
    eval ("y = x;");
  endfunction
endfunction

foo ("x = 5;")
    ⇒ error: can not add variable "x" to a static workspace

foo ("y = 10;")
    ⇒ 10

foo ("")
    ⇒ 100
```

The parent function `foo` is unable to create a new variable *x*, but the child function `bar` was successful. Furthermore, even in an `eval` statement *y* in `bar` is the same *y* as in its parent function `foo`. The use of `eval` in conjunction with nested functions is best avoided.

As with subfunctions, only the first nested function in a file may be called from the outside. Inside a function the rules are more complicated. In general a nested function may call:

0. Globally visible functions
1. Any function that the nested function's parent can call
2. Sibling functions (functions that have the same parents)
3. Direct children

As a complex example consider a parent function `ex_top` with two child functions, `ex_a` and `ex_b`. In addition, `ex_a` has two more child functions, `ex_aa` and `ex_ab`. For example:

```
function ex_top ()
  ## Can call: ex_top, ex_a, and ex_b
  ## Can NOT call: ex_aa and ex_ab

  function ex_a ()
    ## Call call everything

    function ex_aa ()
      ## Can call everything
```

```
    endfunction

    function ex_ab ()
      ## Can call everything
    endfunction
  endfunction

  function ex_b ()
    ## Can call: ex_top, ex_a, and ex_b
    ## Can NOT call: ex_aa and ex_ab
  endfunction
endfunction
```

11.9.5 Overloading and Autoloading

Functions can be overloaded to work with different input arguments. For example, the operator '+' has been overloaded in Octave to work with single, double, uint8, int32, and many other arguments. The preferred way to overload functions is through classes and object oriented programming (see Section 34.4.1 [Function Overloading], page 731). Occasionally, however, one needs to undo user overloading and call the default function associated with a specific type. The `builtin` function exists for this purpose.

`[...] = builtin (f, ...)` [Built-in Function]

Call the base function *f* even if *f* is overloaded to another function for the given type signature.

This is normally useful when doing object-oriented programming and there is a requirement to call one of Octave's base functions rather than the overloaded one of a new class.

A trivial example which redefines the `sin` function to be the `cos` function shows how `builtin` works.

```
sin (0)
  ⇒ 0
function y = sin (x), y = cos (x); endfunction
sin (0)
  ⇒ 1
builtin ("sin", 0)
  ⇒ 0
```

A single dynamically linked file might define several functions. However, as Octave searches for functions based on the functions filename, Octave needs a manner in which to find each of the functions in the dynamically linked file. On operating systems that support symbolic links, it is possible to create a symbolic link to the original file for each of the functions which it contains.

However, there is at least one well known operating system that doesn't support symbolic links. Making copies of the original file for each of the functions is undesirable as it increases the amount of disk space used by Octave. Instead Octave supplies the `autoload` function, that permits the user to define in which file a certain function will be found.

autoload_map = autoload () [Built-in Function]
autoload (*function*, *file*) [Built-in Function]
autoload (..., "*remove*") [Built-in Function]

Define *function* to autoload from *file*.

The second argument, *file*, should be an absolute file name or a file name in the same directory as the function or script from which the autoload command was run. *file should not* depend on the Octave load path.

Normally, calls to `autoload` appear in PKG_ADD script files that are evaluated when a directory is added to Octave's load path. To avoid having to hardcode directory names in *file*, if *file* is in the same directory as the PKG_ADD script then

```
autoload ("foo", "bar.oct");
```

will load the function `foo` from the file `bar.oct`. The above usage when `bar.oct` is not in the same directory, or usages such as

```
autoload ("foo", file_in_loadpath ("bar.oct"))
```

are strongly discouraged, as their behavior may be unpredictable.

With no arguments, return a structure containing the current autoload map.

If a third argument `"remove"` is given, the function is cleared and not loaded anymore during the current Octave session.

See also: [PKG_ADD], page 812.

11.9.6 Function Locking

It is sometime desirable to lock a function into memory with the `mlock` function. This is typically used for dynamically linked functions in Oct-files or mex-files that contain some initialization, and it is desirable that calling `clear` does not remove this initialization.

As an example,

```
function my_function ()
  mlock ();
  ...
```

prevents `my_function` from being removed from memory after it is called, even if `clear` is called. It is possible to determine if a function is locked into memory with the `mislocked`, and to unlock a function with `munlock`, which the following illustrates.

```
my_function ();
mislocked ("my_function")
⇒ ans = 1
munlock ("my_function");
mislocked ("my_function")
⇒ ans = 0
```

A common use of `mlock` is to prevent persistent variables from being removed from memory, as the following example shows:

```
function count_calls ()
  mlock ();
  persistent calls = 0;
  printf ("'count_calls' has been called %d times\n",
          ++calls);
endfunction

count_calls ();
⊣ 'count_calls' has been called 1 times

clear count_calls
count_calls ();
⊣ 'count_calls' has been called 2 times
```

`mlock` might equally be used to prevent changes to a function from having effect in Octave, though a similar effect can be had with the `ignore_function_time_stamp` function.

`mlock ()` [Built-in Function]

Lock the current function into memory so that it can't be cleared.

See also: [munlock], page 197, [mislocked], page 197, [persistent], page 126.

`munlock ()` [Built-in Function]
`munlock (`*fcn*`)` [Built-in Function]

Unlock the named function *fcn*.

If no function is named then unlock the current function.

See also: [mlock], page 197, [mislocked], page 197, [persistent], page 126.

`mislocked ()` [Built-in Function]
`mislocked (`*fcn*`)` [Built-in Function]

Return true if the named function *fcn* is locked.

If no function is named then return true if the current function is locked.

See also: [mlock], page 197, [munlock], page 197, [persistent], page 126.

11.9.7 Function Precedence

Given the numerous different ways that Octave can define a function, it is possible and even likely that multiple versions of a function, might be defined within a particular scope. The precedence of which function will be used within a particular scope is given by

1. Subfunction A subfunction with the required function name in the given scope.
2. Private function A function defined within a private directory of the directory which contains the current function.
3. Class constructor A function that constructs a user class as defined in chapter Chapter 34 [Object Oriented Programming], page 721.
4. Class method An overloaded function of a class as in chapter Chapter 34 [Object Oriented Programming], page 721.
5. Command-line Function A function that has been defined on the command-line.

6. Autoload function A function that is marked as autoloaded with See [autoload], page 195.
7. A Function on the Path A function that can be found on the users load-path. There can also be Oct-file, mex-file or m-file versions of this function and the precedence between these versions are in that order.
8. Built-in function A function that is a part of core Octave such as `numel`, `size`, etc.

11.10 Script Files

A script file is a file containing (almost) any sequence of Octave commands. It is read and evaluated just as if you had typed each command at the Octave prompt, and provides a convenient way to perform a sequence of commands that do not logically belong inside a function.

Unlike a function file, a script file must *not* begin with the keyword `function`. If it does, Octave will assume that it is a function file, and that it defines a single function that should be evaluated as soon as it is defined.

A script file also differs from a function file in that the variables named in a script file are not local variables, but are in the same scope as the other variables that are visible on the command line.

Even though a script file may not begin with the `function` keyword, it is possible to define more than one function in a single script file and load (but not execute) all of them at once. To do this, the first token in the file (ignoring comments and other white space) must be something other than `function`. If you have no other statements to evaluate, you can use a statement that has no effect, like this:

```
# Prevent Octave from thinking that this
# is a function file:

1;

# Define function one:

function one ()
  ...
```

To have Octave read and compile these functions into an internal form, you need to make sure that the file is in Octave's load path (accessible through the `path` function), then simply type the base name of the file that contains the commands. (Octave uses the same rules to search for script files as it does to search for function files.)

If the first token in a file (ignoring comments) is `function`, Octave will compile the function and try to execute it, printing a message warning about any non-whitespace characters that appear after the function definition.

Note that Octave does not try to look up the definition of any identifier until it needs to evaluate it. This means that Octave will compile the following statements if they appear in a script file, or are typed at the command line,

```
# not a function file:
1;
function foo ()
  do_something ();
endfunction
function do_something ()
  do_something_else ();
endfunction
```

even though the function `do_something` is not defined before it is referenced in the function `foo`. This is not an error because Octave does not need to resolve all symbols that are referenced by a function until the function is actually evaluated.

Since Octave doesn't look for definitions until they are needed, the following code will always print '`bar = 3`' whether it is typed directly on the command line, read from a script file, or is part of a function body, even if there is a function or script file called `bar.m` in Octave's path.

```
eval ("bar = 3");
bar
```

Code like this appearing within a function body could fool Octave if definitions were resolved as the function was being compiled. It would be virtually impossible to make Octave clever enough to evaluate this code in a consistent fashion. The parser would have to be able to perform the call to `eval` at compile time, and that would be impossible unless all the references in the string to be evaluated could also be resolved, and requiring that would be too restrictive (the string might come from user input, or depend on things that are not known until the function is evaluated).

Although Octave normally executes commands from script files that have the name *file*`.m`, you can use the function `source` to execute commands from any file.

`source` (*file*) [Built-in Function]

Parse and execute the contents of *file*.

This is equivalent to executing commands from a script file, but without requiring the file to be named *file*`.m`.

See also: [run], page 156.

11.11 Function Handles, Anonymous Functions, Inline Functions

It can be very convenient store a function in a variable so that it can be passed to a different function. For example, a function that performs numerical minimization needs access to the function that should be minimized.

11.11.1 Function Handles

A function handle is a pointer to another function and is defined with the syntax

```
@function-name
```

For example,

```
f = @sin;
```

creates a function handle called `f` that refers to the function `sin`.

Function handles are used to call other functions indirectly, or to pass a function as an argument to another function like `quad` or `fsolve`. For example:

```
f = @sin;
quad (f, 0, pi)
    ⇒ 2
```

You may use `feval` to call a function using function handle, or simply write the name of the function handle followed by an argument list. If there are no arguments, you must use an empty argument list '`()`'. For example:

```
f = @sin;
feval (f, pi/4)
    ⇒ 0.70711
f (pi/4)
    ⇒ 0.70711
```

`is_function_handle` (*x*) [Built-in Function]

Return true if *x* is a function handle.

See also: [isa], page 39, [typeinfo], page 39, [class], page 39, [functions], page 200.

s = `functions` (*fcn_handle*) [Built-in Function]

Return a structure containing information about the function handle *fcn_handle*.

The structure *s* always contains these three fields:

function — The function name. For an anonymous function (no name) this will be the actual function definition.

type — Type of the function.

- anonymous — The function is anonymous.
- private — The function is private.
- overloaded — The function overloads an existing function.
- simple — The function is a built-in or m-file function.
- subfunction — The function is a subfunction within an m-file.

file — The m-file that will be called to perform the function. This field is empty for anonymous and built-in functions.

In addition, some function types may return more information in additional fields.

Warning: `functions` is provided for debugging purposes only. It's behavior may change in the future and programs should not depend on a particular output.

`func2str` (*fcn_handle*) [Built-in Function]

Return a string containing the name of the function referenced by the function handle *fcn_handle*.

See also: [str2func], page 201, [functions], page 200.

`str2func (fcn_name)` [Built-in Function]
`str2func (fcn_name, "global")` [Built-in Function]

Return a function handle constructed from the string *fcn_name*.

If the optional `"global"` argument is passed, locally visible functions are ignored in the lookup.

Note: `str2func` does not currently accept strings which define anonymous functions (those which begin with '`@`'). Use `eval (str)` as a replacement.

See also: [func2str], page 200, [inline], page 202.

11.11.2 Anonymous Functions

Anonymous functions are defined using the syntax

```
@(argument-list) expression
```

Any variables that are not found in the argument list are inherited from the enclosing scope. Anonymous functions are useful for creating simple unnamed functions from expressions or for wrapping calls to other functions to adapt them for use by functions like `quad`. For example,

```
f = @(x) x.^2;
quad (f, 0, 10)
    ⇒ 333.33
```

creates a simple unnamed function from the expression `x.^2` and passes it to `quad`,

```
quad (@(x) sin (x), 0, pi)
    ⇒ 2
```

wraps another function, and

```
a = 1;
b = 2;
quad (@(x) betainc (x, a, b), 0, 0.4)
    ⇒ 0.13867
```

adapts a function with several parameters to the form required by `quad`. In this example, the values of *a* and *b* that are passed to `betainc` are inherited from the current environment.

Note that for performance reasons it is better to use handles to existing Octave functions, rather than to define anonymous functions which wrap an existing function. The integration of `sin (x)` is 5X faster if the code is written as

```
quad (@sin, 0, pi)
```

rather than using the anonymous function `@(x) sin (x)`. There are many operators which have functional equivalents that may be better choices than an anonymous function. Instead of writing

```
f = @(x, y) x + y
```

this should be coded as

```
f = @plus
```

See Section 34.4.2 [Operator Overloading], page 731, for a list of operators which also have a functional form.

11.11.3 Inline Functions

An inline function is created from a string containing the function body using the `inline` function. The following code defines the function $f(x) = x^2 + 2$.

```
f = inline ("x^2 + 2");
```

After this it is possible to evaluate f at any x by writing `f(x)`.

Caution: MATLAB has begun the process of deprecating inline functions. At some point in the future support will be dropped and eventually Octave will follow MATLAB and also remove inline functions. Use anonymous functions in all new code.

`inline` (*str*) [Built-in Function]
`inline` (*str*, *arg1*, ...) [Built-in Function]
`inline` (*str*, *n*) [Built-in Function]

Create an inline function from the character string *str*.

If called with a single argument, the arguments of the generated function are extracted from the function itself. The generated function arguments will then be in alphabetical order. It should be noted that i and j are ignored as arguments due to the ambiguity between their use as a variable or their use as an built-in constant. All arguments followed by a parenthesis are considered to be functions. If no arguments are found, a function taking a single argument named `x` will be created.

If the second and subsequent arguments are character strings, they are the names of the arguments of the function.

If the second argument is an integer *n*, the arguments are `"x"`, `"P1"`, ..., `"PN"`.

Programming Note: The use of `inline` is discouraged and it may be removed from a future version of Octave. The preferred way to create functions from strings is through the use of anonymous functions (see Section 11.11.2 [Anonymous Functions], page 201) or `str2func`.

See also: [argnames], page 202, [formula], page 202, [vectorize], page 490, [str2func], page 201.

`argnames` (*fun*) [Built-in Function]

Return a cell array of character strings containing the names of the arguments of the inline function *fun*.

See also: [inline], page 202, [formula], page 202, [vectorize], page 490.

`formula` (*fun*) [Built-in Function]

Return a character string representing the inline function *fun*.

Note that `char (fun)` is equivalent to `formula (fun)`.

See also: [char], page 71, [argnames], page 202, [inline], page 202, [vectorize], page 490.

vars = `symvar` (*str*) [Function File]

Identify the symbolic variable names in the string *str*.

Common constant names such as `i`, `j`, `pi`, `Inf` and Octave functions such as `sin` or `plot` are ignored.

Any names identified are returned in a cell array of strings. The array is empty if no variables were found.

Example:

```
symvar ("x^2 + y^2 == 4")
⇒ {
     [1,1] = x
     [2,1] = y
   }
```

11.12 Commands

Commands are a special class of functions that only accept string input arguments. A command can be called as an ordinary function, but it can also be called without the parentheses. For example,

```
my_command hello world
```

is equivalent to

```
my_command ("hello", "world")
```

The general form of a command call is

```
cmdname arg1 arg2 ...
```

which translates directly to

```
cmdname ("arg1", "arg2", ...)
```

Any regular function can be used as a command if it accepts string input arguments. For example:

```
toupper lower_case_arg
   ⇒ ans = LOWER_CASE_ARG
```

One difficulty of commands occurs when one of the string input arguments is stored in a variable. Because Octave can't tell the difference between a variable name and an ordinary string, it is not possible to pass a variable as input to a command. In such a situation a command must be called as a function. For example:

```
strvar = "hello world";
toupper strvar
   ⇒ ans = STRVAR
toupper (strvar)
   ⇒ ans = HELLO WORLD
```

11.13 Organization of Functions Distributed with Octave

Many of Octave's standard functions are distributed as function files. They are loosely organized by topic, in subdirectories of *octave-home*/lib/octave/*version*/m, to make it easier to find them.

The following is a list of all the function file subdirectories, and the types of functions you will find there.

`audio` Functions for playing and recording sounds.

`deprecated`
: Out-of-date functions which will eventually be removed from Octave.

`elfun`
: Elementary functions, principally trigonometric.

`@ftp`
: Class functions for the FTP object.

`general`
: Miscellaneous matrix manipulations, like `flipud`, `rot90`, and `triu`, as well as other basic functions, like `ismatrix`, `narginchk`, etc.

`geometry`
: Functions related to Delaunay triangulation.

`help`
: Functions for Octave's built-in help system.

`image`
: Image processing tools. These functions require the X Window System.

`io`
: Input-output functions.

`linear-algebra`
: Functions for linear algebra.

`miscellaneous`
: Functions that don't really belong anywhere else.

`optimization`
: Functions related to minimization, optimization, and root finding.

`path`
: Functions to manage the directory path Octave uses to find functions.

`pkg`
: Package manager for installing external packages of functions in Octave.

`plot`
: Functions for displaying and printing two- and three-dimensional graphs.

`polynomial`
: Functions for manipulating polynomials.

`prefs`
: Functions implementing user-defined preferences.

`set`
: Functions for creating and manipulating sets of unique values.

`signal`
: Functions for signal processing applications.

`sparse`
: Functions for handling sparse matrices.

`specfun`
: Special functions such as `bessel` or `factor`.

`special-matrix`
: Functions that create special matrix forms such as Hilbert or Vandermonde matrices.

`startup`
: Octave's system-wide startup file.

`statistics`
: Statistical functions.

`strings`
: Miscellaneous string-handling functions.

`testfun`
: Functions for performing unit tests on other functions.

`time`
: Functions related to time and date processing.

12 Errors and Warnings

Octave includes several functions for printing error and warning messages. When you write functions that need to take special action when they encounter abnormal conditions, you should print the error messages using the functions described in this chapter.

Since many of Octave's functions use these functions, it is also useful to understand them, so that errors and warnings can be handled.

12.1 Handling Errors

An error is something that occurs when a program is in a state where it doesn't make sense to continue. An example is when a function is called with too few input arguments. In this situation the function should abort with an error message informing the user of the lacking input arguments.

Since an error can occur during the evaluation of a program, it is very convenient to be able to detect that an error occurred, so that the error can be fixed. This is possible with the `try` statement described in Section 10.9 [The try Statement], page 168.

12.1.1 Raising Errors

The most common use of errors is for checking input arguments to functions. The following example calls the `error` function if the function `f` is called without any input arguments.

```
function f (arg1)
  if (nargin == 0)
    error ("not enough input arguments");
  endif
endfunction
```

When the `error` function is called, it prints the given message and returns to the Octave prompt. This means that no code following a call to `error` will be executed.

`error` (*template*, ...) [Built-in Function]
`error` (*id*, *template*, ...) [Built-in Function]

Display an error message and stop m-file execution.

Format the optional arguments under the control of the template string *template* using the same rules as the `printf` family of functions (see Section 14.2.4 [Formatted Output], page 253) and print the resulting message on the `stderr` stream. The message is prefixed by the character string '`error: `'.

Calling `error` also sets Octave's internal error state such that control will return to the top level without evaluating any further commands. This is useful for aborting from functions or scripts.

If the error message does not end with a newline character, Octave will print a traceback of all the function calls leading to the error. For example, given the following function definitions:

```
function f () g (); end
function g () h (); end
function h () nargin == 1 || error ("nargin != 1"); end
```

calling the function `f` will result in a list of messages that can help you to quickly locate the exact location of the error:

```
f ()
error: nargin != 1
error: called from:
error:   error at line -1, column -1
error:   h at line 1, column 27
error:   g at line 1, column 15
error:   f at line 1, column 15
```

If the error message ends in a newline character, Octave will print the message but will not display any traceback messages as it returns control to the top level. For example, modifying the error message in the previous example to end in a newline causes Octave to only print a single message:

```
function h () nargin == 1 || error ("nargin != 1\n"); end
f ()
error: nargin != 1
```

A null string (`""`) input to `error` will be ignored and the code will continue running as if the statement were a NOP. This is for compatibility with MATLAB. It also makes it possible to write code such as

```
err_msg = "";
if (CONDITION 1)
  err_msg = "CONDITION 1 found";
elseif (CONDITION2)
  err_msg = "CONDITION 2 found";
...
endif
error (err_msg);
```

which will only stop execution if an error has been found.

Implementation Note: For compatibility with MATLAB, escape sequences in *template* (e.g., `"\n"` => newline) are processed regardless of whether *template* has been defined with single quotes, as long as there are two or more input arguments. To disable escape sequence expansion use a second backslash before the sequence (e.g., `"\\n"`) or use the `regexptranslate` function.

See also: [warning], page 212, [lasterror], page 208.

Since it is common to use errors when there is something wrong with the input to a function, Octave supports functions to simplify such code. When the `print_usage` function is called, it reads the help text of the function calling `print_usage`, and presents a useful error. If the help text is written in Texinfo it is possible to present an error message that only contains the function prototypes as described by the `@deftypefn` parts of the help text. When the help text isn't written in Texinfo, the error message contains the entire help message.

Consider the following function.

```
## -*- texinfo -*-
## @deftypefn {Function File} f (@var{arg1})
## Function help text goes here...
## @end deftypefn
function f (arg1)
  if (nargin == 0)
    print_usage ();
  endif
endfunction
```

When it is called with no input arguments it produces the following error.

```
f ()

 ⊣  error: Invalid call to f.  Correct usage is:
 ⊣
 ⊣   -- Function File: f (ARG1)
 ⊣
 ⊣
 ⊣  Additional help for built-in functions and operators is
 ⊣  available in the online version of the manual.  Use the command
 ⊣  'doc <topic>' to search the manual index.
 ⊣
 ⊣  Help and information about Octave is also available on the WWW
 ⊣  at http://www.octave.org and via the help@octave.org
 ⊣  mailing list.
```

`print_usage` () [Function File]
`print_usage` (*name*) [Function File]

Print the usage message for the function *name*.

When called with no input arguments the `print_usage` function displays the usage message of the currently executing function.

See also: [help], page 20.

`beep` () [Function File]

Produce a beep from the speaker (or visual bell).

This function sends the alarm character `"\a"` to the terminal. Depending on the user's configuration this may produce an audible beep, a visual bell, or nothing at all.

See also: [puts], page 252, [fputs], page 252, [printf], page 253, [fprintf], page 254.

val = `beep_on_error` () [Built-in Function]
old_val = `beep_on_error` (*new_val*) [Built-in Function]
`beep_on_error` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether Octave will try to ring the terminal bell before printing an error message.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

12.1.2 Catching Errors

When an error occurs, it can be detected and handled using the `try` statement as described in Section 10.9 [The try Statement], page 168. As an example, the following piece of code counts the number of errors that occurs during a `for` loop.

```
number_of_errors = 0;
for n = 1:100
  try
    ...
  catch
    number_of_errors++;
  end_try_catch
endfor
```

The above example treats all errors the same. In many situations it can however be necessary to discriminate between errors, and take different actions depending on the error. The `lasterror` function returns a structure containing information about the last error that occurred. As an example, the code above could be changed to count the number of errors related to the '`*`' operator.

```
number_of_errors = 0;
for n = 1:100
  try
    ...
  catch
    msg = lasterror.message;
    if (strfind (msg, "operator *"))
      number_of_errors++;
    endif
  end_try_catch
endfor
```

Alternatively, the output of the `lasterror` function can be found in a variable indicated immediately after the `catch` keyword, as in the example below showing how to redirect an error as a warning:

```
try
  ...
catch err
  warning(err.identifier, err.message);
  ...
end_try_catch
```

lasterr = `lasterror ()` [Built-in Function]
`lasterror` (*err*) [Built-in Function]
`lasterror` ("*reset*") [Built-in Function]

Query or set the last error message structure.

When called without arguments, return a structure containing the last error message and other information related to this error. The elements of the structure are:

`message` The text of the last error message

`identifier`
The message identifier of this error message

`stack` A structure containing information on where the message occurred. This may be an empty structure if the information cannot be obtained. The fields of the structure are:

`file` The name of the file where the error occurred

`name` The name of function in which the error occurred

`line` The line number at which the error occurred

`column` An optional field with the column number at which the error occurred

The last error structure may be set by passing a scalar structure, *err*, as input. Any fields of *err* that match those above are set while any unspecified fields are initialized with default values.

If `lasterror` is called with the argument `"reset"`, all fields are set to their default values.

See also: [lasterr], page 209, [error], page 205, [lastwarn], page 213.

`[`*msg*`,` *msgid*`] = lasterr ()` [Built-in Function]
`lasterr (`*msg*`)` [Built-in Function]
`lasterr (`*msg*`,` *msgid*`)` [Built-in Function]

Query or set the last error message.

When called without input arguments, return the last error message and message identifier.

With one argument, set the last error message to *msg*.

With two arguments, also set the last message identifier.

See also: [lasterror], page 208, [error], page 205, [lastwarn], page 213.

It is also possible to assign an identification string to an error. If an error has such an ID the user can catch this error as will be shown in the next example. To assign an ID to an error, simply call `error` with two string arguments, where the first is the identification string, and the second is the actual error. Note that error IDs are in the format `"NAMESPACE:ERROR-NAME"`. The namespace `"Octave"` is used for Octave's own errors. Any other string is available as a namespace for user's own errors.

The next example counts indexing errors. The errors are caught using the field identifier of the structure returned by the function `lasterror`.

```
number_of_errors = 0;
for n = 1:100
  try
    ...
  catch
    id = lasterror.identifier;
    if (strcmp (id, "Octave:invalid-indexing"))
      number_of_errors++;
    endif
  end_try_catch
endfor
```

The functions distributed with Octave can issue one of the following errors.

`Octave:invalid-context`
: Indicates the error was generated by an operation that cannot be executed in the scope from which it was called. For example, the function `print_usage ()` when called from the Octave prompt raises this error.

`Octave:invalid-input-arg`
: Indicates that a function was called with invalid input arguments.

`Octave:invalid-fun-call`
: Indicates that a function was called in an incorrect way, e.g., wrong number of input arguments.

`Octave:invalid-indexing`
: Indicates that a data-type was indexed incorrectly, e.g., real-value index for arrays, nonexistent field of a structure.

`Octave:bad-alloc`
: Indicates that memory couldn't be allocated.

`Octave:undefined-function`
: Indicates a call to a function that is not defined. The function may exist but Octave is unable to find it in the search path.

When an error has been handled it is possible to raise it again. This can be useful when an error needs to be detected, but the program should still abort. This is possible using the `rethrow` function. The previous example can now be changed to count the number of errors related to the '`*`' operator, but still abort if another kind of error occurs.

```
number_of_errors = 0;
for n = 1:100
  try
    ...
  catch
    msg = lasterror.message;
    if (strfind (msg, "operator *"))
      number_of_errors++;
    else
      rethrow (lasterror);
    endif
  end_try_catch
endfor
```

`rethrow` (*err*) [Built-in Function]

Reissue a previous error as defined by *err*.

err is a structure that must contain at least the `"message"` and `"identifier"` fields. *err* can also contain a field `"stack"` that gives information on the assumed location of the error. Typically *err* is returned from `lasterror`.

See also: [lasterror], page 208, [lasterr], page 209, [error], page 205.

err = `errno` () [Built-in Function]
err = `errno` (*val*) [Built-in Function]
err = `errno` (*name*) [Built-in Function]

Return the current value of the system-dependent variable errno, set its value to *val* and return the previous value, or return the named error code given *name* as a character string, or -1 if *name* is not found.

See also: [errno_list], page 211.

`errno_list` () [Built-in Function]

Return a structure containing the system-dependent errno values.

See also: [errno], page 211.

12.1.3 Recovering From Errors

Octave provides several ways of recovering from errors. There are `try`/`catch` blocks, `unwind_protect`/`unwind_protect_cleanup` blocks, and finally the `onCleanup` command.

The `onCleanup` command associates an ordinary Octave variable (the trigger) with an arbitrary function (the action). Whenever the Octave variable ceases to exist—whether due to a function return, an error, or simply because the variable has been removed with `clear`—then the assigned function is executed.

The function can do anything necessary for cleanup such as closing open file handles, printing an error message, or restoring global variables to their initial values. The last example is a very convenient idiom for Octave code. For example:

```
function rand42
  old_state = rand ("state");
  restore_state = onCleanup (@() rand ("state", old_state));
  rand ("state", 42);
  ...
endfunction  # rand generator state restored by onCleanup
```

obj = onCleanup (*function*) [Built-in Function]

Create a special object that executes a given function upon destruction.

If the object is copied to multiple variables (or cell or struct array elements) or returned from a function, *function* will be executed after clearing the last copy of the object. Note that if multiple local onCleanup variables are created, the order in which they are called is unspecified. For similar functionality See Section 10.8 [The unwind_protect Statement], page 168.

12.2 Handling Warnings

Like an error, a warning is issued when something unexpected happens. Unlike an error, a warning doesn't abort the currently running program. A simple example of a warning is when a number is divided by zero. In this case Octave will issue a warning and assign the value `Inf` to the result.

```
a = 1/0
     ⊣ warning: division by zero
     ⇒ a = Inf
```

12.2.1 Issuing Warnings

It is possible to issue warnings from any code using the `warning` function. In its most simple form, the `warning` function takes a string describing the warning as its input argument. As an example, the following code controls if the variable '`a`' is non-negative, and if not issues a warning and sets '`a`' to zero.

```
a = -1;
if (a < 0)
  warning ("'a' must be non-negative.  Setting 'a' to zero.");
  a = 0;
endif
     ⊣ 'a' must be non-negative.  Setting 'a' to zero.
```

Since warnings aren't fatal to a running program, it is not possible to catch a warning using the `try` statement or something similar. It is however possible to access the last warning as a string using the `lastwarn` function.

It is also possible to assign an identification string to a warning. If a warning has such an ID the user can enable and disable this warning as will be described in the next section. To assign an ID to a warning, simply call `warning` with two string arguments, where the first is the identification string, and the second is the actual warning. Note that warning IDs are in the format `"NAMESPACE:WARNING-NAME"`. The namespace `"Octave"` is used for Octave's own warnings. Any other string is available as a namespace for user's own warnings.

`warning` (*template*, ...) [Built-in Function]
`warning` (*id*, *template*, ...) [Built-in Function]
`warning` ("*on*", *id*) [Built-in Function]
`warning` ("*off*", *id*) [Built-in Function]
`warning` ("*query*", *id*) [Built-in Function]
`warning` ("*error*", *id*) [Built-in Function]
`warning` (*state*, "*backtrace*") [Built-in Function]
`warning` (*state*, *id*, "*local*") [Built-in Function]

Display a warning message or control the behavior of Octave's warning system.

Format the optional arguments under the control of the template string *template* using the same rules as the `printf` family of functions (see Section 14.2.4 [Formatted Output], page 253) and print the resulting message on the `stderr` stream. The message is prefixed by the character string '`warning: `'. You should use this function when you want to notify the user of an unusual condition, but only when it makes sense for your program to go on.

The optional message identifier allows users to enable or disable warnings tagged by *id*. A message identifier is of the form "NAMESPACE:WARNING-NAME". Octave's own warnings use the `"Octave"` namespace (see [XREFwarning_ids], page 214). The special identifier `"all"` may be used to set the state of all warnings.

If the first argument is `"on"` or `"off"`, set the state of a particular warning using the identifier *id*. If the first argument is `"query"`, query the state of this warning instead. If the identifier is omitted, a value of `"all"` is assumed. If you set the state of a warning to `"error"`, the warning named by *id* is handled as if it were an error instead. So, for example, the following handles all warnings as errors:

```
warning ("error");
```

If the state is `"on"` or `"off"` and the third argument is `"backtrace"`, then a stack trace is printed along with the warning message when warnings occur inside function calls. This option is enabled by default.

If the state is `"on"`, `"off"`, or `"error"` and the third argument is `"local"`, then the warning state will be set temporarily, until the end of the current function. Changes to warning states that are set locally affect the current function and all functions called from the current scope. The previous warning state is restored on return from the current function. The `"local"` option is ignored if used in the top-level workspace.

Implementation Note: For compatibility with MATLAB, escape sequences in *template* (e.g., `"\n"` => newline) are processed regardless of whether *template* has been defined with single quotes, as long as there are two or more input arguments. To disable escape sequence expansion use a second backslash before the sequence (e.g., `"\\n"`) or use the `regexptranslate` function.

See also: [warning_ids], page 214, [lastwarn], page 213, [error], page 205.

[*msg*, *msgid*] = `lastwarn` () [Built-in Function]
`lastwarn` (*msg*) [Built-in Function]
`lastwarn` (*msg*, *msgid*) [Built-in Function]

Query or set the last warning message.

When called without input arguments, return the last warning message and message identifier.

With one argument, set the last warning message to *msg*.

With two arguments, also set the last message identifier.

See also: [warning], page 212, [lasterror], page 208, [lasterr], page 209.

The functions distributed with Octave can issue one of the following warnings.

`Octave:abbreviated-property-match`
: By default, the `Octave:abbreviated-property-match` warning is enabled.

`Octave:array-to-scalar`
: If the `Octave:array-to-scalar` warning is enabled, Octave will warn when an implicit conversion from an array to a scalar value is attempted. By default, the `Octave:array-to-scalar` warning is disabled.

`Octave:array-to-vector`
: If the `Octave:array-to-vector` warning is enabled, Octave will warn when an implicit conversion from an array to a vector value is attempted. By default, the `Octave:array-to-vector` warning is disabled.

`Octave:assign-as-truth-value`
: If the `Octave:assign-as-truth-value` warning is enabled, a warning is issued for statements like

```
if (s = t)
  ...
```

since such statements are not common, and it is likely that the intent was to write

```
if (s == t)
  ...
```

instead.

There are times when it is useful to write code that contains assignments within the condition of a `while` or `if` statement. For example, statements like

```
while (c = getc ())
  ...
```

are common in C programming.

It is possible to avoid all warnings about such statements by disabling the `Octave:assign-as-truth-value` warning, but that may also let real errors like

```
if (x = 1)  # intended to test (x == 1)!
  ...
```

slip by.

In such cases, it is possible suppress errors for specific statements by writing them with an extra set of parentheses. For example, writing the previous example as

```
while ((c = getc ()))
  ...
```

will prevent the warning from being printed for this statement, while allowing Octave to warn about other assignments used in conditional contexts.

By default, the `Octave:assign-as-truth-value` warning is enabled.

`Octave:associativity-change`
: If the `Octave:associativity-change` warning is enabled, Octave will warn about possible changes in the meaning of some code due to changes in associativity for some operators. Associativity changes have typically been made for MATLAB compatibility. By default, the `Octave:associativity-change` warning is enabled.

`Octave:autoload-relative-file-name`
: If the `Octave:autoload-relative-file-name` is enabled, Octave will warn when parsing autoload() function calls with relative paths to function files. This usually happens when using autoload() calls in PKG_ADD files, when the PKG_ADD file is not in the same directory as the .oct file referred to by the autoload() command. By default, the `Octave:autoload-relative-file-name` warning is enabled.

`Octave:built-in-variable-assignment`
: By default, the `Octave:built-in-variable-assignment` warning is enabled.

`Octave:deprecated-keyword`
: If the `Octave:deprecated-keyword` warning is enabled, a warning is issued when Octave encounters a keyword that is obsolete and scheduled for removal from Octave. By default, the `Octave:deprecated-keyword` warning is enabled.

`Octave:divide-by-zero`
: If the `Octave:divide-by-zero` warning is enabled, a warning is issued when Octave encounters a division by zero. By default, the `Octave:divide-by-zero` warning is enabled.

`Octave:fopen-file-in-path`
: By default, the `Octave:fopen-file-in-path` warning is enabled.

`Octave:function-name-clash`
: If the `Octave:function-name-clash` warning is enabled, a warning is issued when Octave finds that the name of a function defined in a function file differs from the name of the file. (If the names disagree, the name declared inside the file is ignored.) By default, the `Octave:function-name-clash` warning is enabled.

`Octave:future-time-stamp`
: If the `Octave:future-time-stamp` warning is enabled, Octave will print a warning if it finds a function file with a time stamp that is in the future. By default, the `Octave:future-time-stamp` warning is enabled.

`Octave:glyph-render`
: By default, the `Octave:glyph-render` warning is enabled.

`Octave:imag-to-real`
: If the `Octave:imag-to-real` warning is enabled, a warning is printed for implicit conversions of complex numbers to real numbers. By default, the `Octave:imag-to-real` warning is disabled.

`Octave:language-extension`
: Print warnings when using features that are unique to the Octave language and that may still be missing in MATLAB. By default, the `Octave:language-extension` warning is disabled. The `--traditional` or `--braindead` startup options for Octave may also be of use, see Section 2.1.1 [Command Line Options], page 15.

`Octave:load-file-in-path`
: By default, the `Octave:load-file-in-path` warning is enabled.

`Octave:logical-conversion`
: By default, the `Octave:logical-conversion` warning is enabled.

`Octave:md5sum-file-in-path`
: By default, the `Octave:md5sum-file-in-path` warning is enabled.

`Octave:missing-glyph`
: By default, the `Octave:missing-glyph` warning is enabled.

`Octave:missing-semicolon`
: If the `Octave:missing-semicolon` warning is enabled, Octave will warn when statements in function definitions don't end in semicolons. By default the `Octave:missing-semicolon` warning is disabled.

`Octave:mixed-string-concat`
: If the `Octave:mixed-string-concat` warning is enabled, print a warning when concatenating a mixture of double and single quoted strings. By default, the `Octave:mixed-string-concat` warning is disabled.

`Octave:neg-dim-as-zero`
: If the `Octave:neg-dim-as-zero` warning is enabled, print a warning for expressions like

```
eye (-1)
```

By default, the `Octave:neg-dim-as-zero` warning is disabled.

`Octave:nested-functions-coerced`
: By default, the `Octave:nested-functions-coerced` warning is enabled.

`Octave:noninteger-range-as-index`
: By default, the `Octave:noninteger-range-as-index` warning is enabled.

`Octave:num-to-str`
: If the `Octave:num-to-str` warning is enable, a warning is printed for implicit conversions of numbers to their ASCII character equivalents when strings are constructed using a mixture of strings and numbers in matrix notation. For example,

```
[ "f", 111, 111 ]
⇒ "foo"
```

elicits a warning if the `Octave:num-to-str` warning is enabled. By default, the `Octave:num-to-str` warning is enabled.

`Octave:possible-matlab-short-circuit-operator`
: If the `Octave:possible-matlab-short-circuit-operator` warning is enabled, Octave will warn about using the not short circuiting operators `&` and `|` inside `if` or `while` conditions. They normally never short circuit, but MATLAB always short circuits if any logical operators are used in a condition. You can turn on the option

```
do_braindead_shortcircuit_evaluation (1)
```

if you would like to enable this short-circuit evaluation in Octave. Note that the `&&` and `||` operators always short circuit in both Octave and MATLAB, so it's only necessary to enable MATLAB-style short-circuiting if it's too arduous to modify existing code that relies on this behavior. By default, the `Octave:possible-matlab-short-circuit-operator` warning is enabled.

`Octave:precedence-change`
: If the `Octave:precedence-change` warning is enabled, Octave will warn about possible changes in the meaning of some code due to changes in precedence for some operators. Precedence changes have typically been made for MATLAB compatibility. By default, the `Octave:precedence-change` warning is enabled.

`Octave:recursive-path-search`
: By default, the `Octave:recursive-path-search` warning is enabled.

`Octave:remove-init-dir`
: The `path` function changes the search path that Octave uses to find functions. It is possible to set the path to a value which excludes Octave's own built-in functions. If the `Octave:remove-init-dir` warning is enabled then Octave will warn when the `path` function has been used in a way that may render Octave unworkable. By default, the `Octave:remove-init-dir` warning is enabled.

`Octave:reload-forces-clear`
: If several functions have been loaded from the same file, Octave must clear all the functions before any one of them can be reloaded. If the `Octave:reload-forces-clear` warning is enabled, Octave will warn you when this happens, and print a list of the additional functions that it is forced to clear. By default, the `Octave:reload-forces-clear` warning is enabled.

`Octave:resize-on-range-error`
: If the `Octave:resize-on-range-error` warning is enabled, print a warning when a matrix is resized by an indexed assignment with indices outside the current bounds. By default, the ## `Octave:resize-on-range-error` warning is disabled.

`Octave:separator-insert`
: Print warning if commas or semicolons might be inserted automatically in literal matrices. By default, the `Octave:separator-insert` warning is disabled.

`Octave:shadowed-function`
: By default, the `Octave:shadowed-function` warning is enabled.

`Octave:single-quote-string`
: Print warning if a single quote character is used to introduce a string constant. By default, the `Octave:single-quote-string` warning is disabled.

`Octave:nearly-singular-matrix`
`Octave:singular-matrix`
: By default, the `Octave:nearly-singular-matrix` and `Octave:singular-matrix` warnings are enabled.

`Octave:sqrtm:SingularMatrix`
: By default, the `Octave:sqrtm:SingularMatrix` warning is enabled.

`Octave:str-to-num`
: If the `Octave:str-to-num` warning is enabled, a warning is printed for implicit conversions of strings to their numeric ASCII equivalents. For example,

```
"abc" + 0
⇒ 97 98 99
```

elicits a warning if the `Octave:str-to-num` warning is enabled. By default, the `Octave:str-to-num` warning is disabled.

`Octave:undefined-return-values`
: If the `Octave:undefined-return-values` warning is disabled, print a warning if a function does not define all the values in the return list which are expected. By default, the `Octave:undefined-return-values` warning is enabled.

`Octave:variable-switch-label`
: If the `Octave:variable-switch-label` warning is enabled, Octave will print a warning if a switch label is not a constant or constant expression. By default, the `Octave:variable-switch-label` warning is disabled.

12.2.2 Enabling and Disabling Warnings

The `warning` function also allows you to control which warnings are actually printed to the screen. If the `warning` function is called with a string argument that is either `"on"` or `"off"` all warnings will be enabled or disabled.

It is also possible to enable and disable individual warnings through their string identifications. The following code will issue a warning

```
warning ("example:non-negative-variable",
         "'a' must be non-negative.  Setting 'a' to zero.");
```

while the following won't issue a warning

```
warning ("off", "example:non-negative-variable");
warning ("example:non-negative-variable",
         "'a' must be non-negative.  Setting 'a' to zero.");
```

13 Debugging

Octave includes a built-in debugger to aid in the development of scripts. This can be used to interrupt the execution of an Octave script at a certain point, or when certain conditions are met. Once execution has stopped, and debug mode is entered, the symbol table at the point where execution has stopped can be examined and modified to check for errors.

The normal command-line editing and history functions are available in debug mode.

13.1 Entering Debug Mode

There are two basic means of interrupting the execution of an Octave script. These are breakpoints (see Section 13.3 [Breakpoints], page 220), discussed in the next section, and interruption based on some condition.

Octave supports three means to stop execution based on the values set in the functions `debug_on_interrupt`, `debug_on_warning`, and `debug_on_error`.

val = `debug_on_interrupt ()` [Built-in Function]
old_val = `debug_on_interrupt (`*new_val*`)` [Built-in Function]
`debug_on_interrupt (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that controls whether Octave will try to enter debugging mode when it receives an interrupt signal (typically generated with *C-c*).

If a second interrupt signal is received before reaching the debugging mode, a normal interrupt will occur.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [debug_on_error], page 219, [debug_on_warning], page 219.

val = `debug_on_warning ()` [Built-in Function]
old_val = `debug_on_warning (`*new_val*`)` [Built-in Function]
`debug_on_warning (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that controls whether Octave will try to enter the debugger when a warning is encountered.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [debug_on_error], page 219, [debug_on_interrupt], page 219.

val = `debug_on_error ()` [Built-in Function]
old_val = `debug_on_error (`*new_val*`)` [Built-in Function]
`debug_on_error (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that controls whether Octave will try to enter the debugger when an error is encountered.

This will also inhibit printing of the normal traceback message (you will only see the top-level error message).

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [debug_on_warning], page 219, [debug_on_interrupt], page 219.

13.2 Leaving Debug Mode

Use either `dbcont` or `return` to leave the debug mode and continue the normal execution of the script.

`dbcont` [Command]

Leave command-line debugging mode and continue code execution normally.

See also: [dbstep], page 224, [dbquit], page 220.

To quit debug mode and return directly to the prompt without executing any additional code use `dbquit`.

`dbquit` [Command]

Quit debugging mode immediately without further code execution and return to the Octave prompt.

See also: [dbcont], page 220, [dbstep], page 224.

Finally, typing `exit` or `quit` at the debug prompt will result in Octave terminating normally.

13.3 Breakpoints

Breakpoints can be set in any m-file function by using the `dbstop` function.

`dbstop` *`func`* [Command]
`dbstop` *`func line`* [Command]
`dbstop` *`func line1 line2 ...`* [Command]
`dbstop` *`line ...`* [Command]
`rline` `= dbstop ("`*`func`*`")` [Built-in Function]
`rline` `= dbstop ("`*`func`*`",` *`line`*`)` [Built-in Function]
`rline` `= dbstop ("`*`func`*`",` *`line1`*`,` *`line2`*`, ...)` [Built-in Function]
`dbstop ("`*`func`*`", [`*`line1`*`, ...])` [Built-in Function]
`dbstop (`*`line`*`, ...)` [Built-in Function]

Set a breakpoint at line number *line* in function *func*.

Arguments are

func Function name as a string variable. When already in debug mode this argument can be omitted and the current function will be used.

line Line number where the breakpoint should be set. Multiple lines may be given as separate arguments or as a vector.

When called with a single argument *func*, the breakpoint is set at the first executable line in the named function.

The optional output *rline* is the real line number where the breakpoint was set. This can differ from the specified line if the line is not executable. For example, if a breakpoint attempted on a blank line then Octave will set the real breakpoint at the next executable line.

See also: [dbclear], page 221, [dbstatus], page 221, [dbstep], page 224, [debug_on_error], page 219, [debug_on_warning], page 219, [debug_on_interrupt], page 219.

Breakpoints in class methods are also supported (e.g., `dbstop ("@class/method")`). However, breakpoints cannot be set in built-in functions (e.g., `sin`, etc.) or dynamically loaded functions (i.e., oct-files).

To set a breakpoint immediately upon entering a function use line number 1, or omit the line number entirely and just give the function name. When setting the breakpoint Octave will ignore the leading comment block, and the breakpoint will be set on the first executable statement in the function. For example:

```
dbstop ("asind", 1)
⇒ 29
```

Note that the return value of `29` means that the breakpoint was effectively set to line 29. The status of breakpoints in a function can be queried with `dbstatus`.

`dbstatus ()` [Built-in Function]
brk_list `= dbstatus ()` [Built-in Function]
brk_list `= dbstatus ("`*func*`")` [Built-in Function]

Report the location of active breakpoints.

When called with no input or output arguments, print the list of all functions with breakpoints and the line numbers where those breakpoints are set.

If a function name *func* is specified then only report breakpoints for the named function.

The optional return argument *brk_list* is a struct array with the following fields.

name The name of the function with a breakpoint.

file The name of the m-file where the function code is located.

line A line number, or vector of line numbers, with a breakpoint.

Note: When `dbstatus` is called from the debug prompt within a function, the list of breakpoints is automatically trimmed to the breakpoints in the current function.

See also: [dbclear], page 221, [dbwhere], page 223.

Reusing the previous example, `dbstatus ("asind")` will return 29. The breakpoints listed can then be cleared with the `dbclear` function.

`dbclear` *func* [Command]
`dbclear` *func line* [Command]
`dbclear` *func line1 line2 ...* [Command]
`dbclear` *line ...* [Command]
`dbclear` *all* [Command]

`dbclear ("`*func*`")` [Built-in Function]
`dbclear ("`*func*`", `*line*`)` [Built-in Function]
`dbclear ("`*func*`", `*line1*`, `*line2*`, ...)` [Built-in Function]
`dbclear ("`*func*`", [`*line1*`, ...])` [Built-in Function]
`dbclear (`*line*`, ...)` [Built-in Function]
`dbclear ("`*all*`")` [Built-in Function]

Delete a breakpoint at line number *line* in the function *func*.

Arguments are

func Function name as a string variable. When already in debug mode this argument can be omitted and the current function will be used.

line Line number from which to remove a breakpoint. Multiple lines may be given as separate arguments or as a vector.

When called without a line number specification all breakpoints in the named function are cleared.

If the requested line is not a breakpoint no action is performed.

The special keyword `"all"` will clear all breakpoints from all files.

See also: [dbstop], page 220, [dbstatus], page 221, [dbwhere], page 223.

A breakpoint may also be set in a subfunction. For example, if a file contains the functions

```
function y = func1 (x)
  y = func2 (x);
endfunction
function y = func2 (x)
  y = x + 1;
endfunction
```

then a breakpoint can be set at the start of the subfunction directly with

```
dbstop (["func1", filemarker(), "func2"])
⇒ 5
```

Note that `filemarker` returns the character that marks subfunctions from the file containing them. Unless the default has been changed this character is '`>`'. Thus, a quicker and more normal way to set the breakpoint would be

```
dbstop func1>func2
```

Another simple way of setting a breakpoint in an Octave script is the use of the `keyboard` function.

`keyboard ()` [Built-in Function]
`keyboard ("`*prompt*`")` [Built-in Function]

Stop m-file execution and enter debug mode.

When the `keyboard` function is executed, Octave prints a prompt and waits for user input. The input strings are then evaluated and the results are printed. This makes it possible to examine the values of variables within a function, and to assign new values if necessary. To leave the prompt and return to normal execution type '`return`' or '`dbcont`'. The `keyboard` function does not return an exit status.

If `keyboard` is invoked without arguments, a default prompt of '`debug> `' is used.

See also: [dbstop], page 220, [dbcont], page 220, [dbquit], page 220.

The `keyboard` function is placed in a script at the point where the user desires that the execution be stopped. It automatically sets the running script into the debug mode.

13.4 Debug Mode

There are three additional support functions that allow the user to find out where in the execution of a script Octave entered the debug mode, and to print the code in the script surrounding the point where Octave entered debug mode.

`dbwhere` [Command]

In debugging mode, report the current file and line number where execution is stopped.

See also: [dbstatus], page 221, [dbcont], page 220, [dbstep], page 224, [dbup], page 225.

`dbtype` [Command]
`dbtype` *`lineno`* [Command]
`dbtype` *`startl:endl`* [Command]
`dbtype` *`startl:end`* [Command]
`dbtype` *`func`* [Command]
`dbtype` *`func lineno`* [Command]
`dbtype` *`func startl:endl`* [Command]
`dbtype` *`func startl:end`* [Command]

Display a script file with line numbers.

When called with no arguments in debugging mode, display the script file currently being debugged.

An optional range specification can be used to list only a portion of the file. The special keyword `"end"` is a valid line number specification for the last line of the file.

When called with the name of a function, list that script file with line numbers.

See also: [dbwhere], page 223, [dbstatus], page 221, [dbstop], page 220.

`dblist` [Command]
`dblist` *`n`* [Command]

In debugging mode, list *n* lines of the function being debugged centered around the current line to be executed.

If unspecified *n* defaults to 10 (+/- 5 lines)

See also: [dbwhere], page 223, [dbtype], page 223.

You may also use `isdebugmode` to determine whether the debugger is currently active.

`isdebugmode ()` [Built-in Function]

Return true if in debugging mode, otherwise false.

See also: [dbwhere], page 223, [dbstack], page 224, [dbstatus], page 221.

Debug mode also allows single line stepping through a function using the command `dbstep`.

`dbstep` [Command]
`dbstep` *n* [Command]
`dbstep` *in* [Command]
`dbstep` *out* [Command]
`dbnext ...` [Command]

In debugging mode, execute the next *n* lines of code.

If *n* is omitted, execute the next single line of code. If the next line of code is itself defined in terms of an m-file remain in the existing function.

Using `dbstep in` will cause execution of the next line to step into any m-files defined on the next line.

Using `dbstep out` will cause execution to continue until the current function returns.

`dbnext` is an alias for `dbstep`.

See also: [dbcont], page 220, [dbquit], page 220.

When in debug mode the `RETURN` key will execute the last entered command. This is useful, for example, after hitting a breakpoint and entering `dbstep` once. After that, one can advance line by line through the code with only a single key stroke.

13.5 Call Stack

The function being debugged may be the leaf node of a series of function calls. After examining values in the current subroutine it may turn out that the problem occurred in earlier pieces of code. Use `dbup` and `dbdown` to move up and down through the series of function calls to locate where variables first took on the wrong values. `dbstack` shows the entire series of function calls and at what level debugging is currently taking place.

`dbstack` [Command]
`dbstack` *n* [Command]
`dbstack` *-completenames* [Command]
`[`*stack*`,` *idx*`] = dbstack (...)` [Built-in Function]

Display or return current debugging function stack information.

With optional argument *n*, omit the *n* innermost stack frames.

Although accepted, the argument *-completenames* is silently ignored. Octave always returns absolute file names.

The arguments *n* and *-completenames* can be both specified in any order.

The optional return argument *stack* is a struct array with the following fields:

file	The name of the m-file where the function code is located.
name	The name of the function with a breakpoint.
line	The line number of an active breakpoint.
column	The column number of the line where the breakpoint begins.
scope	Undocumented.

context Undocumented.

The return argument *idx* specifies which element of the *stack* struct array is currently active.

See also: [dbup], page 225, [dbdown], page 225, [dbwhere], page 223, [dbstatus], page 221.

`dbup` [Command]
`dbup n` [Command]

In debugging mode, move up the execution stack *n* frames.

If *n* is omitted, move up one frame.

See also: [dbstack], page 224, [dbdown], page 225.

`dbdown` [Command]
`dbdown n` [Command]

In debugging mode, move down the execution stack *n* frames.

If *n* is omitted, move down one frame.

See also: [dbstack], page 224, [dbup], page 225.

13.6 Profiling

Octave supports profiling of code execution on a per-function level. If profiling is enabled, each call to a function (supporting built-ins, operators, functions in oct- and mex-files, user-defined functions in Octave code and anonymous functions) is recorded while running Octave code. After that, this data can aid in analyzing the code behavior, and is in particular helpful for finding "hot spots" in the code which use up a lot of computation time and are the best targets to spend optimization efforts on.

The main command for profiling is `profile`, which can be used to start or stop the profiler and also to query collected data afterwards. The data is returned in an Octave data structure which can then be examined or further processed by other routines or tools.

`profile` *on* [Command]
`profile` *off* [Command]
`profile` *resume* [Command]
`profile` *clear* [Command]
S `= profile ("`*status*`")` [Function File]
T `= profile ("`*info*`")` [Function File]

Control the built-in profiler.

`profile on`
: Start the profiler, clearing all previously collected data if there is any.

`profile off`
: Stop profiling. The collected data can later be retrieved and examined with `T = profile ("info")`.

`profile clear`
: Clear all collected profiler data.

`profile resume`
Restart profiling without clearing the old data. All newly collected statistics are added to the existing ones.

S = `profile ("status")`
Return a structure with information about the current status of the profiler. At the moment, the only field is `ProfilerStatus` which is either `"on"` or `"off"`.

T = `profile ("info")`
Return the collected profiling statistics in the structure *T*. The flat profile is returned in the field `FunctionTable` which is an array of structures, each entry corresponding to a function which was called and for which profiling statistics are present. In addition, the field `Hierarchical` contains the hierarchical call tree. Each node has an index into the `FunctionTable` identifying the function it corresponds to as well as data fields for number of calls and time spent at this level in the call tree.

See also: [profshow], page 226, [profexplore], page 226.

An easy way to get an overview over the collected data is `profshow`. This function takes the profiler data returned by `profile` as input and prints a flat profile, for instance:

```
 Function Attr     Time (s)        Calls
----------------------------------------
   >myfib    R        2.195        13529
binary <=             0.061        13529
 binary -             0.050        13528
 binary +             0.026         6764
```

This shows that most of the run time was spent executing the function '`myfib`', and some minor proportion evaluating the listed binary operators. Furthermore, it is shown how often the function was called and the profiler also records that it is recursive.

`profshow (`*data*`)` [Function File]
`profshow (`*data*`, `*n*`)` [Function File]
`profshow ()` [Function File]
`profshow (`*n*`)` [Function File]

Display flat per-function profiler results.

Print out profiler data (execution time, number of calls) for the most critical *n* functions. The results are sorted in descending order by the total time spent in each function. If *n* is unspecified it defaults to 20.

The input *data* is the structure returned by `profile ("info")`. If unspecified, `profshow` will use the current profile dataset.

The attribute column displays '`R`' for recursive functions, and is blank for all other function types.

See also: [profexplore], page 226, [profile], page 225.

`profexplore ()` [Function File]
`profexplore (`*data*`)` [Function File]

Interactively explore hierarchical profiler output.

Assuming *data* is the structure with profile data returned by `profile ("info")`, this command opens an interactive prompt that can be used to explore the call-tree. Type *`help`* to get a list of possible commands. If *data* is omitted, `profile ("info")` is called and used in its place.

See also: [profile], page 225, [profshow], page 226.

13.7 Profiler Example

Below, we will give a short example of a profiler session. See Section 13.6 [Profiling], page 225, for the documentation of the profiler functions in detail. Consider the code:

```
global N A;

N = 300;
A = rand (N, N);

function xt = timesteps (steps, x0, expM)
  global N;

  if (steps == 0)
    xt = NA (N, 0);
  else
    xt = NA (N, steps);
    x1 = expM * x0;
    xt(:, 1) = x1;
    xt(:, 2 : end) = timesteps (steps - 1, x1, expM);
  endif
endfunction

function foo ()
  global N A;

  initial = @(x) sin (x);
  x0 = (initial (linspace (0, 2 * pi, N)))';

  expA = expm (A);
  xt = timesteps (100, x0, expA);
endfunction

function fib = bar (N)
  if (N <= 2)
    fib = 1;
  else
    fib = bar (N - 1) + bar (N - 2);
  endif
endfunction
```

If we execute the two main functions, we get:

```
tic; foo; toc;
⇒ Elapsed time is 2.37338 seconds.

tic; bar (20); toc;
⇒ Elapsed time is 2.04952 seconds.
```

But this does not give much information about where this time is spent; for instance, whether the single call to `expm` is more expensive or the recursive time-stepping itself. To get a more detailed picture, we can use the profiler.

```
profile on;
foo;
profile off;

data = profile ("info");
profshow (data, 10);
```

This prints a table like:

```
   #  Function Attr     Time (s)        Calls
---------------------------------------------
   7      expm             1.034            1
   3  binary *             0.823          117
  41  binary \             0.188            1
  38  binary ^             0.126            2
  43 timesteps    R        0.111          101
  44        NA             0.029          101
  39  binary +             0.024            8
  34      norm             0.011            1
  40  binary -             0.004          101
  33   balance             0.003            1
```

The entries are the individual functions which have been executed (only the 10 most important ones), together with some information for each of them. The entries like '`binary *`' denote operators, while other entries are ordinary functions. They include both built-ins like `expm` and our own routines (for instance `timesteps`). From this profile, we can immediately deduce that `expm` uses up the largest proportion of the processing time, even though it is only called once. The second expensive operation is the matrix-vector product in the routine `timesteps`.[1]

Timing, however, is not the only information available from the profile. The attribute column shows us that `timesteps` calls itself recursively. This may not be that remarkable in this example (since it's clear anyway), but could be helpful in a more complex setting. As to the question of why is there a '`binary \`' in the output, we can easily shed some light on that too. Note that `data` is a structure array (Section 6.1.2 [Structure Arrays], page 103) which contains the field `FunctionTable`. This stores the raw data for the profile shown. The number in the first column of the table gives the index under which the shown function can be found there. Looking up `data.FunctionTable(41)` gives:

[1] We only know it is the binary multiplication operator, but fortunately this operator appears only at one place in the code and thus we know which occurrence takes so much time. If there were multiple places, we would have to use the hierarchical profile to find out the exact place which uses up the time which is not covered in this example.

```
scalar structure containing the fields:

  FunctionName = binary \
  TotalTime =  0.18765
  NumCalls =  1
  IsRecursive = 0
  Parents =  7
  Children = [](1x0)
```

Here we see the information from the table again, but have additional fields `Parents` and `Children`. Those are both arrays, which contain the indices of functions which have directly called the function in question (which is entry 7, `expm`, in this case) or been called by it (no functions). Hence, the backslash operator has been used internally by `expm`.

Now let's take a look at `bar`. For this, we start a fresh profiling session (`profile on` does this; the old data is removed before the profiler is restarted):

```
profile on;
bar (20);
profile off;

profshow (profile ("info"));
```

This gives:

```
   #           Function Attr     Time (s)        Calls
-------------------------------------------------------
   1                bar    R        2.091        13529
   2          binary <=             0.062        13529
   3           binary -             0.042        13528
   4           binary +             0.023         6764
   5            profile             0.000            1
   8              false             0.000            1
   6             nargin             0.000            1
   7          binary !=             0.000            1
   9 __profiler_enable__            0.000            1
```

Unsurprisingly, `bar` is also recursive. It has been called 13,529 times in the course of recursively calculating the Fibonacci number in a suboptimal way, and most of the time was spent in `bar` itself.

Finally, let's say we want to profile the execution of both `foo` and `bar` together. Since we already have the run-time data collected for `bar`, we can restart the profiler without clearing the existing data and collect the missing statistics about `foo`. This is done by:

```
profile resume;
foo;
profile off;

profshow (profile ("info"), 10);
```

As you can see in the table below, now we have both profiles mixed together.

```
   #  Function Attr     Time (s)        Calls
---------------------------------------------
   1       bar    R        2.091        13529
  16      expm             1.122            1
  12  binary *             0.798          117
  46  binary \             0.185            1
  45  binary ^             0.124            2
  48 timesteps    R        0.115          101
   2 binary <=             0.062        13529
   3  binary -             0.045        13629
   4  binary +             0.041         6772
  49        NA             0.036          101
```

14 Input and Output

Octave supports several ways of reading and writing data to or from the prompt or a file. The simplest functions for data Input and Output (I/O) are easy to use, but only provide limited control of how data is processed. For more control, a set of functions modeled after the C standard library are also provided by Octave.

14.1 Basic Input and Output

14.1.1 Terminal Output

Since Octave normally prints the value of an expression as soon as it has been evaluated, the simplest of all I/O functions is a simple expression. For example, the following expression will display the value of 'pi'

```
pi
     ⊣ pi = 3.1416
```

This works well as long as it is acceptable to have the name of the variable (or 'ans') printed along with the value. To print the value of a variable without printing its name, use the function `disp`.

The `format` command offers some control over the way Octave prints values with `disp` and through the normal echoing mechanism.

`disp` (*x*) [Built-in Function]

Display the value of *x*.

For example:

```
disp ("The value of pi is:"), disp (pi)

     ⊣ the value of pi is:
     ⊣ 3.1416
```

Note that the output from `disp` always ends with a newline.

If an output value is requested, `disp` prints nothing and returns the formatted output in a string.

See also: [fdisp], page 242.

`list_in_columns` (*arg*, *width*, *prefix*) [Built-in Function]

Return a string containing the elements of *arg* listed in columns with an overall maximum width of *width* and optional prefix *prefix*.

The argument *arg* must be a cell array of character strings or a character array.

If *width* is not specified or is an empty matrix, or less than or equal to zero, the width of the terminal screen is used. Newline characters are used to break the lines in the output string. For example:

```
list_in_columns ({"abc", "def", "ghijkl", "mnop", "qrs", "tuv"}, 20)
     ⇒ abc     mnop
        def     qrs
        ghijkl  tuv

whos ans
     ⇒
     Variables in the current scope:

       Attr Name        Size                     Bytes  Class
       ==== ====        ====                     =====  =====
            ans         1x37                        37  char

     Total is 37 elements using 37 bytes
```

See also: [terminal_size], page 232.

`terminal_size ()` [Built-in Function]

Return a two-element row vector containing the current size of the terminal window in characters (rows and columns).

See also: [list_in_columns], page 231.

`format` [Command]
`format` *options* [Command]

Reset or specify the format of the output produced by `disp` and Octave's normal echoing mechanism.

This command only affects the display of numbers but not how they are stored or computed. To change the internal representation from the default double use one of the conversion functions such as `single`, `uint8`, `int64`, etc.

By default, Octave displays 5 significant digits in a human readable form (option '`short`' paired with '`loose`' format for matrices). If `format` is invoked without any options, this default format is restored.

Valid formats for floating point numbers are listed in the following table.

`short` Fixed point format with 5 significant figures in a field that is a maximum of 10 characters wide. (default).

If Octave is unable to format a matrix so that columns line up on the decimal point and all numbers fit within the maximum field width then it switches to an exponential '`e`' format.

`long` Fixed point format with 15 significant figures in a field that is a maximum of 20 characters wide.

As with the '`short`' format, Octave will switch to an exponential '`e`' format if it is unable to format a matrix properly using the current format.

`short e`
`long e` Exponential format. The number to be represented is split between a mantissa and an exponent (power of 10). The mantissa has 5 significant digits in the short format and 15 digits in the long format. For example, with the '`short e`' format, `pi` is displayed as `3.1416e+00`.

`short E`
`long E` Identical to '`short e`' or '`long e`' but displays an uppercase '`E`' to indicate the exponent. For example, with the '`long E`' format, `pi` is displayed as `3.14159265358979E+00`.

`short g`
`long g` Optimally choose between fixed point and exponential format based on the magnitude of the number. For example, with the '`short g`' format, `pi .^ [2; 4; 8; 16; 32]` is displayed as

```
ans =

      9.8696
      97.409
      9488.5
  9.0032e+07
  8.1058e+15
```

`short eng`
`long eng` Identical to '`short e`' or '`long e`' but displays the value using an engineering format, where the exponent is divisible by 3. For example, with the '`short eng`' format, `10 * pi` is displayed as `31.4159e+00`.

`long G`
`short G` Identical to '`short g`' or '`long g`' but displays an uppercase '`E`' to indicate the exponent.

`free`
`none` Print output in free format, without trying to line up columns of matrices on the decimal point. This also causes complex numbers to be formatted as numeric pairs like this '`(0.60419, 0.60709)`' instead of like this '`0.60419 + 0.60709i`'.

The following formats affect all numeric output (floating point and integer types).

`"+"`
`"+"` *chars*
`plus`
`plus` *chars*

Print a '`+`' symbol for matrix elements greater than zero, a '`-`' symbol for elements less than zero and a space for zero matrix elements. This format can be very useful for examining the structure of a large sparse matrix.

The optional argument *chars* specifies a list of 3 characters to use for printing values greater than zero, less than zero and equal to zero. For example, with the '`"+" "+-."`' format, `[1, 0, -1; -1, 0, 1]` is displayed as

```
ans =

+.-
-.+
```

`bank` Print in a fixed format with two digits to the right of the decimal point.

`native-hex`
: Print the hexadecimal representation of numbers as they are stored in memory. For example, on a workstation which stores 8 byte real values in IEEE format with the least significant byte first, the value of `pi` when printed in `native-hex` format is `400921fb54442d18`.

`hex`
: The same as `native-hex`, but always print the most significant byte first.

`native-bit`
: Print the bit representation of numbers as stored in memory. For example, the value of `pi` is

```
01000000000010010010000111111011
01010100010001000010110100011000
```

(shown here in two 32 bit sections for typesetting purposes) when printed in native-bit format on a workstation which stores 8 byte real values in IEEE format with the least significant byte first.

`bit`
: The same as `native-bit`, but always print the most significant bits first.

`rat`
: Print a rational approximation, i.e., values are approximated as the ratio of small integers. For example, with the '`rat`' format, `pi` is displayed as `355/113`.

The following two options affect the display of all matrices.

`compact`
: Remove blank lines around column number labels and between matrices producing more compact output with more data per page.

`loose`
: Insert blank lines above and below column number labels and between matrices to produce a more readable output with less data per page. (default).

See also: [fixed_point_format], page 51, [output_max_field_width], page 49, [output_precision], page 50, [split_long_rows], page 50, [print_empty_dimensions], page 51, [rats], page 458.

14.1.1.1 Paging Screen Output

When running interactively, Octave normally sends any output intended for your terminal that is more than one screen long to a paging program, such as `less` or `more`. This avoids the problem of having a large volume of output stream by before you can read it. With `less` (and some versions of `more`) you can also scan forward and backward, and search for specific items.

Normally, no output is displayed by the pager until just before Octave is ready to print the top level prompt, or read from the standard input (for example, by using the `fscanf` or `scanf` functions). This means that there may be some delay before any output appears on your screen if you have asked Octave to perform a significant amount of work with a single command statement. The function `fflush` may be used to force output to be sent to the pager (or any other stream) immediately.

You can select the program to run as the pager using the `PAGER` function, and you can turn paging off by using the function `more`.

`more` [Command]
`more` *on* [Command]
`more` *off* [Command]

Turn output pagination on or off.

Without an argument, `more` toggles the current state.

The current state can be determined via `page_screen_output`.

See also: [page_screen_output], page 235, [page_output_immediately], page 236, [PAGER], page 235, [PAGER_FLAGS], page 235.

val = PAGER () [Built-in Function]
old_val = PAGER (*new_val*) [Built-in Function]
PAGER (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the program to use to display terminal output on your system.

The default value is normally `"less"`, `"more"`, or `"pg"`, depending on what programs are installed on your system. See Appendix G [Installation], page 893.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [PAGER_FLAGS], page 235, [page_output_immediately], page 236, [more], page 234, [page_screen_output], page 235.

val = PAGER_FLAGS () [Built-in Function]
old_val = PAGER_FLAGS (*new_val*) [Built-in Function]
PAGER_FLAGS (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies the options to pass to the pager.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [PAGER], page 235, [more], page 234, [page_screen_output], page 235, [page_output_immediately], page 236.

val = page_screen_output () [Built-in Function]
old_val = page_screen_output (*new_val*) [Built-in Function]
page_screen_output (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether output intended for the terminal window that is longer than one page is sent through a pager.

This allows you to view one screenful at a time. Some pagers (such as `less`—see Appendix G [Installation], page 893) are also capable of moving backward on the output.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [more], page 234, [page_output_immediately], page 236, [PAGER], page 235, [PAGER_FLAGS], page 235.

val = `page_output_immediately ()` [Built-in Function]
old_val = `page_output_immediately (`*new_val*`)` [Built-in Function]
`page_output_immediately (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that controls whether Octave sends output to the pager as soon as it is available.

Otherwise, Octave buffers its output and waits until just before the prompt is printed to flush it to the pager.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [page_screen_output], page 235, [more], page 234, [PAGER], page 235, [PAGER_FLAGS], page 235.

`fflush (`*fid*`)` [Built-in Function]

Flush output to file descriptor *fid*.

`fflush` returns 0 on success and an OS dependent error value (−1 on Unix) on error.

Programming Note: Flushing is useful for ensuring that all pending output makes it to the screen before some other event occurs. For example, it is always a good idea to flush the standard output stream before calling `input`.

See also: [fopen], page 250, [fclose], page 251.

14.1.2 Terminal Input

Octave has three functions that make it easy to prompt users for input. The `input` and `menu` functions are normally used for managing an interactive dialog with a user, and the `keyboard` function is normally used for doing simple debugging.

ans = `input (`*prompt*`)` [Built-in Function]
ans = `input (`*prompt*`, "`*s*`")` [Built-in Function]

Print *prompt* and wait for user input.

For example,

```
input ("Pick a number, any number! ")
```

prints the prompt

```
Pick a number, any number!
```

and waits for the user to enter a value. The string entered by the user is evaluated as an expression, so it may be a literal constant, a variable name, or any other valid Octave code.

The number of return arguments, their size, and their class depend on the expression entered.

If you are only interested in getting a literal string value, you can call `input` with the character string `"s"` as the second argument. This tells Octave to return the string entered by the user directly, without evaluating it first.

Because there may be output waiting to be displayed by the pager, it is a good idea to always call `fflush (stdout)` before calling `input`. This will ensure that all pending output is written to the screen before your prompt.

See also: [yes_or_no], page 237, [kbhit], page 237, [pause], page 755, [menu], page 237, [listdlg], page 800.

choice = menu (*title*, *opt1*, ...) [Function File]
choice = menu (*title*, {*opt1*, ...}) [Function File]

Display a menu with heading *title* and options *opt1*, ..., and wait for user input.

If the GUI is running, or Java is available, the menu is displayed graphically using `listdlg`. Otherwise, the title and menu options are printed on the console.

title is a string and the options may be input as individual strings or as a cell array of strings.

The return value *choice* is the number of the option selected by the user counting from 1.

This function is useful for interactive programs. There is no limit to the number of options that may be passed in, but it may be confusing to present more than will fit easily on one screen.

See also: [input], page 236, [listdlg], page 800.

ans = yes_or_no ("*prompt*") [Built-in Function]

Ask the user a yes-or-no question.

Return logical true if the answer is yes or false if the answer is no.

Takes one argument, *prompt*, which is the string to display when asking the question. *prompt* should end in a space; `yes-or-no` adds the string '`(yes or no)` ' to it. The user must confirm the answer with `RET` and can edit it until it has been confirmed.

See also: [input], page 236.

For `input`, the normal command line history and editing functions are available at the prompt.

Octave also has a function that makes it possible to get a single character from the keyboard without requiring the user to type a carriage return.

kbhit () [Built-in Function]
kbhit (*1*) [Built-in Function]

Read a single keystroke from the keyboard.

If called with an argument, don't wait for a keypress.

For example,

```
x = kbhit ();
```

will set *x* to the next character typed at the keyboard as soon as it is typed.

```
x = kbhit (1);
```

is identical to the above example, but doesn't wait for a keypress, returning the empty string if no key is available.

See also: [input], page 236, [pause], page 755.

14.1.3 Simple File I/O

The `save` and `load` commands allow data to be written to and read from disk files in various formats. The default format of files written by the `save` command can be controlled using the functions `save_default_options` and `save_precision`.

As an example the following code creates a 3-by-3 matrix and saves it to the file '`myfile.mat`'.

```
A = [ 1:3; 4:6; 7:9 ];
save myfile.mat A
```

Once one or more variables have been saved to a file, they can be read into memory using the `load` command.

```
load myfile.mat
A
     ⊣ A =
     ⊣
     ⊣    1   2   3
     ⊣    4   5   6
     ⊣    7   8   9
```

`save` *file* [Command]
`save` *options file* [Command]
`save` *options file v1 v2 . . .* [Command]
`save` *options file -struct STRUCT f1 f2 . . .* [Command]
`save "-"` *v1 v2 . . .* [Command]
s `= save ("-"` *v1 v2 . . .*`)` [Built-in Function]

Save the named variables *v1*, *v2*, . . . , in the file *file*.

The special filename '`-`' may be used to return the content of the variables as a string. If no variable names are listed, Octave saves all the variables in the current scope. Otherwise, full variable names or pattern syntax can be used to specify the variables to save. If the `-struct` modifier is used, fields *f1 f2* . . . of the scalar structure *STRUCT* are saved as if they were variables with corresponding names. Valid options for the `save` command are listed in the following table. Options that modify the output format override the format specified by `save_default_options`.

If save is invoked using the functional form

```
save ("-option1", ..., "file", "v1", ...)
```

then the *options*, *file*, and variable name arguments (*v1*, . . .) must be specified as character strings.

If called with a filename of `"-"`, write the output to stdout if nargout is 0, otherwise return the output in a character string.

`-append` Append to the destination instead of overwriting.

`-ascii` Save a single matrix in a text file without header or any other information.

`-binary` Save the data in Octave's binary data format.

`-float-binary`
: Save the data in Octave's binary data format but only using single precision. Only use this format if you know that all the values to be saved can be represented in single precision.

`-hdf5`
: Save the data in HDF5 format. (HDF5 is a free, portable binary format developed by the National Center for Supercomputing Applications at the University of Illinois.) This format is only available if Octave was built with a link to the HDF5 libraries.

`-float-hdf5`
: Save the data in HDF5 format but only using single precision. Only use this format if you know that all the values to be saved can be represented in single precision.

`-V7`
`-v7`
`-7`
`-mat7-binary`
: Save the data in MATLAB's v7 binary data format.

`-V6`
`-v6`
`-6`
`-mat`
`-mat-binary`
: Save the data in MATLAB's v6 binary data format.

`-V4`
`-v4`
`-4`
`-mat4-binary`
: Save the data in the binary format written by MATLAB version 4.

`-text`
: Save the data in Octave's text data format. (default).

`-zip`
`-z`
: Use the gzip algorithm to compress the file. This works equally on files that are compressed with gzip outside of octave, and gzip can equally be used to convert the files for backward compatibility. This option is only available if Octave was built with a link to the zlib libraries.

The list of variables to save may use wildcard patterns containing the following special characters:

`?`
: Match any single character.

`*`
: Match zero or more characters.

`[ list ]`
: Match the list of characters specified by *list*. If the first character is `!` or `^`, match all characters except those specified by *list*. For example, the pattern `[a-zA-Z]` will match all lower and uppercase alphabetic characters.

Wildcards may also be used in the field name specifications when using the `-struct` modifier (but not in the struct name itself).

Except when using the MATLAB binary data file format or the '`-ascii`' format, saving global variables also saves the global status of the variable. If the variable is restored at a later time using '`load`', it will be restored as a global variable.

The command

```
save -binary data a b*
```

saves the variable '`a`' and all variables beginning with '`b`' to the file `data` in Octave's binary format.

See also: [load], page 241, [save_default_options], page 240, [save_header_format_string], page 240, [dlmread], page 244, [csvread], page 244, [fread], page 262.

There are three functions that modify the behavior of `save`.

val = `save_default_options ()` [Built-in Function]
old_val = `save_default_options (`*new_val*`)` [Built-in Function]
`save_default_options (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that specifies the default options for the `save` command, and defines the default format.

Typical values include `"-ascii"`, `"-text -zip"`. The default value is `-text`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [save], page 238.

val = `save_precision ()` [Built-in Function]
old_val = `save_precision (`*new_val*`)` [Built-in Function]
`save_precision (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that specifies the number of digits to keep when saving data in text format.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

val = `save_header_format_string ()` [Built-in Function]
old_val = `save_header_format_string (`*new_val*`)` [Built-in Function]
`save_header_format_string (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that specifies the format string used for the comment line written at the beginning of text-format data files saved by Octave.

The format string is passed to `strftime` and should begin with the character '`#`' and contain no newline characters. If the value of `save_header_format_string` is the empty string, the header comment is omitted from text-format data files. The default value is

```
"# Created by Octave VERSION, %a %b %d %H:%M:%S %Y %Z <USER@HOST>"
```

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [strftime], page 751, [save], page 238.

`load` *file* [Command]
`load` *options file* [Command]
`load` *options file v1 v2 . . .* [Command]
`S = load ("`*options*`", "`*file*`", "`*v1*`", "`*v2*`", ...)` [Command]
`load` *file options* [Command]
`load` *file options v1 v2 . . .* [Command]
`S = load ("`*file*`", "`*options*`", "`*v1*`", "`*v2*`", ...)` [Command]

Load the named variables *v1*, *v2*, . . ., from the file *file*.

If no variables are specified then all variables found in the file will be loaded. As with `save`, the list of variables to extract can be full names or use a pattern syntax. The format of the file is automatically detected but may be overridden by supplying the appropriate option.

If load is invoked using the functional form

```
load ("-option1", ..., "file", "v1", ...)
```

then the *options*, *file*, and variable name arguments (*v1*, . . .) must be specified as character strings.

If a variable that is not marked as global is loaded from a file when a global symbol with the same name already exists, it is loaded in the global symbol table. Also, if a variable is marked as global in a file and a local symbol exists, the local symbol is moved to the global symbol table and given the value from the file.

If invoked with a single output argument, Octave returns data instead of inserting variables in the symbol table. If the data file contains only numbers (TAB- or space-delimited columns), a matrix of values is returned. Otherwise, `load` returns a structure with members corresponding to the names of the variables in the file.

The `load` command can read data stored in Octave's text and binary formats, and MATLAB's binary format. If compiled with zlib support, it can also load gzip-compressed files. It will automatically detect the type of file and do conversion from different floating point formats (currently only IEEE big and little endian, though other formats may be added in the future).

Valid options for `load` are listed in the following table.

`-force` This option is accepted for backward compatibility but is ignored. Octave now overwrites variables currently in memory with those of the same name found in the file.

`-ascii` Force Octave to assume the file contains columns of numbers in text format without any header or other information. Data in the file will be loaded as a single numeric matrix with the name of the variable derived from the name of the file.

`-binary` Force Octave to assume the file is in Octave's binary format.

`-hdf5` Force Octave to assume the file is in HDF5 format. (HDF5 is a free, portable binary format developed by the National Center for Supercomputing Applications at the University of Illinois.) Note that Octave can read HDF5 files not created by itself, but may skip some datasets in formats that it cannot support. This format is only available if Octave was built with a link to the HDF5 libraries.

`-import` This option is accepted for backward compatibility but is ignored. Octave can now support multi-dimensional HDF data and automatically modifies variable names if they are invalid Octave identifiers.

`-mat`
`-mat-binary`
`-6`
`-v6`
`-7`
`-v7` Force Octave to assume the file is in MATLAB's version 6 or 7 binary format.

`-mat4-binary`
`-4`
`-v4`
`-V4` Force Octave to assume the file is in the binary format written by MATLAB version 4.

`-text` Force Octave to assume the file is in Octave's text format.

See also: [save], page 238, [dlmwrite], page 243, [csvwrite], page 244, [fwrite], page 265.

str = `fileread` (*filename*) [Function File]

Read the contents of *filename* and return it as a string.

See also: [fread], page 262, [textread], page 244, [sscanf], page 260.

`native_float_format ()` [Built-in Function]

Return the native floating point format as a string.

It is possible to write data to a file in a similar way to the `disp` function for writing data to the screen. The `fdisp` works just like `disp` except its first argument is a file pointer as created by `fopen`. As an example, the following code writes to data '`myfile.txt`'.

```
fid = fopen ("myfile.txt", "w");
fdisp (fid, "3/8 is ");
fdisp (fid, 3/8);
fclose (fid);
```

See Section 14.2.1 [Opening and Closing Files], page 249, for details on how to use `fopen` and `fclose`.

`fdisp` (*fid*, *x*) [Built-in Function]

Display the value of *x* on the stream *fid*.

For example:

```
fdisp (stdout, "The value of pi is:"), fdisp (stdout, pi)

     ⊣ the value of pi is:
     ⊣ 3.1416
```

Note that the output from `fdisp` always ends with a newline.

See also: [disp], page 231.

Octave can also read and write matrices text files such as comma separated lists.

`dlmwrite` (*file*, *M*) [Function File]
`dlmwrite` (*file*, *M*, *delim*, *r*, *c*) [Function File]
`dlmwrite` (*file*, *M*, *key*, *val* ...) [Function File]
`dlmwrite` (*file*, *M*, "*-append*", ...) [Function File]
`dlmwrite` (*fid*, ...) [Function File]

Write the matrix *M* to the named file using delimiters.

file should be a file name or writable file ID given by `fopen`.

The parameter *delim* specifies the delimiter to use to separate values on a row.

The value of *r* specifies the number of delimiter-only lines to add to the start of the file.

The value of *c* specifies the number of delimiters to prepend to each line of data.

If the argument `"-append"` is given, append to the end of *file*.

In addition, the following keyword value pairs may appear at the end of the argument list:

`"append"` Either `"on"` or `"off"`. See `"-append"` above.

`"delimiter"`
: See *delim* above.

`"newline"`
: The character(s) to use to separate each row. Three special cases exist for this option. `"unix"` is changed into `"\n"`, `"pc"` is changed into `"\r\n"`, and `"mac"` is changed into `"\r"`. Any other value is used directly as the newline separator.

`"roffset"`
: See *r* above.

`"coffset"`
: See *c* above.

`"precision"`
: The precision to use when writing the file. It can either be a format string (as used by fprintf) or a number of significant digits.

```
dlmwrite ("file.csv", reshape (1:16, 4, 4));
dlmwrite ("file.tex", a, "delimiter", "&", "newline", "\n")
```

See also: [dlmread], page 244, [csvread], page 244, [csvwrite], page 244.

`data = dlmread (file)` [Built-in Function]
`data = dlmread (file, sep)` [Built-in Function]
`data = dlmread (file, sep, r0, c0)` [Built-in Function]
`data = dlmread (file, sep, range)` [Built-in Function]
`data = dlmread (..., "emptyvalue", EMPTYVAL)` [Built-in Function]

Read the matrix *data* from a text file which uses the delimiter *sep* between data values.

If *sep* is not defined the separator between fields is determined from the file itself.

Given two scalar arguments *r0* and *c0*, these define the starting row and column of the data to be read. These values are indexed from zero, such that the first row corresponds to an index of zero.

The *range* parameter may be a 4-element vector containing the upper left and lower right corner `[R0,C0,R1,C1]` where the lowest index value is zero. Alternatively, a spreadsheet style range such as `"A2..Q15"` or `"T1:AA5"` can be used. The lowest alphabetical index `'A'` refers to the first column. The lowest row index is 1.

file should be a file name or file id given by `fopen`. In the latter case, the file is read until end of file is reached.

The `"emptyvalue"` option may be used to specify the value used to fill empty fields. The default is zero.

See also: [csvread], page 244, [textscan], page 246, [textread], page 244, [dlmwrite], page 243.

`csvwrite (filename, x)` [Function File]
`csvwrite (filename, x, dlm_opts)` [Function File]

Write the matrix *x* to the file *filename* in comma-separated-value format.

This function is equivalent to

```
dlmwrite (filename, x, ",", ...)
```

See also: [csvread], page 244, [dlmwrite], page 243, [dlmread], page 244.

`x = csvread (filename)` [Function File]
`x = csvread (filename, dlm_opts)` [Function File]

Read the comma-separated-value file *filename* into the matrix *x*.

This function is equivalent to

```
x = dlmread (filename, "," , ...)
```

See also: [csvwrite], page 244, [dlmread], page 244, [dlmwrite], page 243.

Formatted data from can be read from, or written to, text files as well.

`[a, ...] = textread (filename)` [Function File]
`[a, ...] = textread (filename, format)` [Function File]
`[a, ...] = textread (filename, format, n)` [Function File]
`[a, ...] = textread (filename, format, prop1, value1, ...)` [Function File]
`[a, ...] = textread (filename, format, n, prop1, value1, ...)` [Function File]

Read data from a text file.

The file *filename* is read and parsed according to *format*. The function behaves like `strread` except it works by parsing a file instead of a string. See the documentation of `strread` for details.

In addition to the options supported by `strread`, this function supports two more:

- `"headerlines"`: The first *value* number of lines of *filename* are skipped.
- `"endofline"`: Specify a single character or `"\r\n"`. If no value is given, it will be inferred from the file. If set to `""` (empty string) EOLs are ignored as delimiters.

The optional input *n* (format repeat count) specifies the number of times the format string is to be used or the number of lines to be read, whichever happens first while reading. The former is equivalent to requesting that the data output vectors should be of length *N*. Note that when reading files with format strings referring to multiple lines, *n* should rather be the number of lines to be read than the number of format string uses.

If the format string is empty (not just omitted) and the file contains only numeric data (excluding headerlines), textread will return a rectangular matrix with the number of columns matching the number of numeric fields on the first data line of the file. Empty fields are returned as zero values.

Examples:

```
Assume a data file like:
1 a 2 b
3 c 4 d
5 e

[a, b] = textread (f, "%f %s")
returns two columns of data, one with doubles, the other a
cellstr array:
a = [1; 2; 3; 4; 5]
b = {"a"; "b"; "c"; "d"; "e"}

[a, b] = textread (f, "%f %s", 3)
(read data into two culumns, try to use the format string
three times)
returns
a = [1; 2; 3]
b = {"a"; "b"; "c"}


With a data file like:
1
a
2
b

[a, b] = textread (f, "%f %s", 2)
returns a = 1 and b = {"a"}; i.e., the format string is used
only once because the format string refers to 2 lines of the
data file. To obtain 2x1 data output columns, specify N = 4
```

```
(number of data lines containing all requested data) rather
than 2.
```

See also: [strread], page 84, [load], page 241, [dlmread], page 244, [fscanf], page 259, [textscan], page 246.

C = textscan (*fid*, *format*) [Function File]
C = textscan (*fid*, *format*, *n*) [Function File]
C = textscan (*fid*, *format*, *param*, *value*, ...) [Function File]
C = textscan (*fid*, *format*, *n*, *param*, *value*, ...) [Function File]
C = textscan (*str*, ...) [Function File]
[*C*, *position*] = textscan (*fid*, ...) [Function File]

Read data from a text file or string.

The string *str* or file associated with *fid* is read from and parsed according to *format*. The function behaves like `strread` except it can also read from file instead of a string. See the documentation of `strread` for details.

In addition to the options supported by `strread`, this function supports a few more:

- `"collectoutput"`: A value of 1 or true instructs textscan to concatenate consecutive columns of the same class in the output cell array. A value of 0 or false (default) leaves output in distinct columns.
- `"endofline"`: Specify `"\r"`, `"\n"` or `"\r\n"` (for CR, LF, or CRLF). If no value is given, it will be inferred from the file. If set to `""` (empty string) EOLs are ignored as delimiters and added to whitespace.
- `"headerlines"`: The first *value* number of lines of *fid* are skipped.
- `"returnonerror"`: If set to numerical 1 or true (default), return normally when read errors have been encountered. If set to 0 or false, return an error and no data. As the string or file is read by columns rather than by rows, and because textscan is fairly forgiving as regards read errors, setting this option may have little or no actual effect.

When reading from a character string, optional input argument *n* specifies the number of times *format* should be used (i.e., to limit the amount of data read). When reading from file, *n* specifies the number of data lines to read; in this sense it differs slightly from the format repeat count in strread.

The output *C* is a cell array whose second dimension is determined by the number of format specifiers.

The second output, *position*, provides the position, in characters, from the beginning of the file.

If the format string is empty (not: omitted) and the file contains only numeric data (excluding headerlines), textscan will return data in a number of columns matching the number of numeric fields on the first data line of the file.

See also: [dlmread], page 244, [fscanf], page 259, [load], page 241, [strread], page 84, [textread], page 244.

The `importdata` function has the ability to work with a wide variety of data.

`A = importdata (fname)` [Function File]
`A = importdata (fname, delimiter)` [Function File]
`A = importdata (fname, delimiter, header_rows)` [Function File]
`[A, delimiter] = importdata (...)` [Function File]
`[A, delimiter, header_rows] = importdata (...)` [Function File]

Import data from the file *fname*.

Input parameters:

- *fname* The name of the file containing data.
- *delimiter* The character separating columns of data. Use `\t` for tab. (Only valid for ASCII files)
- *header_rows* The number of header rows before the data begins. (Only valid for ASCII files)

Different file types are supported:

- ASCII table

  Import ASCII table using the specified number of header rows and the specified delimiter.
- Image file
- MATLAB file
- Spreadsheet files (depending on external software)
- WAV file

See also: [textscan], page 246, [dlmread], page 244, [csvread], page 244, [load], page 241.

14.1.3.1 Saving Data on Unexpected Exits

If Octave for some reason exits unexpectedly it will by default save the variables available in the workspace to a file in the current directory. By default this file is named '`octave-workspace`' and can be loaded into memory with the `load` command. While the default behavior most often is reasonable it can be changed through the following functions.

`val = crash_dumps_octave_core ()` [Built-in Function]
`old_val = crash_dumps_octave_core (new_val)` [Built-in Function]
`crash_dumps_octave_core (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether Octave tries to save all current variables to the file `octave-workspace` if it crashes or receives a hangup, terminate or similar signal.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [octave_core_file_limit], page 248, [octave_core_file_name], page 248, [octave_core_file_options], page 248.

`val = sighup_dumps_octave_core ()` [Built-in Function]
`old_val = sighup_dumps_octave_core (new_val)` [Built-in Function]

`sighup_dumps_octave_core (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether Octave tries to save all current variables to the file `octave-workspace` if it receives a hangup signal.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

`val = sigterm_dumps_octave_core ()` [Built-in Function]
`old_val = sigterm_dumps_octave_core (new_val)` [Built-in Function]
`sigterm_dumps_octave_core (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether Octave tries to save all current variables to the file `octave-workspace` if it receives a terminate signal.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

`val = octave_core_file_options ()` [Built-in Function]
`old_val = octave_core_file_options (new_val)` [Built-in Function]
`octave_core_file_options (new_val, "local")` [Built-in Function]

Query or set the internal variable that specifies the options used for saving the workspace data if Octave aborts.

The value of `octave_core_file_options` should follow the same format as the options for the `save` function. The default value is Octave's binary format.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [crash_dumps_octave_core], page 247, [octave_core_file_name], page 248, [octave_core_file_limit], page 248.

`val = octave_core_file_limit ()` [Built-in Function]
`old_val = octave_core_file_limit (new_val)` [Built-in Function]
`octave_core_file_limit (new_val, "local")` [Built-in Function]

Query or set the internal variable that specifies the maximum amount of memory (in kilobytes) of the top-level workspace that Octave will attempt to save when writing data to the crash dump file (the name of the file is specified by *octave_core_file_name*).

If *octave_core_file_options* flags specify a binary format, then *octave_core_file_limit* will be approximately the maximum size of the file. If a text file format is used, then the file could be much larger than the limit. The default value is -1 (unlimited)

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [crash_dumps_octave_core], page 247, [octave_core_file_name], page 248, [octave_core_file_options], page 248.

`val = octave_core_file_name ()` [Built-in Function]
`old_val = octave_core_file_name (new_val)` [Built-in Function]

`octave_core_file_name (new_val, "local")` [Built-in Function]

Query or set the internal variable that specifies the name of the file used for saving data from the top-level workspace if Octave aborts.

The default value is `"octave-workspace"`

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [crash_dumps_octave_core], page 247, [octave_core_file_name], page 248, [octave_core_file_options], page 248.

14.2 C-Style I/O Functions

Octave's C-style input and output functions provide most of the functionality of the C programming language's standard I/O library. The argument lists for some of the input functions are slightly different, however, because Octave has no way of passing arguments by reference.

In the following, *file* refers to a file name and `fid` refers to an integer file number, as returned by `fopen`.

There are three files that are always available. Although these files can be accessed using their corresponding numeric file ids, you should always use the symbolic names given in the table below, since it will make your programs easier to understand.

`stdin ()` [Built-in Function]

Return the numeric value corresponding to the standard input stream.

When Octave is used interactively, stdin is filtered through the command line editing functions.

See also: [stdout], page 249, [stderr], page 249.

`stdout ()` [Built-in Function]

Return the numeric value corresponding to the standard output stream.

Data written to the standard output is normally filtered through the pager.

See also: [stdin], page 249, [stderr], page 249.

`stderr ()` [Built-in Function]

Return the numeric value corresponding to the standard error stream.

Even if paging is turned on, the standard error is not sent to the pager. It is useful for error messages and prompts.

See also: [stdin], page 249, [stdout], page 249.

14.2.1 Opening and Closing Files

When reading data from a file it must be opened for reading first, and likewise when writing to a file. The `fopen` function returns a pointer to an open file that is ready to be read or written. Once all data has been read from or written to the opened file it should be closed.

The `fclose` function does this. The following code illustrates the basic pattern for writing to a file, but a very similar pattern is used when reading a file.

```
filename = "myfile.txt";
fid = fopen (filename, "w");
# Do the actual I/O here...
fclose (fid);
```

fid = fopen (*name*) [Built-in Function]
fid = fopen (*name*, *mode*) [Built-in Function]
fid = fopen (*name*, *mode*, *arch*) [Built-in Function]
[*fid*, *msg*] = fopen (...) [Built-in Function]
fid_list = fopen ("*all*") [Built-in Function]
[*file*, *mode*, *arch*] = fopen (*fid*) [Built-in Function]

Open a file for low-level I/O or query open files and file descriptors.

The first form of the `fopen` function opens the named file with the specified mode (read-write, read-only, etc.) and architecture interpretation (IEEE big endian, IEEE little endian, etc.), and returns an integer value that may be used to refer to the file later. If an error occurs, *fid* is set to -1 and *msg* contains the corresponding system error message. The *mode* is a one or two character string that specifies whether the file is to be opened for reading, writing, or both.

The second form of the `fopen` function returns a vector of file ids corresponding to all the currently open files, excluding the `stdin`, `stdout`, and `stderr` streams.

The third form of the `fopen` function returns information about the open file given its file id.

For example,

```
myfile = fopen ("splat.dat", "r", "ieee-le");
```

opens the file `splat.dat` for reading. If necessary, binary numeric values will be read assuming they are stored in IEEE format with the least significant bit first, and then converted to the native representation.

Opening a file that is already open simply opens it again and returns a separate file id. It is not an error to open a file several times, though writing to the same file through several different file ids may produce unexpected results.

The possible values '`mode`' may have are

'`r`' (default)
: Open a file for reading.

'`w`'
: Open a file for writing. The previous contents are discarded.

'`a`'
: Open or create a file for writing at the end of the file.

'`r+`'
: Open an existing file for reading and writing.

'`w+`'
: Open a file for reading or writing. The previous contents are discarded.

'`a+`'
: Open or create a file for reading or writing at the end of the file.

Append a `"t"` to the mode string to open the file in text mode or a `"b"` to open in binary mode. On Windows and Macintosh systems, text mode reading and writing automatically converts linefeeds to the appropriate line end character for the system (carriage-return linefeed on Windows, carriage-return on Macintosh). The default when no mode is specified is binary mode.

Additionally, you may append a `"z"` to the mode string to open a gzipped file for reading or writing. For this to be successful, you must also open the file in binary mode.

The parameter *arch* is a string specifying the default data format for the file. Valid values for *arch* are:

`"native"` or `"n"` (default)
: The format of the current machine.

`"ieee-be"` or `"b"`
: IEEE big endian format.

`"ieee-le"` or `"l"`
: IEEE little endian format.

However, conversions are currently only supported for '`native`', '`ieee-be`', and '`ieee-le`' formats.

When opening a new file that does not yet exist, permissions will be set to `0666 - umask`.

See also: [fclose], page 251, [fgets], page 253, [fgetl], page 252, [fscanf], page 259, [fread], page 262, [fputs], page 252, [fdisp], page 242, [fprintf], page 254, [fwrite], page 265, [fskipl], page 253, [fseek], page 268, [frewind], page 269, [ftell], page 268, [feof], page 267, [ferror], page 267, [fclear], page 267, [fflush], page 236, [freport], page 268, [umask], page 762.

`fclose (`*fid*`)` [Built-in Function]
`fclose ("`*all*`")` [Built-in Function]
status `= fclose ("`*all*`")` [Built-in Function]

Close the file specified by the file descriptor *fid*.

If successful, `fclose` returns 0, otherwise, it returns -1. The second form of the `fclose` call closes all open files except `stdout`, `stderr`, and `stdin`.

Programming Note: When using `"all"` the file descriptors associated with gnuplot will also be closed. This will prevent further plotting with gnuplot until Octave is closed and restarted.

See also: [fopen], page 250, [fflush], page 236, [freport], page 268.

`is_valid_file_id (`*fid*`)` [Function File]

Return true if *fid* refers to an open file.

See also: [freport], page 268, [fopen], page 250.

14.2.2 Simple Output

Once a file has been opened for writing a string can be written to the file using the `fputs` function. The following example shows how to write the string '`Free Software is needed for Free Science`' to the file '`free.txt`'.

```
filename = "free.txt";
fid = fopen (filename, "w");
fputs (fid, "Free Software is needed for Free Science");
fclose (fid);
```

`fputs (fid, string)` [Built-in Function]
`status = fputs (fid, string)` [Built-in Function]

Write the string *string* to the file with file descriptor *fid*.

The string is written to the file with no additional formatting. Use `fdisp` instead to automatically append a newline character appropriate for the local machine.

Return a non-negative number on success or EOF on error.

See also: [fdisp], page 242, [fprintf], page 254, [fwrite], page 265, [fopen], page 250.

A function much similar to `fputs` is available for writing data to the screen. The `puts` function works just like `fputs` except it doesn't take a file pointer as its input.

`puts (string)` [Built-in Function]
`status = puts (string)` [Built-in Function]

Write a string to the standard output with no formatting.

The string is written verbatim to the standard output. Use `disp` to automatically append a newline character appropriate for the local machine.

Return a non-negative number on success and EOF on error.

See also: [fputs], page 252, [disp], page 231.

14.2.3 Line-Oriented Input

To read from a file it must be opened for reading using `fopen`. Then a line can be read from the file using `fgetl` as the following code illustrates

```
fid = fopen ("free.txt");
txt = fgetl (fid)
     ⊣ Free Software is needed for Free Science
fclose (fid);
```

This of course assumes that the file '`free.txt`' exists and contains the line '`Free Software is needed for Free Science`'.

`str = fgetl (fid)` [Built-in Function]
`str = fgetl (fid, len)` [Built-in Function]

Read characters from a file, stopping after a newline, or EOF, or *len* characters have been read.

The characters read, excluding the possible trailing newline, are returned as a string.

If *len* is omitted, `fgetl` reads until the next newline character.

If there are no more characters to read, `fgetl` returns -1.

To read a line and return the terminating newline see `fgets`.

See also: [fgets], page 253, [fscanf], page 259, [fread], page 262, [fopen], page 250.

str = `fgets` (*fid*) [Built-in Function]
str = `fgets` (*fid*, *len*) [Built-in Function]

Read characters from a file, stopping after a newline, or EOF, or *len* characters have been read.

The characters read, including the possible trailing newline, are returned as a string.

If *len* is omitted, `fgets` reads until the next newline character.

If there are no more characters to read, `fgets` returns -1.

To read a line and discard the terminating newline see `fgetl`.

See also: [fputs], page 252, [fgetl], page 252, [fscanf], page 259, [fread], page 262, [fopen], page 250.

nlines = `fskipl` (*fid*) [Built-in Function]
nlines = `fskipl` (*fid*, *count*) [Built-in Function]
nlines = `fskipl` (*fid*, *Inf*) [Built-in Function]

Read and skip *count* lines from the file specified by the file descriptor *fid*.

`fskipl` discards characters until an end-of-line is encountered exactly *count*-times, or until the end-of-file marker is found.

If *count* is omitted, it defaults to 1. *count* may also be `Inf`, in which case lines are skipped until the end of the file. This form is suitable for counting the number of lines in a file.

Returns the number of lines skipped (end-of-line sequences encountered).

See also: [fgetl], page 252, [fgets], page 253, [fscanf], page 259, [fopen], page 250.

14.2.4 Formatted Output

This section describes how to call `printf` and related functions.

The following functions are available for formatted output. They are modeled after the C language functions of the same name, but they interpret the format template differently in order to improve the performance of printing vector and matrix values.

Implementation Note: For compatibility with MATLAB, escape sequences in the template string (e.g., `"\n"` => newline) are expanded even when the template string is defined with single quotes.

`printf` (*template*, ...) [Built-in Function]

Print optional arguments under the control of the template string *template* to the stream `stdout` and return the number of characters printed.

See the Formatted Output section of the GNU Octave manual for a complete description of the syntax of the template string.

Implementation Note: For compatibility with MATLAB, escape sequences in the template string (e.g., `"\n"` => newline) are expanded even when the template string is defined with single quotes.

See also: [fprintf], page 254, [sprintf], page 254, [scanf], page 260.

`fprintf` (*fid*, *template*, ...) [Built-in Function]
`fprintf` (*template*, ...) [Built-in Function]
numbytes = `fprintf` (...) [Built-in Function]

This function is equivalent to `printf`, except that the output is written to the file descriptor *fid* instead of `stdout`.

If *fid* is omitted, the output is written to `stdout` making the function exactly equivalent to `printf`.

The optional output returns the number of bytes written to the file.

Implementation Note: For compatibility with MATLAB, escape sequences in the template string (e.g., `"\n"` => newline) are expanded even when the template string is defined with single quotes.

See also: [fputs], page 252, [fdisp], page 242, [fwrite], page 265, [fscanf], page 259, [printf], page 253, [sprintf], page 254, [fopen], page 250.

`sprintf` (*template*, ...) [Built-in Function]

This is like `printf`, except that the output is returned as a string.

Unlike the C library function, which requires you to provide a suitably sized string as an argument, Octave's `sprintf` function returns the string, automatically sized to hold all of the items converted.

Implementation Note: For compatibility with MATLAB, escape sequences in the template string (e.g., `"\n"` => newline) are expanded even when the template string is defined with single quotes.

See also: [printf], page 253, [fprintf], page 254, [sscanf], page 260.

The `printf` function can be used to print any number of arguments. The template string argument you supply in a call provides information not only about the number of additional arguments, but also about their types and what style should be used for printing them.

Ordinary characters in the template string are simply written to the output stream as-is, while *conversion specifications* introduced by a '`%`' character in the template cause subsequent arguments to be formatted and written to the output stream. For example,

```
pct = 37;
filename = "foo.txt";
printf ("Processed %d%% of '%s'.\nPlease be patient.\n",
        pct, filename);
```

produces output like

```
Processed 37% of 'foo.txt'.
Please be patient.
```

This example shows the use of the '`%d`' conversion to specify that a scalar argument should be printed in decimal notation, the '`%s`' conversion to specify printing of a string argument, and the '`%%`' conversion to print a literal '`%`' character.

There are also conversions for printing an integer argument as an unsigned value in octal, decimal, or hexadecimal radix ('`%o`', '`%u`', or '`%x`', respectively); or as a character value ('`%c`').

Floating-point numbers can be printed in normal, fixed-point notation using the '%f' conversion or in exponential notation using the '%e' conversion. The '%g' conversion uses either '%e' or '%f' format, depending on what is more appropriate for the magnitude of the particular number.

You can control formatting more precisely by writing *modifiers* between the '%' and the character that indicates which conversion to apply. These slightly alter the ordinary behavior of the conversion. For example, most conversion specifications permit you to specify a minimum field width and a flag indicating whether you want the result left- or right-justified within the field.

The specific flags and modifiers that are permitted and their interpretation vary depending on the particular conversion. They're all described in more detail in the following sections.

14.2.5 Output Conversion for Matrices

When given a matrix value, Octave's formatted output functions cycle through the format template until all the values in the matrix have been printed. For example:

```
printf ("%4.2f %10.2e %8.4g\n", hilb (3));

     ⊣ 1.00   5.00e-01   0.3333
     ⊣ 0.50   3.33e-01     0.25
     ⊣ 0.33   2.50e-01      0.2
```

If more than one value is to be printed in a single call, the output functions do not return to the beginning of the format template when moving on from one value to the next. This can lead to confusing output if the number of elements in the matrices are not exact multiples of the number of conversions in the format template. For example:

```
printf ("%4.2f %10.2e %8.4g\n", [1, 2], [3, 4]);

     ⊣ 1.00   2.00e+00        3
     ⊣ 4.00
```

If this is not what you want, use a series of calls instead of just one.

14.2.6 Output Conversion Syntax

This section provides details about the precise syntax of conversion specifications that can appear in a `printf` template string.

Characters in the template string that are not part of a conversion specification are printed as-is to the output stream.

The conversion specifications in a `printf` template string have the general form:

`% flags width [ . precision ] type conversion`

For example, in the conversion specifier '%-10.8ld', the '-' is a flag, '10' specifies the field width, the precision is '8', the letter 'l' is a type modifier, and 'd' specifies the conversion style. (This particular type specifier says to print a numeric argument in decimal notation, with a minimum of 8 digits left-justified in a field at least 10 characters wide.)

In more detail, output conversion specifications consist of an initial '%' character followed in sequence by:

- Zero or more *flag characters* that modify the normal behavior of the conversion specification.
- An optional decimal integer specifying the *minimum field width*. If the normal conversion produces fewer characters than this, the field is padded with spaces to the specified width. This is a *minimum* value; if the normal conversion produces more characters than this, the field is *not* truncated. Normally, the output is right-justified within the field.

  You can also specify a field width of '`*`'. This means that the next argument in the argument list (before the actual value to be printed) is used as the field width. The value is rounded to the nearest integer. If the value is negative, this means to set the '`-`' flag (see below) and to use the absolute value as the field width.
- An optional *precision* to specify the number of digits to be written for the numeric conversions. If the precision is specified, it consists of a period ('`.`') followed optionally by a decimal integer (which defaults to zero if omitted).

  You can also specify a precision of '`*`'. This means that the next argument in the argument list (before the actual value to be printed) is used as the precision. The value must be an integer, and is ignored if it is negative.
- An optional *type modifier character*. This character is ignored by Octave's `printf` function, but is recognized to provide compatibility with the C language `printf`.
- A character that specifies the conversion to be applied.

The exact options that are permitted and how they are interpreted vary between the different conversion specifiers. See the descriptions of the individual conversions for information about the particular options that they use.

14.2.7 Table of Output Conversions

Here is a table summarizing what all the different conversions do:

'`%d`', '`%i`'
: Print an integer as a signed decimal number. See Section 14.2.8 [Integer Conversions], page 257, for details. '`%d`' and '`%i`' are synonymous for output, but are different when used with `scanf` for input (see Section 14.2.13 [Table of Input Conversions], page 261).

'`%o`'
: Print an integer as an unsigned octal number. See Section 14.2.8 [Integer Conversions], page 257, for details.

'`%u`'
: Print an integer as an unsigned decimal number. See Section 14.2.8 [Integer Conversions], page 257, for details.

'`%x`', '`%X`'
: Print an integer as an unsigned hexadecimal number. '`%x`' uses lowercase letters and '`%X`' uses uppercase. See Section 14.2.8 [Integer Conversions], page 257, for details.

'`%f`'
: Print a floating-point number in normal (fixed-point) notation. See Section 14.2.9 [Floating-Point Conversions], page 258, for details.

'`%e`', '`%E`'
: Print a floating-point number in exponential notation. '`%e`' uses lowercase letters and '`%E`' uses uppercase. See Section 14.2.9 [Floating-Point Conversions], page 258, for details.

'%g', '%G' Print a floating-point number in either normal (fixed-point) or exponential notation, whichever is more appropriate for its magnitude. '%g' uses lowercase letters and '%G' uses uppercase. See Section 14.2.9 [Floating-Point Conversions], page 258, for details.

'%c' Print a single character. See Section 14.2.10 [Other Output Conversions], page 258.

'%s' Print a string. See Section 14.2.10 [Other Output Conversions], page 258.

'%%' Print a literal '%' character. See Section 14.2.10 [Other Output Conversions], page 258.

If the syntax of a conversion specification is invalid, unpredictable things will happen, so don't do this. In particular, MATLAB allows a bare percentage sign '%' with no subsequent conversion character. Octave will emit an error and stop if it sees such code. When the string variable to be processed cannot be guaranteed to be free of potential format codes it is better to use the two argument form of any of the `printf` functions and set the format string to `%s`. Alternatively, for code which is not required to be backwards-compatible with MATLAB the Octave function `puts` or `disp` can be used.

```
printf (strvar);        # Unsafe if strvar contains format codes
printf ("%s", strvar);  # Safe
puts (strvar);          # Safe
```

If there aren't enough function arguments provided to supply values for all the conversion specifications in the template string, or if the arguments are not of the correct types, the results are unpredictable. If you supply more arguments than conversion specifications, the extra argument values are simply ignored; this is sometimes useful.

14.2.8 Integer Conversions

This section describes the options for the '%d', '%i', '%o', '%u', '%x', and '%X' conversion specifications. These conversions print integers in various formats.

The '%d' and '%i' conversion specifications both print an numeric argument as a signed decimal number; while '%o', '%u', and '%x' print the argument as an unsigned octal, decimal, or hexadecimal number (respectively). The '%X' conversion specification is just like '%x' except that it uses the characters 'ABCDEF' as digits instead of 'abcdef'.

The following flags are meaningful:

'-' Left-justify the result in the field (instead of the normal right-justification).

'+' For the signed '%d' and '%i' conversions, print a plus sign if the value is positive.

' ' For the signed '%d' and '%i' conversions, if the result doesn't start with a plus or minus sign, prefix it with a space character instead. Since the '+' flag ensures that the result includes a sign, this flag is ignored if you supply both of them.

'#' For the '%o' conversion, this forces the leading digit to be '0', as if by increasing the precision. For '%x' or '%X', this prefixes a leading '0x' or '0X' (respectively) to the result. This doesn't do anything useful for the '%d', '%i', or '%u' conversions.

'0' Pad the field with zeros instead of spaces. The zeros are placed after any indication of sign or base. This flag is ignored if the '-' flag is also specified, or if a precision is specified.

If a precision is supplied, it specifies the minimum number of digits to appear; leading zeros are produced if necessary. If you don't specify a precision, the number is printed with as many digits as it needs. If you convert a value of zero with an explicit precision of zero, then no characters at all are produced.

14.2.9 Floating-Point Conversions

This section discusses the conversion specifications for floating-point numbers: the '`%f`', '`%e`', '`%E`', '`%g`', and '`%G`' conversions.

The '`%f`' conversion prints its argument in fixed-point notation, producing output of the form [`-`]*ddd*`.`*ddd*, where the number of digits following the decimal point is controlled by the precision you specify.

The '`%e`' conversion prints its argument in exponential notation, producing output of the form [`-`]*d*`.`*ddd*`e`[`+`|`-`]*dd*. Again, the number of digits following the decimal point is controlled by the precision. The exponent always contains at least two digits. The '`%E`' conversion is similar but the exponent is marked with the letter '`E`' instead of '`e`'.

The '`%g`' and '`%G`' conversions print the argument in the style of '`%e`' or '`%E`' (respectively) if the exponent would be less than -4 or greater than or equal to the precision; otherwise they use the '`%f`' style. Trailing zeros are removed from the fractional portion of the result and a decimal-point character appears only if it is followed by a digit.

The following flags can be used to modify the behavior:

'`-`' Left-justify the result in the field. Normally the result is right-justified.

'`+`' Always include a plus or minus sign in the result.

'` `' If the result doesn't start with a plus or minus sign, prefix it with a space instead. Since the '`+`' flag ensures that the result includes a sign, this flag is ignored if you supply both of them.

'`#`' Specifies that the result should always include a decimal point, even if no digits follow it. For the '`%g`' and '`%G`' conversions, this also forces trailing zeros after the decimal point to be left in place where they would otherwise be removed.

'`0`' Pad the field with zeros instead of spaces; the zeros are placed after any sign. This flag is ignored if the '`-`' flag is also specified.

The precision specifies how many digits follow the decimal-point character for the '`%f`', '`%e`', and '`%E`' conversions. For these conversions, the default precision is `6`. If the precision is explicitly `0`, this suppresses the decimal point character entirely. For the '`%g`' and '`%G`' conversions, the precision specifies how many significant digits to print. Significant digits are the first digit before the decimal point, and all the digits after it. If the precision is `0` or not specified for '`%g`' or '`%G`', it is treated like a value of `1`. If the value being printed cannot be expressed precisely in the specified number of digits, the value is rounded to the nearest number that fits.

14.2.10 Other Output Conversions

This section describes miscellaneous conversions for `printf`.

The '%c' conversion prints a single character. The '-' flag can be used to specify left-justification in the field, but no other flags are defined, and no precision or type modifier can be given. For example:

```
printf ("%c%c%c%c%c", "h", "e", "l", "l", "o");
```

prints 'hello'.

The '%s' conversion prints a string. The corresponding argument must be a string. A precision can be specified to indicate the maximum number of characters to write; otherwise characters in the string up to but not including the terminating null character are written to the output stream. The '-' flag can be used to specify left-justification in the field, but no other flags or type modifiers are defined for this conversion. For example:

```
printf ("%3s%-6s", "no", "where");
```

prints ' nowhere ' (note the leading and trailing spaces).

14.2.11 Formatted Input

Octave provides the `scanf`, `fscanf`, and `sscanf` functions to read formatted input. There are two forms of each of these functions. One can be used to extract vectors of data from a file, and the other is more 'C-like'.

[*val*, *count*, *errmsg*] = fscanf (*fid*, *template*, *size*) [Built-in Function]
[*v1*, *v2*, ..., *count*, *errmsg*] = fscanf (*fid*, *template*, "*C*") [Built-in Function]

In the first form, read from *fid* according to *template*, returning the result in the matrix *val*.

The optional argument *size* specifies the amount of data to read and may be one of

`Inf` Read as much as possible, returning a column vector.

nr Read up to *nr* elements, returning a column vector.

[*nr*, Inf] Read as much as possible, returning a matrix with *nr* rows. If the number of elements read is not an exact multiple of *nr*, the last column is padded with zeros.

[*nr*, *nc*] Read up to *nr* * *nc* elements, returning a matrix with *nr* rows. If the number of elements read is not an exact multiple of *nr*, the last column is padded with zeros.

If *size* is omitted, a value of `Inf` is assumed.

A string is returned if *template* specifies only character conversions.

The number of items successfully read is returned in *count*.

If an error occurs, *errmsg* contains a system-dependent error message.

In the second form, read from *fid* according to *template*, with each conversion specifier in *template* corresponding to a single scalar return value. This form is more "C-like", and also compatible with previous versions of Octave. The number of successful conversions is returned in *count*

See the Formatted Input section of the GNU Octave manual for a complete description of the syntax of the template string.

See also: [fgets], page 253, [fgetl], page 252, [fread], page 262, [scanf], page 260, [sscanf], page 260, [fopen], page 250.

[*val*, *count*, *errmsg*] = scanf (*template*, *size*) [Built-in Function]
[*v1*, *v2*, ..., *count*, *errmsg*]] = scanf (*template*, "*C*") [Built-in Function]

This is equivalent to calling `fscanf` with *fid* = `stdin`.

It is currently not useful to call `scanf` in interactive programs.

See also: [fscanf], page 259, [sscanf], page 260, [printf], page 253.

[*val*, *count*, *errmsg*, *pos*] = sscanf (*string*, *template*, *size*) [Built-in Function]
[*v1*, *v2*, ..., *count*, *errmsg*] = sscanf (*string*, *template*, "*C*") [Built-in Function]

This is like `fscanf`, except that the characters are taken from the string *string* instead of from a stream.

Reaching the end of the string is treated as an end-of-file condition. In addition to the values returned by `fscanf`, the index of the next character to be read is returned in *pos*.

See also: [fscanf], page 259, [scanf], page 260, [sprintf], page 254.

Calls to `scanf` are superficially similar to calls to `printf` in that arbitrary arguments are read under the control of a template string. While the syntax of the conversion specifications in the template is very similar to that for `printf`, the interpretation of the template is oriented more towards free-format input and simple pattern matching, rather than fixed-field formatting. For example, most `scanf` conversions skip over any amount of "white space" (including spaces, tabs, and newlines) in the input file, and there is no concept of precision for the numeric input conversions as there is for the corresponding output conversions. Ordinarily, non-whitespace characters in the template are expected to match characters in the input stream exactly.

When a *matching failure* occurs, `scanf` returns immediately, leaving the first non-matching character as the next character to be read from the stream, and `scanf` returns all the items that were successfully converted.

The formatted input functions are not used as frequently as the formatted output functions. Partly, this is because it takes some care to use them properly. Another reason is that it is difficult to recover from a matching error.

14.2.12 Input Conversion Syntax

A `scanf` template string is a string that contains ordinary multibyte characters interspersed with conversion specifications that start with '`%`'.

Any whitespace character in the template causes any number of whitespace characters in the input stream to be read and discarded. The whitespace characters that are matched need not be exactly the same whitespace characters that appear in the template string. For example, write '` , `' in the template to recognize a comma with optional whitespace before and after.

Other characters in the template string that are not part of conversion specifications must match characters in the input stream exactly; if this is not the case, a matching failure occurs.

The conversion specifications in a `scanf` template string have the general form:

`% flags width type conversion`

In more detail, an input conversion specification consists of an initial '`%`' character followed in sequence by:

- An optional *flag character* '`*`', which says to ignore the text read for this specification. When `scanf` finds a conversion specification that uses this flag, it reads input as directed by the rest of the conversion specification, but it discards this input, does not return any value, and does not increment the count of successful assignments.
- An optional decimal integer that specifies the *maximum field width*. Reading of characters from the input stream stops either when this maximum is reached or when a non-matching character is found, whichever happens first. Most conversions discard initial whitespace characters, and these discarded characters don't count towards the maximum field width. Conversions that do not discard initial whitespace are explicitly documented.
- An optional type modifier character. This character is ignored by Octave's `scanf` function, but is recognized to provide compatibility with the C language `scanf`.
- A character that specifies the conversion to be applied.

The exact options that are permitted and how they are interpreted vary between the different conversion specifiers. See the descriptions of the individual conversions for information about the particular options that they allow.

14.2.13 Table of Input Conversions

Here is a table that summarizes the various conversion specifications:

'`%d`' Matches an optionally signed integer written in decimal. See Section 14.2.14 [Numeric Input Conversions], page 262.

'`%i`' Matches an optionally signed integer in any of the formats that the C language defines for specifying an integer constant. See Section 14.2.14 [Numeric Input Conversions], page 262.

'`%o`' Matches an unsigned integer written in octal radix. See Section 14.2.14 [Numeric Input Conversions], page 262.

'`%u`' Matches an unsigned integer written in decimal radix. See Section 14.2.14 [Numeric Input Conversions], page 262.

'`%x`', '`%X`' Matches an unsigned integer written in hexadecimal radix. See Section 14.2.14 [Numeric Input Conversions], page 262.

'`%e`', '`%f`', '`%g`', '`%E`', '`%G`'
Matches an optionally signed floating-point number. See Section 14.2.14 [Numeric Input Conversions], page 262.

'`%s`' Matches a string containing only non-whitespace characters. See Section 14.2.15 [String Input Conversions], page 262.

'`%c`' Matches a string of one or more characters; the number of characters read is controlled by the maximum field width given for the conversion. See Section 14.2.15 [String Input Conversions], page 262.

'%%' This matches a literal '%' character in the input stream. No corresponding argument is used.

If the syntax of a conversion specification is invalid, the behavior is undefined. If there aren't enough function arguments provided to supply addresses for all the conversion specifications in the template strings that perform assignments, or if the arguments are not of the correct types, the behavior is also undefined. On the other hand, extra arguments are simply ignored.

14.2.14 Numeric Input Conversions

This section describes the `scanf` conversions for reading numeric values.

The '%d' conversion matches an optionally signed integer in decimal radix.

The '%i' conversion matches an optionally signed integer in any of the formats that the C language defines for specifying an integer constant.

For example, any of the strings '10', '0xa', or '012' could be read in as integers under the '%i' conversion. Each of these specifies a number with decimal value `10`.

The '%o', '%u', and '%x' conversions match unsigned integers in octal, decimal, and hexadecimal radices, respectively.

The '%X' conversion is identical to the '%x' conversion. They both permit either uppercase or lowercase letters to be used as digits.

Unlike the C language `scanf`, Octave ignores the 'h', 'l', and 'L' modifiers.

14.2.15 String Input Conversions

This section describes the `scanf` input conversions for reading string and character values: '%s' and '%c'.

The '%c' conversion is the simplest: it matches a fixed number of characters, always. The maximum field with says how many characters to read; if you don't specify the maximum, the default is 1. This conversion does not skip over initial whitespace characters. It reads precisely the next *n* characters, and fails if it cannot get that many.

The '%s' conversion matches a string of non-whitespace characters. It skips and discards initial whitespace, but stops when it encounters more whitespace after having read something.

For example, reading the input:

```
hello, world
```

with the conversion '%10c' produces `" hello, wo"`, but reading the same input with the conversion '%10s' produces `"hello,"`.

14.2.16 Binary I/O

Octave can read and write binary data using the functions `fread` and `fwrite`, which are patterned after the standard C functions with the same names. They are able to automatically swap the byte order of integer data and convert among the supported floating point formats as the data are read.

val = fread (*fid*) [Built-in Function]
val = fread (*fid*, *size*) [Built-in Function]

val = fread (*fid*, *size*, *precision*) [Built-in Function]
val = fread (*fid*, *size*, *precision*, *skip*) [Built-in Function]
val = fread (*fid*, *size*, *precision*, *skip*, *arch*) [Built-in Function]
[*val*, *count*] = fread (...) [Built-in Function]

Read binary data from the file specified by the file descriptor *fid*.

The optional argument *size* specifies the amount of data to read and may be one of

`Inf` Read as much as possible, returning a column vector.

nr Read up to *nr* elements, returning a column vector.

[*nr*, Inf] Read as much as possible, returning a matrix with *nr* rows. If the number of elements read is not an exact multiple of *nr*, the last column is padded with zeros.

[*nr*, *nc*] Read up to *nr* * *nc* elements, returning a matrix with *nr* rows. If the number of elements read is not an exact multiple of *nr*, the last column is padded with zeros.

If *size* is omitted, a value of `Inf` is assumed.

The optional argument *precision* is a string specifying the type of data to read and may be one of

"schar"
"signed char"
Signed character.

"uchar"
"unsigned char"
Unsigned character.

"int8"
"integer*1"
8-bit signed integer.

"int16"
"integer*2"
16-bit signed integer.

"int32"
"integer*4"
32-bit signed integer.

"int64"
"integer*8"
64-bit signed integer.

"uint8" 8-bit unsigned integer.

"uint16" 16-bit unsigned integer.

"uint32" 32-bit unsigned integer.

"uint64" 64-bit unsigned integer.

`"single"`
`"float32"`
`"real*4"` 32-bit floating point number.

`"double"`
`"float64"`
`"real*8"` 64-bit floating point number.

`"char"`
`"char*1"` Single character.

`"short"` Short integer (size is platform dependent).

`"int"` Integer (size is platform dependent).

`"long"` Long integer (size is platform dependent).

`"ushort"`
`"unsigned short"`
Unsigned short integer (size is platform dependent).

`"uint"`
`"unsigned int"`
Unsigned integer (size is platform dependent).

`"ulong"`
`"unsigned long"`
Unsigned long integer (size is platform dependent).

`"float"` Single precision floating point number (size is platform dependent).

The default precision is `"uchar"`.

The *precision* argument may also specify an optional repeat count. For example, '`32*single`' causes `fread` to read a block of 32 single precision floating point numbers. Reading in blocks is useful in combination with the *skip* argument.

The *precision* argument may also specify a type conversion. For example, '`int16=>int32`' causes `fread` to read 16-bit integer values and return an array of 32-bit integer values. By default, `fread` returns a double precision array. The special form '`*TYPE`' is shorthand for '`TYPE=>TYPE`'.

The conversion and repeat counts may be combined. For example, the specification '`32*single=>single`' causes `fread` to read blocks of single precision floating point values and return an array of single precision values instead of the default array of double precision values.

The optional argument *skip* specifies the number of bytes to skip after each element (or block of elements) is read. If it is not specified, a value of 0 is assumed. If the final block read is not complete, the final skip is omitted. For example,

```
fread (f, 10, "3*single=>single", 8)
```

will omit the final 8-byte skip because the last read will not be a complete block of 3 values.

The optional argument *arch* is a string specifying the data format for the file. Valid values are

`"native"` or `"n"`
: The format of the current machine.

`"ieee-be"` or `"b"`
: IEEE big endian.

`"ieee-le"` or `"l"`
: IEEE little endian.

If no *arch* is given the value used in the call to `fopen` which created the file descriptor is used. Otherwise, the value specified with `fread` overrides that of `fopen` and determines the data format.

The output argument *val* contains the data read from the file.

The optional return value *count* contains the number of elements read.

See also: [fwrite], page 265, [fgets], page 253, [fgetl], page 252, [fscanf], page 259, [fopen], page 250.

`fwrite (`*fid*, *data*`)` [Built-in Function]
`fwrite (`*fid*, *data*, *precision*`)` [Built-in Function]
`fwrite (`*fid*, *data*, *precision*, *skip*`)` [Built-in Function]
`fwrite (`*fid*, *data*, *precision*, *skip*, *arch*`)` [Built-in Function]
count `= fwrite (...)` [Built-in Function]

Write data in binary form to the file specified by the file descriptor *fid*, returning the number of values *count* successfully written to the file.

The argument *data* is a matrix of values that are to be written to the file. The values are extracted in column-major order.

The remaining arguments *precision*, *skip*, and *arch* are optional, and are interpreted as described for `fread`.

The behavior of `fwrite` is undefined if the values in *data* are too large to fit in the specified precision.

See also: [fread], page 262, [fputs], page 252, [fprintf], page 254, [fopen], page 250.

14.2.17 Temporary Files

Sometimes one needs to write data to a file that is only temporary. This is most commonly used when an external program launched from within Octave needs to access data. When Octave exits all temporary files will be deleted, so this step need not be executed manually.

`[`*fid*, *name*, *msg*`] = mkstemp ("`*template*`")` [Built-in Function]
`[`*fid*, *name*, *msg*`] = mkstemp ("`*template*`", `*delete*`)` [Built-in Function]

Return the file descriptor *fid* corresponding to a new temporary file with a unique name created from *template*.

The last six characters of *template* must be `"XXXXXX"` and these are replaced with a string that makes the filename unique. The file is then created with mode read/write and permissions that are system dependent (on GNU/Linux systems, the permissions will be 0600 for versions of glibc 2.0.7 and later). The file is opened in binary mode and with the `O_EXCL` flag.

If the optional argument *delete* is supplied and is true, the file will be deleted automatically when Octave exits.

If successful, *fid* is a valid file ID, *name* is the name of the file, and *msg* is an empty string. Otherwise, *fid* is -1, *name* is empty, and *msg* contains a system-dependent error message.

See also: [tempname], page 266, [tempdir], page 266, [P_tmpdir], page 266, [tmpfile], page 266, [fopen], page 250.

`[`*fid*`,` *msg*`] = tmpfile ()` [Built-in Function]

Return the file ID corresponding to a new temporary file with a unique name.

The file is opened in binary read/write (`"w+b"`) mode and will be deleted automatically when it is closed or when Octave exits.

If successful, *fid* is a valid file ID and *msg* is an empty string. Otherwise, *fid* is -1 and *msg* contains a system-dependent error message.

See also: [tempname], page 266, [mkstemp], page 265, [tempdir], page 266, [P_tmpdir], page 266.

fname `= tempname ()` [Built-in Function]
fname `= tempname (`*dir*`)` [Built-in Function]
fname `= tempname (`*dir*`,` *prefix*`)` [Built-in Function]

Return a unique temporary file name as a string.

If *prefix* is omitted, a value of `"oct-"` is used.

If *dir* is also omitted, the default directory for temporary files (`P_tmpdir`) is used. If *dir* is provided, it must exist, otherwise the default directory for temporary files is used.

Programming Note: Because the named file is not opened by `tempname`, it is possible, though relatively unlikely, that it will not be available by the time your program attempts to open it. If this is a concern, see `tmpfile`.

See also: [mkstemp], page 265, [tempdir], page 266, [P_tmpdir], page 266, [tmpfile], page 266.

dir `= tempdir ()` [Function File]

Return the name of the host system's directory for temporary files.

The directory name is taken first from the environment variable `TMPDIR`. If that does not exist the system default returned by `P_tmpdir` is used.

See also: [P_tmpdir], page 266, [tempname], page 266, [mkstemp], page 265, [tmpfile], page 266.

`P_tmpdir ()` [Built-in Function]

Return the name of the host system's **default** directory for temporary files.

Programming Note: The value returned by `P_tmpdir` is always the default location. This value may not agree with that returned from `tempdir` if the user has overridden the default with the `TMPDIR` environment variable.

See also: [tempdir], page 266, [tempname], page 266, [mkstemp], page 265, [tmpfile], page 266.

14.2.18 End of File and Errors

Once a file has been opened its status can be acquired. As an example the `feof` functions determines if the end of the file has been reached. This can be very useful when reading small parts of a file at a time. The following example shows how to read one line at a time from a file until the end has been reached.

```
filename = "myfile.txt";
fid = fopen (filename, "r");
while (! feof (fid) )
  text_line = fgetl (fid);
endwhile
fclose (fid);
```

Note that in some situations it is more efficient to read the entire contents of a file and then process it, than it is to read it line by line. This has the potential advantage of removing the loop in the above code.

`status = feof (fid)` [Built-in Function]

Return 1 if an end-of-file condition has been encountered for the file specified by file descriptor *fid* and 0 otherwise.

Note that `feof` will only return 1 if the end of the file has already been encountered, not if the next read operation will result in an end-of-file condition.

See also: [fread], page 262, [frewind], page 269, [fseek], page 268, [fclear], page 267, [fopen], page 250.

`msg = ferror (fid)` [Built-in Function]
`[msg, err] = ferror (fid)` [Built-in Function]
`[dots] = ferror (fid, "clear")` [Built-in Function]

Query the error status of the stream specified by file descriptor *fid*

If an error condition exists then return a string *msg* describing the error. Otherwise, return an empty string `""`.

The second input `"clear"` is optional. If supplied, the error state on the stream will be cleared.

The optional second output is a numeric indication of the error status. *err* is 1 if an error condition has been encountered and 0 otherwise.

Note that `ferror` indicates if an error has already occurred, not whether the next operation will result in an error condition.

See also: [fclear], page 267, [fopen], page 250.

`fclear (fid)` [Built-in Function]

Clear the stream state for the file specified by the file descriptor *fid*.

See also: [ferror], page 267, [fopen], page 250.

`freport ()` [Built-in Function]

Print a list of which files have been opened, and whether they are open for reading, writing, or both.

For example:

```
freport ()

     ⊣  number  mode  arch       name
     ⊣  ------  ----  ----       ----
     ⊣     0     r    ieee-le    stdin
     ⊣     1     w    ieee-le    stdout
     ⊣     2     w    ieee-le    stderr
     ⊣     3     r    ieee-le    myfile
```

See also: [fopen], page 250, [fclose], page 251, [is_valid_file_id], page 251.

14.2.19 File Positioning

Three functions are available for setting and determining the position of the file pointer for a given file.

pos = `ftell` (*fid*) [Built-in Function]

Return the position of the file pointer as the number of characters from the beginning of the file specified by file descriptor *fid*.

See also: [fseek], page 268, [frewind], page 269, [feof], page 267, [fopen], page 250.

`fseek` (*fid*, *offset*) [Built-in Function]
`fseek` (*fid*, *offset*, *origin*) [Built-in Function]
status = `fseek` (*...*) [Built-in Function]

Set the file pointer to the location *offset* within the file *fid*.

The pointer is positioned *offset* characters from the *origin*, which may be one of the predefined variables `SEEK_CUR` (current position), `SEEK_SET` (beginning), or `SEEK_END` (end of file) or strings `"cof"`, `"bof"` or `"eof"`. If *origin* is omitted, `SEEK_SET` is assumed. *offset* may be positive, negative, or zero but not all combinations of *origin* and *offset* can be realized.

`fseek` returns 0 on success and -1 on error.

See also: [fskipl], page 253, [frewind], page 269, [ftell], page 268, [fopen], page 250.

`SEEK_SET ()` [Built-in Function]
`SEEK_CUR ()` [Built-in Function]
`SEEK_END ()` [Built-in Function]

Return the numerical value to pass to `fseek` to perform one of the following actions:

`SEEK_SET` Position file relative to the beginning.

`SEEK_CUR` Position file relative to the current position.

`SEEK_END` Position file relative to the end.

See also: [fseek], page 268.

`frewind` (*fid*) [Built-in Function]
status = `frewind` (*fid*) [Built-in Function]

Move the file pointer to the beginning of the file specified by file descriptor *fid*.

`frewind` returns 0 for success, and -1 if an error is encountered. It is equivalent to `fseek (`*fid*`, 0, SEEK_SET)`.

See also: [fseek], page 268, [ftell], page 268, [fopen], page 250.

The following example stores the current file position in the variable `marker`, moves the pointer to the beginning of the file, reads four characters, and then returns to the original position.

```
marker = ftell (myfile);
frewind (myfile);
fourch = fgets (myfile, 4);
fseek (myfile, marker, SEEK_SET);
```

15 Plotting

15.1 Introduction to Plotting

Earlier versions of Octave provided plotting through the use of gnuplot. This capability is still available. But, a newer plotting capability is provided by access to OpenGL. Which plotting system is used is controlled by the `graphics_toolkit` function. See Section 15.4.7 [Graphics Toolkits], page 402.

The function call `graphics_toolkit ("fltk")` selects the FLTK/OpenGL system, and `graphics_toolkit ("gnuplot")` selects the gnuplot system. The two systems may be used selectively through the use of the `graphics_toolkit` property of the graphics handle for each figure. This is explained in Section 15.3 [Graphics Data Structures], page 354. **Caution:** The FLTK toolkit uses single precision variables internally which limits the maximum value that can be displayed to approximately 10^{38}. If your data contains larger values you must use the gnuplot toolkit which supports values up to 10^{308}.

15.2 High-Level Plotting

Octave provides simple means to create many different types of two- and three-dimensional plots using high-level functions.

If you need more detailed control, see Section 15.3 [Graphics Data Structures], page 354, and Section 15.4 [Advanced Plotting], page 387.

15.2.1 Two-Dimensional Plots

The `plot` function allows you to create simple x-y plots with linear axes. For example,

```
x = -10:0.1:10;
plot (x, sin (x));
```

displays a sine wave shown in Figure 15.1. On most systems, this command will open a separate plot window to display the graph.

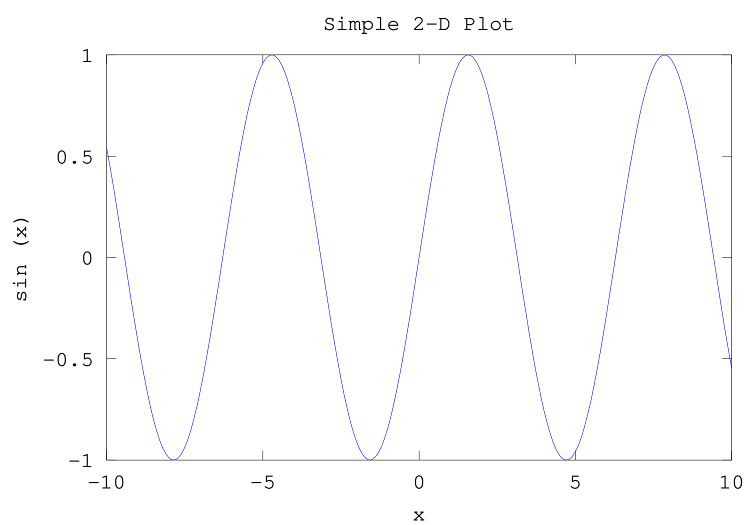

Figure 15.1: Simple Two-Dimensional Plot.

`plot (`*y*`)` [Function File]
`plot (`*x*`, `*y*`)` [Function File]
`plot (`*x*`, `*y*`, `*fmt*`)` [Function File]
`plot (..., `*property*`, `*value*`, ...)` [Function File]
`plot (`*x1*`, `*y1*`, ..., `*xn*`, `*yn*`)` [Function File]
`plot (`*hax*`, ...)` [Function File]
h `= plot (...)` [Function File]

Produce 2-D plots.

Many different combinations of arguments are possible. The simplest form is

```
plot (y)
```

where the argument is taken as the set of *y* coordinates and the *x* coordinates are taken to be the range `1:numel (y)`.

If more than one argument is given, they are interpreted as

```
plot (y, property, value, ...)
```

or

```
plot (x, y, property, value, ...)
```

or

```
plot (x, y, fmt, ...)
```

and so on. Any number of argument sets may appear. The *x* and *y* values are interpreted as follows:

- If a single data argument is supplied, it is taken as the set of *y* coordinates and the *x* coordinates are taken to be the indices of the elements, starting with 1.
- If *x* and *y* are scalars, a single point is plotted.

- `squeeze()` is applied to arguments with more than two dimensions, but no more than two singleton dimensions.
- If both arguments are vectors, the elements of *y* are plotted versus the elements of *x*.
- If *x* is a vector and *y* is a matrix, then the columns (or rows) of *y* are plotted versus *x*. (using whichever combination matches, with columns tried first.)
- If the *x* is a matrix and *y* is a vector, *y* is plotted versus the columns (or rows) of *x*. (using whichever combination matches, with columns tried first.)
- If both arguments are matrices, the columns of *y* are plotted versus the columns of *x*. In this case, both matrices must have the same number of rows and columns and no attempt is made to transpose the arguments to make the number of rows match.

Multiple property-value pairs may be specified, but they must appear in pairs. These arguments are applied to the line objects drawn by `plot`. Useful properties to modify are `"linestyle"`, `"linewidth"`, `"color"`, `"marker"`, `"markersize"`, `"markeredgecolor"`, `"markerfacecolor"`. See Section 15.3.3.4 [Line Properties], page 374.

The *fmt* format argument can also be used to control the plot style. It is a string composed of four optional parts: "<linestyle><marker><color><;displayname;>". When a marker is specified, but no linestyle, only the markers are plotted. Similarly, if a linestyle is specified, but no marker, then only lines are drawn. If both are specified then lines and markers will be plotted. If no *fmt* and no *property*/*value* pairs are given, then the default plot style is solid lines with no markers and the color determined by the `"colororder"` property of the current axes.

Format arguments:

linestyle

'`-`'	Use solid lines (default).
'`--`'	Use dashed lines.
'`:`'	Use dotted lines.
'`-.`'	Use dash-dotted lines.

marker

'`+`'	crosshair
'`o`'	circle
'`*`'	star
'`.`'	point
'`x`'	cross
'`s`'	square
'`d`'	diamond
'`^`'	upward-facing triangle
'`v`'	downward-facing triangle
'`>`'	right-facing triangle
'`<`'	left-facing triangle
'`p`'	pentagram
'`h`'	hexagram

color

'k'	blacK
'r'	Red
'g'	Green
'b'	Blue
'm'	Magenta
'c'	Cyan
'w'	White

";displayname;"
: Here "displayname" is the label to use for the plot legend.

The *fmt* argument may also be used to assign legend labels. To do so, include the desired label between semicolons after the formatting sequence described above, e.g., "+b;Key Title;". Note that the last semicolon is required and Octave will generate an error if it is left out.

Here are some plot examples:

```
plot (x, y, "or", x, y2, x, y3, "m", x, y4, "+")
```

This command will plot y with red circles, y2 with solid lines, y3 with solid magenta lines, and y4 with points displayed as '+'.

```
plot (b, "*", "markersize", 10)
```

This command will plot the data in the variable b, with points displayed as '*' and a marker size of 10.

```
t = 0:0.1:6.3;
plot (t, cos(t), "-;cos(t);", t, sin(t), "-b;sin(t);");
```

This will plot the cosine and sine functions and label them accordingly in the legend.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by gca.

The optional return value *h* is a vector of graphics handles to the created line objects.

To save a plot, in one of several image formats such as PostScript or PNG, use the print command.

See also: [axis], page 299, [box], page 331, [grid], page 331, [hold], page 339, [legend], page 328, [title], page 328, [xlabel], page 330, [ylabel], page 330, [xlim], page 301, [ylim], page 301, [ezplot], page 302, [errorbar], page 288, [fplot], page 301, [line], page 357, [plot3], page 318, [polar], page 292, [loglog], page 276, [semilogx], page 275, [semilogy], page 275, [subplot], page 335.

The plotyy function may be used to create a plot with two independent y axes.

plotyy (*x1*, *y1*, *x2*, *y2*) [Function File]
plotyy (..., *fun*) [Function File]
plotyy (..., *fun1*, *fun2*) [Function File]
plotyy (*hax*, ...) [Function File]
[*ax*, *h1*, *h2*] = plotyy (...) [Function File]

Plot two sets of data with independent y-axes and a common x-axis.

The arguments *x1* and *y1* define the arguments for the first plot and *x1* and *y2* for the second.

By default the arguments are evaluated with `feval (@plot, x, y)`. However the type of plot can be modified with the *fun* argument, in which case the plots are generated by `feval (fun, x, y)`. *fun* can be a function handle, an inline function, or a string of a function name.

The function to use for each of the plots can be independently defined with *fun1* and *fun2*.

If the first argument *hax* is an axes handle, then it defines the principal axis in which to plot the *x1* and *y1* data.

The return value *ax* is a vector with the axis handles of the two y-axes. *h1* and *h2* are handles to the objects generated by the plot commands.

```
x = 0:0.1:2*pi;
y1 = sin (x);
y2 = exp (x - 1);
ax = plotyy (x, y1, x - 1, y2, @plot, @semilogy);
xlabel ("X");
ylabel (ax(1), "Axis 1");
ylabel (ax(2), "Axis 2");
```

See also: [plot], page 272.

The functions `semilogx`, `semilogy`, and `loglog` are similar to the `plot` function, but produce plots in which one or both of the axes use log scales.

`semilogx` (*y*) [Function File]
`semilogx` (*x*, *y*) [Function File]
`semilogx` (*x*, *y*, *property*, *value*, ...) [Function File]
`semilogx` (*x*, *y*, *fmt*) [Function File]
`semilogx` (*hax*, ...) [Function File]
h = `semilogx` (...) [Function File]

Produce a 2-D plot using a logarithmic scale for the x-axis.

See the documentation of `plot` for a description of the arguments that `semilogx` will accept.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

See also: [plot], page 272, [semilogy], page 275, [loglog], page 276.

`semilogy` (*y*) [Function File]
`semilogy` (*x*, *y*) [Function File]
`semilogy` (*x*, *y*, *property*, *value*, ...) [Function File]
`semilogy` (*x*, *y*, *fmt*) [Function File]
`semilogy` (*h*, ...) [Function File]
h = `semilogy` (...) [Function File]

Produce a 2-D plot using a logarithmic scale for the y-axis.

See the documentation of `plot` for a description of the arguments that `semilogy` will accept.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

See also: [plot], page 272, [semilogx], page 275, [loglog], page 276.

`loglog (`*y*`)` [Function File]
`loglog (`*x*, *y*`)` [Function File]
`loglog (`*x*, *y*, *prop*, *value*, ...`)` [Function File]
`loglog (`*x*, *y*, *fmt*`)` [Function File]
`loglog (`*hax*, ...`)` [Function File]
h `= loglog (...)` [Function File]

Produce a 2-D plot using logarithmic scales for both axes.

See the documentation of `plot` for a description of the arguments that `loglog` will accept.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

See also: [plot], page 272, [semilogx], page 275, [semilogy], page 275.

The functions `bar`, `barh`, `stairs`, and `stem` are useful for displaying discrete data. For example,

```
hist (randn (10000, 1), 30);
```

produces the histogram of 10,000 normally distributed random numbers shown in Figure 15.2.

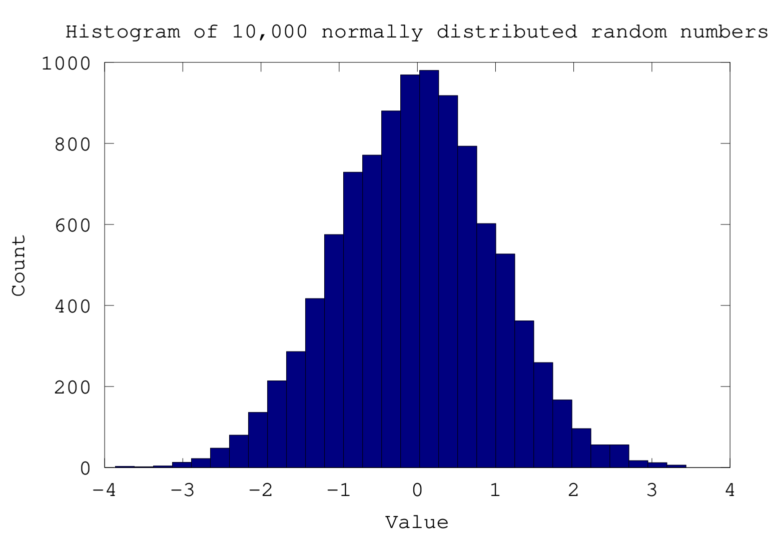

Figure 15.2: Histogram.

`bar` (*y*) [Function File]
`bar` (*x*, *y*) [Function File]
`bar` (. . . , *w*) [Function File]
`bar` (. . . , *style*) [Function File]
`bar` (. . . , *prop*, *val*, . . .) [Function File]
`bar` (*hax*, . . .) [Function File]
h = `bar` (. . . , *prop*, *val*, . . .) [Function File]

Produce a bar graph from two vectors of X-Y data.

If only one argument is given, *y*, it is taken as a vector of Y values and the X coordinates are the range `1:numel (y)`.

The optional input *w* controls the width of the bars. A value of 1.0 will cause each bar to exactly touch any adjacent bars. The default width is 0.8.

If *y* is a matrix, then each column of *y* is taken to be a separate bar graph plotted on the same graph. By default the columns are plotted side-by-side. This behavior can be changed by the *style* argument which can take the following values:

`"grouped"` (default)
: Side-by-side bars with a gap between bars and centered over the X-coordinate.

`"stacked"`
: Bars are stacked so that each X value has a single bar composed of multiple segments.

`"hist"`
: Side-by-side bars with no gap between bars and centered over the X-coordinate.

`"histc"`
: Side-by-side bars with no gap between bars and left-aligned to the X-coordinate.

Optional property/value pairs are passed directly to the underlying patch objects.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of handles to the created `"bar series"` hggroups with one handle per column of the variable *y*. This series makes it possible to change a common element in one bar series object and have the change reflected in the other `"bar series"`. For example,

```
h = bar (rand (5, 10));
set (h(1), "basevalue", 0.5);
```

changes the position on the base of all of the bar series.

The following example modifies the face and edge colors using property/value pairs.

```
bar (randn (1, 100), "facecolor", "r", "edgecolor", "b");
```

The color of the bars is taken from the figure's colormap, such that

```
bar (rand (10, 3));
colormap (summer (64));
```

will change the colors used for the bars. The color of bars can also be set manually using the "facecolor" property as shown below.

```
h = bar (rand (10, 3));
set (h(1), "facecolor", "r")
set (h(2), "facecolor", "g")
set (h(3), "facecolor", "b")
```

See also: [barh], page 278, [hist], page 279, [pie], page 293, [plot], page 272, [patch], page 357.

barh (*y*) [Function File]
barh (*x*, *y*) [Function File]
barh (..., *w*) [Function File]
barh (..., *style*) [Function File]
barh (..., *prop*, *val*, ...) [Function File]
barh (*hax*, ...) [Function File]
h = barh (..., *prop*, *val*, ...) [Function File]

Produce a horizontal bar graph from two vectors of X-Y data.

If only one argument is given, it is taken as a vector of Y values and the X coordinates are the range `1:numel (y)`.

The optional input *w* controls the width of the bars. A value of 1.0 will cause each bar to exactly touch any adjacent bars. The default width is 0.8.

If *y* is a matrix, then each column of *y* is taken to be a separate bar graph plotted on the same graph. By default the columns are plotted side-by-side. This behavior can be changed by the *style* argument which can take the following values:

"grouped" (default)
: Side-by-side bars with a gap between bars and centered over the Y-coordinate.

"stacked"
: Bars are stacked so that each Y value has a single bar composed of multiple segments.

"hist"
: Side-by-side bars with no gap between bars and centered over the Y-coordinate.

"histc"
: Side-by-side bars with no gap between bars and left-aligned to the Y-coordinate.

Optional property/value pairs are passed directly to the underlying patch objects.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created bar series hggroup. For a description of the use of the bar series, see [bar], page 276.

See also: [bar], page 276, [hist], page 279, [pie], page 293, [plot], page 272, [patch], page 357.

`hist (y)` [Function File]
`hist (y, x)` [Function File]
`hist (y, nbins)` [Function File]
`hist (y, x, norm)` [Function File]
`hist (..., prop, val, ...)` [Function File]
`hist (hax, ...)` [Function File]
`[nn, xx] = hist (...)` [Function File]

Produce histogram counts or plots.

With one vector input argument, *y*, plot a histogram of the values with 10 bins. The range of the histogram bins is determined by the range of the data. With one matrix input argument, *y*, plot a histogram where each bin contains a bar per input column.

Given a second vector argument, *x*, use that as the centers of the bins, with the width of the bins determined from the adjacent values in the vector.

If scalar, the second argument, *nbins*, defines the number of bins.

If a third argument is provided, the histogram is normalized such that the sum of the bars is equal to *norm*.

Extreme values are lumped into the first and last bins.

The histogram's appearance may be modified by specifying property/value pairs. For example the face and edge color may be modified.

```
hist (randn (1, 100), 25, "facecolor", "r", "edgecolor", "b");
```

The histogram's colors also depend upon the current colormap.

```
hist (rand (10, 3));
colormap (summer ());
```

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

With two output arguments, produce the values *nn* (numbers of elements) and *xx* (bin centers) such that `bar (xx, nn)` will plot the histogram.

See also: [histc], page 606, [bar], page 276, [pie], page 293, [rose], page 285.

`stemleaf (x, caption)` [Function File]
`stemleaf (x, caption, stem_sz)` [Function File]
`plotstr = stemleaf (...)` [Function File]

Compute and display a stem and leaf plot of the vector *x*.

The input *x* should be a vector of integers. Any non-integer values will be converted to integer by `x = fix (x)`. By default each element of *x* will be plotted with the last digit of the element as a leaf value and the remaining digits as the stem. For example, 123 will be plotted with the stem '12' and the leaf '3'. The second argument, *caption*, should be a character array which provides a description of the data. It is included as a heading for the output.

The optional input *stem_sz* sets the width of each stem. The stem width is determined by `10^(stem_sz + 1)`. The default stem width is 10.

The output of `stemleaf` is composed of two parts: a "Fenced Letter Display," followed by the stem-and-leaf plot itself. The Fenced Letter Display is described in *Exploratory Data Analysis*. Briefly, the entries are as shown:

```
          Fenced Letter Display
#% nx|___________________     nx = numel (x)
M% mi|       md         |     mi median index, md median
H% hi|hl              hu| hs  hi lower hinge index, hl,hu hinges,
1    |x(1)         x(nx)|     hs h_spreadx(1), x(nx) first
            _______           and last data value.
      ______|step |_______    step 1.5*h_spread
     f|ifl            ifh|    inner fence, lower and higher
      |nfl            nfh|    no.\ of data points within fences
     F|ofl            ofh|    outer fence, lower and higher
      |nFl            nFh|    no.\ of data points outside outer
                              fences
```

The stem-and-leaf plot shows on each line the stem value followed by the string made up of the leaf digits. If the *stem_sz* is not 1 the successive leaf values are separated by ",".

With no return argument, the plot is immediately displayed. If an output argument is provided, the plot is returned as an array of strings.

The leaf digits are not sorted. If sorted leaf values are desired, use `xs = sort (x)` before calling `stemleaf (xs)`.

The stem and leaf plot and associated displays are described in: Ch. 3, *Exploratory Data Analysis* by J. W. Tukey, Addison-Wesley, 1977.

See also: [hist], page 279, [printd], page 280.

`printd (obj, filename)` [Function File]
`out_file = printd (...)` [Function File]

Convert any object acceptable to `disp` into the format selected by the suffix of *filename*.

If the return argument *out_file* is given, the name of the created file is returned.

This function is intended to facilitate manipulation of the output of functions such as `stemleaf`.

See also: [stemleaf], page 279.

`stairs (y)` [Function File]
`stairs (x, y)` [Function File]
`stairs (..., style)` [Function File]
`stairs (..., prop, val, ...)` [Function File]
`stairs (hax, ...)` [Function File]
`h = stairs (...)` [Function File]
`[xstep, ystep] = stairs (...)` [Function File]

Produce a stairstep plot.

The arguments *x* and *y* may be vectors or matrices. If only one argument is given, it is taken as a vector of Y values and the X coordinates are taken to be the indices of the elements.

The style to use for the plot can be defined with a line style *style* of the same format as the `plot` command.

Multiple property/value pairs may be specified, but they must appear in pairs.

If the first argument *hax* is an axis handle, then plot into this axis, rather than the current axis handle returned by `gca`.

If one output argument is requested, return a graphics handle to the created plot. If two output arguments are specified, the data are generated but not plotted. For example,

```
stairs (x, y);
```

and

```
[xs, ys] = stairs (x, y);
plot (xs, ys);
```

are equivalent.

See also: [bar], page 276, [hist], page 279, [plot], page 272, [stem], page 281.

`stem` (*y*) [Function File]
`stem` (*x*, *y*) [Function File]
`stem` (. . ., *linespec*) [Function File]
`stem` (. . ., "*filled*") [Function File]
`stem` (. . ., *prop*, *val*, . . .) [Function File]
`stem` (*hax*, . . .) [Function File]
h = `stem` (. . .) [Function File]

Plot a 2-D stem graph.

If only one argument is given, it is taken as the y-values and the x-coordinates are taken from the indices of the elements.

If *y* is a matrix, then each column of the matrix is plotted as a separate stem graph. In this case *x* can either be a vector, the same length as the number of rows in *y*, or it can be a matrix of the same size as *y*.

The default color is `"b"` (blue), the default line style is `"-"`, and the default marker is `"o"`. The line style can be altered by the `linespec` argument in the same manner as the `plot` command. If the `"filled"` argument is present the markers at the top of the stems will be filled in. For example,

```
x = 1:10;
y = 2*x;
stem (x, y, "r");
```

plots 10 stems with heights from 2 to 20 in red;

Optional property/value pairs may be specified to control the appearance of the plot.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a handle to a "stem series" hggroup. The single hggroup handle has all of the graphical elements comprising the plot as its children; This allows the properties of multiple graphics objects to be changed by modifying just a single property of the "stem series" hggroup.

For example,

```
x = [0:10]';
y = [sin(x), cos(x)]
h = stem (x, y);
set (h(2), "color", "g");
set (h(1), "basevalue", -1)
```

changes the color of the second "stem series" and moves the base line of the first.

Stem Series Properties

linestyle
: The linestyle of the stem. (Default: `"-"`)

linewidth
: The width of the stem. (Default: 0.5)

color
: The color of the stem, and if not separately specified, the marker. (Default: `"b"` [blue])

marker
: The marker symbol to use at the top of each stem. (Default: `"o"`)

markeredgecolor
: The edge color of the marker. (Default: `"color"` property)

markerfacecolor
: The color to use for "filling" the marker. (Default: `"none"` [unfilled])

markersize
: The size of the marker. (Default: 6)

baseline
: The handle of the line object which implements the baseline. Use `set` with the returned handle to change graphic properties of the baseline.

basevalue
: The y-value where the baseline is drawn. (Default: 0)

See also: [stem3], page 282, [bar], page 276, [hist], page 279, [plot], page 272, [stairs], page 280.

`stem3` (*x*, *y*, *z*) [Function File]
`stem3` (. . ., *linespec*) [Function File]
`stem3` (. . ., "*filled*") [Function File]
`stem3` (. . ., *prop*, *val*, . . .) [Function File]
`stem3` (*hax*, . . .) [Function File]
h = `stem3` (. . .) [Function File]

Plot a 3-D stem graph.

Stems are drawn from the height *z* to the location in the x-y plane determined by *x* and *y*. The default color is `"b"` (blue), the default line style is `"-"`, and the default marker is `"o"`.

The line style can be altered by the `linespec` argument in the same manner as the `plot` command. If the `"filled"` argument is present the markers at the top of the stems will be filled in.

Optional property/value pairs may be specified to control the appearance of the plot.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a handle to the "stem series" hggroup containing the line and marker objects used for the plot. See [stem], page 281, for a description of the "stem series" object.

Example:

```
theta = 0:0.2:6;
stem3 (cos (theta), sin (theta), theta);
```

plots 31 stems with heights from 0 to 6 lying on a circle.

Implementation Note: Color definitions with RGB-triples are not valid.

See also: [stem], page 281, [bar], page 276, [hist], page 279, [plot], page 272.

`scatter` (*x*, *y*) [Function File]
`scatter` (*x*, *y*, *s*) [Function File]
`scatter` (*x*, *y*, *s*, *c*) [Function File]
`scatter` (. . ., *style*) [Function File]
`scatter` (. . ., "*filled*") [Function File]
`scatter` (. . ., *prop*, *val*, . . .) [Function File]
`scatter` (*hax*, . . .) [Function File]
h = `scatter` (. . .) [Function File]

Draw a 2-D scatter plot.

A marker is plotted at each point defined by the coordinates in the vectors *x* and *y*.

The size of the markers is determined by *s*, which can be a scalar or a vector of the same length as *x* and *y*. If *s* is not given, or is an empty matrix, then a default value of 8 points is used.

The color of the markers is determined by *c*, which can be a string defining a fixed color; a 3-element vector giving the red, green, and blue components of the color; a vector of the same length as *x* that gives a scaled index into the current colormap; or an Nx3 matrix defining the RGB color of each marker individually.

The marker to use can be changed with the *style* argument, that is a string defining a marker in the same manner as the `plot` command. If no marker is specified it defaults to `"o"` or circles. If the argument `"filled"` is given then the markers are filled.

Additional property/value pairs are passed directly to the underlying patch object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created patch object.

Example:

```
x = randn (100, 1);
y = randn (100, 1);
scatter (x, y, [], sqrt (x.^2 + y.^2));
```

See also: [scatter3], page 321, [patch], page 357, [plot], page 272.

`plotmatrix` (*x*, *y*) [Function File]
`plotmatrix` (*x*) [Function File]
`plotmatrix` (. . ., *style*) [Function File]
`plotmatrix` (*hax*, . . .) [Function File]

`[h, ax, bigax, p, pax] = plotmatrix (...)` [Function File]

Scatter plot of the columns of one matrix against another.

Given the arguments *x* and *y* that have a matching number of rows, `plotmatrix` plots a set of axes corresponding to

```
plot (x(:, i), y(:, j))
```

When called with a single argument *x* this is equivalent to

```
plotmatrix (x, x)
```

except that the diagonal of the set of axes will be replaced with the histogram `hist (x(:, i))`.

The marker to use can be changed with the *style* argument, that is a string defining a marker in the same manner as the `plot` command.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* provides handles to the individual graphics objects in the scatter plots, whereas *ax* returns the handles to the scatter plot axis objects.

bigax is a hidden axis object that surrounds the other axes, such that the commands `xlabel`, `title`, etc., will be associated with this hidden axis.

Finally, *p* returns the graphics objects associated with the histogram and *pax* the corresponding axes objects.

Example:

```
plotmatrix (randn (100, 3), "g+")
```

See also: [scatter], page 283, [plot], page 272.

`pareto (y)` [Function File]
`pareto (y, x)` [Function File]
`pareto (hax, ...)` [Function File]
`h = pareto (...)` [Function File]

Draw a Pareto chart.

A Pareto chart is a bar graph that arranges information in such a way that priorities for process improvement can be established; It organizes and displays information to show the relative importance of data. The chart is similar to the histogram or bar chart, except that the bars are arranged in decreasing magnitude from left to right along the x-axis.

The fundamental idea (Pareto principle) behind the use of Pareto diagrams is that the majority of an effect is due to a small subset of the causes. For quality improvement, the first few contributing causes (leftmost bars as presented on the diagram) to a problem usually account for the majority of the result. Thus, targeting these "major causes" for elimination results in the most cost-effective improvement scheme.

Typically only the magnitude data *y* is present in which case *x* is taken to be the range `1 : length (y)`. If *x* is given it may be a string array, a cell array of strings, or a numerical vector.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a 2-element vector with a graphics handle for the created bar plot and a second handle for the created line plot.

An example of the use of `pareto` is

```
Cheese = {"Cheddar", "Swiss", "Camembert", ...
          "Munster", "Stilton", "Blue"};
Sold = [105, 30, 70, 10, 15, 20];
pareto (Sold, Cheese);
```

See also: [bar], page 276, [barh], page 278, [hist], page 279, [pie], page 293, [plot], page 272.

`rose (`*th*`)` [Function File]
`rose (`*th*`, `*nbins*`)` [Function File]
`rose (`*th*`, `*bins*`)` [Function File]
`rose (`*hax*`, ...)` [Function File]
h `= rose (...)` [Function File]
`[`*thout rout*`] = rose (...)` [Function File]

Plot an angular histogram.

With one vector argument, *th*, plot the histogram with 20 angular bins. If *th* is a matrix then each column of *th* produces a separate histogram.

If *nbins* is given and is a scalar, then the histogram is produced with *nbin* bins. If *bins* is a vector, then the center of each bin is defined by the values of *bins* and the number of bins is given by the number of elements in *bins*.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of graphics handles to the line objects representing each histogram.

If two output arguments are requested then no plot is made and the polar vectors necessary to plot the histogram are returned instead.

```
[th, r] = rose ([2*randn(1e5,1), pi + 2*randn(1e5,1)]);
polar (th, r);
```

See also: [hist], page 279, [polar], page 292.

The `contour`, `contourf` and `contourc` functions produce two-dimensional contour plots from three-dimensional data.

`contour (`*z*`)` [Function File]
`contour (`*z*`, `*vn*`)` [Function File]
`contour (`*x*`, `*y*`, `*z*`)` [Function File]
`contour (`*x*`, `*y*`, `*z*`, `*vn*`)` [Function File]
`contour (..., `*style*`)` [Function File]
`contour (`*hax*`, ...)` [Function File]
`[`*c*`, `*h*`] = contour (...)` [Function File]

Create a 2-D contour plot.

Plot level curves (contour lines) of the matrix *z*, using the contour matrix *c* computed by `contourc` from the same arguments; see the latter for their interpretation.

The appearance of contour lines can be defined with a line style *style* in the same manner as `plot`. Only line style and color are used; Any markers defined by *style* are ignored.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional output *c* contains the contour levels in `contourc` format.

The optional return value *h* is a graphics handle to the hggroup comprising the contour lines.

Example:

```
x = 0:2;
y = x;
z = x' * y;
contour (x, y, z, 2:3)
```

See also: [ezcontour], page 303, [contourc], page 287, [contourf], page 286, [contour3], page 287, [clabel], page 330, [meshc], page 308, [surfc], page 310, [caxis], page 300, [colormap], page 704, [plot], page 272.

`contourf` (*z*) [Function File]
`contourf` (*z*, *vn*) [Function File]
`contourf` (*x*, *y*, *z*) [Function File]
`contourf` (*x*, *y*, *z*, *vn*) [Function File]
`contourf` (..., *style*) [Function File]
`contourf` (*hax*, ...) [Function File]
[*c*, *h*] = `contourf` (...) [Function File]

Create a 2-D contour plot with filled intervals.

Plot level curves (contour lines) of the matrix *z* and fill the region between lines with colors from the current colormap.

The level curves are taken from the contour matrix *c* computed by `contourc` for the same arguments; see the latter for their interpretation.

The appearance of contour lines can be defined with a line style *style* in the same manner as `plot`. Only line style and color are used; Any markers defined by *style* are ignored.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional output *c* contains the contour levels in `contourc` format.

The optional return value *h* is a graphics handle to the hggroup comprising the contour lines.

The following example plots filled contours of the `peaks` function.

```
[x, y, z] = peaks (50);
contourf (x, y, z, -7:9)
```

See also: [ezcontourf], page 303, [contour], page 285, [contourc], page 287, [contour3], page 287, [clabel], page 330, [meshc], page 308, [surfc], page 310, [caxis], page 300, [colormap], page 704, [plot], page 272.

`[c, lev] = contourc (z)` [Function File]
`[c, lev] = contourc (z, vn)` [Function File]
`[c, lev] = contourc (x, y, z)` [Function File]
`[c, lev] = contourc (x, y, z, vn)` [Function File]

Compute contour lines (isolines of constant Z value).

The matrix *z* contains height values above the rectangular grid determined by *x* and *y*. If only a single input *z* is provided then *x* is taken to be `1:rows (z)` and *y* is taken to be `1:columns (z)`.

The optional input *vn* is either a scalar denoting the number of contour lines to compute or a vector containing the Z values where lines will be computed. When *vn* is a vector the number of contour lines is `numel (vn)`. However, to compute a single contour line at a given value use `vn = [val, val]`. If *vn* is omitted it defaults to 10.

The return value *c* is a 2x*n* matrix containing the contour lines in the following format

```
c = [lev1, x1, x2, ..., levn, x1, x2, ...
     len1, y1, y2, ..., lenn, y1, y2, ...]
```

in which contour line *n* has a level (height) of *levn* and length of *lenn*.

The optional return value *lev* is a vector with the Z values of the contour levels.

Example:

```
x = 0:2;
y = x;
z = x' * y;
contourc (x, y, z, 2:3)
   ⇒   2.0000   2.0000   1.0000   3.0000   1.5000   2.0000
       2.0000   1.0000   2.0000   2.0000   2.0000   1.5000
```

See also: [contour], page 285, [contourf], page 286, [contour3], page 287, [clabel], page 330.

`contour3 (z)` [Function File]
`contour3 (z, vn)` [Function File]
`contour3 (x, y, z)` [Function File]
`contour3 (x, y, z, vn)` [Function File]
`contour3 (..., style)` [Function File]
`contour3 (hax, ...)` [Function File]
`[c, h] = contour3 (...)` [Function File]

Create a 3-D contour plot.

`contour3` plots level curves (contour lines) of the matrix *z* at a Z level corresponding to each contour. This is in contrast to `contour` which plots all of the contour lines at the same Z level and produces a 2-D plot.

The level curves are taken from the contour matrix *c* computed by `contourc` for the same arguments; see the latter for their interpretation.

The appearance of contour lines can be defined with a line style *style* in the same manner as `plot`. Only line style and color are used; Any markers defined by *style* are ignored.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional output *c* are the contour levels in `contourc` format.

The optional return value *h* is a graphics handle to the hggroup comprising the contour lines.

Example:

```
contour3 (peaks (19));
colormap cool;
hold on;
surf (peaks (19), "facecolor", "none", "edgecolor", "black");
```

See also: [contour], page 285, [contourc], page 287, [contourf], page 286, [clabel], page 330, [meshc], page 308, [surfc], page 310, [caxis], page 300, [colormap], page 704, [plot], page 272.

The `errorbar`, `semilogxerr`, `semilogyerr`, and `loglogerr` functions produce plots with error bar markers. For example,

```
x = 0:0.1:10;
y = sin (x);
lerr = 0.1 .* rand (size (x));
uerr = 0.1 .* rand (size (x));
errorbar (x, y, lerr, uerr);
```

produces the figure shown in Figure 15.3.

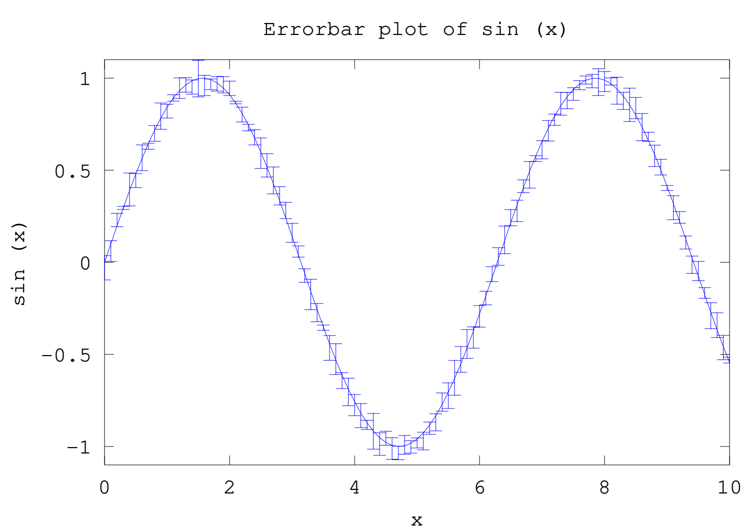

Figure 15.3: Errorbar plot.

`errorbar` (*y*, *ey*) [Function File]
`errorbar` (*y*, . . ., *fmt*) [Function File]
`errorbar` (*x*, *y*, *ey*) [Function File]
`errorbar` (*x*, *y*, *err*, *fmt*) [Function File]
`errorbar` (*x*, *y*, *lerr*, *uerr*, *fmt*) [Function File]
`errorbar` (*x*, *y*, *ex*, *ey*, *fmt*) [Function File]

`errorbar` (*x*, *y*, *lx*, *ux*, *ly*, *uy*, *fmt*) [Function File]
`errorbar` (*x1*, *y1*, ..., *fmt*, *xn*, *yn*, ...) [Function File]
`errorbar` (*hax*, ...) [Function File]
h = `errorbar` (...) [Function File]

Create a 2-D plot with errorbars.

Many different combinations of arguments are possible. The simplest form is

```
errorbar (y, ey)
```

where the first argument is taken as the set of *y* coordinates, the second argument *ey* are the errors around the *y* values, and the *x* coordinates are taken to be the indices of the elements (`1:numel (y)`).

The general form of the function is

```
errorbar (x, y, err1, ..., fmt, ...)
```

After the *x* and *y* arguments there can be 1, 2, or 4 parameters specifying the error values depending on the nature of the error values and the plot format *fmt*.

err (scalar)
: When the error is a scalar all points share the same error value. The errorbars are symmetric and are drawn from *data*-*err* to *data*+*err*. The *fmt* argument determines whether *err* is in the x-direction, y-direction (default), or both.

err (vector or matrix)
: Each data point has a particular error value. The errorbars are symmetric and are drawn from *data*(n)-*err*(n) to *data*(n)+*err*(n).

lerr, *uerr* (scalar)
: The errors have a single low-side value and a single upper-side value. The errorbars are not symmetric and are drawn from *data*-*lerr* to *data*+*uerr*.

lerr, *uerr* (vector or matrix)
: Each data point has a low-side error and an upper-side error. The errorbars are not symmetric and are drawn from *data*(n)-*lerr*(n) to *data*(n)+*uerr*(n).

Any number of data sets (*x1*,*y1*, *x2*,*y2*, ...) may appear as long as they are separated by a format string *fmt*.

If *y* is a matrix, *x* and the error parameters must also be matrices having the same dimensions. The columns of *y* are plotted versus the corresponding columns of *x* and errorbars are taken from the corresponding columns of the error parameters.

If *fmt* is missing, the yerrorbars (`"~"`) plot style is assumed.

If the *fmt* argument is supplied then it is interpreted, as in normal plots, to specify the line style, marker, and color. In addition, *fmt* may include an errorbar style which **must precede** the ordinary format codes. The following errorbar styles are supported:

'`~`' Set yerrorbars plot style (default).

'`>`' Set xerrorbars plot style.

'`~>`' Set xyerrorbars plot style.

‘#~’ Set yboxes plot style.

‘#’ Set xboxes plot style.

‘#~>’ Set xyboxes plot style.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a handle to the hggroup object representing the data plot and errorbars.

Note: For compatibility with MATLAB a line is drawn through all data points. However, most scientific errorbar plots are a scatter plot of points with errorbars. To accomplish this, add a marker style to the *fmt* argument such as `"."`. Alternatively, remove the line by modifying the returned graphic handle with `set (h, "linestyle", "none")`.

Examples:

```
errorbar (x, y, ex, ">.r")
```

produces an xerrorbar plot of *y* versus *x* with *x* errorbars drawn from *x*-*ex* to *x*+*ex*. The marker `"."` is used so no connecting line is drawn and the errorbars appear in red.

```
errorbar (x, y1, ey, "~",
          x, y2, ly, uy)
```

produces yerrorbar plots with *y1* and *y2* versus *x*. Errorbars for *y1* are drawn from *y1*-*ey* to *y1*+*ey*, errorbars for *y2* from *y2*-*ly* to *y2*+*uy*.

```
errorbar (x, y, lx, ux,
          ly, uy, "~>")
```

produces an xyerrorbar plot of *y* versus *x* in which *x* errorbars are drawn from *x*-*lx* to *x*+*ux* and *y* errorbars from *y*-*ly* to *y*+*uy*.

See also: [semilogxerr], page 290, [semilogyerr], page 291, [loglogerr], page 291, [plot], page 272.

`semilogxerr (y, ey)` [Function File]
`semilogxerr (y, ..., fmt)` [Function File]
`semilogxerr (x, y, ey)` [Function File]
`semilogxerr (x, y, err, fmt)` [Function File]
`semilogxerr (x, y, lerr, uerr, fmt)` [Function File]
`semilogxerr (x, y, ex, ey, fmt)` [Function File]
`semilogxerr (x, y, lx, ux, ly, uy, fmt)` [Function File]
`semilogxerr (x1, y1, ..., fmt, xn, yn, ...)` [Function File]
`semilogxerr (hax, ...)` [Function File]
`h = semilogxerr (...)` [Function File]

Produce 2-D plots using a logarithmic scale for the x-axis and errorbars at each data point.

Many different combinations of arguments are possible. The most common form is

```
semilogxerr (x, y, ey, fmt)
```

which produces a semi-logarithmic plot of *y* versus *x* with errors in the *y*-scale defined by *ey* and the plot format defined by *fmt*. See [errorbar], page 288, for available formats and additional information.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

See also: [errorbar], page 288, [semilogyerr], page 291, [loglogerr], page 291.

`semilogyerr (`*y*, *ey*`)` [Function File]
`semilogyerr (`*y*, ..., *fmt*`)` [Function File]
`semilogyerr (`*x*, *y*, *ey*`)` [Function File]
`semilogyerr (`*x*, *y*, *err*, *fmt*`)` [Function File]
`semilogyerr (`*x*, *y*, *lerr*, *uerr*, *fmt*`)` [Function File]
`semilogyerr (`*x*, *y*, *ex*, *ey*, *fmt*`)` [Function File]
`semilogyerr (`*x*, *y*, *lx*, *ux*, *ly*, *uy*, *fmt*`)` [Function File]
`semilogyerr (`*x1*, *y1*, ..., *fmt*, *xn*, *yn*, ...`)` [Function File]
`semilogyerr (`*hax*, ...`)` [Function File]
h `= semilogyerr (...)` [Function File]

Produce 2-D plots using a logarithmic scale for the y-axis and errorbars at each data point.

Many different combinations of arguments are possible. The most common form is

```
semilogyerr (x, y, ey, fmt)
```

which produces a semi-logarithmic plot of *y* versus *x* with errors in the *y*-scale defined by *ey* and the plot format defined by *fmt*. See [errorbar], page 288, for available formats and additional information.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

See also: [errorbar], page 288, [semilogxerr], page 290, [loglogerr], page 291.

`loglogerr (`*y*, *ey*`)` [Function File]
`loglogerr (`*y*, ..., *fmt*`)` [Function File]
`loglogerr (`*x*, *y*, *ey*`)` [Function File]
`loglogerr (`*x*, *y*, *err*, *fmt*`)` [Function File]
`loglogerr (`*x*, *y*, *lerr*, *uerr*, *fmt*`)` [Function File]
`loglogerr (`*x*, *y*, *ex*, *ey*, *fmt*`)` [Function File]
`loglogerr (`*x*, *y*, *lx*, *ux*, *ly*, *uy*, *fmt*`)` [Function File]
`loglogerr (`*x1*, *y1*, ..., *fmt*, *xn*, *yn*, ...`)` [Function File]
`loglogerr (`*hax*, ...`)` [Function File]
h `= loglogerr (...)` [Function File]

Produce 2-D plots on a double logarithm axis with errorbars.

Many different combinations of arguments are possible. The most common form is

```
loglogerr (x, y, ey, fmt)
```

which produces a double logarithm plot of *y* versus *x* with errors in the *y*-scale defined by *ey* and the plot format defined by *fmt*. See [errorbar], page 288, for available formats and additional information.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

See also: [errorbar], page 288, [semilogxerr], page 290, [semilogyerr], page 291.

Finally, the `polar` function allows you to easily plot data in polar coordinates. However, the display coordinates remain rectangular and linear. For example,

```
polar (0:0.1:10*pi, 0:0.1:10*pi);
```

produces the spiral plot shown in Figure 15.4.

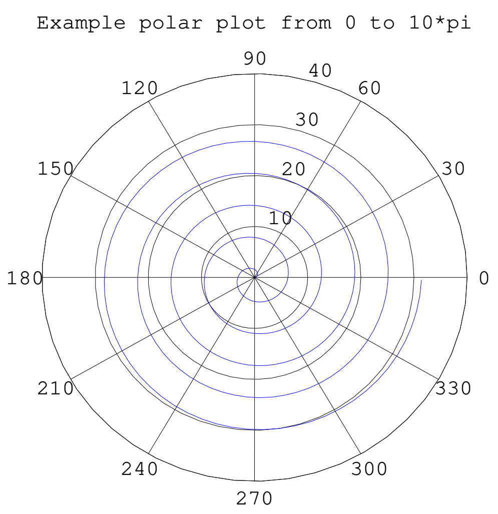

Figure 15.4: Polar plot.

`polar (`*theta*`,` *rho*`)` [Function File]
`polar (`*theta*`,` *rho*`,` *fmt*`)` [Function File]
`polar (`*cplx*`)` [Function File]
`polar (`*cplx*`,` *fmt*`)` [Function File]
`polar (`*hax*`, ...)` [Function File]
h `= polar (...)` [Function File]

Create a 2-D plot from polar coordinates *theta* and *rho*.

If a single complex input *cplx* is given then the real part is used for *theta* and the imaginary part is used for *rho*.

The optional argument *fmt* specifies the line format in the same way as `plot`.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

Implementation Note: The polar axis is drawn using line and text objects encapsulated in an hggroup. The hggroup properties are linked to the original axes object such that altering an appearance property, for example `fontname`, will update the polar axis. Two new properties are added to the original axes–`rtick`, `ttick`–which replace `xtick`, `ytick`. The first is a list of tick locations in the radial (rho) direction; The second is a list of tick locations in the angular (theta) direction specified in degrees, i.e., in the range 0–359.

See also: [rose], page 285, [compass], page 295, [plot], page 272.

`pie` (*x*) [Function File]
`pie` (. . ., *explode*) [Function File]
`pie` (. . ., *labels*) [Function File]
`pie` (*hax*, . . .); [Function File]
h = `pie` (. . .); [Function File]

Plot a 2-D pie chart.

When called with a single vector argument, produce a pie chart of the elements in *x*. The size of the ith slice is the percentage that the element *x*i represents of the total sum of *x*: `pct = x(i) / sum (x)`.

The optional input *explode* is a vector of the same length as *x* that, if nonzero, "explodes" the slice from the pie chart.

The optional input *labels* is a cell array of strings of the same length as *x* specifying the label for each slice.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a list of handles to the patch and text objects generating the plot.

Note: If `sum (x)` $\leq$ `1` then the elements of *x* are interpreted as percentages directly and are not normalized by `sum (x)`. Furthermore, if the sum is less than 1 then there will be a missing slice in the pie plot to represent the missing, unspecified percentage.

See also: [pie3], page 293, [bar], page 276, [hist], page 279, [rose], page 285.

`pie3` (*x*) [Function File]
`pie3` (. . ., *explode*) [Function File]
`pie3` (. . ., *labels*) [Function File]
`pie3` (*hax*, . . .); [Function File]
h = `pie3` (. . .); [Function File]

Plot a 3-D pie chart.

Called with a single vector argument, produces a 3-D pie chart of the elements in *x*. The size of the ith slice is the percentage that the element *x*i represents of the total sum of *x*: `pct = x(i) / sum (x)`.

The optional input *explode* is a vector of the same length as *x* that, if nonzero, "explodes" the slice from the pie chart.

The optional input *labels* is a cell array of strings of the same length as *x* specifying the label for each slice.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a list of graphics handles to the patch, surface, and text objects generating the plot.

Note: If `sum (x)` $\leq$ `1` then the elements of *x* are interpreted as percentages directly and are not normalized by `sum (x)`. Furthermore, if the sum is less than 1 then there will be a missing slice in the pie plot to represent the missing, unspecified percentage.

See also: [pie], page 293, [bar], page 276, [hist], page 279, [rose], page 285.

`quiver (u, v)` [Function File]
`quiver (x, y, u, v)` [Function File]
`quiver (..., s)` [Function File]
`quiver (..., style)` [Function File]
`quiver (..., "filled")` [Function File]
`quiver (hax, ...)` [Function File]
`h = quiver (...)` [Function File]

Plot a 2-D vector field with arrows.

Plot the (*u*, *v*) components of a vector field in an (*x*, *y*) meshgrid. If the grid is uniform then *x* and *y* can be specified as vectors.

If *x* and *y* are undefined they are assumed to be `(1:m, 1:n)` where `[m, n] = size (u)`.

The variable *s* is a scalar defining a scaling factor to use for the arrows of the field relative to the mesh spacing. A value of 0 disables all scaling. The default value is 0.9.

The style to use for the plot can be defined with a line style *style* of the same format as the `plot` command. If a marker is specified then markers at the grid points of the vectors are drawn rather than arrows. If the argument `"filled"` is given then the markers are filled.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to a quiver object. A quiver object regroups the components of the quiver plot (body, arrow, and marker), and allows them to be changed together.

Example:

```
[x, y] = meshgrid (1:2:20);
h = quiver (x, y, sin (2*pi*x/10), sin (2*pi*y/10));
set (h, "maxheadsize", 0.33);
```

See also: [quiver3], page 294, [compass], page 295, [feather], page 296, [plot], page 272.

`quiver3 (u, v, w)` [Function File]
`quiver3 (x, y, z, u, v, w)` [Function File]
`quiver3 (..., s)` [Function File]
`quiver3 (..., style)` [Function File]
`quiver3 (..., "filled")` [Function File]
`quiver3 (hax, ...)` [Function File]
`h = quiver3 (...)` [Function File]

Plot a 3-D vector field with arrows.

Plot the (*u*, *v*, *w*) components of a vector field in an (*x*, *y*, *z*) meshgrid. If the grid is uniform then *x*, *y*, and *z* can be specified as vectors.

If *x*, *y*, and *z* are undefined they are assumed to be `(1:m, 1:n, 1:p)` where `[m, n] = size (u)` and `p = max (size (w))`.

The variable *s* is a scalar defining a scaling factor to use for the arrows of the field relative to the mesh spacing. A value of 0 disables all scaling. The default value is 0.9.

The style to use for the plot can be defined with a line style *style* of the same format as the `plot` command. If a marker is specified then markers at the grid points of the vectors are drawn rather than arrows. If the argument `"filled"` is given then the markers are filled.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to a quiver object. A quiver object regroups the components of the quiver plot (body, arrow, and marker), and allows them to be changed together.

```
[x, y, z] = peaks (25);
surf (x, y, z);
hold on;
[u, v, w] = surfnorm (x, y, z / 10);
h = quiver3 (x, y, z, u, v, w);
set (h, "maxheadsize", 0.33);
```

See also: [quiver], page 294, [compass], page 295, [feather], page 296, [plot], page 272.

`compass (u, v)` [Function File]
`compass (z)` [Function File]
`compass (..., style)` [Function File]
`compass (hax, ...)` [Function File]
`h = compass (...)` [Function File]

Plot the `(u, v)` components of a vector field emanating from the origin of a polar plot.

The arrow representing each vector has one end at the origin and the tip at [*u*(i), *v*(i)]. If a single complex argument *z* is given, then `u = real (z)` and `v = imag (z)`.

The style to use for the plot can be defined with a line style *style* of the same format as the `plot` command.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of graphics handles to the line objects representing the drawn vectors.

```
a = toeplitz ([1;randn(9,1)], [1,randn(1,9)]);
compass (eig (a));
```

See also: [polar], page 292, [feather], page 296, [quiver], page 294, [rose], page 285, [plot], page 272.

`feather (u, v)` [Function File]
`feather (z)` [Function File]
`feather (..., style)` [Function File]
`feather (hax, ...)` [Function File]
`h = feather (...)` [Function File]

Plot the `(u, v)` components of a vector field emanating from equidistant points on the x-axis.

If a single complex argument *z* is given, then `u = real (z)` and `v = imag (z)`.

The style to use for the plot can be defined with a line style *style* of the same format as the `plot` command.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of graphics handles to the line objects representing the drawn vectors.

```
phi = [0 : 15 : 360] * pi/180;
feather (sin (phi), cos (phi));
```

See also: [plot], page 272, [quiver], page 294, [compass], page 295.

`pcolor (x, y, c)` [Function File]
`pcolor (c)` [Function File]
`pcolor (hax, ...)` [Function File]
`h = pcolor (...)` [Function File]

Produce a 2-D density plot.

A `pcolor` plot draws rectangles with colors from the matrix *c* over the two-dimensional region represented by the matrices *x* and *y*. *x* and *y* are the coordinates of the mesh's vertices and are typically the output of `meshgrid`. If *x* and *y* are vectors, then a typical vertex is (*x*(j), *y*(i), *c*(i,j)). Thus, columns of *c* correspond to different *x* values and rows of *c* correspond to different *y* values.

The values in *c* are scaled to span the range of the current colormap. Limits may be placed on the color axis by the command `caxis`, or by setting the `clim` property of the parent axis.

The face color of each cell of the mesh is determined by interpolating the values of *c* for each of the cell's vertices; Contrast this with `imagesc` which renders one cell for each element of *c*.

`shading` modifies an attribute determining the manner by which the face color of each cell is interpolated from the values of *c*, and the visibility of the cells' edges. By default the attribute is `"faceted"`, which renders a single color for each cell's face with the edge visible.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

See also: [caxis], page 300, [shading], page 320, [meshgrid], page 317, [contour], page 285, [imagesc], page 702.

`area (y)` [Function File]
`area (x, y)` [Function File]
`area (..., lvl)` [Function File]
`area (..., prop, val, ...)` [Function File]
`area (hax, ...)` [Function File]
`h = area (...)` [Function File]

Area plot of the columns of *y*.

This plot shows the contributions of each column value to the row sum. It is functionally similar to `plot (x, cumsum (y, 2))`, except that the area under the curve is shaded.

If the *x* argument is omitted it defaults to `1:rows (y)`. A value *lvl* can be defined that determines where the base level of the shading under the curve should be defined. The default level is 0.

Additional property/value pairs are passed directly to the underlying patch object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the hggroup object comprising the area patch objects. The `"BaseValue"` property of the hggroup can be used to adjust the level where shading begins.

Example: Verify identity sin^2 + cos^2 = 1

```
t = linspace (0, 2*pi, 100)';
y = [sin(t).^2, cos(t).^2];
area (t, y);
legend ("sin^2", "cos^2", "location", "NorthEastOutside");
```

See also: [plot], page 272, [patch], page 357.

`fill (x, y, c)` [Function File]
`fill (x1, y1, c1, x2, y2, c2)` [Function File]
`fill (..., prop, val)` [Function File]
`fill (hax, ...)` [Function File]
`h = fill (...)` [Function File]

Create one or more filled 2-D polygons.

The inputs *x* and *y* are the coordinates of the polygon vertices. If the inputs are matrices then the rows represent different vertices and each column produces a different polygon. `fill` will close any open polygons before plotting.

The input *c* determines the color of the polygon. The simplest form is a single color specification such as a `plot` format or an RGB-triple. In this case the polygon(s) will have one unique color. If *c* is a vector or matrix then the color data is first scaled using `caxis` and then indexed into the current colormap. A row vector will color each polygon (a column from matrices *x* and *y*) with a single computed color. A matrix *c* of the same size as *x* and *y* will compute the color of each vertex and then interpolate the face color between the vertices.

Multiple property/value pairs for the underlying patch object may be specified, but they must appear in pairs.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of graphics handles to the created patch objects.

Example: red square

```
vertices = [0 0
            1 0
            1 1
            0 1];
fill (vertices(:,1), vertices(:,2), "r");
axis ([-0.5 1.5, -0.5 1.5])
axis equal
```

See also: [patch], page 357, [caxis], page 300, [colormap], page 704.

`comet (`*y*`)` [Function File]
`comet (`*x*`, `*y*`)` [Function File]
`comet (`*x*`, `*y*`, `*p*`)` [Function File]
`comet (`*hax*`, ...)` [Function File]

Produce a simple comet style animation along the trajectory provided by the input coordinate vectors (*x*, *y*).

If *x* is not specified it defaults to the indices of *y*.

The speed of the comet may be controlled by *p*, which represents the time each point is displayed before moving to the next one. The default for *p* is 0.1 seconds.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

See also: [comet3], page 298.

`comet3 (`*z*`)` [Function File]
`comet3 (`*x*`, `*y*`, `*z*`)` [Function File]
`comet3 (`*x*`, `*y*`, `*z*`, `*p*`)` [Function File]
`comet3 (`*hax*`, ...)` [Function File]

Produce a simple comet style animation along the trajectory provided by the input coordinate vectors (*x*, *y*, *z*).

If only *z* is specified then *x*, *y* default to the indices of *z*.

The speed of the comet may be controlled by *p*, which represents the time each point is displayed before moving to the next one. The default for *p* is 0.1 seconds.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

See also: [comet], page 298.

`[`*x*`, `*map*`] = frame2im (`*f*`)` [Function File]

Convert movie frame to indexed image.

A movie frame is simply a struct with the fields `"cdata"` and `"colormap"`.

Support for N-dimensional images or movies is given when *f* is a struct array. In such cases, *x* will be a MxNx1xK or MxNx3xK for indexed and RGB movies respectively, with each frame concatenated along the 4th dimension.

See also: [im2frame], page 299.

`im2frame (rgb)` [Function File]
`im2frame (x, map)` [Function File]

Convert image to movie frame.

A movie frame is simply a struct with the fields `"cdata"` and `"colormap"`.

Support for N-dimensional images is given when each image projection, matrix sizes of MxN and MxNx3 for RGB images, is concatenated along the fourth dimension. In such cases, the returned value is a struct array.

See also: [frame2im], page 298.

15.2.1.1 Axis Configuration

The axis function may be used to change the axis limits of an existing plot and various other axis properties, such as the aspect ratio and the appearance of tic marks.

`axis ()` [Function File]
`axis ([x_lo x_hi])` [Function File]
`axis ([x_lo x_hi y_lo y_hi])` [Function File]
`axis ([x_lo x_hi y_lo y_hi z_lo z_hi])` [Function File]
`axis (option)` [Function File]
`axis (..., option)` [Function File]
`axis (hax, ...)` [Function File]
`limits = axis ()` [Function File]

Set axis limits and appearance.

The argument *limits* should be a 2-, 4-, or 6-element vector. The first and second elements specify the lower and upper limits for the x-axis. The third and fourth specify the limits for the y-axis, and the fifth and sixth specify the limits for the z-axis. The special values -Inf and Inf may be used to indicate that the limit should automatically be computed based on the data in the axis.

Without any arguments, `axis` turns autoscaling on.

With one output argument, `limits = axis` returns the current axis limits.

The vector argument specifying limits is optional, and additional string arguments may be used to specify various axis properties. For example,

```
axis ([1, 2, 3, 4], "square");
```

forces a square aspect ratio, and

```
axis ("tic", "labely");
```

turns tic marks on for all axes and tic mark labels on for the y-axis only.

The following options control the aspect ratio of the axes.

`"square"` Force a square aspect ratio.

`"equal"` Force x distance to equal y-distance.

`"normal"` Restore default aspect ratio.

The following options control the way axis limits are interpreted.

`"auto"` Set the specified axes to have nice limits around the data or all if no axes are specified.

`"manual"` Fix the current axes limits.

`"tight"` Fix axes to the limits of the data.

`"image"` Equivalent to `"tight"` and `"equal"`.

The following options affect the appearance of tic marks.

`"on"` Turn tic marks and labels on for all axes.

`"off"` Turn tic marks off for all axes.

`"tic[xyz]"`
: Turn tic marks on for all axes, or turn them on for the specified axes and off for the remainder.

`"label[xyz]"`
: Turn tic labels on for all axes, or turn them on for the specified axes and off for the remainder.

`"nolabel"`
: Turn tic labels off for all axes.

Note, if there are no tic marks for an axis, there can be no labels.

The following options affect the direction of increasing values on the axes.

`"ij"` Reverse y-axis, so lower values are nearer the top.

`"xy"` Restore y-axis, so higher values are nearer the top.

If the first argument *hax* is an axes handle, then operate on this axes rather than the current axes returned by `gca`.

See also: [xlim], page 301, [ylim], page 301, [zlim], page 301, [daspect], page 322, [pbaspect], page 323, [box], page 331, [grid], page 331.

Similarly the axis limits of the colormap can be changed with the caxis function.

`caxis ([`*cmin cmax*`])` [Function File]
`caxis ("`*auto*`")` [Function File]
`caxis ("`*manual*`")` [Function File]
`caxis (`*hax*`, ...)` [Function File]
`limits` `= caxis ()` [Function File]

Query or set color axis limits for plots.

The limits argument should be a 2-element vector specifying the lower and upper limits to assign to the first and last value in the colormap. Data values outside this range are clamped to the first and last colormap entries.

If the `"auto"` option is given then automatic colormap limits are applied. The automatic algorithm sets *cmin* to the minimum data value and *cmax* to the maximum data value. If "manual" is specified then the "climmode" property is set to "manual" and the numeric values in the "clim" property are used for limits.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by gca.

Called without arguments the current color axis limits are returned.

See also: [colormap], page 704.

The xlim, ylim, and zlim functions may be used to get or set individual axis limits. Each has the same form.

xlimits = xlim () [Function File]
xmode = xlim ("*mode*") [Function File]
xlim ([*x_lo x_hi*]) [Function File]
xlim ("*auto*") [Function File]
xlim ("*manual*") [Function File]
xlim (*hax*, ...) [Function File]

Query or set the limits of the x-axis for the current plot.

Called without arguments xlim returns the x-axis limits of the current plot.

With the input query "mode", return the current x-limit calculation mode which is either "auto" or "manual".

If passed a 2-element vector [*x_lo x_hi*], the limits of the x-axis are set to these values and the mode is set to "manual".

The current plotting mode can be changed by using either "auto" or "manual" as the argument.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by gca.

See also: [ylim], page 301, [zlim], page 301, [axis], page 299, [set], page 361, [get], page 361, [gca], page 360.

15.2.1.2 Two-dimensional Function Plotting

Octave can plot a function from a function handle, inline function, or string defining the function without the user needing to explicitly create the data to be plotted. The function fplot also generates two-dimensional plots with linear axes using a function name and limits for the range of the x-coordinate instead of the x and y data. For example,

```
fplot (@sin, [-10, 10], 201);
```

produces a plot that is equivalent to the one above, but also includes a legend displaying the name of the plotted function.

fplot (*fn*, *limits*) [Function File]
fplot (..., *tol*) [Function File]
fplot (..., *n*) [Function File]
fplot (..., *fmt*) [Function File]
[*x*, *y*] = fplot (...) [Function File]

Plot a function *fn* within the range defined by *limits*.

fn is a function handle, inline function, or string containing the name of the function to evaluate.

The limits of the plot are of the form `[xlo, xhi]` or `[xlo, xhi, ylo, yhi]`.

The next three arguments are all optional and any number of them may be given in any order.

tol is the relative tolerance to use for the plot and defaults to 2e-3 (.2%).

n is the minimum number of points to use. When *n* is specified, the maximum stepsize will be `xhi - xlo / n`. More than *n* points may still be used in order to meet the relative tolerance requirement.

The *fmt* argument specifies the linestyle to be used by the plot command.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

With no output arguments the results are immediately plotted. With two output arguments the 2-D plot data is returned. The data can subsequently be plotted manually with `plot (x, y)`.

Example:

```
fplot (@cos, [0, 2*pi])
fplot ("[cos(x), sin(x)]", [0, 2*pi])
```

Programming Notes:

`fplot` works best with continuous functions. Functions with discontinuities are unlikely to plot well. This restriction may be removed in the future.

`fplot` requires that the function accept and return a vector argument. Consider this when writing user-defined functions and use `.*`, `./`, etc. See the function `vectorize` for potentially converting inline or anonymous functions to vectorized versions.

See also: [ezplot], page 302, [plot], page 272, [vectorize], page 490.

Other functions that can create two-dimensional plots directly from a function include `ezplot`, `ezcontour`, `ezcontourf` and `ezpolar`.

`ezplot (f)` [Function File]
`ezplot (f2v)` [Function File]
`ezplot (fx, fy)` [Function File]
`ezplot (..., dom)` [Function File]
`ezplot (..., n)` [Function File]
`ezplot (hax, ...)` [Function File]
`h = ezplot (...)` [Function File]

Plot the 2-D curve defined by the function *f*.

The function *f* may be a string, inline function, or function handle and can have either one or two variables. If *f* has one variable, then the function is plotted over the domain `-2*pi < x < 2*pi` with 500 points.

If *f2v* is a function of two variables then the implicit function `f(x,y) = 0` is calculated over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

For example:

```
ezplot (@(x, y) x.^2 - y.^2 - 1)
```

If two functions are passed as inputs then the parametric function

```
x = fx (t)
y = fy (t)
```

is plotted over the domain `-2*pi <= t <= 2*pi` with 500 points.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*, or *t* for a parametric plot. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in plotting the function.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of graphics handles to the created line objects.

See also: [plot], page 272, [ezplot3], page 323, [ezpolar], page 304, [ezcontour], page 303, [ezcontourf], page 303, [ezmesh], page 323, [ezmeshc], page 324, [ezsurf], page 325, [ezsurfc], page 326.

`ezcontour` (*f*) [Function File]
`ezcontour` (..., *dom*) [Function File]
`ezcontour` (..., *n*) [Function File]
`ezcontour` (*hax*, ...) [Function File]
h = `ezcontour` (...) [Function File]

Plot the contour lines of a function.

f is a string, inline function, or function handle with two arguments defining the function. By default the plot is over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in each dimension.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

Example:

```
f = @(x,y) sqrt (abs (x .* y)) ./ (1 + x.^2 + y.^2);
ezcontour (f, [-3, 3]);
```

See also: [contour], page 285, [ezcontourf], page 303, [ezplot], page 302, [ezmeshc], page 324, [ezsurfc], page 326.

`ezcontourf` (*f*) [Function File]
`ezcontourf` (..., *dom*) [Function File]
`ezcontourf` (..., *n*) [Function File]
`ezcontourf` (*hax*, ...) [Function File]
h = `ezcontourf` (...) [Function File]

Plot the filled contour lines of a function.

f is a string, inline function, or function handle with two arguments defining the function. By default the plot is over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in each dimension.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

Example:

```
f = @(x,y) sqrt (abs (x .* y)) ./ (1 + x.^2 + y.^2);
ezcontourf (f, [-3, 3]);
```

See also: [contourf], page 286, [ezcontour], page 303, [ezplot], page 302, [ezmeshc], page 324, [ezsurfc], page 326.

`ezpolar (`*f*`)` [Function File]
`ezpolar (...,` *dom*`)` [Function File]
`ezpolar (...,` *n*`)` [Function File]
`ezpolar (`*hax*`, ...)` [Function File]
h `= ezpolar (...)` [Function File]

Plot a 2-D function in polar coordinates.

The function *f* is a string, inline function, or function handle with a single argument. The expected form of the function is *rho* `=` *f*`(`*theta*`)`. By default the plot is over the domain `0 <=` *theta* `<= 2*pi` with 500 points.

If *dom* is a two element vector, it represents the minimum and maximum values of *theta*.

n is a scalar defining the number of points to use in plotting the function.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

Example:

```
ezpolar (@(t) sin (5/4 * t), [0, 8*pi]);
```

See also: [polar], page 292, [ezplot], page 302.

15.2.1.3 Two-dimensional Geometric Shapes

`rectangle ()` [Function File]
`rectangle (..., "`*Position*`",` *pos*`)` [Function File]
`rectangle (..., "`*Curvature*`",` *curv*`)` [Function File]
`rectangle (..., "`*EdgeColor*`",` *ec*`)` [Function File]
`rectangle (..., "`*FaceColor*`",` *fc*`)` [Function File]
`rectangle (`*hax*`, ...)` [Function File]

`h = rectangle (...)` [Function File]

Draw a rectangular patch defined by *pos* and *curv*.

The variable *pos*(1:2) defines the lower left-hand corner of the patch and *pos*(3:4) defines its width and height. By default, the value of *pos* is [0, 0, 1, 1].

The variable *curv* defines the curvature of the sides of the rectangle and may be a scalar or two-element vector with values between 0 and 1. A value of 0 represents no curvature of the side, whereas a value of 1 means that the side is entirely curved into the arc of a circle. If *curv* is a two-element vector, then the first element is the curvature along the x-axis of the patch and the second along y-axis.

If *curv* is a scalar, it represents the curvature of the shorter of the two sides of the rectangle and the curvature of the other side is defined by

```
min (pos(1:2)) / max (pos(1:2)) * curv
```

Additional property/value pairs are passed to the underlying patch command.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created rectangle object.

See also: [patch], page 357, [line], page 357, [cylinder], page 326, [ellipsoid], page 327, [sphere], page 327.

15.2.2 Three-Dimensional Plots

The function `mesh` produces mesh surface plots. For example,

```
tx = ty = linspace (-8, 8, 41)';
[xx, yy] = meshgrid (tx, ty);
r = sqrt (xx .^ 2 + yy .^ 2) + eps;
tz = sin (r) ./ r;
mesh (tx, ty, tz);
```

produces the familiar "sombrero" plot shown in Figure 15.5. Note the use of the function `meshgrid` to create matrices of X and Y coordinates to use for plotting the Z data. The `ndgrid` function is similar to `meshgrid`, but works for N-dimensional matrices.

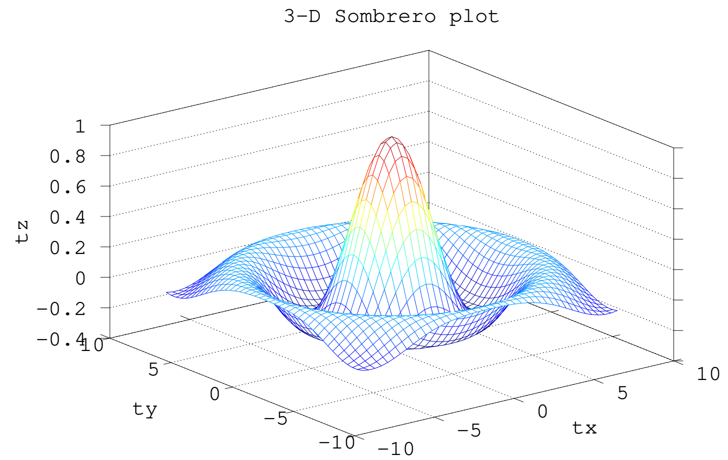

Figure 15.5: Mesh plot.

The `meshc` function is similar to `mesh`, but also produces a plot of contours for the surface.

The `plot3` function displays arbitrary three-dimensional data, without requiring it to form a surface. For example,

```
t = 0:0.1:10*pi;
r = linspace (0, 1, numel (t));
z = linspace (0, 1, numel (t));
plot3 (r.*sin(t), r.*cos(t), z);
```

displays the spiral in three dimensions shown in Figure 15.6.

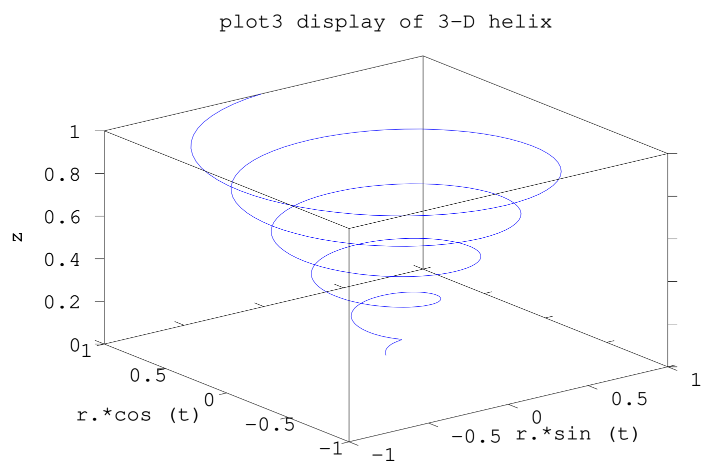

Figure 15.6: Three-dimensional spiral.

Finally, the `view` function changes the viewpoint for three-dimensional plots.

mesh (*x*, *y*, *z*) [Function File]
mesh (*z*) [Function File]
mesh (. . ., *c*) [Function File]
mesh (. . ., *prop*, *val*, . . .) [Function File]
mesh (*hax*, . . .) [Function File]
h = **mesh** (. . .) [Function File]

Plot a 3-D wireframe mesh.

The wireframe mesh is plotted using rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 1:columns (z), y = 1:rows (z)`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The color of the mesh is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally, the color of the mesh can be specified independently of *z* by supplying a color matrix, *c*.

Any property/value pairs are passed directly to the underlying surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

See also: [ezmesh], page 323, [meshc], page 308, [meshz], page 308, [trimesh], page 667, [contour], page 285, [surf], page 309, [surface], page 358, [meshgrid], page 317, [hidden], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

`meshc (x, y, z)` [Function File]
`meshc (z)` [Function File]
`meshc (..., c)` [Function File]
`meshc (..., prop, val, ...)` [Function File]
`meshc (hax, ...)` [Function File]
`h = meshc (...)` [Function File]

Plot a 3-D wireframe mesh with underlying contour lines.

The wireframe mesh is plotted using rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 1:columns (z), y = 1:rows (z)`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The color of the mesh is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally the color of the mesh can be specified independently of *z* by supplying a color matrix, *c*.

Any property/value pairs are passed directly to the underlying surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a 2-element vector with a graphics handle to the created surface object and to the created contour plot.

See also: [ezmeshc], page 324, [mesh], page 307, [meshz], page 308, [contour], page 285, [surfc], page 310, [surface], page 358, [meshgrid], page 317, [hidden], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

`meshz (x, y, z)` [Function File]
`meshz (z)` [Function File]
`meshz (..., c)` [Function File]
`meshz (..., prop, val, ...)` [Function File]
`meshz (hax, ...)` [Function File]
`h = meshz (...)` [Function File]

Plot a 3-D wireframe mesh with a surrounding curtain.

The wireframe mesh is plotted using rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 0:columns (z) - 1, y = 0:rows (z) - 1`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The color of the mesh is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally the color of the mesh can be specified independently of *z* by supplying a color matrix, *c*.

Any property/value pairs are passed directly to the underlying surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

See also: [mesh], page 307, [meshc], page 308, [contour], page 285, [surf], page 309, [surface], page 358, [waterfall], page 321, [meshgrid], page 317, [hidden], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

`hidden` [Command]
`hidden` *on* [Command]
`hidden` *off* [Command]
mode `= hidden (...)` [Function File]

Control mesh hidden line removal.

When called with no argument the hidden line removal state is toggled.

When called with one of the modes `"on"` or `"off"` the state is set accordingly.

The optional output argument *mode* is the current state.

Hidden Line Removal determines what graphic objects behind a mesh plot are visible. The default is for the mesh to be opaque and lines behind the mesh are not visible. If hidden line removal is turned off then objects behind the mesh can be seen through the faces (openings) of the mesh, although the mesh grid lines are still opaque.

See also: [mesh], page 307, [meshc], page 308, [meshz], page 308, [ezmesh], page 323, [ezmeshc], page 324, [trimesh], page 667, [waterfall], page 321.

`surf (`*x*, *y*, *z*`)` [Function File]
`surf (`*z*`)` [Function File]
`surf (..., `*c*`)` [Function File]
`surf (..., `*prop*, *val*`, ...)` [Function File]
`surf (`*hax*`, ...)` [Function File]
h `= surf (...)` [Function File]

Plot a 3-D surface mesh.

The surface mesh is plotted using shaded rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 1:columns (z), y = 1:rows (z)`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The color of the surface is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally, the color of the surface can be specified independently of *z* by supplying a color matrix, *c*.

Any property/value pairs are passed directly to the underlying surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

Note: The exact appearance of the surface can be controlled with the `shading` command or by using `set` to control surface object properties.

See also: [ezsurf], page 325, [surfc], page 310, [surfl], page 310, [surfnorm], page 311, [trisurf], page 668, [contour], page 285, [mesh], page 307, [surface], page 358, [meshgrid], page 317, [hidden], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

`surfc (x, y, z)` [Function File]
`surfc (z)` [Function File]
`surfc (..., c)` [Function File]
`surfc (..., prop, val, ...)` [Function File]
`surfc (hax, ...)` [Function File]
`h = surfc (...)` [Function File]

Plot a 3-D surface mesh with underlying contour lines.

The surface mesh is plotted using shaded rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 1:columns (z), y = 1:rows (z)`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The color of the surface is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally, the color of the surface can be specified independently of *z* by supplying a color matrix, *c*.

Any property/value pairs are passed directly to the underlying surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

Note: The exact appearance of the surface can be controlled with the `shading` command or by using `set` to control surface object properties.

See also: [ezsurfc], page 326, [surf], page 309, [surfl], page 310, [surfnorm], page 311, [trisurf], page 668, [contour], page 285, [mesh], page 307, [surface], page 358, [meshgrid], page 317, [hidden], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

`surfl (z)` [Function File]
`surfl (x, y, z)` [Function File]
`surfl (..., lsrc)` [Function File]
`surfl (x, y, z, lsrc, P)` [Function File]
`surfl (..., "cdata")` [Function File]
`surfl (..., "light")` [Function File]
`surfl (hax, ...)` [Function File]
`h = surfl (...)` [Function File]

Plot a 3-D surface using shading based on various lighting models.

The surface mesh is plotted using shaded rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 1:columns (z), y = 1:rows (z)`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The default lighting mode `"cdata"`, changes the cdata property of the surface object to give the impression of a lighted surface. **Warning:** The alternative mode `"light"` mode which creates a light object to illuminate the surface is not implemented (yet).

The light source location can be specified using *lsrc*. It can be given as a 2-element vector [azimuth, elevation] in degrees, or as a 3-element vector [lx, ly, lz]. The default value is rotated 45 degrees counterclockwise to the current view.

The material properties of the surface can specified using a 4-element vector $P = [AM\ D\ SP\ exp]$ which defaults to $p = [0.55\ 0.6\ 0.4\ 10]$.

`"AM"` strength of ambient light
`"D"` strength of diffuse reflection
`"SP"` strength of specular reflection
`"EXP"` specular exponent

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

Example:

```
colormap (bone (64));
surfl (peaks);
shading interp;
```

See also: [diffuse], page 317, [specular], page 317, [surf], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

`surfnorm (x, y, z)` [Function File]
`surfnorm (z)` [Function File]
`surfnorm (..., prop, val, ...)` [Function File]
`surfnorm (hax, ...)` [Function File]
`[nx, ny, nz] = surfnorm (...)` [Function File]

Find the vectors normal to a meshgridded surface.

If *x* and *y* are vectors, then a typical vertex is (*x*(j), *y*(i), *z*(i,j)). Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values. If only a single input *z* is given then *x* is taken to be `1:rows (z)` and *y* is `1:columns (z)`.

If no return arguments are requested, a surface plot with the normal vectors to the surface is plotted.

Any property/value input pairs are assigned to the surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

If output arguments are requested then the components of the normal vectors are returned in *nx*, *ny*, and *nz* and no plot is made.

An example of the use of `surfnorm` is

```
surfnorm (peaks (25));
```

Algorithm: The normal vectors are calculated by taking the cross product of the diagonals of each of the quadrilaterals in the meshgrid to find the normal vectors of the centers of these quadrilaterals. The four nearest normal vectors to the meshgrid points are then averaged to obtain the normal to the surface at the meshgridded points.

See also: [isonormals], page 313, [quiver3], page 294, [surf], page 309, [meshgrid], page 317.

[*fv*] = isosurface (*val*, *iso*) [Function File]
[*fv*] = isosurface (*x*, *y*, *z*, *val*, *iso*) [Function File]
[*fv*] = isosurface (. . . , "*noshare*", "*verbose*") [Function File]
[*fvc*] = isosurface (. . . , *col*) [Function File]
[*f*, *v*] = isosurface (*x*, *y*, *z*, *val*, *iso*) [Function File]
[*f*, *v*, *c*] = isosurface (*x*, *y*, *z*, *val*, *iso*, *col*) [Function File]
isosurface (*x*, *y*, *z*, *val*, *iso*, *col*, *opt*) [Function File]

Calculate isosurface of 3-D data.

If called with one output argument and the first input argument *val* is a three-dimensional array that contains the data of an isosurface geometry and the second input argument *iso* keeps the isovalue as a scalar value then return a structure array *fv* that contains the fields *Faces* and *Vertices* at computed points `[x, y, z] = meshgrid (1:l, 1:m, 1:n)`. The output argument *fv* can directly be taken as an input argument for the `patch` function.

If called with further input arguments *x*, *y* and *z* which are three–dimensional arrays with the same size than *val* then the volume data is taken at those given points.

The string input argument `"noshare"` is only for compatibility and has no effect. If given the string input argument `"verbose"` then print messages to the command line interface about the current progress.

If called with the input argument *col* which is a three-dimensional array of the same size than *val* then take those values for the interpolation of coloring the isosurface geometry. Add the field *FaceVertexCData* to the structure array *fv*.

If called with two or three output arguments then return the information about the faces *f*, vertices *v* and color data *c* as separate arrays instead of a single structure array.

If called with no output argument then directly process the isosurface geometry with the `patch` command.

For example,

```
[x, y, z] = meshgrid (1:5, 1:5, 1:5);
val = rand (5, 5, 5);
isosurface (x, y, z, val, .5);
```

will directly draw a random isosurface geometry in a graphics window. Another example for an isosurface geometry with different additional coloring

```
N = 15;    # Increase number of vertices in each direction
iso = .4;  # Change isovalue to .1 to display a sphere
lin = linspace (0, 2, N);
[x, y, z] = meshgrid (lin, lin, lin);
c = abs ((x-.5).^2 + (y-.5).^2 + (z-.5).^2);
figure (); # Open another figure window

subplot (2,2,1); view (-38, 20);
[f, v] = isosurface (x, y, z, c, iso);
p = patch ("Faces", f, "Vertices", v, "EdgeColor", "none");
set (gca, "PlotBoxAspectRatioMode", "manual", ...
          "PlotBoxAspectRatio", [1 1 1]);
# set (p, "FaceColor", "green", "FaceLighting", "phong");
# light ("Position", [1 1 5]); # Available with the JHandles package

subplot (2,2,2); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "EdgeColor", "blue");
set (gca, "PlotBoxAspectRatioMode", "manual", ...
          "PlotBoxAspectRatio", [1 1 1]);
# set (p, "FaceColor", "none", "FaceLighting", "phong");
# light ("Position", [1 1 5]);

subplot (2,2,3); view (-38, 20);
[f, v, c] = isosurface (x, y, z, c, iso, y);
p = patch ("Faces", f, "Vertices", v, "FaceVertexCData", c, ...
           "FaceColor", "interp", "EdgeColor", "none");
set (gca, "PlotBoxAspectRatioMode", "manual", ...
          "PlotBoxAspectRatio", [1 1 1]);
# set (p, "FaceLighting", "phong");
# light ("Position", [1 1 5]);

subplot (2,2,4); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "FaceVertexCData", c, ...
           "FaceColor", "interp", "EdgeColor", "blue");
set (gca, "PlotBoxAspectRatioMode", "manual", ...
          "PlotBoxAspectRatio", [1 1 1]);
# set (p, "FaceLighting", "phong");
# light ("Position", [1 1 5]);
```

See also: [isonormals], page 313, [isocolors], page 314.

`[`*n*`] = isonormals (`*val*, *v*`)` [Function File]
`[`*n*`] = isonormals (`*val*, *p*`)` [Function File]
`[`*n*`] = isonormals (`*x*, *y*, *z*, *val*, *v*`)` [Function File]
`[`*n*`] = isonormals (`*x*, *y*, *z*, *val*, *p*`)` [Function File]
`[`*n*`] = isonormals (`..., "*negate*"`)` [Function File]
`isonormals (`..., *p*`)` [Function File]

Calculate normals to an isosurface.

If called with one output argument and the first input argument *val* is a three-dimensional array that contains the data for an isosurface geometry and the second input argument *v* keeps the vertices of an isosurface then return the normals *n* in form of a matrix with the same size than *v* at computed points `[x, y, z] =`

`meshgrid (1:l, 1:m, 1:n)`. The output argument *n* can be taken to manually set *VertexNormals* of a patch.

If called with further input arguments *x*, *y* and *z* which are three–dimensional arrays with the same size than *val* then the volume data is taken at those given points. Instead of the vertices data *v* a patch handle *p* can be passed to this function.

If given the string input argument `"negate"` as last input argument then compute the reverse vector normals of an isosurface geometry.

If no output argument is given then directly redraw the patch that is given by the patch handle *p*.

For example:

```
function [] = isofinish (p)
  set (gca, "PlotBoxAspectRatioMode", "manual", ...
            "PlotBoxAspectRatio", [1 1 1]);
  set (p, "VertexNormals", -get (p,"VertexNormals")); # Revert normals
  set (p, "FaceColor", "interp");
  ## set (p, "FaceLighting", "phong");
  ## light ("Position", [1 1 5]); # Available with JHandles
endfunction

N = 15;    # Increase number of vertices in each direction
iso = .4;  # Change isovalue to .1 to display a sphere
lin = linspace (0, 2, N);
[x, y, z] = meshgrid (lin, lin, lin);
c = abs ((x-.5).^2 + (y-.5).^2 + (z-.5).^2);
figure (); # Open another figure window

subplot (2,2,1); view (-38, 20);
[f, v, cdat] = isosurface (x, y, z, c, iso, y);
p = patch ("Faces", f, "Vertices", v, "FaceVertexCData", cdat, ...
           "FaceColor", "interp", "EdgeColor", "none");
isofinish (p);  # Call user function isofinish

subplot (2,2,2); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "FaceVertexCData", cdat, ...
           "FaceColor", "interp", "EdgeColor", "none");
isonormals (x, y, z, c, p); # Directly modify patch
isofinish (p);

subplot (2,2,3); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "FaceVertexCData", cdat, ...
           "FaceColor", "interp", "EdgeColor", "none");
n = isonormals (x, y, z, c, v); # Compute normals of isosurface
set (p, "VertexNormals", n);    # Manually set vertex normals
isofinish (p);

subplot (2,2,4); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "FaceVertexCData", cdat, ...
           "FaceColor", "interp", "EdgeColor", "none");
isonormals (x, y, z, c, v, "negate"); # Use reverse directly
isofinish (p);
```

See also: [isosurface], page 312, [isocolors], page 314.

[*cd*] = **isocolors** (*c*, *v*) [Function File]
[*cd*] = **isocolors** (*x*, *y*, *z*, *c*, *v*) [Function File]

`[cd] = isocolors (x, y, z, r, g, b, v)` [Function File]
`[cd] = isocolors (r, g, b, v)` [Function File]
`[cd] = isocolors (..., p)` [Function File]
`isocolors (...)` [Function File]

Compute isosurface colors.

If called with one output argument and the first input argument *c* is a three-dimensional array that contains color values and the second input argument *v* keeps the vertices of a geometry then return a matrix *cd* with color data information for the geometry at computed points `[x, y, z] = meshgrid (1:l, 1:m, 1:n)`. The output argument *cd* can be taken to manually set FaceVertexCData of a patch.

If called with further input arguments *x*, *y* and *z* which are three–dimensional arrays of the same size than *c* then the color data is taken at those given points. Instead of the color data *c* this function can also be called with RGB values *r*, *g*, *b*. If input argumnets *x*, *y*, *z* are not given then again `meshgrid` computed values are taken.

Optionally, the patch handle *p* can be given as the last input argument to all variations of function calls instead of the vertices data *v*. Finally, if no output argument is given then directly change the colors of a patch that is given by the patch handle *p*.

For example:

```
function [] = isofinish (p)
  set (gca, "PlotBoxAspectRatioMode", "manual", ...
            "PlotBoxAspectRatio", [1 1 1]);
  set (p, "FaceColor", "interp");
  ## set (p, "FaceLighting", "flat");
  ## light ("Position", [1 1 5]);  # Available with JHandles
endfunction

N = 15;    # Increase number of vertices in each direction
iso = .4;  # Change isovalue to .1 to display a sphere
lin = linspace (0, 2, N);
[x, y, z] = meshgrid (lin, lin, lin);
c = abs ((x-.5).^2 + (y-.5).^2 + (z-.5).^2);
figure (); # Open another figure window

subplot (2,2,1); view (-38, 20);
[f, v] = isosurface (x, y, z, c, iso);
p = patch ("Faces", f, "Vertices", v, "EdgeColor", "none");
cdat = rand (size (c));       # Compute random patch color data
isocolors (x, y, z, cdat, p); # Directly set colors of patch
isofinish (p);                # Call user function isofinish

subplot (2,2,2); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "EdgeColor", "none");
[r, g, b] = meshgrid (lin, 2-lin, 2-lin);
cdat = isocolors (x, y, z, c, v); # Compute color data vertices
set (p, "FaceVertexCData", cdat); # Set color data manually
isofinish (p);
```

```
subplot (2,2,3); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "EdgeColor", "none");
cdat = isocolors (r, g, b, c, p); # Compute color data patch
set (p, "FaceVertexCData", cdat); # Set color data manually
isofinish (p);

subplot (2,2,4); view (-38, 20);
p = patch ("Faces", f, "Vertices", v, "EdgeColor", "none");
r = g = b = repmat ([1:N] / N, [N, 1, N]); # Black to white
cdat = isocolors (x, y, z, r, g, b, v);
set (p, "FaceVertexCData", cdat);
isofinish (p);
```

See also: [isosurface], page 312, [isonormals], page 313.

`shrinkfaces (`*p*`, `*sf*`)` [Function File]
nfv `= shrinkfaces (`*p*`, `*sf*`)` [Function File]
nfv `= shrinkfaces (`*fv*`, `*sf*`)` [Function File]
nfv `= shrinkfaces (`*f*`, `*v*`, `*sf*`)` [Function File]
`[`*nf*`, `*nv*`] = shrinkfaces (...)` [Function File]

Reduce the size of faces in a patch by the shrink factor *sf*.

The patch object can be specified by a graphics handle (*p*), a patch structure (*fv*) with the fields `"faces"` and `"vertices"`, or as two separate matrices (*f*, *v*) of faces and vertices.

The shrink factor *sf* is a positive number specifying the percentage of the original area the new face will occupy. If no factor is given the default is 0.3 (a reduction to 30% of the original size). A factor greater than 1.0 will result in the expansion of faces.

Given a patch handle as the first input argument and no output parameters, perform the shrinking of the patch faces in place and redraw the patch.

If called with one output argument, return a structure with fields `"faces"`, `"vertices"`, and `"facevertexdata"` containing the data after shrinking. This structure can be used directly as an input argument to the `patch` function.

Caution:: Performing the shrink operation on faces which are not convex can lead to undesirable results.

Example: a triangulated 3/4 circle and the corresponding shrunken version.

```
[phi r] = meshgrid (linspace (0, 1.5*pi, 16), linspace (1, 2, 4));
tri = delaunay (phi(:), r(:));
v = [r(:).*sin(phi(:)) r(:).*cos(phi(:))];
clf ()
p = patch ("Faces", tri, "Vertices", v, "FaceColor", "none");
fv = shrinkfaces (p);
patch (fv)
axis equal
grid on
```

See also: [patch], page 357.

`diffuse` (*sx*, *sy*, *sz*, *lv*) [Function File]

Calculate the diffuse reflection strength of a surface defined by the normal vector elements *sx*, *sy*, *sz*.

The light source location vector *lv* can be given as a 2-element vector [azimuth, elevation] in degrees or as a 3-element vector [x, y, z].

See also: [specular], page 317, [surfl], page 310.

`specular` (*sx*, *sy*, *sz*, *lv*, *vv*) [Function File]
`specular` (*sx*, *sy*, *sz*, *lv*, *vv*, *se*) [Function File]

Calculate the specular reflection strength of a surface defined by the normal vector elements *sx*, *sy*, *sz* using Phong's approximation.

The light source location and viewer location vectors are specified using parameters *lv* and *vv* respectively. The location vectors can given as 2-element vectors [azimuth, elevation] in degrees or as 3-element vectors [x, y, z].

An optional sixth argument specifies the specular exponent (spread) *se*. If not given, *se* defaults to 10.

See also: [diffuse], page 317, [surfl], page 310.

[*xx*, *yy*] = `meshgrid` (*x*, *y*) [Function File]
[*xx*, *yy*, *zz*] = `meshgrid` (*x*, *y*, *z*) [Function File]
[*xx*, *yy*] = `meshgrid` (*x*) [Function File]
[*xx*, *yy*, *zz*] = `meshgrid` (*x*) [Function File]

Given vectors of *x* and *y* coordinates, return matrices *xx* and *yy* corresponding to a full 2-D grid.

The rows of *xx* are copies of *x*, and the columns of *yy* are copies of *y*. If *y* is omitted, then it is assumed to be the same as *x*.

If the optional *z* input is given, or *zz* is requested, then the output will be a full 3-D grid.

`meshgrid` is most frequently used to produce input for a 2-D or 3-D function that will be plotted. The following example creates a surface plot of the "sombrero" function.

```
f = @(x,y) sin (sqrt (x.^2 + y.^2)) ./ sqrt (x.^2 + y.^2);
range = linspace (-8, 8, 41);
[X, Y] = meshgrid (range, range);
Z = f (X, Y);
surf (X, Y, Z);
```

Programming Note: `meshgrid` is restricted to 2-D or 3-D grid generation. The `ndgrid` function will generate 1-D through N-D grids. However, the functions are not completely equivalent. If *x* is a vector of length M and *y* is a vector of length N, then `meshgrid` will produce an output grid which is NxM. `ndgrid` will produce an output which is MxN (transpose) for the same input. Some core functions expect `meshgrid` input and others expect `ndgrid` input. Check the documentation for the function in question to determine the proper input format.

See also: [ndgrid], page 318, [mesh], page 307, [contour], page 285, [surf], page 309.

`[y1, y2, ..., yn] = ndgrid (x1, x2, ..., xn)` [Function File]
`[y1, y2, ..., yn] = ndgrid (x)` [Function File]

Given n vectors *x1*, ..., *x*n, `ndgrid` returns n arrays of dimension n.

The elements of the i-th output argument contains the elements of the vector *x*i repeated over all dimensions different from the i-th dimension. Calling ndgrid with only one input argument *x* is equivalent to calling ndgrid with all n input arguments equal to *x*:

[*y1*, *y2*, ..., *y*n] = ndgrid (x, ..., *x*)

Programming Note: `ndgrid` is very similar to the function `meshgrid` except that the first two dimensions are transposed in comparison to `meshgrid`. Some core functions expect `meshgrid` input and others expect `ndgrid` input. Check the documentation for the function in question to determine the proper input format.

See also: [meshgrid], page 317.

`plot3 (x, y, z)` [Function File]
`plot3 (x, y, z, prop, value, ...)` [Function File]
`plot3 (x, y, z, fmt)` [Function File]
`plot3 (x, cplx)` [Function File]
`plot3 (cplx)` [Function File]
`plot3 (hax, ...)` [Function File]
`h = plot3 (...)` [Function File]

Produce 3-D plots.

Many different combinations of arguments are possible. The simplest form is

```
plot3 (x, y, z)
```

in which the arguments are taken to be the vertices of the points to be plotted in three dimensions. If all arguments are vectors of the same length, then a single continuous line is drawn. If all arguments are matrices, then each column of is treated as a separate line. No attempt is made to transpose the arguments to make the number of rows match.

If only two arguments are given, as

```
plot3 (x, cplx)
```

the real and imaginary parts of the second argument are used as the *y* and *z* coordinates, respectively.

If only one argument is given, as

```
plot3 (cplx)
```

the real and imaginary parts of the argument are used as the *y* and *z* values, and they are plotted versus their index.

Arguments may also be given in groups of three as

```
plot3 (x1, y1, z1, x2, y2, z2, ...)
```

in which each set of three arguments is treated as a separate line or set of lines in three dimensions.

To plot multiple one- or two-argument groups, separate each group with an empty format string, as

```
plot3 (x1, c1, "", c2, "", ...)
```

Multiple property-value pairs may be specified which will affect the line objects drawn by `plot3`. If the *fmt* argument is supplied it will format the line objects in the same manner as `plot`.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created plot.

Example:

```
z = [0:0.05:5];
plot3 (cos (2*pi*z), sin (2*pi*z), z, ";helix;");
plot3 (z, exp (2i*pi*z), ";complex sinusoid;");
```

See also: [ezplot3], page 323, [plot], page 272.

`view (`*azimuth*`,` *elevation*`)` [Function File]
`view ([`*azimuth* *elevation*`])` [Function File]
`view ([`*x y z*`])` [Function File]
`view (`*2*`)` [Function File]
`view (`*3*`)` [Function File]
`view (`*hax*`, ...)` [Function File]
`[`*azimuth*`,` *elevation*`] = view ()` [Function File]

Query or set the viewpoint for the current axes.

The parameters *azimuth* and *elevation* can be given as two arguments or as 2-element vector. The viewpoint can also be specified with Cartesian coordinates *x*, *y*, and *z*.

The call `view (2)` sets the viewpoint to *azimuth* = 0 and *elevation* = 90, which is the default for 2-D graphs.

The call `view (3)` sets the viewpoint to *azimuth* = -37.5 and *elevation* = 30, which is the default for 3-D graphs.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by `gca`.

If no inputs are given, return the current *azimuth* and *elevation*.

`slice (`*x*`,` *y*`,` *z*`,` *v*`,` *sx*`,` *sy*`,` *sz*`)` [Function File]
`slice (`*x*`,` *y*`,` *z*`,` *v*`,` *xi*`,` *yi*`,` *zi*`)` [Function File]
`slice (`*v*`,` *sx*`,` *sy*`,` *sz*`)` [Function File]
`slice (`*v*`,` *xi*`,` *yi*`,` *zi*`)` [Function File]
`slice (...,` *method*`)` [Function File]
`slice (`*hax*`, ...)` [Function File]
h `= slice (...)` [Function File]

Plot slices of 3-D data/scalar fields.

Each element of the 3-dimensional array *v* represents a scalar value at a location given by the parameters *x*, *y*, and *z*. The parameters *x*, *x*, and *z* are either 3-dimensional arrays of the same size as the array *v* in the `"meshgrid"` format or vectors. The parameters *xi*, etc. respect a similar format to *x*, etc., and they represent the points at which the array *vi* is interpolated using interp3. The vectors *sx*, *sy*, and *sz* contain points of orthogonal slices of the respective axes.

If *x*, *y*, *z* are omitted, they are assumed to be `x = 1:size (v, 2)`, `y = 1:size (v, 1)` and `z = 1:size (v, 3)`.

method is one of:

`"nearest"`
: Return the nearest neighbor.

`"linear"`
: Linear interpolation from nearest neighbors.

`"cubic"`
: Cubic interpolation from four nearest neighbors (not implemented yet).

`"spline"`
: Cubic spline interpolation—smooth first and second derivatives throughout the curve.

The default method is `"linear"`.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

Examples:

```
[x, y, z] = meshgrid (linspace (-8, 8, 32));
v = sin (sqrt (x.^2 + y.^2 + z.^2)) ./ (sqrt (x.^2 + y.^2 + z.^2));
slice (x, y, z, v, [], 0, []);

[xi, yi] = meshgrid (linspace (-7, 7));
zi = xi + yi;
slice (x, y, z, v, xi, yi, zi);
```

See also: [interp3], page 660, [surface], page 358, [pcolor], page 296.

`ribbon` (*y*) [Function File]
`ribbon` (*x*, *y*) [Function File]
`ribbon` (*x*, *y*, *width*) [Function File]
`ribbon` (*hax*, ...) [Function File]
h = `ribbon` (...) [Function File]

Draw a ribbon plot for the columns of *y* vs. *x*.

The optional parameter *width* specifies the width of a single ribbon (default is 0.75). If *x* is omitted, a vector containing the row numbers is assumed (`1:rows (Y)`).

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a vector of graphics handles to the surface objects representing each ribbon.

See also: [surface], page 358, [waterfall], page 321.

`shading` (*type*) [Function File]
`shading` (*hax*, *type*) [Function File]

Set the shading of patch or surface graphic objects.

Valid arguments for *type* are

`"flat"`
: Single colored patches with invisible edges.

`"faceted"`
Single colored patches with visible edges.

`"interp"` Color between patch vertices are interpolated and the patch edges are invisible.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

See also: [fill], page 297, [mesh], page 307, [patch], page 357, [pcolor], page 296, [surf], page 309, [surface], page 358, [hidden], page 309.

`scatter3` (*x*, *y*, *z*) [Function File]
`scatter3` (*x*, *y*, *z*, *s*) [Function File]
`scatter3` (*x*, *y*, *z*, *s*, *c*) [Function File]
`scatter3` (. . ., *style*) [Function File]
`scatter3` (. . ., "*filled*") [Function File]
`scatter3` (. . ., *prop*, *val*) [Function File]
`scatter3` (*hax*, . . .) [Function File]
h = `scatter3` (. . .) [Function File]

Draw a 3-D scatter plot.

A marker is plotted at each point defined by the coordinates in the vectors *x*, *y*, and *z*.

The size of the markers is determined by *s*, which can be a scalar or a vector of the same length as *x*, *y*, and *z*. If *s* is not given, or is an empty matrix, then a default value of 8 points is used.

The color of the markers is determined by *c*, which can be a string defining a fixed color; a 3-element vector giving the red, green, and blue components of the color; a vector of the same length as *x* that gives a scaled index into the current colormap; or an Nx3 matrix defining the RGB color of each marker individually.

The marker to use can be changed with the *style* argument, that is a string defining a marker in the same manner as the `plot` command. If no marker is specified it defaults to `"o"` or circles. If the argument `"filled"` is given then the markers are filled.

Additional property/value pairs are passed directly to the underlying patch object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the hggroup object representing the points.

```
[x, y, z] = peaks (20);
scatter3 (x(:), y(:), z(:), [], z(:));
```

See also: [scatter], page 283, [patch], page 357, [plot], page 272.

`waterfall` (*x*, *y*, *z*) [Function File]
`waterfall` (*z*) [Function File]
`waterfall` (. . ., *c*) [Function File]
`waterfall` (. . ., *prop*, *val*, . . .) [Function File]
`waterfall` (*hax*, . . .) [Function File]

`h = waterfall (...)` [Function File]

Plot a 3-D waterfall plot.

A waterfall plot is similar to a `meshz` plot except only mesh lines for the rows of *z* (x-values) are shown.

The wireframe mesh is plotted using rectangles. The vertices of the rectangles [*x*, *y*] are typically the output of `meshgrid`. over a 2-D rectangular region in the x-y plane. *z* determines the height above the plane of each vertex. If only a single *z* matrix is given, then it is plotted over the meshgrid `x = 1:columns (z), y = 1:rows (z)`. Thus, columns of *z* correspond to different *x* values and rows of *z* correspond to different *y* values.

The color of the mesh is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally the color of the mesh can be specified independently of *z* by supplying a color matrix, *c*.

Any property/value pairs are passed directly to the underlying surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

See also: [meshz], page 308, [mesh], page 307, [meshc], page 308, [contour], page 285, [surf], page 309, [surface], page 358, [ribbon], page 320, [meshgrid], page 317, [hidden], page 309, [shading], page 320, [colormap], page 704, [caxis], page 300.

15.2.2.1 Aspect Ratio

For three-dimensional plots the aspect ratio can be set for data with `daspect` and for the plot box with `pbaspect`. See Section 15.2.1.1 [Axis Configuration], page 299, for controlling the x-, y-, and z-limits for plotting.

`data_aspect_ratio = daspect ()` [Function File]
`daspect (data_aspect_ratio)` [Function File]
`daspect (mode)` [Function File]
`data_aspect_ratio_mode = daspect ("mode")` [Function File]
`daspect (hax, ...)` [Function File]

Query or set the data aspect ratio of the current axes.

The aspect ratio is a normalized 3-element vector representing the span of the x, y, and z-axis limits.

`daspect (mode)`

Set the data aspect ratio mode of the current axes. *mode* is either `"auto"` or `"manual"`.

`daspect ("mode")`

Return the data aspect ratio mode of the current axes.

`daspect (hax, ...)`

Operate on the axes in handle *hax* instead of the current axes.

See also: [axis], page 299, [pbaspect], page 323, [xlim], page 301, [ylim], page 301, [zlim], page 301.

plot_box_aspect_ratio = pbaspect ( ) [Function File]
pbaspect (*plot_box_aspect_ratio*) [Function File]
pbaspect (*mode*) [Function File]
plot_box_aspect_ratio_mode = pbaspect ("*mode*") [Function File]
pbaspect (*hax*, . . .) [Function File]

Query or set the plot box aspect ratio of the current axes.

The aspect ratio is a normalized 3-element vector representing the rendered lengths of the x, y, and z axes.

pbaspect(*mode*)

Set the plot box aspect ratio mode of the current axes. *mode* is either "auto" or "manual".

pbaspect ("mode")

Return the plot box aspect ratio mode of the current axes.

pbaspect (*hax*, ...)

Operate on the axes in handle *hax* instead of the current axes.

See also: [axis], page 299, [daspect], page 322, [xlim], page 301, [ylim], page 301, [zlim], page 301.

15.2.2.2 Three-dimensional Function Plotting

ezplot3 (*fx*, *fy*, *fz*) [Function File]
ezplot3 (. . ., *dom*) [Function File]
ezplot3 (. . ., *n*) [Function File]
ezplot3 (*hax*, . . .) [Function File]
h = ezplot3 (. . .) [Function File]

Plot a parametrically defined curve in three dimensions.

fx, *fy*, and *fz* are strings, inline functions, or function handles with one argument defining the function. By default the plot is over the domain 0 <= *t* <= 2*pi with 500 points.

If *dom* is a two element vector, it represents the minimum and maximum values of *t*.

n is a scalar defining the number of points to use in plotting the function.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by gca.

The optional return value *h* is a graphics handle to the created plot.

```
fx = @(t) cos (t);
fy = @(t) sin (t);
fz = @(t) t;
ezplot3 (fx, fy, fz, [0, 10*pi], 100);
```

See also: [plot3], page 318, [ezplot], page 302, [ezmesh], page 323, [ezsurf], page 325.

ezmesh (*f*) [Function File]
ezmesh (*fx*, *fy*, *fz*) [Function File]
ezmesh (. . ., *dom*) [Function File]
ezmesh (. . ., *n*) [Function File]

`ezmesh (..., "circ")` [Function File]
`ezmesh (hax, ...)` [Function File]
`h = ezmesh (...)` [Function File]

Plot the mesh defined by a function.

f is a string, inline function, or function handle with two arguments defining the function. By default the plot is over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

If three functions are passed, then plot the parametrically defined function `[fx (s, t), fy (s, t), fz (s, t)]`.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in each dimension.

If the argument `"circ"` is given, then the function is plotted over a disk centered on the middle of the domain *dom*.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

Example 1: 2-argument function

```
f = @(x,y) sqrt (abs (x .* y)) ./ (1 + x.^2 + y.^2);
ezmesh (f, [-3, 3]);
```

Example 2: parametrically defined function

```
fx = @(s,t) cos (s) .* cos (t);
fy = @(s,t) sin (s) .* cos (t);
fz = @(s,t) sin (t);
ezmesh (fx, fy, fz, [-pi, pi, -pi/2, pi/2], 20);
```

See also: [mesh], page 307, [ezmeshc], page 324, [ezplot], page 302, [ezsurf], page 325, [ezsurfc], page 326, [hidden], page 309.

`ezmeshc (f)` [Function File]
`ezmeshc (fx, fy, fz)` [Function File]
`ezmeshc (..., dom)` [Function File]
`ezmeshc (..., n)` [Function File]
`ezmeshc (..., "circ")` [Function File]
`ezmeshc (hax, ...)` [Function File]
`h = ezmeshc (...)` [Function File]

Plot the mesh and contour lines defined by a function.

f is a string, inline function, or function handle with two arguments defining the function. By default the plot is over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

If three functions are passed, then plot the parametrically defined function `[fx (s, t), fy (s, t), fz (s, t)]`.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in each dimension.

If the argument `"circ"` is given, then the function is plotted over a disk centered on the middle of the domain *dom*.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a 2-element vector with a graphics handle for the created mesh plot and a second handle for the created contour plot.

Example: 2-argument function

```
f = @(x,y) sqrt (abs (x .* y)) ./ (1 + x.^2 + y.^2);
ezmeshc (f, [-3, 3]);
```

See also: [meshc], page 308, [ezmesh], page 323, [ezplot], page 302, [ezsurf], page 325, [ezsurfc], page 326, [hidden], page 309.

`ezsurf (`*f*`)` [Function File]
`ezsurf (`*fx*`, `*fy*`, `*fz*`)` [Function File]
`ezsurf (`. . . , *dom*`)` [Function File]
`ezsurf (`. . . , *n*`)` [Function File]
`ezsurf (`. . . , `"`*circ*`"`)` [Function File]
`ezsurf (`*hax*`, `. . .`)` [Function File]
h `= ezsurf (`. . .`)` [Function File]

Plot the surface defined by a function.

f is a string, inline function, or function handle with two arguments defining the function. By default the plot is over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

If three functions are passed, then plot the parametrically defined function `[fx (s, t), fy (s, t), fz (s, t)]`.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in each dimension.

If the argument `"circ"` is given, then the function is plotted over a disk centered on the middle of the domain *dom*.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

Example 1: 2-argument function

```
f = @(x,y) sqrt (abs (x .* y)) ./ (1 + x.^2 + y.^2);
ezsurf (f, [-3, 3]);
```

Example 2: parametrically defined function

```
fx = @(s,t) cos (s) .* cos (t);
fy = @(s,t) sin (s) .* cos (t);
fz = @(s,t) sin (t);
ezsurf (fx, fy, fz, [-pi, pi, -pi/2, pi/2], 20);
```

See also: [surf], page 309, [ezsurfc], page 326, [ezplot], page 302, [ezmesh], page 323, [ezmeshc], page 324, [shading], page 320.

`ezsurfc (`*f*`)` [Function File]
`ezsurfc (`*fx*, *fy*, *fz*`)` [Function File]
`ezsurfc (...,` *dom*`)` [Function File]
`ezsurfc (...,` *n*`)` [Function File]
`ezsurfc (..., "`*circ*`")` [Function File]
`ezsurfc (`*hax*`, ...)` [Function File]
h `= ezsurfc (...)` [Function File]

Plot the surface and contour lines defined by a function.

f is a string, inline function, or function handle with two arguments defining the function. By default the plot is over the meshed domain `-2*pi <= x | y <= 2*pi` with 60 points in each dimension.

If three functions are passed, then plot the parametrically defined function `[`*fx*` (`*s*`, `*t*`), `*fy*` (`*s*`, `*t*`), `*fz*` (`*s*`, `*t*`)]`.

If *dom* is a two element vector, it represents the minimum and maximum values of both *x* and *y*. If *dom* is a four element vector, then the minimum and maximum values are `[xmin xmax ymin ymax]`.

n is a scalar defining the number of points to use in each dimension.

If the argument `"circ"` is given, then the function is plotted over a disk centered on the middle of the domain *dom*.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a 2-element vector with a graphics handle for the created surface plot and a second handle for the created contour plot.

Example:

```
f = @(x,y) sqrt (abs (x .* y)) ./ (1 + x.^2 + y.^2);
ezsurfc (f, [-3, 3]);
```

See also: [surfc], page 310, [ezsurf], page 325, [ezplot], page 302, [ezmesh], page 323, [ezmeshc], page 324, [shading], page 320.

15.2.2.3 Three-dimensional Geometric Shapes

`cylinder` [Command]
`cylinder (`*r*`)` [Function File]
`cylinder (`*r*, *n*`)` [Function File]
`cylinder (`*hax*`, ...)` [Function File]
`[`*x*`, `*y*`, `*z*`] = cylinder (...)` [Function File]

Plot a 3-D unit cylinder.

The optional input *r* is a vector specifying the radius along the unit z-axis. The default is [1 1] indicating radius 1 at `Z == 0` and at `Z == 1`.

The optional input *n* determines the number of faces around the circumference of the cylinder. The default value is 20.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

If outputs are requested `cylinder` returns three matrices in `meshgrid` format, such that `surf (x, y, z)` generates a unit cylinder.

Example:

```
[x, y, z] = cylinder (10:-1:0, 50);
surf (x, y, z);
title ("a cone");
```

See also: [ellipsoid], page 327, [rectangle], page 304, [sphere], page 327.

`sphere ()` [Function File]
`sphere (n)` [Function File]
`sphere (hax, ...)` [Function File]
`[x, y, z] = sphere (...)` [Function File]

Plot a 3-D unit sphere.

The optional input *n* determines the number of faces around the circumference of the sphere. The default value is 20.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

If outputs are requested `sphere` returns three matrices in `meshgrid` format such that `surf (x, y, z)` generates a unit sphere.

Example:

```
[x, y, z] = sphere (40);
surf (3*x, 3*y, 3*z);
axis equal;
title ("sphere of radius 3");
```

See also: [cylinder], page 326, [ellipsoid], page 327, [rectangle], page 304.

`ellipsoid (xc, yc, zc, xr, yr, zr, n)` [Function File]
`ellipsoid (..., n)` [Function File]
`ellipsoid (hax, ...)` [Function File]
`[x, y, z] = ellipsoid (...)` [Function File]

Plot a 3-D ellipsoid.

The inputs *xc*, *yc*, *zc* specify the center of the ellipsoid. The inputs *xr*, *yr*, *zr* specify the semi-major axis lengths.

The optional input *n* determines the number of faces around the circumference of the cylinder. The default value is 20.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

If outputs are requested `ellipsoid` returns three matrices in `meshgrid` format, such that `surf (x, y, z)` generates the ellipsoid.

See also: [cylinder], page 326, [rectangle], page 304, [sphere], page 327.

15.2.3 Plot Annotations

You can add titles, axis labels, legends, and arbitrary text to an existing plot. For example:

```
x = -10:0.1:10;
plot (x, sin (x));
title ("sin(x) for x = -10:0.1:10");
xlabel ("x");
ylabel ("sin (x)");
text (pi, 0.7, "arbitrary text");
legend ("sin (x)");
```

The functions `grid` and `box` may also be used to add grid and border lines to the plot. By default, the grid is off and the border lines are on.

Finally, arrows, text and rectangular or elliptic boxes can be added to highlight parts of a plot using the `annotation` function. Those objects are drawn in an invisible axes, on top of every other axes.

`title` (*string*) [Function File]
`title` (*string*, *prop*, *val*, ...) [Function File]
`title` (*hax*, ...) [Function File]
h = `title` (...) [Function File]

Specify the string used as a title for the current axis.

An optional list of *property*/*value* pairs can be used to change the appearance of the created title text object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created text object.

See also: [xlabel], page 330, [ylabel], page 330, [zlabel], page 330, [text], page 330.

`legend` (*str1*, *str2*, ...) [Function File]
`legend` (*matstr*) [Function File]
`legend` (*cellstr*) [Function File]
`legend` (..., "*location*", *pos*) [Function File]
`legend` (..., "*orientation*", *orient*) [Function File]
`legend` (*hax*, ...) [Function File]
`legend` (*hobjs*, ...) [Function File]
`legend` (*hax*, *hobjs*, ...) [Function File]
`legend` ("*option*") [Function File]
[*hleg*, *hleg_obj*, *hplot*, *labels*] = `legend` (...) [Function File]

Display a legend for the current axes using the specified strings as labels.

Legend entries may be specified as individual character string arguments, a character array, or a cell array of character strings.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`. If the handles, *hobjs*, are not specified then the legend's strings will be associated with the axes' descendants. `legend` works on line graphs, bar graphs, etc. A plot must exist before legend is called.

The optional parameter *pos* specifies the location of the legend as follows:

pos	**location of the legend**
north	center top
south	center bottom
east	right center
west	left center
northeast	right top (default)
northwest	left top
southeast	right bottom
southwest	left bottom
outside	can be appended to any location string

The optional parameter *orient* determines if the key elements are placed vertically or horizontally. The allowed values are `"vertical"` (default) or `"horizontal"`.

The following customizations are available using *option*:

`"show"` Show legend on the plot

`"hide"` Hide legend on the plot

`"toggle"` Toggles between `"hide"` and `"show"`

`"boxon"` Show a box around legend (default)

`"boxoff"` Hide the box around legend

`"right"` Place label text to the right of the keys (default)

`"left"` Place label text to the left of the keys

`"off"` Delete the legend object

The optional output values are

hleg The graphics handle of the legend object.

hleg_obj Graphics handles to the text and line objects which make up the legend.

hplot Graphics handles to the plot objects which were used in making the legend.

labels A cell array of strings of the labels in the legend.

The legend label text is either provided in the call to `legend` or is taken from the DisplayName property of graphics objects. If no labels or DisplayNames are available, then the label text is simply `"data1"`, `"data2"`, ..., `"dataN"`.

Implementation Note: A legend is implemented as an additional axes object of the current figure with the `"tag"` set to `"legend"`. Properties of the legend object may be manipulated directly by using `set`.

`text (x, y, string)` [Function File]
`text (x, y, z, string)` [Function File]
`text (..., prop, val, ...)` [Function File]
`h = text (...)` [Function File]

Create a text object with text *string* at position *x*, *y*, (*z*) on the current axes.

Multiple locations can be specified if *x*, *y*, (*z*) are vectors. Multiple strings can be specified with a character matrix or a cell array of strings.

Optional property/value pairs may be used to control the appearance of the text.

The optional return value *h* is a vector of graphics handles to the created text objects.

See also: [gtext], page 352, [title], page 328, [xlabel], page 330, [ylabel], page 330, [zlabel], page 330.

See Section 15.3.3.5 [Text Properties], page 376, for the properties that you can set.

`xlabel (string)` [Function File]
`xlabel (string, property, val, ...)` [Function File]
`xlabel (hax, ...)` [Function File]
`h = xlabel (...)` [Function File]

Specify the string used to label the x-axis of the current axis.

An optional list of *property*/*value* pairs can be used to change the properties of the created text label.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created text object.

See also: [ylabel], page 330, [zlabel], page 330, [datetick], page 760, [title], page 328, [text], page 330.

`clabel (c, h)` [Function File]
`clabel (c, h, v)` [Function File]
`clabel (c, h, "manual")` [Function File]
`clabel (c)` [Function File]
`clabel (..., prop, val, ...)` [Function File]
`h = clabel (...)` [Function File]

Add labels to the contours of a contour plot.

The contour levels are specified by the contour matrix *c* which is returned by `contour`, `contourc`, `contourf`, and `contour3`. Contour labels are rotated to match the local line orientation and centered on the line. The position of labels along the contour line is chosen randomly.

If the argument *h* is a handle to a contour group object, then label this plot rather than the one in the current axes returned by `gca`.

By default, all contours are labeled. However, the contours to label can be specified by the vector *v*. If the `"manual"` argument is given then the contours to label can be selected with the mouse.

Additional property/value pairs that are valid properties of text objects can be given and are passed to the underlying text objects. Moreover, the contour group property "LabelSpacing" is available which determines the spacing between labels on a contour to be specified. The default is 144 points, or 2 inches.

The optional return value *h* is a vector of graphics handles to the text objects representing each label. The "userdata" property of the text objects contains the numerical value of the contour label.

An example of the use of `clabel` is

```
[c, h] = contour (peaks (), -4 : 6);
clabel (c, h, -4:2:6, "fontsize", 12);
```

See also: [contour], page 285, [contourf], page 286, [contour3], page 287, [meshc], page 308, [surfc], page 310, [text], page 330.

`box` [Command]
`box` *on* [Command]
`box` *off* [Command]
`box` (*hax*, . . .) [Function File]

Control display of the axis border.

The argument may be either "on" or "off". If it is omitted, the current box state is toggled.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by `gca`.

See also: [axis], page 299, [grid], page 331.

`grid` [Command]
`grid` *on* [Command]
`grid` *off* [Command]
`grid` *minor* [Command]
`grid` *minor on* [Command]
`grid` *minor off* [Command]
`grid` (*hax*, . . .) [Function File]

Control the display of plot grid lines.

The function state input may be either "on" or "off". If it is omitted, the current grid state is toggled.

When the first argument is "minor" all subsequent commands modify the minor grid rather than the major grid.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by `gca`.

To control the grid lines for an individual axis use the `set` function. For example:

```
set (gca, "ygrid", "on");
```

See also: [axis], page 299, [box], page 331.

`colorbar` [Command]
`colorbar` (*loc*) [Function File]

`colorbar` (*delete_option*) [Function File]
`colorbar` (*hcb*, ...) [Function File]
`colorbar` (*hax*, ...) [Function File]
`colorbar` (..., "*peer*", *hax*, ...) [Function File]
`colorbar` (..., "*location*", *loc*, ...) [Function File]
`colorbar` (..., *prop*, *val*, ...) [Function File]
h = `colorbar` (...) [Function File]

Add a colorbar to the current axes.

A colorbar displays the current colormap along with numerical rulings so that the color scale can be interpreted.

The optional input *loc* determines the location of the colorbar. Valid values for *loc* are

`"EastOutside"`
: Place the colorbar outside the plot to the right. This is the default.

`"East"`
: Place the colorbar inside the plot to the right.

`"WestOutside"`
: Place the colorbar outside the plot to the left.

`"West"`
: Place the colorbar inside the plot to the left.

`"NorthOutside"`
: Place the colorbar above the plot.

`"North"`
: Place the colorbar at the top of the plot.

`"SouthOutside"`
: Place the colorbar under the plot.

`"South"`
: Place the colorbar at the bottom of the plot.

To remove a colorbar from a plot use any one of the following keywords for the *delete_option*: `"delete"`, `"hide"`, `"off"`.

If the argument `"peer"` is given, then the following argument is treated as the axes handle in which to add the colorbar. Alternatively, If the first argument *hax* is an axes handle, then the colorbar is added to this axis, rather than the current axes returned by `gca`.

If the first argument *hcb* is a handle to a colorbar object, then operate on this colorbar directly.

Additional property/value pairs are passed directly to the underlying axes object.

The optional return value *h* is a graphics handle to the created colorbar object.

Implementation Note: A colorbar is created as an additional axes to the current figure with the `"tag"` property set to `"colorbar"`. The created axes object has the extra property `"location"` which controls the positioning of the colorbar.

See also: [colormap], page 704.

`annotation` (*type*) [Function File]
`annotation` ("*line*", *x*, *y*) [Function File]

`annotation ("`*arrow*`", `*x*`, `*y*`)` [Function File]
`annotation ("`*doublearrow*`", `*x*`, `*y*`)` [Function File]
`annotation ("`*textarrow*`", `*x*`, `*y*`)` [Function File]
`annotation ("`*textbox*`", `*pos*`)` [Function File]
`annotation ("`*rectangle*`", `*pos*`)` [Function File]
`annotation ("`*ellipse*`", `*pos*`)` [Function File]
`annotation (`*. . .*`, `*prop*`, `*val*`)` [Function File]
`annotation (`*hf*`, `*. . .*`)` [Function File]
h `= annotation (`*. . .*`)` [Function File]

Draw annotations to emphasize parts of a figure.

You may build a default annotation by specifying only the *type* of the annotation.

Otherwise you can select the type of annotation and then set its position using either *x* and *y* coordinates for line-based annotations or a position vector *pos* for others. In either case, coordinates are interpreted using the `"units"` property of the annotation object. The default is `"normalized"`, which means the lower left hand corner of the figure has coordinates '`[0 0]`' and the upper right hand corner '`[1 1]`'.

If the first argument *hf* is a figure handle, then plot into this figure, rather than the current figure returned by `gcf`.

Further arguments can be provided in the form of *prop*/*val* pairs to customize the annotation appearance.

The optional return value *h* is a graphics handle to the created annotation object. This can be used with the `set` function to customize an existing annotation object.

All annotation objects share two properties:

- `"units"`: the units in which coordinates are interpreted.
  Its value may be one of `"centimeters" | "characters" | "inches" | "{normalized}" | "pixels" | "points"`.
- `"position"`: a four-element vector [x0 y0 width height].
  The vector specifies the coordinates (x0,y0) of the origin of the annotation object, its width, and its height. The width and height may be negative, depending on the orientation of the object.

Valid annotation types and their specific properties are described below:

`"line"` Constructs a line. *x* and *y* must be two-element vectors specifying the x and y coordinates of the two ends of the line.

The line can be customized using `"linewidth"`, `"linestyle"`, and `"color"` properties the same way as for `line` objects.

`"arrow"` Construct an arrow. The second point in vectors *x* and *y* specifies the arrowhead coordinates.

Besides line properties, the arrowhead can be customized using `"headlength"`, `"headwidth"`, and `"headstyle"` properties. Supported values for `"headstyle"` property are: [`"diamond" | "ellipse" | "plain" | "rectangle" | "vback1" | "{vback2}" | "vback3"`]

`"doublearrow"`

Construct a double arrow. Vectors *x* and *y* specify the arrowhead coordinates.

The line and the arrowhead can be customized as for arrow annotations, but some property names are duplicated: "head1length"/"head2length", "head1width"/"head2width", etc. The index 1 marks the properties of the arrowhead at the first point in *x* and *y* coordinates.

"textarrow"

Construct an arrow with a text label at the opposite end from the arrowhead.

Use the "string" property to change the text string. The line and the arrowhead can be customized as for arrow annotations, and the text can be customized using the same properties as `text` graphics objects. Note, however, that some text property names are prefixed with "text" to distinguish them from arrow properties: "textbackgroundcolor", "textcolor", "textedgecolor", "textlinewidth", "textmargin", "textrotation".

"textbox"

Construct a box with text inside. *pos* specifies the "position" property of the annotation.

Use the "string" property to change the text string. You may use "backgroundcolor", "edgecolor", "linestyle", and "linewidth" properties to customize the box background color and edge appearance. A limited set of `text` objects properties are also available; Besides "font..." properties, you may also use "horizontalalignment" and "verticalalignment" to position the text inside the box.

Finally, the "fitboxtotext" property controls the actual extent of the box. If "on" (the default) the box limits are fitted to the text extent.

"rectangle"

Construct a rectangle. *pos* specifies the "position" property of the annotation.

You may use "facecolor", "color", "linestyle", and "linewidth" properties to customize the rectangle background color and edge appearance.

"ellipse"

Construct an ellipse. *pos* specifies the "position" property of the annotation.

See "rectangle" annotations for customization.

See also: [xlabel], page 330, [ylabel], page 330, [zlabel], page 330, [title], page 328, [text], page 330, [gtext], page 352, [legend], page 328, [colorbar], page 331.

15.2.4 Multiple Plots on One Page

Octave can display more than one plot in a single figure. The simplest way to do this is to use the `subplot` function to divide the plot area into a series of subplot windows that are indexed by an integer. For example,

```
subplot (2, 1, 1)
fplot (@sin, [-10, 10]);
subplot (2, 1, 2)
fplot (@cos, [-10, 10]);
```

creates a figure with two separate axes, one displaying a sine wave and the other a cosine wave. The first call to subplot divides the figure into two plotting areas (two rows and one column) and makes the first plot area active. The grid of plot areas created by `subplot` is numbered in column-major order (top to bottom, left to right).

`subplot` (*rows*, *cols*, *index*) [Function File]
`subplot` (*rcn*) [Function File]
`subplot` (*hax*) [Function File]
`subplot` (. . ., "*align*") [Function File]
`subplot` (. . ., "*replace*") [Function File]
`subplot` (. . ., "*position*", *pos*) [Function File]
`subplot` (. . ., *prop*, *val*, . . .) [Function File]
hax = `subplot` (. . .) [Function File]

Set up a plot grid with *rows* by *cols* subwindows and set the current axes for plotting (`gca`) to the location given by *index*.

If only one numeric argument is supplied, then it must be a three digit value specifying the number of rows in digit 1, the number of columns in digit 2, and the plot index in digit 3.

The plot index runs row-wise; First, all columns in a row are numbered and then the next row is filled.

For example, a plot with 2x3 grid will have plot indices running as follows:

1	2	3
4	5	6

index may also be a vector. In this case, the new axis will enclose the grid locations specified. The first demo illustrates this:

```
demo ("subplot", 1)
```

The index of the subplot to make active may also be specified by its axes handle, *hax*, returned from a previous `subplot` command.

If the option `"align"` is given then the plot boxes of the subwindows will align, but this may leave no room for axis tick marks or labels.

If the option `"replace"` is given then the subplot axis will be reset, rather than just switching the current axis for plotting to the requested subplot.

The `"position"` property can be used to exactly position the subplot axes within the current figure. The option *pos* is a 4-element vector [x, y, width, height] that determines the location and size of the axes. The values in *pos* are normalized in the range [0,1].

Any property/value pairs are passed directly to the underlying axes object.

If the output *hax* is requested, subplot returns the axis handle for the subplot. This is useful for modifying the properties of a subplot using `set`.

See also: [axes], page 357, [plot], page 272, [gca], page 360, [set], page 361.

15.2.5 Multiple Plot Windows

You can open multiple plot windows using the `figure` function. For example,

```
figure (1);
fplot (@sin, [-10, 10]);
figure (2);
fplot (@cos, [-10, 10]);
```

creates two figures, with the first displaying a sine wave and the second a cosine wave. Figure numbers must be positive integers.

`figure` [Command]
`figure` *n* [Command]
`figure` (*n*) [Function File]
`figure` (..., "*property*", *value*, ...) [Function File]
h = `figure` (...) [Function File]

Create a new figure window for plotting.

If no arguments are specified, a new figure with the next available number is created.

If called with an integer *n*, and no such numbered figure exists, then a new figure with the specified number is created. If the figure already exists then it is made visible and becomes the current figure for plotting.

Multiple property-value pairs may be specified for the figure object, but they must appear in pairs.

The optional return value *h* is a graphics handle to the created figure object.

See also: [axes], page 357, [gcf], page 359, [clf], page 340, [close], page 341.

15.2.6 Manipulation of Plot Objects

`pan` [Command]
`pan` *on* [Command]
`pan` *off* [Command]
`pan` *xon* [Command]
`pan` *yon* [Command]
`pan` (*hfig*, *option*) [Function File]

Control the interactive panning mode of a figure in the GUI.

Given the option `"on"` or `"off"`, set the interactive pan mode on or off.

With no arguments, toggle the current pan mode on or off.

Given the option `"xon"` or `"yon"`, enable pan mode for the x or y axis only.

If the first argument *hfig* is a figure, then operate on the given figure rather than the current figure as returned by `gcf`.

See also: [rotate3d], page 337, [zoom], page 337.

`rotate` (*h*, *dir*, *alpha*) [Function File]
`rotate` (. . . , *origin*) [Function File]

Rotate the plot object *h* through *alpha* degrees around the line with direction *dir* and origin *origin*.

The default value of *origin* is the center of the axes object that is the parent of *h*.

If *h* is a vector of handles, they must all have the same parent axes object.

Graphics objects that may be rotated are lines, surfaces, patches, and images.

`rotate3d` [Command]
`rotate3d` *on* [Command]
`rotate3d` *off* [Command]
`rotate3d` (*hfig*, *option*) [Function File]

Control the interactive 3-D rotation mode of a figure in the GUI.

Given the option `"on"` or `"off"`, set the interactive rotate mode on or off.

With no arguments, toggle the current rotate mode on or off.

If the first argument *hfig* is a figure, then operate on the given figure rather than the current figure as returned by `gcf`.

See also: [pan], page 336, [zoom], page 337.

`zoom` [Command]
`zoom` (*factor*) [Command]
`zoom` *on* [Command]
`zoom` *off* [Command]
`zoom` *xon* [Command]
`zoom` *yon* [Command]
`zoom` *out* [Command]
`zoom` *reset* [Command]
`zoom` (*hfig*, *option*) [Command]

Zoom the current axes object or control the interactive zoom mode of a figure in the GUI.

Given a numeric argument greater than zero, zoom by the given factor. If the zoom factor is greater than one, zoom in on the plot. If the factor is less than one, zoom out. If the zoom factor is a two- or three-element vector, then the elements specify the zoom factors for the x, y, and z axes respectively.

Given the option `"on"` or `"off"`, set the interactive zoom mode on or off.

With no arguments, toggle the current zoom mode on or off.

Given the option `"xon"` or `"yon"`, enable zoom mode for the x or y-axis only.

Given the option `"out"`, zoom to the initial zoom setting.

Given the option `"reset"`, store the current zoom setting so that `zoom out` will return to this zoom level.

If the first argument *hfig* is a figure, then operate on the given figure rather than the current figure as returned by `gcf`.

See also: [pan], page 336, [rotate3d], page 337.

15.2.7 Manipulation of Plot Windows

By default, Octave refreshes the plot window when a prompt is printed, or when waiting for input. The `drawnow` function is used to cause a plot window to be updated.

`drawnow ()` [Built-in Function]
`drawnow ("`*expose*`")` [Built-in Function]
`drawnow (`*term*, *file*, *mono*, *debug_file*`)` [Built-in Function]

Update figure windows and their children.

The event queue is flushed and any callbacks generated are executed.

With the optional argument `"expose"`, only graphic objects are updated and no other events or callbacks are processed.

The third calling form of `drawnow` is for debugging and is undocumented.

See also: [refresh], page 338.

Only figures that are modified will be updated. The `refresh` function can also be used to cause an update of the current figure, even if it is not modified.

`refresh ()` [Function File]
`refresh (`*h*`)` [Function File]

Refresh a figure, forcing it to be redrawn.

When called without an argument the current figure is redrawn. Otherwise, the figure with graphic handle *h* is redrawn.

See also: [drawnow], page 338.

Normally, high-level plot functions like `plot` or `mesh` call `newplot` to initialize the state of the current axes so that the next plot is drawn in a blank window with default property settings. To have two plots superimposed over one another, use the `hold` function. For example,

```
hold on;
x = -10:0.1:10;
plot (x, sin (x));
plot (x, cos (x));
hold off;
```

displays sine and cosine waves on the same axes. If the hold state is off, consecutive plotting commands like this will only display the last plot.

`newplot ()` [Function File]
`newplot (`*hfig*`)` [Function File]
`newplot (`*hax*`)` [Function File]
hax `= newplot (...)` [Function File]

Prepare graphics engine to produce a new plot.

This function is called at the beginning of all high-level plotting functions. It is not normally required in user programs. `newplot` queries the `"NextPlot"` field of the current figure and axis to determine what to do.

Figure NextPlot	**Action**
`"new"`	Create a new figure and make it the current figure.
`"add"` (default)	Add new graphic objects to the current figure.
`"replacechildren"`	Delete child objects whose HandleVisibility is set to `"on"`. Set NextPlot property to `"add"`. This typically clears a figure, but leaves in place hidden objects such as menubars. This is equivalent to `clf`.
`"replace"`	Delete all child objects of the figure and reset all figure properties to their defaults. However, the following four properties are not reset: Position, Units, PaperPosition, PaperUnits. This is equivalent to `clf reset`.

Axis NextPlot	**Action**
`"add"`	Add new graphic objects to the current axes. This is equivalent to `hold on`.
`"replacechildren"`	Delete child objects whose HandleVisibility is set to `"on"`, but leave axis properties unmodified. This typically clears a plot, but preserves special settings such as log scaling for axes. This is equivalent to `cla`.
`"replace"` (default)	Delete all child objects of the axis and reset all axis properties to their defaults. However, the following properties are not reset: Position, Units. This is equivalent to `cla reset`.

If the optional input *hfig* or *hax* is given then prepare the specified figure or axes rather than the current figure and axes.

The optional return value *hax* is a graphics handle to the created axes object (not figure).

Caution: Calling `newplot` may change the current figure and current axis.

`hold` [Command]
`hold` *on* [Command]
`hold` *off* [Command]
`hold` *all* [Command]
`hold (`*hax*`, ...)` [Function File]

Toggle or set the `"hold"` state of the plotting engine which determines whether new graphic objects are added to the plot or replace the existing objects.

`hold on` Retain plot data and settings so that subsequent plot commands are displayed on a single graph.

`hold all` Retain plot line color, line style, data, and settings so that subsequent plot commands are displayed on a single graph with the next line color and style.

`hold off` Restore default graphics settings which clear the graph and reset axis properties before each new plot command. (default).

`hold` Toggle the current hold state.

When given the additional argument *hax*, the hold state is modified for this axis rather than the current axes returned by `gca`.

To query the current hold state use the `ishold` function.

See also: [ishold], page 340, [cla], page 340, [clf], page 340, [newplot], page 338.

`ishold` [Command]
`ishold (`*hax*`)` [Function File]
`ishold (`*hfig*`)` [Function File]

Return true if the next plot will be added to the current plot, or false if the plot device will be cleared before drawing the next plot.

If the first argument is an axes handle *hax* or figure handle *hfig* then operate on this plot rather than the current one.

See also: [hold], page 339, [newplot], page 338.

To clear the current figure, call the `clf` function. To clear the current axis, call the `cla` function. To bring the current figure to the top of the window stack, call the `shg` function. To delete a graphics object, call `delete` on its index. To close the figure window, call the `close` function.

`clf` [Command]
`clf` *reset* [Command]
`clf (`*hfig*`)` [Function File]
`clf (`*hfig*`, "`*reset*`")` [Function File]
h `= clf (...)` [Function File]

Clear the current figure window.

`clf` operates by deleting child graphics objects with visible handles (HandleVisibility = `"on"`).

If the optional argument `"reset"` is specified, delete all child objects including those with hidden handles and reset all figure properties to their defaults. However, the following properties are not reset: Position, Units, PaperPosition, PaperUnits.

If the first argument *hfig* is a figure handle, then operate on this figure rather than the current figure returned by `gcf`.

The optional return value *h* is the graphics handle of the figure window that was cleared.

See also: [cla], page 340, [close], page 341, [delete], page 341, [reset], page 386.

`cla` [Command]
`cla` *reset* [Command]
`cla (`*hax*`)` [Function File]
`cla (`*hax*`, "`*reset*`")` [Function File]

Clear the current axes.

`cla` operates by deleting child graphic objects with visible handles (HandleVisibility = `"on"`).

If the optional argument `"reset"` is specified, delete all child objects including those with hidden handles and reset all axis properties to their defaults. However, the following properties are not reset: Position, Units.

If the first argument *hax* is an axes handle, then operate on this axis rather than the current axes returned by `gca`.

See also: [clf], page 340, [delete], page 341, [reset], page 386.

`shg` [Command]

Show the graph window.

Currently, this is the same as executing `drawnow`.

See also: [drawnow], page 338, [figure], page 336.

`delete (`*`file`*`)` [Function File]
`delete (`*`file1`*`, `*`file2`*`, ...)` [Function File]
`delete (`*`handle`*`)` [Function File]

Delete the named file or graphics handle.

file may contain globbing patterns such as '`*`'. Multiple files to be deleted may be specified in the same function call.

handle may be a scalar or vector of graphic handles to delete.

Programming Note: Deleting graphics objects is the proper way to remove features from a plot without clearing the entire figure.

See also: [clf], page 340, [cla], page 340, [unlink], page 761, [rmdir], page 762.

`close` [Command]
`close (`*`h`*`)` [Command]
`close `*`h`* [Command]
`close` *all* [Command]
`close` *all hidden* [Command]
`close` *all force* [Command]

Close figure window(s).

When called with no arguments, close the current figure. This is equivalent to `close (gcf)`. If the input *h* is a graphic handle, or vector of graphics handles, then close each figure in *h*.

If the argument `"all"` is given then all figures with visible handles (HandleVisibility = `"on"`) are closed.

If the argument `"all hidden"` is given then all figures, including hidden ones, are closed.

If the argument `"all force"` is given then all figures are closed even when `"closerequestfcn"` has been altered to prevent closing the window.

Implementation Note: `close` operates by calling the function specified by the `"closerequestfcn"` property for each figure. By default, the function `closereq` is used. It is possible that the function invoked will delay or abort removing the figure.

To remove a figure without executing any callback functions use `delete`. When writing a callback function to close a window do not use `close` to avoid recursion.

See also: [closereq], page 342, [delete], page 341.

`closereq` () [Function File]

Close the current figure and delete all graphics objects associated with it.

By default, the `"closerequestfcn"` property of a new plot figure points to this function.

See also: [close], page 341, [delete], page 341.

15.2.8 Use of the `interpreter` Property

All text objects—such as titles, labels, legends, and text—include the property `"interpreter"` that determines the manner in which special control sequences in the text are rendered.

The interpreter property can take three values: `"none"`, `"tex"`, `"latex"`. If the interpreter is set to `"none"` then no special rendering occurs—the displayed text is a verbatim copy of the specified text. Currently, the `"latex"` interpreter is not implemented and is equivalent to `"none"`.

The `"tex"` option implements a subset of TEX functionality when rendering text. This allows the insertion of special glyphs such as Greek characters or mathematical symbols. Special characters are inserted by using a backslash (\) character followed by a code, as shown in Table 15.1.

Note that for on-screen display the interpreter property is honored by all graphics toolkits. However for printing, **only** the `"gnuplot"` toolkit renders TEX instructions.

Besides special glyphs, the formatting of the text can be changed within the string by using the codes

\bf	Bold font
\it	Italic font
\sl	Oblique Font
\rm	Normal font

These codes may be used in conjunction with the { and } characters to limit the change to a part of the string. For example,

```
xlabel ('{\bf H} = a {\bf V}')
```

where the character `'a'` will not appear in bold font. Note that to avoid having Octave interpret the backslash character in the strings, the strings themselves should be in single quotes.

It is also possible to change the fontname and size within the text

\fontname{*fontname*}	Specify the font to use
\fontsize{*size*}	Specify the size of the font to use

The color of the text may also be changed inline using either a string (e.g., `"red"`) or numerically with a Red-Green-Blue (RGB) specification (.e.g., [1 0 0], also red).

\color{*color*}	Specify the color as a string
\color[rgb]{R G B}	Specify the color numerically

Finally, superscripting and subscripting can be controlled with the `'^'` and `'_'` characters. If the `'^'` or `'_'` is followed by a { character, then all of the block surrounded by the { } pair is superscripted or subscripted. Without the { } pair, only the character immediately following the `'^'` or `'_'` is changed.

Greek Lowercase Letters

Code	Sym	Code	Sym	Code	Sym
\alpha	α	\beta	β	\gamma	γ
\delta	δ	\epsilon	ϵ	\zeta	ζ
\eta	η	\theta	θ	\vartheta	ϑ
\iota	ι	\kappa	κ	\lambda	λ
\mu	μ	\nu	ν	\xi	ξ
\o	o	\pi	π	\varpi	ϖ
\rho	ρ	\sigma	σ	\varsigma	ς
\tau	τ	\upsilon	υ	\phi	ϕ
\chi	χ	\psi	ψ	\omega	ω

Greek Uppercase Letters

Code	Sym	Code	Sym	Code	Sym
\Gamma	Γ	\Delta	Δ	\Theta	Θ
\Lambda	Λ	\Xi	Ξ	\Pi	Π
\Sigma	Σ	\Upsilon	Υ	\Phi	Φ
\Psi	Ψ	\Omega	Ω		

Misc Symbols Type Ord

Code	Sym	Code	Sym	Code	Sym
\aleph	$\aleph$	\wp	$\wp$	\Re	$\Re$
\Im	$\Im$	\partial	∂	\infty	∞
\prime	$\prime$	\nabla	∇	\surd	$\surd$
\angle	$\angle$	\forall	$\forall$	\exists	$\exists$
\neg	$\neg$	\clubsuit	$\clubsuit$	\diamondsuit	$\diamondsuit$
\heartsuit	$\heartsuit$	\spadesuit	$\spadesuit$		

"Large" Operators

Code	Sym	Code	Sym	Code	Sym
\int	$\int$				

Binary operators

Code	Sym	Code	Sym	Code	Sym
\pm	$\pm$	\cdot	$\cdot$	\times	$\times$
\ast	$\ast$	\circ	$\circ$	\bullet	$\bullet$
\div	$\div$	\cap	$\cap$	\cup	$\cup$
\vee	$\vee$	\wedge	$\wedge$	\oplus	$\oplus$
\otimes	$\otimes$	\oslash	$\oslash$		

Table 15.1: Available special characters in TeX mode

Relations

Code	Sym	Code	Sym	Code	Sym
\leq	$\leq$	\subset	$\subset$	\subseteq	$\subseteq$
\in	$\in$	\geq	$\geq$	\supset	$\supset$
\supseteq	$\supseteq$	\ni	$\ni$	\mid	$\mid$
\equiv	$\equiv$	\sim	$\sim$	\approx	$\approx$
\cong	$\cong$	\propto	$\propto$	\perp	$\perp$

Arrows

Code	Sym	Code	Sym	Code	Sym
\leftarrow	$\leftarrow$	\Leftarrow	$\Leftarrow$	\rightarrow	$\rightarrow$
\Rightarrow	$\Rightarrow$	\leftrightarrow	$\leftrightarrow$	\uparrow	$\uparrow$
\downarrow	$\downarrow$				

Openings and Closings

Code	Sym	Code	Sym	Code	Sym
\lfloor	$\lfloor$	\langle	$\langle$	\lceil	$\lceil$
\rfloor	$\rfloor$	\rangle	$\rangle$	\rceil	$\rceil$

Alternate Names

Code	Sym	Code	Sym	Code	Sym
\neq	$\neq$				

Other (not in Appendix F Tables)

Code	Sym	Code	Sym	Code	Sym
\ldots	$\ldots$	\0	$\oslash$	\copyright	©
\deg	$\circ$				

Table 15.1: Available special characters in TEX mode (cont.)

A complete example showing the capabilities of the extended text is

```
x = 0:0.01:3;
plot (x, erf (x));
hold on;
plot (x,x,"r");
axis ([0, 3, 0, 1]);
text (0.65, 0.6175, strcat ('\leftarrow x = {2/\surd\pi',
' {\fontsize{16}\int_{\fontsize{8}0}^{\fontsize{8}x}}',
' e^{-t^2} dt} = 0.6175'))
```

The result of which can be seen in Figure 15.7

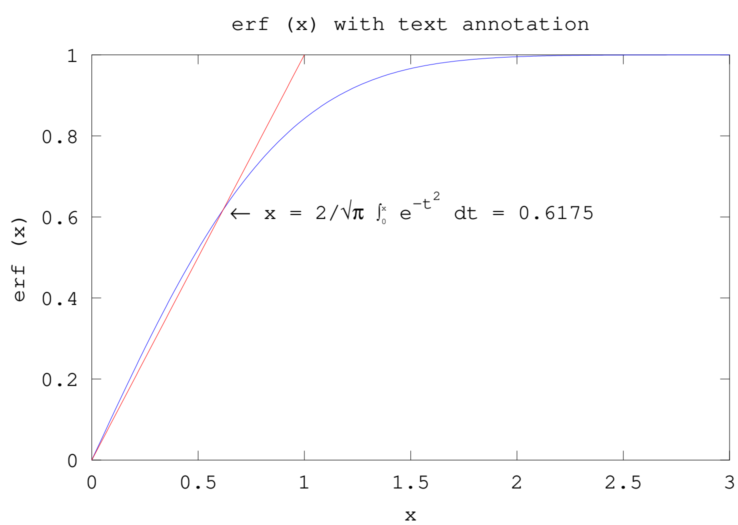

Figure 15.7: Example of inclusion of text with the TEX interpreter

15.2.9 Printing and Saving Plots

The `print` command allows you to send plots to you printer and to save plots in a variety of formats. For example,

```
print -dpsc
```

prints the current figure to a color PostScript printer. And,

```
print -deps foo.eps
```

saves the current figure to an encapsulated PostScript file called `foo.eps`.

The different graphic toolkits have different print capabilities. In particular, the OpenGL based toolkits such as `fltk` do not support the `"interpreter"` property of text objects. This means special symbols drawn with the `"tex"` interpreter will appear correctly on-screen but will be rendered with interpreter `"none"` when printing. Switch graphics toolkits for printing if this is a concern.

print () [Function File]
print (*options*) [Function File]
print (*filename*, *options*) [Function File]
print (*h*, *filename*, *options*) [Function File]

Print a plot, or save it to a file.

Both output formatted for printing (PDF and PostScript), and many bitmapped and vector image formats are supported.

filename defines the name of the output file. If the file name has no suffix, one is inferred from the specified device and appended to the file name. If no filename is specified, the output is sent to the printer.

h specifies the handle of the figure to print. If no handle is specified the current figure is used.

For output to a printer, PostScript file, or PDF file, the paper size is specified by the figure's `papersize` property. The location and size of the image on the page are specified by the figure's `paperposition` property. The orientation of the page is specified by the figure's `paperorientation` property.

The width and height of images are specified by the figure's `paperpositon(3:4)` property values.

The `print` command supports many *options*:

`-fh` Specify the handle, *h*, of the figure to be printed. The default is the current figure.

`-Pprinter`
: Set the *printer* name to which the plot is sent if no *filename* is specified.

`-Gghostscript_command`
: Specify the command for calling Ghostscript. For Unix and Windows the defaults are `"gs"` and `"gswin32c"`, respectively.

`-color`
`-mono` Color or monochrome output.

`-solid`
`-dashed` Force all lines to be solid or dashed, respectively.

`-portrait`
`-landscape`
: Specify the orientation of the plot for printed output. For non-printed output the aspect ratio of the output corresponds to the plot area defined by the `"paperposition"` property in the orientation specified. This option is equivalent to changing the figure's `"paperorientation"` property.

`-TextAlphaBits=n`
`-GraphicsAlphaBits=n`
: Octave is able to produce output for various printers, bitmaps, and vector formats by using Ghostscript. For bitmap and printer output anti-aliasing is applied using Ghostscript's TextAlphaBits and GraphicsAlphaBits options. The default number of bits for each is 4. Allowed values for *N* are 1, 2, or 4.

`-ddevice` The available output format is specified by the option *device*, and is one of:

`ps`
`ps2`
`psc`
`psc2` PostScript (level 1 and 2, mono and color). The FLTK graphics toolkit generates PostScript level 3.0.

`eps`
`eps2`
`epsc`
`epsc2` Encapsulated PostScript (level 1 and 2, mono and color). The FLTK graphic toolkit generates PostScript level 3.0.

`pslatex`
`epslatex`
`pdflatex`
`pslatexstandalone`
`epslatexstandalone`
`pdflatexstandalone`
: Generate a LaTeX file *filename*`.tex` for the text portions of a plot and a file *filename*`.(ps|eps|pdf)` for the remaining graphics. The graphics file suffix .ps|eps|pdf is determined by the specified device type. The LaTeX file produced by the '`standalone`' option can be processed directly by LaTeX. The file generated without the '`standalone`' option is intended to be included from another LaTeX document. In either case, the LaTeX file contains an `\includegraphics` command so that the generated graphics file is automatically included when the LaTeX file is processed. The text that is written to the LaTeX file contains the strings **exactly** as they were specified in the plot. If any special characters of the TeX mode interpreter were used, the file must be edited before LaTeX processing. Specifically, the special characters must be enclosed with dollar signs (`$ ... $`), and other characters that are recognized by LaTeX may also need editing (.e.g., braces). The '`pdflatex`' device, and any of the '`standalone`' formats, are not available with the Gnuplot toolkit.

`tikz`
: Generate a LaTeX file using PGF/TikZ. For the FLTK toolkit the result is PGF.

`ill`
`aifm`
: Adobe Illustrator (Obsolete for Gnuplot versions > 4.2)

`cdr`
`corel`
: CorelDraw

`dxf`
: AutoCAD

`emf`
`meta`
: Microsoft Enhanced Metafile

`fig`
: XFig. For the Gnuplot graphics toolkit, the additional options `-textspecial` or `-textnormal` can be used to control whether the special flag should be set for the text in the figure. (default is `-textnormal`)

`hpgl`
: HP plotter language

`mf`
: Metafont

`png`
: Portable network graphics

`jpg`
`jpeg`
: JPEG image

`gif` GIF image (only available for the Gnuplot graphics toolkit)

`pbm` PBMplus

`svg` Scalable vector graphics

`pdf` Portable document format

If the device is omitted, it is inferred from the file extension, or if there is no filename it is sent to the printer as PostScript.

`-d`*`ghostscript_device`*
: Additional devices are supported by Ghostscript. Some examples are;

`pdfwrite` Produces pdf output from eps

`ljet2p` HP LaserJet IIP

`pcx24b` 24-bit color PCX file format

`ppm` Portable Pixel Map file format

For a complete list, type `system ("gs -h")` to see what formats and devices are available.

When Ghostscript output is sent to a printer the size is determined by the figure's `"papersize"` property. When the output is sent to a file the size is determined by the plot box defined by the figure's `"paperposition"` property.

`-append` Append PostScript or PDF output to a pre-existing file of the same type.

`-r`*`NUM`* Resolution of bitmaps in pixels per inch. For both metafiles and SVG the default is the screen resolution; for other formats it is 150 dpi. To specify screen resolution, use `"-r0"`.

`-loose`
`-tight` Force a tight or loose bounding box for eps files. The default is loose.

`-`*`preview`* Add a preview to eps files. Supported formats are:

`-interchange`
: Provide an interchange preview.

`-metafile`
: Provide a metafile preview.

`-pict` Provide pict preview.

`-tiff` Provide a tiff preview.

`-S`*`xsize`*`,`*`ysize`*
: Plot size in pixels for EMF, GIF, JPEG, PBM, PNG, and SVG. For PS, EPS, PDF, and other vector formats the plot size is in points. This option is equivalent to changing the size of the plot box associated with the `"paperposition"` property. When using the command form of the print function you must quote the *xsize,ysize* option. For example, by writing `"-S640,480"`.

`-Ffontname`
`-Ffontname:size`
`-F:size` Use *fontname* and/or *fontsize* for all text. *fontname* is ignored for some devices: dxf, fig, hpgl, etc.

The filename and options can be given in any order.

Example: Print to a file using the pdf device.

```
figure (1);
clf ();
surf (peaks);
print figure1.pdf
```

Example: Print to a file using jpg device.

```
clf ();
surf (peaks);
print -djpg figure2.jpg
```

Example: Print to printer named PS_printer using ps format.

```
clf ();
surf (peaks);
print -dpswrite -PPS_printer
```

See also: [saveas], page 350, [hgsave], page 351, [orient], page 350, [figure], page 336.

`saveas (h, filename)` [Function File]
`saveas (h, filename, fmt)` [Function File]

Save graphic object *h* to the file *filename* in graphic format *fmt*.

fmt should be one of the following formats:

`ps` PostScript

`eps` Encapsulated PostScript

`jpg` JPEG Image

`png` PNG Image

`emf` Enhanced Meta File

`pdf` Portable Document Format

All device formats specified in `print` may also be used. If *fmt* is omitted it is extracted from the extension of *filename*. The default format is `"pdf"`.

```
clf ();
surf (peaks);
saveas (1, "figure1.png");
```

See also: [print], page 346, [hgsave], page 351, [orient], page 350.

`orient (orientation)` [Function File]
`orient (hfig, orientation)` [Function File]
`orientation = orient ()` [Function File]
`orientation = orient (hfig)` [Function File]

Query or set the print orientation for figure *hfig*.

Valid values for *orientation* are `"portrait"`, `"landscape"`, and `"tall"`.

The `"landscape"` option changes the orientation so the plot width is larger than the plot height. The `"paperposition"` is also modified so that the plot fills the page, while leaving a 0.25 inch border.

The `"tall"` option sets the orientation to `"portrait"` and fills the page with the plot, while leaving a 0.25 inch border.

The `"portrait"` option (default) changes the orientation so the plot height is larger than the plot width. It also restores the default `"paperposition"` property.

When called with no arguments, return the current print orientation.

If the argument *hfig* is omitted, then operate on the current figure returned by `gcf`.

See also: [print], page 346, [saveas], page 350.

`print` and `saveas` are used when work on a plot has finished and the output must be in a publication-ready format. During intermediate stages it is often better to save the graphics object and all of its associated information so that changes—to colors, axis limits, marker styles, etc.—can be made easily from within Octave. The `hgsave`/`hgload` commands can be used to save and re-create a graphics object.

`hgsave (`*`filename`*`)` [Function File]
`hgsave (`*`h`*`, `*`filename`*`)` [Function File]
`hgsave (`*`h`*`, `*`filename`*`, `*`fmt`*`)` [Function File]

Save the graphics handle *h* to the file *filename* in the format *fmt*.

If unspecified, *h* is the current figure as returned by `gcf`.

When *filename* does not have an extension the default filename extension `.ofig` will be appended.

If present, *fmt* should be one of the following:

- `-binary`, `-float-binary`
- `-hdf5`, `-float-hdf5`
- `-V7`, `-v7`, `-7`, `-mat7-binary`
- `-V6`, `-v6`, `-6`, `-mat6-binary`
- `-text`
- `-zip`, `-z`

When producing graphics for final publication use `print` or `saveas`. When it is important to be able to continue to edit a figure as an Octave object, use `hgsave`/`hgload`.

See also: [hgload], page 351, [hdl2struct], page 363, [saveas], page 350, [print], page 346.

`h` `= hgload (`*`filename`*`)` [Function File]

Load the graphics object in *filename* into the graphics handle *h*.

If *filename* has no extension, Octave will try to find the file with and without the standard extension of `.ofig`.

See also: [hgsave], page 351, [struct2hdl], page 363.

15.2.10 Interacting with Plots

The user can select points on a plot with the `ginput` function or selection the position at which to place text on the plot with the `gtext` function using the mouse. Menus may also be created and populated with specific user commands via the `uimenu` function.

`[x, y, buttons] = ginput (n)` [Function File]
`[x, y, buttons] = ginput ()` [Function File]

Return the position and type of mouse button clicks and/or key strokes in the current figure window.

If *n* is defined, then capture *n* events before returning. When *n* is not defined `ginput` will loop until the return key `RET` is pressed.

The return values *x*, *y* are the coordinates where the mouse was clicked in the units of the current axes. The return value *button* is 1, 2, or 3 for the left, middle, or right button. If a key is pressed the ASCII value is returned in *button*.

Implementation Note: `ginput` is intenteded for 2-D plots. For 3-D plots see the *currentpoint* property of the current axes which can be transformed with knowledge of the current `view` into data units.

See also: [gtext], page 352, [waitforbuttonpress], page 352.

`waitforbuttonpress ()` [Function File]
`b = waitforbuttonpress ()` [Function File]

Wait for mouse click or key press over the current figure window.

The return value of *b* is 0 if a mouse button was pressed or 1 if a key was pressed.

See also: [waitfor], page 746, [ginput], page 352, [kbhit], page 237.

`gtext (s)` [Function File]
`gtext ({s1, s2, ...})` [Function File]
`gtext ({s1; s2; ...})` [Function File]
`gtext (..., prop, val, ...)` [Function File]
`h = gtext (...)` [Function File]

Place text on the current figure using the mouse.

The text is defined by the string *s*. If *s* is a cell string organized as a row vector then each string of the cell array is written to a separate line. If *s* is organized as a column vector then one string element of the cell array is placed for every mouse click.

Optional property/value pairs are passed directly to the underlying text objects.

The optional return value *h* is a graphics handle to the created text object(s).

See also: [ginput], page 352, [text], page 330.

`hui = uimenu (property, value, ...)` [Function File]
`hui = uimenu (h, property, value, ...)` [Function File]

Create a uimenu object and return a handle to it.

If *h* is omitted then a top-level menu for the current figure is created. If *h* is given then a submenu relative to *h* is created.

uimenu objects have the following specific properties:

"accelerator"
: A string containing the key combination together with CTRL to execute this menu entry (e.g., "x" for CTRL+x).

"callback"
: Is the function called when this menu entry is executed. It can be either a function string (e.g., "myfun"), a function handle (e.g., @myfun) or a cell array containing the function handle and arguments for the callback function (e.g., {@myfun, arg1, arg2}).

"checked"
: Can be set "on" or "off". Sets a mark at this menu entry.

"enable"
: Can be set "on" or "off". If disabled the menu entry cannot be selected and it is grayed out.

"foregroundcolor"
: A color value setting the text color for this menu entry.

"label"
: A string containing the label for this menu entry. A "&"-symbol can be used to mark the "accelerator" character (e.g., "E&xit")

"position"
: An scalar value containing the relative menu position. The entry with the lowest value is at the first position starting from left or top.

"separator"
: Can be set "on" or "off". If enabled it draws a separator line above the current position. It is ignored for top level entries.

Examples:

```
f = uimenu ("label", "&File", "accelerator", "f");
e = uimenu ("label", "&Edit", "accelerator", "e");
uimenu (f, "label", "Close", "accelerator", "q", ...
           "callback", "close (gcf)");
uimenu (e, "label", "Toggle &Grid", "accelerator", "g", ...
           "callback", "grid (gca)");
```

See also: [figure], page 336.

15.2.11 Test Plotting Functions

The functions sombrero and peaks provide a way to check that plotting is working. Typing either sombrero or peaks at the Octave prompt should display a three-dimensional plot.

sombrero () [Function File]
sombrero (*n*) [Function File]
z = sombrero (...) [Function File]
[*x*, *y*, *z*] = sombrero (...) [Function File]

Plot the familiar 3-D sombrero function.

The function plotted is

$$z = \frac{\sin(\sqrt{(\mathrm{x}^2 + \mathrm{y}^2)})}{\sqrt{(\mathrm{x}^2 + \mathrm{y}^2)}}$$

Called without a return argument, `sombrero` plots the surface of the above function over the meshgrid [-8,8] using `surf`.

If *n* is a scalar the plot is made with *n* grid lines. The default value for *n* is 41.

When called with output arguments, return the data for the function evaluated over the meshgrid. This can subsequently be plotted with `surf (x, y, z)`.

See also: [peaks], page 354, [meshgrid], page 317, [mesh], page 307, [surf], page 309.

`peaks ()` [Function File]
`peaks (n)` [Function File]
`peaks (x, y)` [Function File]
`z = peaks (...)` [Function File]
`[x, y, z] = peaks (...)` [Function File]

Plot a function with lots of local maxima and minima.

The function has the form

$$f(x,y) = 3(1-x)^2 e^{\left(-x^2-(y+1)^2\right)} - 10\left(\frac{x}{5} - x^3 - y^5\right) - \frac{1}{3} e^{\left(-(x+1)^2-y^2\right)}$$

Called without a return argument, `peaks` plots the surface of the above function using `surf`.

If *n* is a scalar, `peaks` plots the value of the above function on an *n*-by-*n* mesh over the range [-3,3]. The default value for *n* is 49.

If *n* is a vector, then it represents the grid values over which to calculate the function. If *x* and *y* are specified then the function value is calculated over the specified grid of vertices.

When called with output arguments, return the data for the function evaluated over the meshgrid. This can subsequently be plotted with `surf (x, y, z)`.

See also: [sombrero], page 353, [meshgrid], page 317, [mesh], page 307, [surf], page 309.

15.3 Graphics Data Structures

15.3.1 Introduction to Graphics Structures

The graphics functions use pointers, which are of class graphics_handle, in order to address the data structures which control visual display. A graphics handle may point to any one of a number of different base object types and these objects are the graphics data structures themselves. The primitive graphic object types are: `figure`, `axes`, `line`, `text`, `patch`, `surface`, `text`, and `image`.

Each of these objects has a function by the same name, and, each of these functions returns a graphics handle pointing to an object of the corresponding type. In addition there are several functions which operate on properties of the graphics objects and which also return handles: the functions `plot` and `plot3` return a handle pointing to an object of type line, the function `subplot` returns a handle pointing to an object of type axes, the function `fill` returns a handle pointing to an object of type patch, the functions `area`, `bar`, `barh`, `contour`, `contourf`, `contour3`, `surf`, `mesh`, `surfc`, `meshc`, `errorbar`, `quiver`, `quiver3`, `scatter`, `scatter3`, `stair`, `stem`, `stem3` each return a handle to a complex data structure as documented in [Data Sources], page 394.

The graphics objects are arranged in a hierarchy:

1. The root is at 0. In other words, `get (0)` returns the properties of the root object.
2. Below the root are `figure` objects.
3. Below the `figure` objects are `axes` objects.
4. Below the `axes` objects are `line`, `text`, `patch`, `surface`, and `image` objects.

Graphics handles may be distinguished from function handles (see Section 11.11.1 [Function Handles], page 199) by means of the function `ishandle`. `ishandle` returns true if its argument is a handle of a graphics object. In addition, a figure or axes object may be tested using `isfigure` or `isaxes` respectively. The test functions return true only if the argument is both a handle and of the correct type (figure or axes).

The `whos` function can be used to show the object type of each currently defined graphics handle. (Note: this is not true today, but it is, I hope, considered an error in whos. It may be better to have whos just show graphics_handle as the class, and provide a new function which, given a graphics handle, returns its object type. This could generalize the ishandle() functions and, in fact, replace them.)

The `get` and `set` commands are used to obtain and set the values of properties of graphics objects. In addition, the `get` command may be used to obtain property names.

For example, the property `"type"` of the graphics object pointed to by the graphics handle h may be displayed by:

```
get (h, "type")
```

The properties and their current values are returned by `get (h)` where h is a handle of a graphics object. If only the names of the allowed properties are wanted they may be displayed by: `get (h, "")`.

Thus, for example:

```
h = figure ();
get (h, "type")
ans = figure
get (h, "");
error: get: ambiguous figure property name ; possible matches:

__enhanced__          hittest               resize
__graphics_toolkit__  integerhandle         resizefcn
__guidata__           interruptible         selected
__modified__          inverthardcopy        selectionhighlight
__myhandle__          keypressfcn           selectiontype
__plot_stream__       keyreleasefcn         tag
alphamap              menubar               toolbar
beingdeleted          mincolormap           type
busyaction            name                  uicontextmenu
buttondownfcn         nextplot              units
children              numbertitle           userdata
clipping              outerposition         visible
```

```
closerequestfcn        paperorientation        windowbuttondownfcn
color                  paperposition           windowbuttonmotionfcn
colormap               paperpositionmode       windowbuttonupfcn
createfcn              papersize               windowkeypressfcn
currentaxes            papertype               windowkeyreleasefcn
currentcharacter       paperunits              windowscrollwheelfcn
currentobject          parent                  windowstyle
currentpoint           pointer                 wvisual
deletefcn              pointershapecdata       wvisualmode
dockcontrols           pointershapehotspot     xdisplay
doublebuffer           position                xvisual
filename               renderer                xvisualmode
handlevisibility       renderermode
```

The root figure has index 0. Its properties may be displayed by: `get (0, "")`.

The uses of `get` and `set` are further explained in [get], page 361, [set], page 361.

res = isprop (*obj*, "*prop*") [Function File]

Return true if *prop* is a property of the object *obj*.

obj may also be an array of objects in which case *res* will be a logical array indicating whether each handle has the property *prop*.

For plotting, *obj* is a handle to a graphics object. Otherwise, *obj* should be an instance of a class.

See also: [get], page 361, [set], page 361, [ismethod], page 723, [isobject], page 722.

15.3.2 Graphics Objects

The hierarchy of graphics objects was explained above. See Section 15.3.1 [Introduction to Graphics Structures], page 354. Here the specific objects are described, and the properties contained in these objects are discussed. Keep in mind that graphics objects are always referenced by *handle*.

root figure
: the top level of the hierarchy and the parent of all figure objects. The handle index of the root figure is 0.

figure
: A figure window.

axes
: A set of axes. This object is a child of a figure object and may be a parent of line, text, image, patch, or surface objects.

line
: A line in two or three dimensions.

text
: Text annotations.

image
: A bitmap image.

patch
: A filled polygon, currently limited to two dimensions.

surface
: A three-dimensional surface.

15.3.2.1 Creating Graphics Objects

You can create any graphics object primitive by calling the function of the same name as the object; In other words, `figure`, `axes`, `line`, `text`, `image`, `patch`, and `surface` functions. These fundamental graphic objects automatically become children of the current axes object as if `hold on` was in place. Seperately, axes will automatically become children of the current figure object and figures will become children of the root object 0.

If this auto-joining feature is not desired then it is important to call `newplot` first to prepare a new figure and axes for plotting. Alternatively, the easier way is to call a high-level graphics routine which will both create the plot and then populate it with low-level graphics objects. Instead of calling `line`, use `plot`. Or use `surf` instead of `surface`. Or use `fill` instead of `patch`.

`axes ()` [Function File]
`axes (`*property*`,` *value*`, ...)` [Function File]
`axes (`*hax*`)` [Function File]
h `= axes (...)` [Function File]

Create an axes object and return a handle to it, or set the current axes to *hax*.

Called without any arguments, or with *property*/*value* pairs, construct a new axes. For accepted properties and corresponding values, see [set], page 361.

Called with a single axes handle argument *hax*, the function makes *hax* the current axis. It also restacks the axes in the corresponding figure so that *hax* is the first entry in the list of children. This causes *hax* to be displayed on top of any other axes objects (Z-order stacking).

See also: [gca], page 360, [set], page 361, [get], page 361.

`line ()` [Function File]
`line (`*x*`,` *y*`)` [Function File]
`line (`*x*`,` *y*`,` *property*`,` *value*`, ...)` [Function File]
`line (`*x*`,` *y*`,` *z*`)` [Function File]
`line (`*x*`,` *y*`,` *z*`,` *property*`,` *value*`, ...)` [Function File]
`line (`*property*`,` *value*`, ...)` [Function File]
`line (`*hax*`, ...)` [Function File]
h `= line (...)` [Function File]

Create line object from *x* and *y* (and possibly *z*) and insert in the current axes.

Multiple property-value pairs may be specified for the line object, but they must appear in pairs.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle (or vector of handles) to the line objects created.

See also: [image], page 701, [patch], page 357, [rectangle], page 304, [surface], page 358, [text], page 330.

`patch ()` [Function File]
`patch (`*x*`,` *y*`,` *c*`)` [Function File]
`patch (`*x*`,` *y*`,` *z*`,` *c*`)` [Function File]
`patch (`*fv*`)` [Function File]
`patch ("`*Faces*`",` *faces*`, "`*Vertices*`",` *verts*`, ...)` [Function File]
`patch (...,` *prop*`,` *val*`, ...)` [Function File]
`patch (`*hax*`, ...)` [Function File]

`h = patch (...)` [Function File]

Create patch object in the current axes with vertices at locations (x, y) and of color c.

If the vertices are matrices of size MxN then each polygon patch has M vertices and a total of N polygons will be created. If some polygons do not have M vertices use NaN to represent "no vertex". If the z input is present then 3-D patches will be created.

The color argument c can take many forms. To create polygons which all share a single color use a string value (e.g., `"r"` for red), a scalar value which is scaled by `caxis` and indexed into the current colormap, or a 3-element RGB vector with the precise TrueColor.

If c is a vector of length N then the ith polygon will have a color determined by scaling entry c(i) according to `caxis` and then indexing into the current colormap. More complicated coloring situations require directly manipulating patch property/value pairs.

Instead of specifying polygons by matrices x and y, it is possible to present a unique list of vertices and then a list of polygon faces created from those vertices. In this case the `"Vertices"` matrix will be an Nx2 (2-D patch) or Nx3 (3-D patch). The MxN `"Faces"` matrix describes M polygons having N vertices—each row describes a single polygon and each column entry is an index into the `"Vertices"` matrix to identify a vertex. The patch object can be created by directly passing the property/value pairs `"Vertices"`/*verts*, `"Faces"`/*faces* as inputs.

A third input form is to create a structure *fv* with the fields `"vertices"`, `"faces"`, and optionally `"facevertexcdata"`.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created patch object.

Implementation Note: Patches are highly configurable objects. To truly customize them requires setting patch properties directly. Useful patch properties are: `"cdata"`, `"edgecolor"`, `"facecolor"`, `"faces"`, `"facevertexcdata"`.

See also: [fill], page 297, [get], page 361, [set], page 361.

`surface (x, y, z, c)` [Function File]
`surface (x, y, z)` [Function File]
`surface (z, c)` [Function File]
`surface (z)` [Function File]
`surface (..., prop, val, ...)` [Function File]
`surface (hax, ...)` [Function File]
`h = surface (...)` [Function File]

Create a surface graphic object given matrices x and y from `meshgrid` and a matrix of values z corresponding to the x and y coordinates of the surface.

If x and y are vectors, then a typical vertex is (x(j), y(i), z(i,j)). Thus, columns of z correspond to different x values and rows of z correspond to different y values. If only a single input z is given then x is taken to be `1:rows (z)` and y is `1:columns (z)`.

Any property/value input pairs are assigned to the surface object.

If the first argument *hax* is an axes handle, then plot into this axis, rather than the current axes returned by `gca`.

The optional return value *h* is a graphics handle to the created surface object.

See also: [surf], page 309, [mesh], page 307, [patch], page 357, [line], page 357.

15.3.2.2 Handle Functions

To determine whether a variable is a graphics object index, or an index to an axes or figure, use the functions `ishandle`, `isaxes`, and `isfigure`.

`ishandle (`*h*`)` [Built-in Function]

Return true if *h* is a graphics handle and false otherwise.

h may also be a matrix of handles in which case a logical array is returned that is true where the elements of *h* are graphics handles and false where they are not.

See also: [isaxes], page 359, [isfigure], page 359.

`ishghandle (`*h*`)` [Function File]

Return true if *h* is a graphics handle and false otherwise.

This function is equivalent to `ishandle` and is provided for compatibility with MATLAB.

See also: [ishandle], page 359.

`isaxes (`*h*`)` [Function File]

Return true if *h* is an axes graphics handle and false otherwise.

If *h* is a matrix then return a logical array which is true where the elements of *h* are axes graphics handles and false where they are not.

See also: [isaxes], page 359, [ishandle], page 359.

`isfigure (`*h*`)` [Function File]

Return true if *h* is a figure graphics handle and false otherwise.

If *h* is a matrix then return a logical array which is true where the elements of *h* are figure graphics handles and false where they are not.

See also: [isaxes], page 359, [ishandle], page 359.

The function `gcf` returns an index to the current figure object, or creates one if none exists. Similarly, `gca` returns the current axes object, or creates one (and its parent figure object) if none exists.

h `= gcf ()` [Function File]

Return a handle to the current figure.

The current figure is the default target for graphics output. If multiple figures exist, `gcf` returns the last created figure or the last figure that was clicked on with the mouse.

If a current figure does not exist, create one and return its handle. The handle may then be used to examine or set properties of the figure. For example,

```
fplot (@sin, [-10, 10]);
fig = gcf ();
set (fig, "numbertitle", "off", "name", "sin plot")
```

plots a sine wave, finds the handle of the current figure, and then renames the figure window to describe the contents.

Note: To find the current figure without creating a new one if it does not exist, query the "CurrentFigure" property of the root graphics object.

```
get (0, "currentfigure");
```

See also: [gca], page 360, [gco], page 360, [gcbf], page 389, [gcbo], page 388, [get], page 361, [set], page 361.

h = gca () [Function File]

Return a handle to the current axis object.

The current axis is the default target for graphics output. In the case of a figure with multiple axes, gca returns the last created axes or the last axes that was clicked on with the mouse.

If no current axes object exists, create one and return its handle. The handle may then be used to examine or set properties of the axes. For example,

```
ax = gca ();
set (ax, "position", [0.5, 0.5, 0.5, 0.5]);
```

creates an empty axes object and then changes its location and size in the figure window.

Note: To find the current axis without creating a new axes object if it does not exist, query the "CurrentAxes" property of a figure.

```
get (gcf, "currentaxes");
```

See also: [gcf], page 359, [gco], page 360, [gcbf], page 389, [gcbo], page 388, [get], page 361, [set], page 361.

h = gco () [Function File]
h = gco (*fig*) [Function File]

Return a handle to the current object of the current figure, or a handle to the current object of the figure with handle *fig*.

The current object of a figure is the object that was last clicked on. It is stored in the "CurrentObject" property of the target figure.

If the last mouse click did not occur on any child object of the figure, then the current object is the figure itself.

If no mouse click occurred in the target figure, this function returns an empty matrix.

Programming Note: The value returned by this function is not necessarily the same as the one returned by gcbo during callback execution. An executing callback can be interrupted by another callback and the current object may be changed.

See also: [gcbo], page 388, [gca], page 360, [gcf], page 359, [gcbf], page 389, [get], page 361, [set], page 361.

The get and set functions may be used to examine and set properties for graphics objects. For example,

```
get (0)
    ⇒ ans =
       {
         type = root
         currentfigure = [](0x0)
         children = [](0x0)
         visible = on
         ...
       }
```

returns a structure containing all the properties of the root figure. As with all functions in Octave, the structure is returned by value, so modifying it will not modify the internal root figure plot object. To do that, you must use the `set` function. Also, note that in this case, the `currentfigure` property is empty, which indicates that there is no current figure window.

The `get` function may also be used to find the value of a single property. For example,

```
get (gca (), "xlim")
    ⇒ [ 0 1 ]
```

returns the range of the x-axis for the current axes object in the current figure.

To set graphics object properties, use the set function. For example,

```
set (gca (), "xlim", [-10, 10]);
```

sets the range of the x-axis for the current axes object in the current figure to '`[-10, 10]`'.

Default property values can also be queried if the `set` function is called without a value argument. When only one argument is given (a graphic handle) then a structure with defaults for all properties of the given object type is returned. For example,

```
set (gca ())
```

returns a structure containing the default property values for axes objects. If `set` is called with two arguments (a graphic handle and a property name) then only the defaults for the requested property are returned.

val = get (*h*) [Built-in Function]
val = get (*h*, *p*) [Built-in Function]

Return the value of the named property *p* from the graphics handle *h*.

If *p* is omitted, return the complete property list for *h*.

If *h* is a vector, return a cell array including the property values or lists respectively.

See also: [set], page 361.

set (*h*, *property*, *value*, ...) [Built-in Function]
set (*h*, *properties*, *values*) [Built-in Function]
set (*h*, *pv*) [Built-in Function]
value_list = set (*h*, *property*) [Built-in Function]
all_value_list = set (*h*) [Built-in Function]

Set named property values for the graphics handle (or vector of graphics handles) *h*.

There are three ways to give the property names and values:

- as a comma separated list of *property*, *value* pairs

  Here, each *property* is a string containing the property name, each *value* is a value of the appropriate type for the property.

- as a cell array of strings *properties* containing property names and a cell array *values* containing property values.

  In this case, the number of columns of *values* must match the number of elements in *properties*. The first column of *values* contains values for the first entry in *properties*, etc. The number of rows of *values* must be 1 or match the number of elements of *h*. In the first case, each handle in *h* will be assigned the same values. In the latter case, the first handle in *h* will be assigned the values from the first row of *values* and so on.

- as a structure array *pv*

  Here, the field names of *pv* represent the property names, and the field values give the property values. In contrast to the previous case, all elements of *pv* will be set in all handles in *h* independent of the dimensions of *pv*.

`set` is also used to query the list of values a named property will take. `clist = set (h, "property")` will return the list of possible values for `"property"` in the cell list *clist*. If no output variable is used then the list is formatted and printed to the screen.

If no property is specified (`slist = set (h)`) then a structure *slist* is returned where the fieldnames are the properties of the object *h* and the fields are the list of possible values for each property. If no output variable is used then the list is formatted and printed to the screen.

For example,

```
hf = figure ();
set (hf, "paperorientation")
⇒  paperorientation:  [ landscape | {portrait} | rotated ]
```

shows the paperorientation property can take three values with the default being `"portrait"`.

See also: [get], page 361.

`parent = ancestor (h, type)` [Function File]
`parent = ancestor (h, type, "toplevel")` [Function File]

Return the first ancestor of handle object *h* whose type matches *type*, where *type* is a character string.

If *type* is a cell array of strings, return the first parent whose type matches any of the given type strings.

If the handle object *h* itself is of type *type*, return *h*.

If `"toplevel"` is given as a third argument, return the highest parent in the object hierarchy that matches the condition, instead of the first (nearest) one.

See also: [findobj], page 384, [findall], page 385, [allchild], page 363.

`h = allchild (handles)` [Function File]

Find all children, including hidden children, of a graphics object.

This function is similar to `get (h, "children")`, but also returns hidden objects (HandleVisibility = `"off"`).

If *handles* is a scalar, *h* will be a vector. Otherwise, *h* will be a cell matrix of the same size as *handles* and each cell will contain a vector of handles.

See also: [findall], page 385, [findobj], page 384, [get], page 361, [set], page 361.

`findfigs ()` [Function File]

Find all visible figures that are currently off the screen and move them onto the screen.

See also: [allchild], page 363, [figure], page 336, [get], page 361, [set], page 361.

Figures can be printed or saved in many graphics formats with `print` and `saveas`. Occasionally, however, it may be useful to save the original Octave handle graphic directly so that further modifications can be made such as modifying a title or legend.

This can be accomplished with the following functions by

```
fig_struct = hdl2struct (gcf);
save myplot.fig -struct fig_struct;
...
fig_struct = load ("myplot.fig");
struct2hdl (fig_struct);
```

`s = hdl2struct (h)` [Function File]

Return a structure, *s*, whose fields describe the properties of the object, and its children, associated with the handle, *h*.

The fields of the structure *s* are `"type"`, `"handle"`, `"properties"`, `"children"`, and `"special"`.

See also: [struct2hdl], page 363, [hgsave], page 351, [findobj], page 384.

`h = struct2hdl (s)` [Function File]
`h = struct2hdl (s, p)` [Function File]
`h = struct2hdl (s, p, hilev)` [Function File]

Construct a graphics handle object *h* from the structure *s*.

The structure must contain the fields `"handle"`, `"type"`, `"children"`, `"properties"`, and `"special"`.

If the handle of an existing figure or axes is specified, *p*, the new object will be created as a child of that object. If no parent handle is provided then a new figure and the necessary children will be constructed using the default values from the root figure.

A third boolean argument *hilev* can be passed to specify whether the function should preserve listeners/callbacks, e.g., for legends or hggroups. The default is false.

See also: [hdl2struct], page 363, [hgload], page 351, [findobj], page 384.

`hnew = copyobj (horig)` [Function File]
`hnew = copyobj (horig, hparent)` [Function File]

Construct a copy of the graphic object associated with handle *horig* and return a handle *hnew* to the new object.

If a parent handle *hparent* (root, figure, axes, or hggroup) is specified, the copied object will be created as a child of *hparent*.

See also: [struct2hdl], page 363, [hdl2struct], page 363, [findobj], page 384.

15.3.3 Graphics Object Properties

In this Section the graphics object properties are discussed in detail, starting with the root figure properties and continuing through the objects hierarchy. The documentation about a specific graphics object can be displayed using `doc` function, e.g., `doc ("axes properties")` will show Section 15.3.3.3 [Axes Properties], page 370.

The allowed values for radio (string) properties can be retrieved programmatically or displayed using the one or two arguments call to `set` function. See [set], page 361.

In the following documentation, default values are enclosed in { }.

15.3.3.1 Root Figure Properties

The `root figure` properties are:

`__modified__`: `"off"` | {`"on"`}
`beingdeleted`: {`"off"`} | `"on"`
: `beingdeleted` is unused.

`busyaction`: `"cancel"` | {`"queue"`}
: `busyaction` is unused.

`buttondownfcn`: string | function handle, def. `[](0x0)`
: `buttondownfcn` is unused.

`callbackobject` (read-only): graphics handle, def. `[](0x0)`
`children` (read-only): vector of graphics handles, def. `[](0x1)`
: Graphics handles of the root's children.

`clipping`: `"off"` | {`"on"`}
: `clipping` is unused.

`commandwindowsize` (read-only): def. `[0 0]`
`createfcn`: string | function handle, def. `[](0x0)`
: `createfcn` is unused.

`currentfigure`: graphics handle, def. `[](0x0)`
: Graphics handle of the current figure.

`deletefcn`: string | function handle, def. `[](0x0)`
: `deletefcn` is unused.

`diary`: {`"off"`} | `"on"`
: If `diary` is `"on"`, the Octave command window session is saved to file. See [diaryfile property], page 364.

`diaryfile`: string, def. `"diary"`
: The name of the diary file. See [diary function], page 34.

`echo`: {`"off"`} | `"on"`
: Control whether Octave displays commands executed from scripts. See [echo function], page 34.

`errormessage` (read-only): string, def. `""`
: The last error message raised. See [lasterr function], page 209.

`fixedwidthfontname`: string, def. `"Courier"`
`format`: `"+"` | `"bank"` | `"bit"` | `"hex"` | `"long"` | `"longe"` | `"longeng"` | `"longg"` | `"native-bit"` | `"native-hex"` | `"none"` | `"rat"` | {`"short"`} | `"shorte"` | `"shorteng"` | `"shortg"`
: This property is a wrapper around the `format` function. See [format function], page 232.

`formatspacing`: `"compact"` | {`"loose"`}
: This property is a wrapper around the `format` function. See [format function], page 232.

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}
: `handlevisibility` is unused.

`hittest`: `"off"` | {`"on"`}
: `hittest` is unused.

`interruptible`: `"off"` | {`"on"`}
: `interruptible` is unused.

`language`: string, def. `"ascii"`
`monitorpositions`:
: `monitorpositions` is unused.

`parent`: graphics handle, def. `[](0x0)`
: Root figure has no parent graphics object. `parent` is always empty.

`pointerlocation`: two-element vector, def. `[0 0]`
: `pointerlocation` is unused.

`pointerwindow` (read-only): graphics handle, def. `0`
: `pointerwindow` is unused.

`recursionlimit`: double, def. `256`
: The maximum number of times a function can be called recursively. See [max_recursion_depth function], page 141.

`screendepth` (read-only): double
`screenpixelsperinch` (read-only): double
`screensize` (read-only): four-element vector
`selected`: {`"off"`} | `"on"`
: `selected` is unused.

`selectionhighlight`: `"off"` | {`"on"`}
: `selectionhighlight` is unused.

`showhiddenhandles`: {`"off"`} | `"on"`
: If `showhiddenhandles` is `"on"`, all graphics objects handles are visible in their parents' children list, regardless of the value of their `handlevisibility` property.

`tag`: string, def. `""`
    A user-defined string to label the graphics object.

`type` (read-only): string
    Class name of the graphics object. `type` is always `"root"`

`uicontextmenu`: graphics handle, def. `[](0x0)`
    `uicontextmenu` is unused.

`units`: `"centimeters"` | `"inches"` | `"normalized"` | {`"pixels"`} | `"points"`
`userdata`: Any Octave data, def. `[](0x0)`
    User-defined data to associate with the graphics object.

`visible`: `"off"` | {`"on"`}
    `visible` is unused.

15.3.3.2 Figure Properties

The `figure` properties are:

`__modified__`: `"off"` | {`"on"`}
`alphamap`: def. 64-by-1 double
    Transparency is not yet implemented for figure objects. `alphamap` is unused.

`beingdeleted`: {`"off"`} | `"on"`
`busyaction`: `"cancel"` | {`"queue"`}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`children` (read-only): vector of graphics handles, def. `[](0x1)`
    Graphics handles of the figure's children.

`clipping`: `"off"` | {`"on"`}
    `clipping` is unused.

`closerequestfcn`: string | function handle, def. `"closereq"`
`color`: colorspec, def. `[1 1 1]`
    Color of the figure background. See Section 15.4.1 [colorspec], page 387.

`colormap`: N-by-3 matrix, def. 64-by-3 double
    A matrix containing the RGB color map for the current axes.

`createfcn`: string | function handle, def. `[](0x0)`
    Callback function executed immediately after figure has been created. Function is set by using default property on root object, e.g., `set (0, "defaultfigurecreatefcn", 'disp ("figure created!")')`.

`currentaxes`: graphics handle, def. `[](0x0)`
    Handle to the graphics object of the current axes.

`currentcharacter` (read-only): def. `""`
    `currentcharacter` is unused.

`currentobject` (read-only): graphics handle, def. `[](0x0)`
`currentpoint` (read-only): two-element vector, def. `[0; 0]`
    A 1-by-2 matrix which holds the coordinates of the point over which the mouse pointer was when a mouse event occurred. The X and Y coordinates are in

units defined by the figure's `units` property and their origin is the lower left corner of the plotting area.

Events which set `currentpoint` are

A mouse button was pressed
: always

A mouse button was released
: only if the figure's callback `windowbuttonupfcn` is defined

The pointer was moved while pressing the mouse button (drag)
: only if the figure's callback `windowbuttonmotionfcn` is defined

`deletefcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately before figure is deleted.

`dockcontrols`: {`"off"`} | `"on"`
: `dockcontrols` is unused.

`doublebuffer`: `"off"` | {`"on"`}

`filename`: string, def. `""`
: The filename used when saving the plot figure

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}
: If `handlevisibility` is `"off"`, the figure's handle is not visible in its parent's `"children"` property.

`hittest`: `"off"` | {`"on"`}

`integerhandle`: `"off"` | {`"on"`}
: Assign the next lowest unused integer as the Figure number.

`interruptible`: `"off"` | {`"on"`}

`inverthardcopy`: {`"off"`} | `"on"`

`keypressfcn`: string | function handle, def. `[](0x0)`

`keyreleasefcn`: string | function handle, def. `[](0x0)`
: With `keypressfcn`, the keyboard callback functions. These callback functions are called when a key is pressed/released respectively. The functions are called with two input arguments. The first argument holds the handle of the calling figure. The second argument holds an event structure which has the following members:

  `Character:`
  : The ASCII value of the key

  `Key:` Lowercase value of the key

  `Modifier:`
  : A cell array containing strings representing the modifiers pressed with the key.

`menubar`: {`"figure"`} | `"none"`
: Control the display of the figure menu bar in the upper left of the figure.

`mincolormap`: def. `64`

`name`: string, def. `""`

Name to be displayed in the figure title bar. The name is displayed to the right of any title determined by the `numbertitle` property.

`nextplot`: {`"add"`} | `"new"` | `"replace"` | `"replacechildren"`

`numbertitle`: `"off"` | {`"on"`}

Display "Figure" followed by the numerical figure handle value in the figure title bar.

`outerposition`: four-element vector, def. `[-1 -1 -1 -1]`

`paperorientation`: `"landscape"` | {`"portrait"`} | `"rotated"`

`paperposition`: four-element vector, def. `[0.25000 2.50000 8.00000 6.00000]`

Vector `[x0 y0 width height]` defining the position of the figure (in `paperunits` units) on the printed page. Setting `paperposition` also forces the `paperpositionmode` property to be set to `"manual"`.

`paperpositionmode`: `"auto"` | {`"manual"`}

If `paperpositionmode` is set to `"auto"`, the `paperposition` property is automatically computed: the printed figure will have the same size as the on-screen figure and will be centered on the output page.

`papersize`: two-element vector, def. `[8.5000 11.0000]`

Vector `[width height]` defining the size of the paper for printing. Setting this property forces the `papertype` property to be set to `"<custom>"`.

`papertype`: `"<custom>"` | `"a"` | `"a0"` | `"a1"` | `"a2"` | `"a3"` | `"a4"` | `"a5"` | `"arch-a"` | `"arch-b"` | `"arch-c"` | `"arch-d"` | `"arch-e"` | `"b"` | `"b0"` | `"b1"` | `"b2"` | `"b3"` | `"b4"` | `"b5"` | `"c"` | `"d"` | `"e"` | `"tabloid"` | `"uslegal"` | {`"usletter"`}

Name of the paper used for printed output. Setting `papertype` also changes `papersize` accordingly.

`paperunits`: `"centimeters"` | {`"inches"`} | `"normalized"` | `"points"`

The unit used to compute the `paperposition` property.

`parent`: graphics handle, def. `0`

Handle of the parent graphics object.

`pointer`: {`"arrow"`} | `"botl"` | `"botr"` | `"bottom"` | `"circle"` | `"cross"` | `"crosshair"` | `"custom"` | `"fleur"` | `"fullcrosshair"` | `"hand"` | `"ibeam"` | `"left"` | `"right"` | `"top"` | `"topl"` | `"topr"` | `"watch"`

`pointer` is unused.

`pointershapecdata`: def. 16-by-16 double

`pointershapecdata` is unused.

`pointershapehotspot`: def. `[0 0]`

`pointershapehotspot` is unused.

`position`: four-element vector, def. `[300 200 560 420]`
`renderer`: `"none"` | `"opengl"` | {`"painters"`} | `"zbuffer"`
`renderermode`: {`"auto"`} | `"manual"`
`resize`: `"off"` | {`"on"`}
`resizefcn`: string | function handle, def. `[](0x0)`
`selected`: {`"off"`} | `"on"`
`selectionhighlight`: `"off"` | {`"on"`}
`selectiontype`: `"alt"` | `"extend"` | {`"normal"`} | `"open"`

    `selectiontype` is unused.

`tag`: string, def. `""`

    A user-defined string to label the graphics object.

`toolbar`: {`"auto"`} | `"figure"` | `"none"`

    `toolbar` is unused.

`type` (read-only): string

    Class name of the graphics object. `type` is always `"figure"`

`uicontextmenu`: graphics handle, def. `[](0x0)`

    Graphics handle of the uicontextmenu object that is currently associated to this figure object.

`units`: `"centimeters"` | `"characters"` | `"inches"` | `"normalized"` | {`"pixels"`} | `"points"`

    The unit used to compute the `position` and `outerposition` properties.

`userdata`: Any Octave data, def. `[](0x0)`

    User-defined data to associate with the graphics object.

`visible`: `"off"` | {`"on"`}

    If `visible` is `"off"`, the figure is not rendered on screen.

`windowbuttondownfcn`: string | function handle, def. `[](0x0)`

    See [windowbuttonupfcn property], page 369.

`windowbuttonmotionfcn`: string | function handle, def. `[](0x0)`

    See [windowbuttonupfcn property], page 369.

`windowbuttonupfcn`: string | function handle, def. `[](0x0)`

    With `windowbuttondownfcn` and `windowbuttonmotionfcn`, the mouse callback functions. These callback functions are called when a mouse button is pressed, dragged, or released respectively. When these callback functions are executed, the `currentpoint` property holds the current coordinates of the cursor.

`windowkeypressfcn`: string | function handle, def. `[](0x0)`
`windowkeyreleasefcn`: string | function handle, def. `[](0x0)`
`windowscrollwheelfcn`: string | function handle, def. `[](0x0)`
`windowstyle`: `"docked"` | `"modal"` | {`"normal"`}
`wvisual`: def. `""`
`wvisualmode`: {`"auto"`} | `"manual"`
`xdisplay`: def. `""`
`xvisual`: def. `""`
`xvisualmode`: {`"auto"`} | `"manual"`

15.3.3.3 Axes Properties

The `axes` properties are:

`__modified__`: `"off"` | {`"on"`}
`activepositionproperty`: {`"outerposition"`} | `"position"`
`alim`: def. `[0 1]`

    Transparency is not yet implemented for axes objects. `alim` is unused.

`alimmode`: {`"auto"`} | `"manual"`
`ambientlightcolor`: def. `[1 1 1]`

    Light is not yet implemented for axes objects. `ambientlightcolor` is unused.

`beingdeleted`: {`"off"`} | `"on"`
`box`: `"off"` | {`"on"`}

    Control whether the axes has a surrounding box.

`busyaction`: `"cancel"` | {`"queue"`}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`cameraposition`: three-element vector, def. `[0.50000 0.50000 9.16025]`
`camerapositionmode`: {`"auto"`} | `"manual"`
`cameratarget`: three-element vector, def. `[0.50000 0.50000 0.50000]`
`cameratargetmode`: {`"auto"`} | `"manual"`
`cameraupvector`: three-element vector, def. `[-0 1 0]`
`cameraupvectormode`: {`"auto"`} | `"manual"`
`cameraviewangle`: scalar, def. `6.6086`
`cameraviewanglemode`: {`"auto"`} | `"manual"`
`children` (read-only): vector of graphics handles, def. `[](0x1)`

    Graphics handles of the axes's children.

`clim`: two-element vector, def. `[0 1]`

    Define the limits for the color axis of image children. Setting `clim` also forces the `climmode` property to be set to `"manual"`. See [pcolor function], page 296.

`climmode`: {`"auto"`} | `"manual"`
`clipping`: `"off"` | {`"on"`}

    `clipping` is unused.

`color`: colorspec, def. `[1 1 1]`

    Color of the axes background. See Section 15.4.1 [colorspec], page 387.

`colororder`: N-by-3 RGB matrix, def. 7-by-3 double

    RGB values used by plot function for automatic line coloring.

`createfcn`: string | function handle, def. `[](0x0)`

    Callback function executed immediately after axes has been created. Function is set by using default property on root object, e.g., `set (0, "defaultaxescreatefcn", 'disp ("axes created!")')`.

`currentpoint`: 2-by-3 matrix, def. 2-by-3 double

    Matrix `[xf, yf, zf; xb, yb, zb]` which holds the coordinates (in axes data units) of the point over which the mouse pointer was when the mouse button was pressed. If a mouse callback function is defined, `currentpoint` holds the pointer coordinates at the time the mouse button was pressed. For 3-D plots, the first row of the returned matrix specifies the point nearest to the current camera position and the second row the furthest point. The two points forms a line which is perpendicular to the screen.

`dataaspectratio`: three-element vector, def. `[1 1 1]`
: Specify the relative height and width of the data displayed in the axes. Setting `dataaspectratio` to `[1, 2]` causes the length of one unit as displayed on the x-axis to be the same as the length of 2 units on the y-axis. Setting `dataaspectratio` also forces the `dataaspectratiomode` property to be set to `"manual"`.

`dataaspectratiomode`: {`"auto"`} | `"manual"`

`deletefcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately before axes is deleted.

`drawmode`: `"fast"` | {`"normal"`}

`fontangle`: `"italic"` | {`"normal"`} | `"oblique"`

`fontname`: string, def. `"*"`
: Name of the font used for axes annotations.

`fontsize`: scalar, def. `10`
: Size of the font used for axes annotations. See [fontunits property], page 371.

`fontunits`: `"centimeters"` | `"inches"` | `"normalized"` | `"pixels"` | {`"points"`}
: Unit used to interpret `fontsize` property.

`fontweight`: `"bold"` | `"demi"` | `"light"` | {`"normal"`}

`gridlinestyle`: `"-"` | `"--"` | `"-."` | {`":"`} | `"none"`

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}
: If `handlevisibility` is `"off"`, the axes's handle is not visible in its parent's "children" property.

`hittest`: `"off"` | {`"on"`}

`interpreter`: `"latex"` | {`"none"`} | `"tex"`

`interruptible`: `"off"` | {`"on"`}

`layer`: {`"bottom"`} | `"top"`
: Control whether the axes is drawn below child graphics objects (ticks, labels, etc. covered by plotted objects) or above.

`linestyleorder`: def. `"-"`

`linewidth`: def. `0.50000`

`minorgridlinestyle`: `"-"` | `"--"` | `"-."` | {`":"`} | `"none"`

`mousewheelzoom`: scalar in the range (0, 1), def. `0.50000`
: Fraction of axes limits to zoom for each wheel movement.

`nextplot`: `"add"` | {`"replace"`} | `"replacechildren"`

`outerposition`: four-element vector, def. `[0 0 1 1]`
: Specify the position of the plot including titles, axes, and legend. The four elements of the vector are the coordinates of the lower left corner and width and height of the plot, in units normalized to the width and height of the plot window. For example, `[0.2, 0.3, 0.4, 0.5]` sets the lower left corner of the

axes at $(0.2, 0.3)$ and the width and height to be 0.4 and 0.5 respectively. See [position property], page 372.

`parent`: graphics handle
: Handle of the parent graphics object.

`plotboxaspectratio`: def. `[1 1 1]`
`plotboxaspectratiomode`: {`"auto"`} | `"manual"`
`position`: four-element vector, def. `[0.13000 0.11000 0.77500 0.81500]`
: Specify the position of the plot excluding titles, axes, and legend. The four elements of the vector are the coordinates of the lower left corner and width and height of the plot, in units normalized to the width and height of the plot window. For example, `[0.2, 0.3, 0.4, 0.5]` sets the lower left corner of the axes at $(0.2, 0.3)$ and the width and height to be 0.4 and 0.5 respectively. See [outerposition property], page 371.

`projection`: {`"orthographic"`} | `"perspective"`
`selected`: {`"off"`} | `"on"`
`selectionhighlight`: `"off"` | {`"on"`}
`tag`: string, def. `""`
: A user-defined string to label the graphics object.

`tickdir`: {`"in"`} | `"out"`
: Control whether axes tick marks project `"in"` to the plot box or `"out"`.

`tickdirmode`: {`"auto"`} | `"manual"`
`ticklength`: two-element vector, def. `[0.010000 0.025000]`
: Two-element vector `[2Dlen 3Dlen]` specifying the length of the tickmarks relative to the longest visible axis.

`tightinset` (read-only): def. `[0.042857 0.038106 0.000000 0.023810]`
`title`: graphics handle
: Graphics handle of the title text object.

`type` (read-only): string
: Class name of the graphics object. `type` is always `"axes"`

`uicontextmenu`: graphics handle, def. `[](0x0)`
: Graphics handle of the uicontextmenu object that is currently associated to this axes object.

`units`: `"centimeters"` | `"characters"` | `"inches"` | {`"normalized"`} | `"pixels"` | `"points"`
`userdata`: Any Octave data, def. `[](0x0)`
: User-defined data to associate with the graphics object.

`view`: two-element vector, def. `[0 90]`
: Two-element vector `[azimuth elevation]` specifying the viewpoint for three-dimensional plots

`visible`: `"off"` | {`"on"`}
: If `visible` is `"off"`, the axes is not rendered on screen.

`xaxislocation`: {`"bottom"`} | `"top"` | `"zero"`
`xcolor`: {colorspec} | `"none"`, def. `[0 0 0]`
Color of the x-axis. See Section 15.4.1 [colorspec], page 387.

`xdir`: {`"normal"`} | `"reverse"`
`xgrid`: {`"off"`} | `"on"`
Control whether major x grid lines are displayed.

`xlabel`: graphics handle
Graphics handle of the x label text object.

`xlim`: two-element vector, def. `[0 1]`
Two-element vector `[xmin xmax]` specifying the limits for the x-axis. Setting `xlim` also forces the `xlimmode` property to be set to `"manual"`. See [xlim function], page 301.

`xlimmode`: {`"auto"`} | `"manual"`
`xminorgrid`: {`"off"`} | `"on"`
Control whether minor x grid lines are displayed.

`xminortick`: {`"off"`} | `"on"`
`xscale`: {`"linear"`} | `"log"`
`xtick`: vector
Position of x tick marks. Setting `xtick` also forces the `xtickmode` property to be set to `"manual"`.

`xticklabel`: string | cell array of strings, def. 1-by-6 cell
Labels of x tick marks. Setting `xticklabel` also forces the `xticklabelmode` property to be set to `"manual"`.

`xticklabelmode`: {`"auto"`} | `"manual"`
`xtickmode`: {`"auto"`} | `"manual"`
`yaxislocation`: {`"left"`} | `"right"` | `"zero"`
`ycolor`: {colorspec} | `"none"`, def. `[0 0 0]`
Color of the y-axis. See Section 15.4.1 [colorspec], page 387.

`ydir`: {`"normal"`} | `"reverse"`
`ygrid`: {`"off"`} | `"on"`
Control whether major y grid lines are displayed.

`ylabel`: graphics handle
Graphics handle of the y label text object.

`ylim`: two-element vector, def. `[0 1]`
Two-element vector `[ymin ymax]` specifying the limits for the y-axis. Setting `ylim` also forces the `ylimmode` property to be set to `"manual"`. See [ylim function], page 301.

`ylimmode`: {`"auto"`} | `"manual"`
`yminorgrid`: {`"off"`} | `"on"`
Control whether minor y grid lines are displayed.

`yminortick`: {"off"} | "on"
`yscale`: {"linear"} | "log"
`ytick`: vector
: Position of y tick marks. Setting `ytick` also forces the `ytickmode` property to be set to "manual".

`yticklabel`: string | cell array of strings, def. 1-by-6 cell
: Labels of y tick marks. Setting `yticklabel` also forces the `yticklabelmode` property to be set to "manual".

`yticklabelmode`: {"auto"} | "manual"
`ytickmode`: {"auto"} | "manual"
`zcolor`: {colorspec} | "none", def. [0 0 0]
: Color of the z-axis. See Section 15.4.1 [colorspec], page 387.

`zdir`: {"normal"} | "reverse"
`zgrid`: {"off"} | "on"
: Control whether major z grid lines are displayed.

`zlabel`: graphics handle
: Graphics handle of the z label text object.

`zlim`: two-element vector, def. [0 1]
: Two-element vector `[zmin zmaz]` specifying the limits for the z-axis. Setting `zlim` also forces the `zlimmode` property to be set to "manual". See [zlim function], page 301.

`zlimmode`: {"auto"} | "manual"
`zminorgrid`: {"off"} | "on"
: Control whether minor z grid lines are displayed.

`zminortick`: {"off"} | "on"
`zscale`: {"linear"} | "log"
`ztick`: vector
: Position of z tick marks. Setting `ztick` also forces the `ztickmode` property to be set to "manual".

`zticklabel`: string | cell array of strings, def. 1-by-6 cell
: Labels of z tick marks. Setting `zticklabel` also forces the `zticklabelmode` property to be set to "manual".

`zticklabelmode`: {"auto"} | "manual"
`ztickmode`: {"auto"} | "manual"

15.3.3.4 Line Properties

The `line` properties are:

`__modified__`: "off" | {"on"}
`beingdeleted`: {"off"} | "on"
`busyaction`: "cancel" | {"queue"}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`children` (read-only): vector of graphics handles, def. `[](0x1)`
: `children` is unused.

`clipping`: `"off"` | {`"on"`}
: If `clipping` is `"on"`, the line is clipped in its parent axes limits.

`color`: colorspec, def. `[0 0 0]`
: Color of the line object. See Section 15.4.1 [colorspec], page 387.

`createfcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately after line has been created. Function is set by using default property on root object, e.g., `set (0, "defaultlinecreatefcn", 'disp ("line created!")')`.

`deletefcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately before line is deleted.

`displayname`: string | cell array of strings, def. `""`
: Text for the legend entry corresponding to this line.

`erasemode`: `"background"` | `"none"` | {`"normal"`} | `"xor"`
: `erasemode` is unused.

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}
: If `handlevisibility` is `"off"`, the line's handle is not visible in its parent's "children" property.

`hittest`: `"off"` | {`"on"`}

`interpreter`: `"latex"` | `"none"` | {`"tex"`}

`interruptible`: `"off"` | {`"on"`}

`linestyle`: {`"-"`} | `"--"` | `"-."` | `":"` | `"none"`
: See Section 15.4.2 [Line Styles], page 387.

`linewidth`: def. `0.50000`
: Width of the line object measured in points.

`marker`: `"*"` | `"+"` | `"."` | `"<"` | `">"` | `"^"` | `"d"` | `"diamond"` | `"h"` | `"hexagram"` | {`"none"`} | `"o"` | `"p"` | `"pentagram"` | `"s"` | `"square"` | `"v"` | `"x"`
: Shape of the marker for each data point. See Section 15.4.3 [Marker Styles], page 387.

`markeredgecolor`: {`"auto"`} | `"none"`
: Color of the edge of the markers. When set to `"auto"`, the marker edges have the same color as the line. If set to `"none"`, no marker edges are displayed. This property can also be set to any color. See Section 15.4.1 [colorspec], page 387.

`markerfacecolor`: `"auto"` | {`"none"`}
: Color of the face of the markers. When set to `"auto"`, the marker faces have the same color as the line. If set to `"none"`, the marker faces are not displayed. This property can also be set to any color. See Section 15.4.1 [colorspec], page 387.

`markersize`: scalar, def. `6`
: Size of the markers measured in points.

`parent`: graphics handle
: Handle of the parent graphics object.

`selected`: {"off"} | "on"
`selectionhighlight`: "off" | {"on"}
`tag`: string, def. ""
A user-defined string to label the graphics object.

`type` (read-only): string
Class name of the graphics object. `type` is always "line"

`uicontextmenu`: graphics handle, def. `[](0x0)`
Graphics handle of the uicontextmenu object that is currently associated to this line object.

`userdata`: Any Octave data, def. `[](0x0)`
User-defined data to associate with the graphics object.

`visible`: "off" | {"on"}
If `visible` is "off", the line is not rendered on screen.

`xdata`: vector, def. `[0 1]`
Vector of x data to be plotted.

`xdatasource`: string, def. ""
Name of a vector in the current base workspace to use as x data.

`ydata`: vector, def. `[0 1]`
Vector of y data to be plotted.

`ydatasource`: string, def. ""
Name of a vector in the current base workspace to use as y data.

`zdata`: vector, def. `[](0x0)`
Vector of z data to be plotted.

`zdatasource`: string, def. ""
Name of a vector in the current base workspace to use as z data.

15.3.3.5 Text Properties

The `text` properties are:

`__modified__`: "off" | {"on"}
`backgroundcolor`: colorspec, def. "none"
Background area is not yet implemented for text objects. `backgroundcolor` is unused.

`beingdeleted`: {"off"} | "on"
`busyaction`: "cancel" | {"queue"}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`children` (read-only): vector of graphics handles, def. `[](0x1)`
`children` is unused.

`clipping`: "off" | {"on"}
If `clipping` is "on", the text is clipped in its parent axes limits.

`color`: colorspec, def. `[0 0 0]`
Color of the text. See Section 15.4.1 [colorspec], page 387.

`createfcn`: string | function handle, def. `[](0x0)`
Callback function executed immediately after text has been created. Function is set by using default property on root object, e.g., `set (0, "defaulttextcreatefcn", 'disp ("text created!")')`.

`deletefcn`: string | function handle, def. `[](0x0)`
Callback function executed immediately before text is deleted.

`displayname`: def. `""`

`edgecolor`: colorspec, def. `"none"`
Background area is not yet implemented for text objects. `edgecolor` is unused.

`editing`: {`"off"`} | `"on"`

`erasemode`: `"background"` | `"none"` | {`"normal"`} | `"xor"`
`erasemode` is unused.

`extent` (read-only): def. `[0.000000 -0.005843 0.000000 0.032136]`

`fontangle`: `"italic"` | {`"normal"`} | `"oblique"`
Flag whether the font is italic or normal. `fontangle` is currently unused.

`fontname`: string, def. `"*"`
The font used for the text.

`fontsize`: scalar, def. `10`
The font size of the text as measured in `fontunits`.

`fontunits`: `"centimeters"` | `"inches"` | `"normalized"` | `"pixels"` | {`"points"`}
The units used to interpret `fontsize` property.

`fontweight`: `"bold"` | `"demi"` | `"light"` | {`"normal"`}
Control variant of base font used: bold, light, normal, etc.

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}
If `handlevisibility` is `"off"`, the text's handle is not visible in its parent's "children" property.

`hittest`: `"off"` | {`"on"`}

`horizontalalignment`: `"center"` | {`"left"`} | `"right"`

`interpreter`: `"latex"` | `"none"` | {`"tex"`}

`interruptible`: `"off"` | {`"on"`}

`linestyle`: {`"-"`} | `"--"` | `"-."` | `":"` | `"none"`
Background area is not yet implemented for text objects. `linestyle` is unused.

`linewidth`: scalar, def. `0.50000`
Background area is not yet implemented for text objects. `linewidth` is unused.

`margin`: scalar, def. `2`
Background area is not yet implemented for text objects. `margin` is unused.

`parent`: graphics handle
Handle of the parent graphics object.

`position`: four-element vector, def. `[0 0 0]`
Vector `[X0 Y0 Z0]` where X0, Y0 and Z0 indicate the position of the text anchor as defined by `verticalalignment` and `horizontalalignment`.

`rotation`: scalar, def. `0`
: The angle of rotation for the displayed text, measured in degrees.

`selected`: {`"off"`} | `"on"`
`selectionhighlight`: `"off"` | {`"on"`}
`string`: string, def. `""`
: The text object string content.

`tag`: string, def. `""`
: A user-defined string to label the graphics object.

`type` (read-only): string
: Class name of the graphics object. `type` is always `"text"`

`uicontextmenu`: graphics handle, def. `[](0x0)`
: Graphics handle of the uicontextmenu object that is currently associated to this text object.

`units`: `"centimeters"` | {`"data"`} | `"inches"` | `"normalized"` | `"pixels"` | `"points"`
`userdata`: Any Octave data, def. `[](0x0)`
: User-defined data to associate with the graphics object.

`verticalalignment`: `"baseline"` | `"bottom"` | `"cap"` | {`"middle"`} | `"top"`
`visible`: `"off"` | {`"on"`}
: If `visible` is `"off"`, the text is not rendered on screen.

15.3.3.6 Image Properties

The `image` properties are:

`__modified__`: `"off"` | {`"on"`}
`alphadata`: scalar | matrix, def. `1`
: Transparency is not yet implemented for image objects. `alphadata` is unused.

`alphadatamapping`: `"direct"` | {`"none"`} | `"scaled"`
: Transparency is not yet implemented for image objects. `alphadatamapping` is unused.

`beingdeleted`: {`"off"`} | `"on"`
`busyaction`: `"cancel"` | {`"queue"`}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`cdata`: matrix, def. 64-by-64 double
`cdatamapping`: {`"direct"`} | `"scaled"`
`children` (read-only): vector of graphics handles, def. `[](0x1)`
: `children` is unused.

`clipping`: `"off"` | {`"on"`}
: If `clipping` is `"on"`, the image is clipped in its parent axes limits.

`createfcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately after image has been created. Function is set by using default property on root object, e.g., `set (0, "defaultimagecreatefcn", 'disp ("image created!")')`.

`deletefcn`: string | function handle, def. `[](0x0)`
    Callback function executed immediately before image is deleted.

`displayname`: string | cell array of strings, def. `""`
    Text for the legend entry corresponding to this image.

`erasemode`: `"background"` | `"none"` | {`"normal"`} | `"xor"`
    `erasemode` is unused.

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}
    If `handlevisibility` is `"off"`, the image's handle is not visible in its parent's "children" property.

`hittest`: `"off"` | {`"on"`}
`interruptible`: `"off"` | {`"on"`}
`parent`: graphics handle
    Handle of the parent graphics object.

`selected`: {`"off"`} | `"on"`
`selectionhighlight`: `"off"` | {`"on"`}
`tag`: string, def. `""`
    A user-defined string to label the graphics object.

`type` (read-only): string
    Class name of the graphics object. `type` is always `"image"`

`uicontextmenu`: graphics handle, def. `[](0x0)`
    Graphics handle of the uicontextmenu object that is currently associated to this image object.

`userdata`: Any Octave data, def. `[](0x0)`
    User-defined data to associate with the graphics object.

`visible`: `"off"` | {`"on"`}
    If `visible` is `"off"`, the image is not rendered on screen.

`xdata`: two-element vector, def. `[1 64]`
    Two-element vector `[xmin xmax]` specifying the x coordinates of the first and last columns of the image.

    Setting `xdata` to the empty matrix ([]) will restore the default value of `[1 columns(image)]`.

`ydata`: two-element vector, def. `[1 64]`
    Two-element vector `[ymin ymax]` specifying the y coordinates of the first and last rows of the image.

    Setting `ydata` to the empty matrix ([]) will restore the default value of `[1 rows(image)]`.

15.3.3.7 Patch Properties

The `patch` properties are:

`__modified__`: `"off"` | {`"on"`}
`alphadatamapping`: `"direct"` | `"none"` | {`"scaled"`}
    Transparency is not yet implemented for patch objects. `alphadatamapping` is unused.

`ambientstrength`: scalar, def. `0.30000`
: Light is not yet implemented for patch objects. `ambientstrength` is unused.

`backfacelighting`: `"lit"` | {`"reverselit"`} | `"unlit"`
: Light is not yet implemented for patch objects. `backfacelighting` is unused.

`beingdeleted`: {`"off"`} | `"on"`
`busyaction`: `"cancel"` | {`"queue"`}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`cdata`: scalar | matrix, def. `[](0x0)`
: Data defining the patch object color. Patch color can be defined for faces or for vertices.

If `cdata` is a scalar index into the current colormap or a RGB triplet, it defines the color of all faces.

If `cdata` is an N-by-1 vector of indices or an N-by-3 (RGB) matrix, it defines the color of each one of the N faces.

If `cdata` is an N-by-M or an N-by-M-by-3 (RGB) matrix, it defines the color at each vertex.

`cdatamapping`: `"direct"` | {`"scaled"`}
`children` (read-only): vector of graphics handles, def. `[](0x1)`
: `children` is unused.

`clipping`: `"off"` | {`"on"`}
: If `clipping` is `"on"`, the patch is clipped in its parent axes limits.

`createfcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately after patch has been created. Function is set by using default property on root object, e.g., `set (0, "defaultpatchcreatefcn", 'disp ("patch created!")')`.

`deletefcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately before patch is deleted.

`diffusestrength`: scalar, def. `0.60000`
: Light is not yet implemented for patch objects. `diffusestrength` is unused.

`displayname`: def. `""`
: Text of the legend entry corresponding to this patch.

`edgealpha`: scalar | matrix, def. `1`
: Transparency is not yet implemented for patch objects. `edgealpha` is unused.

`edgecolor`: def. `[0 0 0]`
`edgelighting`: `"flat"` | `"gouraud"` | {`"none"`} | `"phong"`
: Light is not yet implemented for patch objects. `edgelighting` is unused.

`erasemode`: `"background"` | `"none"` | {`"normal"`} | `"xor"`
: `erasemode` is unused.

`facealpha`: scalar | matrix, def. `1`
: Transparency is not yet implemented for patch objects. `facealpha` is unused.

`facecolor`: {colorspec} | `"none"` | `"flat"` | `"interp"`, def. `[0 0 0]`

`facelighting`: `"flat"` | `"gouraud"` | {`"none"`} | `"phong"`

Light is not yet implemented for patch objects. `facelighting` is unused.

`faces`: def. `[1 2 3]`

`facevertexalphadata`: scalar | matrix, def. `[](0x0)`

Transparency is not yet implemented for patch objects. `facevertexalphadata` is unused.

`facevertexcdata`: def. `[](0x0)`

`handlevisibility`: `"callback"` | `"off"` | {`"on"`}

If `handlevisibility` is `"off"`, the patch's handle is not visible in its parent's `"children"` property.

`hittest`: `"off"` | {`"on"`}

`interpreter`: `"latex"` | `"none"` | {`"tex"`}

`interpreter` is unused.

`interruptible`: `"off"` | {`"on"`}

`linestyle`: {`"-"`} | `"--"` | `"-."` | `":"` | `"none"`

`linewidth`: def. `0.50000`

`marker`: `"*"` | `"+"` | `"."` | `"<"` | `">"` | `"^"` | `"d"` | `"diamond"` | `"h"` | `"hexagram"` | {`"none"`} | `"o"` | `"p"` | `"pentagram"` | `"s"` | `"square"` | `"v"` | `"x"`

See [line marker property], page 375.

`markeredgecolor`: {`"auto"`} | `"flat"` | `"none"`

See [line markeredgecolor property], page 375.

`markerfacecolor`: `"auto"` | `"flat"` | {`"none"`}

See [line markerfacecolor property], page 375.

`markersize`: scalar, def. `6`

See [line markersize property], page 375.

`normalmode`: {`"auto"`} | `"manual"`

`parent`: graphics handle

Handle of the parent graphics object.

`selected`: {`"off"`} | `"on"`

`selectionhighlight`: `"off"` | {`"on"`}

`specularcolorreflectance`: scalar, def. `1`

Light is not yet implemented for patch objects. `specularcolorreflectance` is unused.

`specularexponent`: scalar, def. `10`

Light is not yet implemented for patch objects. `specularexponent` is unused.

`specularstrength`: scalar, def. `0.90000`

Light is not yet implemented for patch objects. `specularstrength` is unused.

`tag`: string, def. `""`

A user-defined string to label the graphics object.

`type` (read-only): string

Class name of the graphics object. `type` is always `"patch"`

`uicontextmenu`: graphics handle, def. `[](0x0)`
: Graphics handle of the uicontextmenu object that is currently associated to this patch object.

`userdata`: Any Octave data, def. `[](0x0)`
: User-defined data to associate with the graphics object.

`vertexnormals`: def. `[](0x0)`
`vertices`: vector | matrix, def. 3-by-2 double
`visible`: `"off"` | {`"on"`}
: If `visible` is `"off"`, the patch is not rendered on screen.

`xdata`: vector | matrix, def. `[0; 1; 0]`
`ydata`: vector | matrix, def. `[1; 1; 0]`
`zdata`: vector | matrix, def. `[](0x0)`

15.3.3.8 Surface Properties

The `surface` properties are:

`__modified__`: `"off"` | {`"on"`}
`alphadata`: scalar | matrix, def. `1`
: Transparency is not yet implemented for surface objects. `alphadata` is unused.

`alphadatamapping`: `"direct"` | `"none"` | {`"scaled"`}
: Transparency is not yet implemented for surface objects. `alphadatamapping` is unused.

`ambientstrength`: def. `0.30000`
: Light is not yet implemented for surface objects. `ambientstrength` is unused.

`backfacelighting`: `"lit"` | {`"reverselit"`} | `"unlit"`
: Light is not yet implemented for surface objects. `backfacelighting` is unused.

`beingdeleted`: {`"off"`} | `"on"`
`busyaction`: `"cancel"` | {`"queue"`}
`buttondownfcn`: string | function handle, def. `[](0x0)`
`cdata`: matrix, def. 3-by-3 double
`cdatamapping`: `"direct"` | {`"scaled"`}
`cdatasource`: def. `""`
`children` (read-only): vector of graphics handles, def. `[](0x1)`
: `children` is unused.

`clipping`: `"off"` | {`"on"`}
: If `clipping` is `"on"`, the surface is clipped in its parent axes limits.

`createfcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately after surface has been created. Function is set by using default property on root object, e.g., `set (0, "defaultsurfacecreatefcn", 'disp ("surface created!")')`.

`deletefcn`: string | function handle, def. `[](0x0)`
: Callback function executed immediately before surface is deleted.

diffusestrength: def. 0.60000
: Light is not yet implemented for surface objects. diffusestrength is unused.

displayname: def. ""
: Text for the legend entry corresponding to this surface.

edgealpha: scalar, def. 1
: Transparency is not yet implemented for surface objects. edgealpha is unused.

edgecolor: def. [0 0 0]

edgelighting: "flat" | "gouraud" | {"none"} | "phong"
: Light is not yet implemented for surface objects. edgelighting is unused.

erasemode: "background" | "none" | {"normal"} | "xor"
: erasemode is unused.

facealpha: scalar | matrix, def. 1
: Transparency is not yet implemented for surface objects. facealpha is unused.

facecolor: {"flat"} | "interp" | "none" | "texturemap"

facelighting: "flat" | "gouraud" | {"none"} | "phong"
: Light is not yet implemented for surface objects. facelighting is unused.

handlevisibility: "callback" | "off" | {"on"}
: If handlevisibility is "off", the surface's handle is not visible in its parent's "children" property.

hittest: "off" | {"on"}

interpreter: "latex" | "none" | {"tex"}

interruptible: "off" | {"on"}

linestyle: {"-"} | "--" | "-." | ":" | "none"
: See Section 15.4.2 [Line Styles], page 387.

linewidth: def. 0.50000
: See [line linewidth property], page 375.

marker: "*" | "+" | "." | "<" | ">" | "^" | "d" | "diamond" | "h" | "hexagram" | {"none"} | "o" | "p" | "pentagram" | "s" | "square" | "v" | "x"
: See Section 15.4.3 [Marker Styles], page 387.

markeredgecolor: {"auto"} | "flat" | "none"
: See [line markeredgecolor property], page 375.

markerfacecolor: "auto" | "flat" | {"none"}
: See [line markerfacecolor property], page 375.

markersize: scalar, def. 6
: See [line markersize property], page 375.

meshstyle: {"both"} | "column" | "row"

normalmode: {"auto"} | "manual"

parent: graphics handle
: Handle of the parent graphics object.

`selected`: {"off"} | "on"
`selectionhighlight`: "off" | {"on"}
`specularcolorreflectance`: def. 1

Light is not yet implemented for surface objects. `specularcolorreflectance` is unused.

`specularexponent`: def. 10

Light is not yet implemented for surface objects. `specularexponent` is unused.

`specularstrength`: def. 0.90000

Light is not yet implemented for surface objects. `specularstrength` is unused.

`tag`: string, def. ""

A user-defined string to label the graphics object.

`type` (read-only): string

Class name of the graphics object. `type` is always "surface"

`uicontextmenu`: graphics handle, def. `[](0x0)`

Graphics handle of the uicontextmenu object that is currently associated to this surface object.

`userdata`: Any Octave data, def. `[](0x0)`

User-defined data to associate with the graphics object.

`vertexnormals`: def. 3-by-3-by-3 double
`visible`: "off" | {"on"}

If `visible` is "off", the surface is not rendered on screen.

`xdata`: matrix, def. `[1 2 3]`
`xdatasource`: def. ""
`ydata`: matrix, def. `[1; 2; 3]`
`ydatasource`: def. ""
`zdata`: matrix, def. 3-by-3 double
`zdatasource`: def. ""

15.3.4 Searching Properties

h = `findobj` () [Function File]
h = `findobj` (*prop_name*, *prop_value*, ...) [Function File]
h = `findobj` (*prop_name*, *prop_value*, "*-logical_op*", *prop_name*, *prop_value*) [Function File]
h = `findobj` ("*-property*", *prop_name*) [Function File]
h = `findobj` ("*-regexp*", *prop_name*, *pattern*) [Function File]
h = `findobj` (*hlist*, ...) [Function File]
h = `findobj` (*hlist*, "*flat*", ...) [Function File]
h = `findobj` (*hlist*, "*-depth*", *d*, ...) [Function File]

Find graphics object with specified property values.

The simplest form is

```
findobj (prop_name, prop_value)
```

which returns the handles of all objects which have a property named *prop_name* that has the value *prop_value*. If multiple property/value pairs are specified then only objects meeting all of the conditions are returned.

The search can be limited to a particular set of objects and their descendants, by passing a handle or set of handles *hlist* as the first argument.

The depth of the object hierarchy to search can be limited with the `"-depth"` argument. An example of searching only three generations of children is:

```
findobj (hlist, "-depth", 3, prop_name, prop_value)
```

Specifying a depth *d* of 0, limits the search to the set of objects passed in *hlist*. A depth *d* of 0 is equivalent to the `"flat"` argument.

A specified logical operator may be applied to the pairs of *prop_name* and *prop_value*. The supported logical operators are: `"-and"`, `"-or"`, `"-xor"`, `"-not"`.

Objects may also be matched by comparing a regular expression to the property values, where property values that match `regexp (prop_value, pattern)` are returned.

Finally, objects may be matched by property name only by using the `"-property"` option.

Implementation Note: The search only includes objects with visible handles (HandleVisibility = `"on"`). See [findall], page 385, to search for all objects including hidden ones.

See also: [findall], page 385, [allchild], page 363, [get], page 361, [set], page 361.

`h = findall ()` [Function File]
`h = findall (prop_name, prop_value, ...)` [Function File]
`h = findall (prop_name, prop_value, "-logical_op", prop_name, prop_value)` [Function File]
`h = findall ("-property", prop_name)` [Function File]
`h = findall ("-regexp", prop_name, pattern)` [Function File]
`h = findall (hlist, ...)` [Function File]
`h = findall (hlist, "flat", ...)` [Function File]
`h = findall (hlist, "-depth", d, ...)` [Function File]

Find graphics object, including hidden ones, with specified property values.

The return value *h* is a list of handles to the found graphic objects.

`findall` performs the same search as `findobj`, but it includes hidden objects (HandleVisibility = `"off"`). For full documentation, see [findobj], page 384.

See also: [findobj], page 384, [allchild], page 363, [get], page 361, [set], page 361.

15.3.5 Managing Default Properties

Object properties have two classes of default values, *factory defaults* (the initial values) and *user-defined defaults*, which may override the factory defaults.

Although default values may be set for any object, they are set in parent objects and apply to child objects, of the specified object type. For example, setting the default `color` property of `line` objects to `"green"`, for the `root` object, will result in all `line` objects inheriting the `color` `"green"` as the default value.

```
set (0, "defaultlinecolor", "green");
```

sets the default line color for all objects. The rule for constructing the property name to set a default value is

```
default + object-type + property-name
```

This rule can lead to some strange looking names, for example `defaultlinelinewidth"` specifies the default `linewidth` property for `line` objects.

The example above used the root figure object, 0, so the default property value will apply to all line objects. However, default values are hierarchical, so defaults set in a figure objects override those set in the root figure object. Likewise, defaults set in axes objects override those set in figure or root figure objects. For example,

```
subplot (2, 1, 1);
set (0, "defaultlinecolor", "red");
set (1, "defaultlinecolor", "green");
set (gca (), "defaultlinecolor", "blue");
line (1:10, rand (1, 10));
subplot (2, 1, 2);
line (1:10, rand (1, 10));
figure (2)
line (1:10, rand (1, 10));
```

produces two figures. The line in first subplot window of the first figure is blue because it inherits its color from its parent axes object. The line in the second subplot window of the first figure is green because it inherits its color from its parent figure object. The line in the second figure window is red because it inherits its color from the global root figure parent object.

To remove a user-defined default setting, set the default property to the value `"remove"`. For example,

```
set (gca (), "defaultlinecolor", "remove");
```

removes the user-defined default line color setting from the current axes object. To quickly remove all user-defined defaults use the `reset` function.

`reset (`*h*`)` [Built-in Function]

Reset the properties of the graphic object *h* to their default values.

For figures, the properties `"position"`, `"units"`, `"windowstyle"`, and `"paperunits"` are not affected. For axes, the properties `"position"` and `"units"` are not affected.

The input *h* may also be a vector of graphic handles in which case each individual object will be reset.

See also: [cla], page 340, [clf], page 340, [newplot], page 338.

Getting the `"default"` property of an object returns a list of user-defined defaults set for the object. For example,

```
get (gca (), "default");
```

returns a list of user-defined default values for the current axes object.

Factory default values are stored in the root figure object. The command

```
get (0, "factory");
```

returns a list of factory defaults.

15.4 Advanced Plotting

15.4.1 Colors

Colors may be specified as RGB triplets with values ranging from zero to one, or by name. Recognized color names include `"blue"`, `"black"`, `"cyan"`, `"green"`, `"magenta"`, `"red"`, `"white"`, and `"yellow"`.

15.4.2 Line Styles

Line styles are specified by the following properties:

`linestyle`
: May be one of

  `"-"` Solid line. [default]

  `"--"` Dashed line.

  `":"` Dotted line.

  `"-."` A dash-dot line.

  `"none"` No line. Points will still be marked using the current Marker Style.

`linewidth`
: A number specifying the width of the line. The default is 1. A value of 2 is twice as wide as the default, etc.

15.4.3 Marker Styles

Marker styles are specified by the following properties:

`marker` A character indicating a plot marker to be place at each data point, or `"none"`, meaning no markers should be displayed.

`markeredgecolor`
: The color of the edge around the marker, or `"auto"`, meaning that the edge color is the same as the face color. See Section 15.4.1 [Colors], page 387.

`markerfacecolor`
: The color of the marker, or `"none"` to indicate that the marker should not be filled. See Section 15.4.1 [Colors], page 387.

`markersize`
: A number specifying the size of the marker. The default is 1. A value of 2 is twice as large as the default, etc.

The `colstyle` function will parse a `plot`-style specification and will return the color, line, and marker values that would result.

`[`*`style`*`, `*`color`*`, `*`marker`*`, `*`msg`*`] = colstyle (`*`linespec`*`)` [Function File]

Parse *linespec* and return the line style, color, and markers given.

In the case of an error, the string *msg* will return the text of the error.

15.4.4 Callbacks

Callback functions can be associated with graphics objects and triggered after certain events occur. The basic structure of all callback function is

```
function mycallback (src, data)
...
endfunction
```

where `src` gives a handle to the source of the callback, and `code` gives some event specific data. This can then be associated with an object either at the objects creation or later with the `set` function. For example,

```
plot (x, "DeleteFcn", @(s, e) disp ("Window Deleted"))
```

where at the moment that the plot is deleted, the message "Window Deleted" will be displayed.

Additional user arguments can be passed to callback functions, and will be passed after the 2 default arguments. For example:

```
plot (x, "DeleteFcn", {@mycallback, "1"})
...
function mycallback (src, data, a1)
  fprintf ("Closing plot %d\n", a1);
endfunction
```

The basic callback functions that are available for all graphics objects are

- CreateFcn This is the callback that is called at the moment of the objects creation. It is not called if the object is altered in any way, and so it only makes sense to define this callback in the function call that defines the object. Callbacks that are added to `CreateFcn` later with the `set` function will never be executed.
- DeleteFcn This is the callback that is called at the moment an object is deleted.
- ButtonDownFcn This is the callback that is called if a mouse button is pressed while the pointer is over this object. Note, that the gnuplot interface does not respect this callback.

The object and figure that the event occurred in that resulted in the callback being called can be found with the `gcbo` and `gcbf` functions.

h = gcbo () [Function File]
[*h*, *fig*] = gcbo () [Function File]

Return a handle to the object whose callback is currently executing.

If no callback is executing, this function returns the empty matrix. This handle is obtained from the root object property `"CallbackObject"`.

When called with a second output argument, return the handle of the figure containing the object whose callback is currently executing. If no callback is executing the second output is also set to the empty matrix.

See also: [gcbf], page 389, [gco], page 360, [gca], page 360, [gcf], page 359, [get], page 361, [set], page 361.

fig = gcbf () [Function File]

Return a handle to the figure containing the object whose callback is currently executing.

If no callback is executing, this function returns the empty matrix. The handle returned by this function is the same as the second output argument of `gcbo`.

See also: [gcbo], page 388, [gcf], page 359, [gco], page 360, [gca], page 360, [get], page 361, [set], page 361.

Callbacks can equally be added to properties with the `addlistener` function described below.

15.4.5 Application-defined Data

Octave has a provision for attaching application-defined data to a graphics handle. The data can be anything which is meaningful to the application, and will be completely ignored by Octave.

setappdata (*h*, *name*, *value*) [Function File]
setappdata (*h*, *name1*, *value1*, *name2*, *value3*, . . .) [Function File]

Set the application data *name* to *value* for the graphics object with handle *h*.

h may also be a vector of graphics handles. If the application data with the specified *name* does not exist, it is created. Multiple *name*/*value* pairs can be specified at a time.

See also: [getappdata], page 389, [isappdata], page 389, [rmappdata], page 389, [guidata], page 744, [get], page 361, [set], page 361, [getpref], page 746, [setpref], page 747.

value = getappdata (*h*, *name*) [Function File]
appdata = getappdata (*h*) [Function File]

Return the *value* of the application data *name* for the graphics object with handle *h*.

h may also be a vector of graphics handles. If no second argument *name* is given then `getappdata` returns a structure, *appdata*, whose fields correspond to the appdata properties.

See also: [setappdata], page 389, [isappdata], page 389, [rmappdata], page 389, [guidata], page 744, [get], page 361, [set], page 361, [getpref], page 746, [setpref], page 747.

rmappdata (*h*, *name*) [Function File]
rmappdata (*h*, *name1*, *name2*, . . .) [Function File]

Delete the application data *name* from the graphics object with handle *h*.

h may also be a vector of graphics handles. Multiple application data names may be supplied to delete several properties at once.

See also: [setappdata], page 389, [getappdata], page 389, [isappdata], page 389.

valid = isappdata (*h*, *name*) [Function File]

Return true if the named application data, *name*, exists for the graphics object with handle *h*.

h may also be a vector of graphics handles.

See also: [getappdata], page 389, [setappdata], page 389, [rmappdata], page 389, [guidata], page 744, [get], page 361, [set], page 361, [getpref], page 746, [setpref], page 747.

15.4.6 Object Groups

A number of Octave high level plot functions return groups of other graphics objects or they return graphics objects that have their properties linked in such a way that changes to one of the properties results in changes in the others. A graphic object that groups other objects is an `hggroup`

`hggroup ()` [Function File]
`hggroup (hax)` [Function File]
`hggroup (..., property, value, ...)` [Function File]
`h = hggroup (...)` [Function File]

Create handle graphics group object with axes parent *hax*.

If no parent is specified, the group is created in the current axes.

Multiple property/value pairs may be specified for the hggroup, but they must appear in pairs.

The optional return value *h* is a graphics handle to the created hggroup object.

Programming Note: An hggroup is a way to group base graphics objects such as line objects or patch objects into a single unit which can react appropriately. For example, the individual lines of a contour plot are collected into a single hggroup so that they can be made visible/invisible with a single command, `set (hg_handle, "visible", "off")`.

See also: [addproperty], page 390, [addlistener], page 391.

For example a simple use of a `hggroup` might be

```
x = 0:0.1:10;
hg = hggroup ();
plot (x, sin (x), "color", [1, 0, 0], "parent", hg);
hold on
plot (x, cos (x), "color", [0, 1, 0], "parent", hg);
set (hg, "visible", "off");
```

which groups the two plots into a single object and controls their visibility directly. The default properties of an `hggroup` are the same as the set of common properties for the other graphics objects. Additional properties can be added with the `addproperty` function.

`addproperty (name, h, type)` [Built-in Function]
`addproperty (name, h, type, arg, ...)` [Built-in Function]

Create a new property named *name* in graphics object *h*.

type determines the type of the property to create. *args* usually contains the default value of the property, but additional arguments might be given, depending on the type of the property.

The supported property types are:

`string` A string property. *arg* contains the default string value.

`any` An un-typed property. This kind of property can hold any octave value. *args* contains the default value.

`radio` A string property with a limited set of accepted values. The first argument must be a string with all accepted values separated by a vertical bar ('`|`'). The default value can be marked by enclosing it with a '{' '}' pair. The default value may also be given as an optional second string argument.

`boolean` A boolean property. This property type is equivalent to a radio property with "on|off" as accepted values. *arg* contains the default property value.

`double` A scalar double property. *arg* contains the default value.

`handle` A handle property. This kind of property holds the handle of a graphics object. *arg* contains the default handle value. When no default value is given, the property is initialized to the empty matrix.

`data` A data (matrix) property. *arg* contains the default data value. When no default value is given, the data is initialized to the empty matrix.

`color` A color property. *arg* contains the default color value. When no default color is given, the property is set to black. An optional second string argument may be given to specify an additional set of accepted string values (like a radio property).

type may also be the concatenation of a core object type and a valid property name for that object type. The property created then has the same characteristics as the referenced property (type, possible values, hidden state...). This allows one to clone an existing property into the graphics object *h*.

Examples:

```
addproperty ("my_property", gcf, "string", "a string value");
addproperty ("my_radio", gcf, "radio", "val_1|val_2|{val_3}");
addproperty ("my_style", gcf, "linelinestyle", "--");
```

See also: [addlistener], page 391, [hggroup], page 390.

Once a property in added to an `hggroup`, it is not linked to any other property of either the children of the group, or any other graphics object. Add so to control the way in which this newly added property is used, the `addlistener` function is used to define a callback function that is executed when the property is altered.

`addlistener (`*h*`,` *prop*`,` *fcn*`)` [Built-in Function]

Register *fcn* as listener for the property *prop* of the graphics object *h*.

Property listeners are executed (in order of registration) when the property is set. The new value is already available when the listeners are executed.

prop must be a string naming a valid property in *h*.

fcn can be a function handle, a string or a cell array whose first element is a function handle. If *fcn* is a function handle, the corresponding function should accept at least 2 arguments, that will be set to the object handle and the empty matrix respectively. If *fcn* is a string, it must be any valid octave expression. If *fcn* is a cell array, the first element must be a function handle with the same signature as described above. The next elements of the cell array are passed as additional arguments to the function.

Example:

```
function my_listener (h, dummy, p1)
  fprintf ("my_listener called with p1=%s\n", p1);
endfunction

addlistener (gcf, "position", {@my_listener, "my string"})
```

See also: [addproperty], page 390, [hggroup], page 390.

`dellistener` (*h*, *prop*, *fcn*) [Built-in Function]

Remove the registration of *fcn* as a listener for the property *prop* of the graphics object *h*.

The function *fcn* must be the same variable (not just the same value), as was passed to the original call to `addlistener`.

If *fcn* is not defined then all listener functions of *prop* are removed.

Example:

```
function my_listener (h, dummy, p1)
  fprintf ("my_listener called with p1=%s\n", p1);
endfunction

c = {@my_listener, "my string"};
addlistener (gcf, "position", c);
dellistener (gcf, "position", c);
```

An example of the use of these two functions might be

```
x = 0:0.1:10;
hg = hggroup ();
h = plot (x, sin (x), "color", [1, 0, 0], "parent", hg);
addproperty ("linestyle", hg, "linelinestyle", get (h, "linestyle"));
addlistener (hg, "linestyle", @update_props);
hold on
plot (x, cos (x), "color", [0, 1, 0], "parent", hg);

function update_props (h, d)
  set (get (h, "children"), "linestyle", get (h, "linestyle"));
endfunction
```

that adds a `linestyle` property to the `hggroup` and propagating any changes its value to the children of the group. The `linkprop` function can be used to simplify the above to be

```
x = 0:0.1:10;
hg = hggroup ();
h1 = plot (x, sin (x), "color", [1, 0, 0], "parent", hg);
addproperty ("linestyle", hg, "linelinestyle", get (h, "linestyle"));
hold on
h2 = plot (x, cos (x), "color", [0, 1, 0], "parent", hg);
hlink = linkprop ([hg, h1, h2], "color");
```

hlink = linkprop (*h*, "*prop*") [Function File]
hlink = linkprop (*h*, {"*prop1*", "*prop2*", ...}) [Function File]

Link graphic object properties, such that a change in one is propagated to the others.

The input *h* is a vector of graphic handles to link.

prop may be a string when linking a single property, or a cell array of strings for multiple properties. During the linking process all properties in *prop* will initially be set to the values that exist on the first object in the list *h*.

The function returns *hlink* which is a special object describing the link. As long as the reference *hlink* exists the link between graphic objects will be active. This means that *hlink* must be preserved in a workspace variable, a global variable, or otherwise stored using a function such as `setappdata`, `guidata`. To unlink properties, execute `clear hlink`.

An example of the use of `linkprop` is

```
x = 0:0.1:10;
subplot (1,2,1);
h1 = plot (x, sin (x));
subplot (1,2,2);
h2 = plot (x, cos (x));
hlink = linkprop ([h1, h2], {"color","linestyle"});
set (h1, "color", "green");
set (h2, "linestyle", "--");
```

See also: [linkaxes], page 393.

linkaxes (*hax*) [Function File]
linkaxes (*hax*, *optstr*) [Function File]

Link the axis limits of 2-D plots such that a change in one is propagated to the others.

The axes handles to be linked are passed as the first argument *hax*.

The optional second argument is a string which defines which axis limits will be linked. The possible values for *optstr* are:

`"x"` Link x-axes

`"y"` Link y-axes

`"xy"` (default)
Link both axes

`"off"` Turn off linking

If unspecified the default is to link both X and Y axes.

When linking, the limits from the first axes in *hax* are applied to the other axes in the list. Subsequent changes to any one of the axes will be propagated to the others.

See also: [linkprop], page 392, [addproperty], page 390.

These capabilities are used in a number of basic graphics objects. The `hggroup` objects created by the functions of Octave contain one or more graphics object and are used to:

- group together multiple graphics objects,

- create linked properties between different graphics objects, and
- to hide the nominal user data, from the actual data of the objects.

For example the `stem` function creates a stem series where each `hggroup` of the stem series contains two line objects representing the body and head of the stem. The `ydata` property of the `hggroup` of the stem series represents the head of the stem, whereas the body of the stem is between the baseline and this value. For example

```
h = stem (1:4)
get (h, "xdata")
⇒ [  1   2   3   4]'
get (get (h, "children")(1), "xdata")
⇒ [  1   1 NaN   2   2 NaN   3   3 NaN   4   4 NaN]'
```

shows the difference between the `xdata` of the `hggroup` of a stem series object and the underlying line.

The basic properties of such group objects is that they consist of one or more linked `hggroup`, and that changes in certain properties of these groups are propagated to other members of the group. Whereas, certain properties of the members of the group only apply to the current member.

In addition the members of the group can also be linked to other graphics objects through callback functions. For example the baseline of the `bar` or `stem` functions is a line object, whose length and position are automatically adjusted, based on changes to the corresponding hggroup elements.

15.4.6.1 Data Sources in Object Groups

All of the group objects contain data source parameters. There are string parameters that contain an expression that is evaluated to update the relevant data property of the group when the `refreshdata` function is called.

refreshdata () [Function File]
refreshdata (*h*) [Function File]
refreshdata (*h*, *workspace*) [Function File]

Evaluate any '`datasource`' properties of the current figure and update the plot if the corresponding data has changed.

If the first argument *h* is a list of graphic handles, then operate on these objects rather than the current figure returned by `gcf`.

The optional second argument *workspace* can take the following values:

`"base"` Evaluate the datasource properties in the base workspace. (default).

`"caller"` Evaluate the datasource properties in the workspace of the function that called `refreshdata`.

An example of the use of `refreshdata` is:

```
x = 0:0.1:10;
y = sin (x);
plot (x, y, "ydatasource", "y");
for i = 1 : 100
  pause (0.1);
  y = sin (x + 0.1*i);
  refreshdata ();
endfor
```

15.4.6.2 Area Series

Area series objects are created by the `area` function. Each of the `hggroup` elements contains a single patch object. The properties of the area series are

`basevalue`
: The value where the base of the area plot is drawn.

`linewidth`
`linestyle`
: The line width and style of the edge of the patch objects making up the areas. See Section 15.4.2 [Line Styles], page 387.

`edgecolor`
`facecolor`
: The line and fill color of the patch objects making up the areas. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
: The x and y coordinates of the original columns of the data passed to `area` prior to the cumulative summation used in the `area` function.

`xdatasource`
`ydatasource`
: Data source variables.

15.4.6.3 Bar Series

Bar series objects are created by the `bar` or `barh` functions. Each `hggroup` element contains a single patch object. The properties of the bar series are

`showbaseline`
`baseline`
`basevalue`
: The property `showbaseline` flags whether the baseline of the bar series is displayed (default is `"on"`). The handle of the graphics object representing the baseline is given by the `baseline` property and the y-value of the baseline by the `basevalue` property.

  Changes to any of these properties are propagated to the other members of the bar series and to the baseline itself. Equally, changes in the properties of the base line itself are propagated to the members of the corresponding bar series.

`barwidth`
`barlayout`
`horizontal`
The property `barwidth` is the width of the bar corresponding to the *width* variable passed to `bar` or *barh*. Whether the bar series is `"grouped"` or `"stacked"` is determined by the `barlayout` property and whether the bars are horizontal or vertical by the `horizontal` property.

Changes to any of these property are propagated to the other members of the bar series.

`linewidth`
`linestyle`
The line width and style of the edge of the patch objects making up the bars. See Section 15.4.2 [Line Styles], page 387.

`edgecolor`
`facecolor`
The line and fill color of the patch objects making up the bars. See Section 15.4.1 [Colors], page 387.

`xdata` The nominal x positions of the bars. Changes in this property and propagated to the other members of the bar series.

`ydata` The y value of the bars in the `hggroup`.

`xdatasource`
`ydatasource`
Data source variables.

15.4.6.4 Contour Groups

Contour group objects are created by the `contour`, `contourf` and `contour3` functions. The are equally one of the handles returned by the `surfc` and `meshc` functions. The properties of the contour group are

`contourmatrix`
A read only property that contains the data return by `contourc` used to create the contours of the plot.

`fill` A radio property that can have the values `"on"` or `"off"` that flags whether the contours to plot are to be filled.

`zlevelmode`
`zlevel` The radio property `zlevelmode` can have the values `"none"`, `"auto"`, or `"manual"`. When its value is `"none"` there is no z component to the plotted contours. When its value is `"auto"` the z value of the plotted contours is at the same value as the contour itself. If the value is `"manual"`, then the z value at which to plot the contour is determined by the `zlevel` property.

`levellistmode`
`levellist`
`levelstepmode`
`levelstep`
: If `levellistmode` is `"manual"`, then the levels at which to plot the contours is determined by `levellist`. If `levellistmode` is set to `"auto"`, then the distance between contours is determined by `levelstep`. If both `levellistmode` and `levelstepmode` are set to `"auto"`, then there are assumed to be 10 equal spaced contours.

`textlistmode`
`textlist`
`textstepmode`
`textstep`
: If `textlistmode` is `"manual"`, then the labeled contours is determined by `textlist`. If `textlistmode` is set to `"auto"`, then the distance between labeled contours is determined by `textstep`. If both `textlistmode` and `textstepmode` are set to `"auto"`, then there are assumed to be 10 equal spaced labeled contours.

`showtext`
: Flag whether the contour labels are shown or not.

`labelspacing`
: The distance between labels on a single contour in points.

`linewidth`
`linestyle`
`linecolor`
: The properties of the contour lines. The properties `linewidth` and `linestyle` are similar to the corresponding properties for lines. The property `linecolor` is a color property (see Section 15.4.1 [Colors], page 387), that can also have the values of `"none"` or `"auto"`. If `linecolor` is `"none"`, then no contour line is drawn. If `linecolor` is `"auto"` then the line color is determined by the colormap.

`xdata`
`ydata`
`zdata`
: The original x, y, and z data of the contour lines.

`xdatasource`
`ydatasource`
`zdatasource`
: Data source variables.

15.4.6.5 Error Bar Series

Error bar series are created by the `errorbar` function. Each `hggroup` element contains two line objects representing the data and the errorbars separately. The properties of the error bar series are

`color`
: The RGB color or color name of the line objects of the error bars. See Section 15.4.1 [Colors], page 387.

`linewidth`
`linestyle`
: The line width and style of the line objects of the error bars. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markeredgecolor`
`markerfacecolor`
`markersize`
: The line and fill color of the markers on the error bars. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
`ldata`
`udata`
`xldata`
`xudata` The original x, y, l, u, xl, xu data of the error bars.

`xdatasource`
`ydatasource`
`ldatasource`
`udatasource`
`xldatasource`
`xudatasource`
: Data source variables.

15.4.6.6 Line Series

Line series objects are created by the `plot` and `plot3` functions and are of the type `line`. The properties of the line series with the ability to add data sources.

`color` The RGB color or color name of the line objects. See Section 15.4.1 [Colors], page 387.

`linewidth`
`linestyle`
: The line width and style of the line objects. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markeredgecolor`
`markerfacecolor`
`markersize`
: The line and fill color of the markers. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
`zdata` The original x, y and z data.

`xdatasource`
`ydatasource`
`zdatasource`
: Data source variables.

15.4.6.7 Quiver Group

Quiver series objects are created by the `quiver` or `quiver3` functions. Each `hggroup` element of the series contains three line objects as children representing the body and head of the arrow, together with a marker as the point of origin of the arrows. The properties of the quiver series are

`autoscale`
`autoscalefactor`
: Flag whether the length of the arrows is scaled or defined directly from the *u*, *v* and *w* data. If the arrow length is flagged as being scaled by the `autoscale` property, then the length of the autoscaled arrow is controlled by the `autoscalefactor`.

`maxheadsize`
: This property controls the size of the head of the arrows in the quiver series. The default value is 0.2.

`showarrowhead`
: Flag whether the arrow heads are displayed in the quiver plot.

`color`
: The RGB color or color name of the line objects of the quiver. See Section 15.4.1 [Colors], page 387.

`linewidth`
`linestyle`
: The line width and style of the line objects of the quiver. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markerfacecolor`
`markersize`
: The line and fill color of the marker objects at the original of the arrows. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
`zdata`
: The origins of the values of the vector field.

`udata`
`vdata`
`wdata`
: The values of the vector field to plot.

`xdatasource`
`ydatasource`
`zdatasource`
`udatasource`
`vdatasource`
`wdatasource`
: Data source variables.

15.4.6.8 Scatter Group

Scatter series objects are created by the `scatter` or `scatter3` functions. A single hggroup element contains as many children as there are points in the scatter plot, with each child representing one of the points. The properties of the stem series are

`linewidth`
: The line width of the line objects of the points. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markeredgecolor`
`markerfacecolor`
: The line and fill color of the markers of the points. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
`zdata`
: The original x, y and z data of the stems.

`cdata`
: The color data for the points of the plot. Each point can have a separate color, or a unique color can be specified.

`sizedata`
: The size data for the points of the plot. Each point can its own size or a unique size can be specified.

`xdatasource`
`ydatasource`
`zdatasource`
`cdatasource`
`sizedatasource`
: Data source variables.

15.4.6.9 Stair Group

Stair series objects are created by the `stair` function. Each `hggroup` element of the series contains a single line object as a child representing the stair. The properties of the stair series are

`color`
: The RGB color or color name of the line objects of the stairs. See Section 15.4.1 [Colors], page 387.

`linewidth`
`linestyle`
: The line width and style of the line objects of the stairs. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markeredgecolor`
`markerfacecolor`
`markersize`
: The line and fill color of the markers on the stairs. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
: The original x and y data of the stairs.

`xdatasource`
`ydatasource`
: Data source variables.

15.4.6.10 Stem Series

Stem series objects are created by the `stem` or `stem3` functions. Each `hggroup` element contains a single line object as a child representing the stems. The properties of the stem series are

`showbaseline`
`baseline`
`basevalue`
: The property `showbaseline` flags whether the baseline of the stem series is displayed (default is `"on"`). The handle of the graphics object representing the baseline is given by the `baseline` property and the y-value (or z-value for `stem3`) of the baseline by the `basevalue` property.

  Changes to any of these property are propagated to the other members of the stem series and to the baseline itself. Equally changes in the properties of the base line itself are propagated to the members of the corresponding stem series.

`color`
: The RGB color or color name of the line objects of the stems. See Section 15.4.1 [Colors], page 387.

`linewidth`
`linestyle`
: The line width and style of the line objects of the stems. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markeredgecolor`
`markerfacecolor`
`markersize`
: The line and fill color of the markers on the stems. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
`zdata`
: The original x, y and z data of the stems.

`xdatasource`
`ydatasource`
`zdatasource`
Data source variables.

15.4.6.11 Surface Group

Surface group objects are created by the `surf` or `mesh` functions, but are equally one of the handles returned by the `surfc` or `meshc` functions. The surface group is of the type `surface`.

The properties of the surface group are

`edgecolor`
`facecolor`
The RGB color or color name of the edges or faces of the surface. See Section 15.4.1 [Colors], page 387.

`linewidth`
`linestyle`
The line width and style of the lines on the surface. See Section 15.4.2 [Line Styles], page 387.

`marker`
`markeredgecolor`
`markerfacecolor`
`markersize`
The line and fill color of the markers on the surface. See Section 15.4.1 [Colors], page 387.

`xdata`
`ydata`
`zdata`
`cdata` The original x, y, z and c data.

`xdatasource`
`ydatasource`
`zdatasource`
`cdatasource`
Data source variables.

15.4.7 Graphics Toolkits

name = `graphics_toolkit ()` [Function File]
name = `graphics_toolkit (`*hlist*`)` [Function File]
`graphics_toolkit (`*name*`)` [Function File]
`graphics_toolkit (`*hlist*, *name*`)` [Function File]

Query or set the default graphics toolkit which is assigned to new figures.

With no inputs, return the current default graphics toolkit. If the input is a list of figure graphic handles, *hlist*, then return the name of the graphics toolkit in use for each figure.

When called with a single input *name* set the default graphics toolkit to *name*. If the toolkit is not already loaded, it is initialized by calling the function `__init_name__`. If the first input is a list of figure handles, *hlist*, then the graphics toolkit is set to *name* for these figures only.

See also: [available_graphics_toolkits], page 403.

`available_graphics_toolkits ()` [Built-in Function]

Return a cell array of registered graphics toolkits.

See also: [graphics_toolkit], page 402, [register_graphics_toolkit], page 403.

`loaded_graphics_toolkits ()` [Built-in Function]

Return a cell array of the currently loaded graphics toolkits.

See also: [available_graphics_toolkits], page 403.

`register_graphics_toolkit (toolkit)` [Built-in Function]

List *toolkit* as an available graphics toolkit.

See also: [available_graphics_toolkits], page 403.

15.4.7.1 Customizing Toolkit Behavior

The specific behavior of the backend toolkit may be modified using the following utility functions. Note: Not all functions apply to every graphics toolkit.

`[prog, args] = gnuplot_binary ()` [Loadable Function]
`[old_prog, old_args] = gnuplot_binary (new_prog, arg1, ...)` [Loadable Function]

Query or set the name of the program invoked by the plot command when the graphics toolkit is set to "gnuplot".

Additional arguments to pass to the external plotting program may also be given. The default value is `"gnuplot"` with no additional arguments. See Appendix G [Installation], page 893.

See also: [graphics_toolkit], page 402.

16 Matrix Manipulation

There are a number of functions available for checking to see if the elements of a matrix meet some condition, and for rearranging the elements of a matrix. For example, Octave can easily tell you if all the elements of a matrix are finite, or are less than some specified value. Octave can also rotate the elements, extract the upper- or lower-triangular parts, or sort the columns of a matrix.

16.1 Finding Elements and Checking Conditions

The functions `any` and `all` are useful for determining whether any or all of the elements of a matrix satisfy some condition. The `find` function is also useful in determining which elements of a matrix meet a specified condition.

`any` (*x*) [Built-in Function]
`any` (*x*, *dim*) [Built-in Function]

For a vector argument, return true (logical 1) if any element of the vector is nonzero.

For a matrix argument, return a row vector of logical ones and zeros with each element indicating whether any of the elements of the corresponding column of the matrix are nonzero. For example:

```
any (eye (2, 4))
 ⇒ [ 1, 1, 0, 0 ]
```

If the optional argument *dim* is supplied, work along dimension *dim*. For example:

```
any (eye (2, 4), 2)
 ⇒ [ 1; 1 ]
```

See also: [all], page 405.

`all` (*x*) [Built-in Function]
`all` (*x*, *dim*) [Built-in Function]

For a vector argument, return true (logical 1) if all elements of the vector are nonzero.

For a matrix argument, return a row vector of logical ones and zeros with each element indicating whether all of the elements of the corresponding column of the matrix are nonzero. For example:

```
all ([2, 3; 1, 0])
    ⇒ [ 1, 0 ]
```

If the optional argument *dim* is supplied, work along dimension *dim*.

See also: [any], page 405.

Since the comparison operators (see Section 8.4 [Comparison Ops], page 145) return matrices of ones and zeros, it is easy to test a matrix for many things, not just whether the elements are nonzero. For example,

```
all (all (rand (5) < 0.9))
     ⇒ 0
```

tests a random 5 by 5 matrix to see if all of its elements are less than 0.9.

Note that in conditional contexts (like the test clause of `if` and `while` statements) Octave treats the test as if you had typed `all (all (condition))`.

z = `xor` (*x*, *y*) [Function File]
z = `xor` (*x1*, *x2*, ...) [Function File]

Return the *exclusive or* of *x* and *y*.

For boolean expressions *x* and *y*, `xor (x, y)` is true if and only if one of *x* or *y* is true. Otherwise, if *x* and *y* are both true or both false, `xor` returns false.

The truth table for the xor operation is

x	*y*	*z*
-	-	-
0	0	0
1	0	1
0	1	1
1	1	0

If more than two arguments are given the xor operation is applied cumulatively from left to right:

```
(...((x1 XOR x2) XOR x3) XOR ...)
```

See also: [and], page 147, [or], page 147, [not], page 147.

`diff` (*x*) [Built-in Function]
`diff` (*x*, *k*) [Built-in Function]
`diff` (*x*, *k*, *dim*) [Built-in Function]

If *x* is a vector of length n, `diff (x)` is the vector of first differences $x_2 - x_1, \ldots, x_n - x_{n-1}$.

If *x* is a matrix, `diff (x)` is the matrix of column differences along the first non-singleton dimension.

The second argument is optional. If supplied, `diff (x, k)`, where *k* is a non-negative integer, returns the *k*-th differences. It is possible that *k* is larger than the first non-singleton dimension of the matrix. In this case, `diff` continues to take the differences along the next non-singleton dimension.

The dimension along which to take the difference can be explicitly stated with the optional variable *dim*. In this case the *k*-th order differences are calculated along this dimension. In the case where *k* exceeds `size (x, dim)` an empty matrix is returned.

See also: [sort], page 414, [merge], page 149.

`isinf` (*x*) [Mapping Function]

Return a logical array which is true where the elements of *x* are infinite and false where they are not.

For example:

```
isinf ([13, Inf, NA, NaN])
      ⇒ [ 0, 1, 0, 0 ]
```

See also: [isfinite], page 407, [isnan], page 406, [isna], page 43.

`isnan` (*x*) [Mapping Function]

Return a logical array which is true where the elements of *x* are NaN values and false where they are not.

NA values are also considered NaN values. For example:

```
isnan ([13, Inf, NA, NaN])
      ⇒ [ 0, 0, 1, 1 ]
```

See also: [isna], page 43, [isinf], page 406, [isfinite], page 407.

`isfinite (x)` [Mapping Function]

Return a logical array which is true where the elements of *x* are finite values and false where they are not.

For example:

```
isfinite ([13, Inf, NA, NaN])
     ⇒ [ 1, 0, 0, 0 ]
```

See also: [isinf], page 406, [isnan], page 406, [isna], page 43.

`[err, y1, ...] = common_size (x1, ...)` [Function File]

Determine if all input arguments are either scalar or of common size.

If true, *err* is zero, and *yi* is a matrix of the common size with all entries equal to *xi* if this is a scalar or *xi* otherwise. If the inputs cannot be brought to a common size, *err* is 1, and *yi* is *xi*. For example:

```
[errorcode, a, b] = common_size ([1 2; 3 4], 5)
     ⇒ errorcode = 0
     ⇒ a = [ 1, 2; 3, 4 ]
     ⇒ b = [ 5, 5; 5, 5 ]
```

This is useful for implementing functions where arguments can either be scalars or of common size.

`idx = find (x)` [Built-in Function]
`idx = find (x, n)` [Built-in Function]
`idx = find (x, n, direction)` [Built-in Function]
`[i, j] = find (...)` [Built-in Function]
`[i, j, v] = find (...)` [Built-in Function]

Return a vector of indices of nonzero elements of a matrix, as a row if *x* is a row vector or as a column otherwise.

To obtain a single index for each matrix element, Octave pretends that the columns of a matrix form one long vector (like Fortran arrays are stored). For example:

```
find (eye (2))
  ⇒ [ 1; 4 ]
```

If two inputs are given, *n* indicates the maximum number of elements to find from the beginning of the matrix or vector.

If three inputs are given, *direction* should be one of `"first"` or `"last"`, requesting only the first or last *n* indices, respectively. However, the indices are always returned in ascending order.

If two outputs are requested, `find` returns the row and column indices of nonzero elements of a matrix. For example:

```
[i, j] = find (2 * eye (2))
    ⇒ i = [ 1; 2 ]
    ⇒ j = [ 1; 2 ]
```

If three outputs are requested, `find` also returns a vector containing the nonzero values. For example:

```
[i, j, v] = find (3 * eye (2))
       ⇒ i = [ 1; 2 ]
       ⇒ j = [ 1; 2 ]
       ⇒ v = [ 3; 3 ]
```

Note that this function is particularly useful for sparse matrices, as it extracts the nonzero elements as vectors, which can then be used to create the original matrix. For example:

```
sz = size (a);
[i, j, v] = find (a);
b = sparse (i, j, v, sz(1), sz(2));
```

See also: [nonzeros], page 526.

idx = `lookup` (*table*, *y*) [Built-in Function]
idx = `lookup` (*table*, *y*, *opt*) [Built-in Function]

Lookup values in a sorted table.

This function is usually used as a prelude to interpolation.

If table is increasing and `idx = lookup (table, y)`, then `table(idx(i)) <= y(i) < table(idx(i+1))` for all `y(i)` within the table. If `y(i) < table(1)` then `idx(i)` is 0. If `y(i) >= table(end)` or `isnan (y(i))` then `idx(i)` is `n`.

If the table is decreasing, then the tests are reversed. For non-strictly monotonic tables, empty intervals are always skipped. The result is undefined if *table* is not monotonic, or if *table* contains a NaN.

The complexity of the lookup is O(M*log(N)) where N is the size of *table* and M is the size of *y*. In the special case when *y* is also sorted, the complexity is O(min(M*log(N),M+N)).

table and *y* can also be cell arrays of strings (or *y* can be a single string). In this case, string lookup is performed using lexicographical comparison.

If *opts* is specified, it must be a string with letters indicating additional options.

`m` `table(idx(i)) == val(i)` if `val(i)` occurs in table; otherwise, `idx(i)` is zero.

`b` `idx(i)` is a logical 1 or 0, indicating whether `val(i)` is contained in table or not.

`l` For numeric lookups the leftmost subinterval shall be extended to infinity (i.e., all indices at least 1)

`r` For numeric lookups the rightmost subinterval shall be extended to infinity (i.e., all indices at most n-1).

If you wish to check if a variable exists at all, instead of properties its elements may have, consult Section 7.3 [Status of Variables], page 127.

16.2 Rearranging Matrices

`fliplr (x)` [Function File]

Flip array left to right.

Return a copy of *x* with the order of the columns reversed. In other words, *x* is flipped left-to-right about a vertical axis. For example:

```
fliplr ([1, 2; 3, 4])
     ⇒  2  1
        4  3
```

See also: [flipud], page 409, [flip], page 409, [rot90], page 410, [rotdim], page 410.

`flipud (x)` [Function File]

Flip array upside down.

Return a copy of *x* with the order of the rows reversed. In other words, *x* is flipped upside-down about a horizontal axis. For example:

```
flipud ([1, 2; 3, 4])
     ⇒  3  4
        1  2
```

See also: [fliplr], page 409, [flip], page 409, [rot90], page 410, [rotdim], page 410.

`flip (x)` [Function File]
`flip (x, dim)` [Function File]

Flip array across dimension *dim*.

Return a copy of *x* flipped about the dimension *dim*. *dim* defaults to the first non-singleton dimension. For example:

```
flip ([1  2  3  4])
      ⇒  4  3  2  1

flip ([1; 2; 3; 4])
      ⇒  4
         3
         2
         1

flip ([1 2; 3 4])
      ⇒  3  4
         1  2

flip ([1 2; 3 4], 2)
      ⇒  2  1
         4  3
```

See also: [fliplr], page 409, [flipud], page 409, [rot90], page 410, [rotdim], page 410, [permute], page 411, [transpose], page 144.

`rot90` (*A*) [Function File]
`rot90` (*A*, *k*) [Function File]

Rotate array by 90 degree increments.

Return a copy of *A* with the elements rotated counterclockwise in 90-degree increments.

The second argument is optional, and specifies how many 90-degree rotations are to be applied (the default value is 1). Negative values of *k* rotate the matrix in a clockwise direction. For example,

```
rot90 ([1, 2; 3, 4], -1)
    ⇒  3  1
       4  2
```

rotates the given matrix clockwise by 90 degrees. The following are all equivalent statements:

```
rot90 ([1, 2; 3, 4], -1)
rot90 ([1, 2; 3, 4], 3)
rot90 ([1, 2; 3, 4], 7)
```

The rotation is always performed on the plane of the first two dimensions, i.e., rows and columns. To perform a rotation on any other plane, use `rotdim`.

See also: [rotdim], page 410, [fliplr], page 409, [flipud], page 409, [flip], page 409.

`rotdim` (*x*) [Function File]
`rotdim` (*x*, *n*) [Function File]
`rotdim` (*x*, *n*, *plane*) [Function File]

Return a copy of *x* with the elements rotated counterclockwise in 90-degree increments.

The second argument *n* is optional, and specifies how many 90-degree rotations are to be applied (the default value is 1). Negative values of *n* rotate the matrix in a clockwise direction.

The third argument is also optional and defines the plane of the rotation. If present, *plane* is a two element vector containing two different valid dimensions of the matrix. When *plane* is not given the first two non-singleton dimensions are used.

For example,

```
rotdim ([1, 2; 3, 4], -1, [1, 2])
     ⇒  3  1
        4  2
```

rotates the given matrix clockwise by 90 degrees. The following are all equivalent statements:

```
rotdim ([1, 2; 3, 4], -1, [1, 2])
rotdim ([1, 2; 3, 4], 3, [1, 2])
rotdim ([1, 2; 3, 4], 7, [1, 2])
```

See also: [rot90], page 410, [fliplr], page 409, [flipud], page 409, [flip], page 409.

`cat (dim, array1, array2, ..., arrayN)` [Built-in Function]

Return the concatenation of N-D array objects, *array1*, *array2*, ..., *arrayN* along dimension *dim*.

```
A = ones (2, 2);
B = zeros (2, 2);
cat (2, A, B)
  ⇒ 1 1 0 0
     1 1 0 0
```

Alternatively, we can concatenate A and B along the second dimension in the following way:

```
[A, B]
```

dim can be larger than the dimensions of the N-D array objects and the result will thus have *dim* dimensions as the following example shows:

```
cat (4, ones (2, 2), zeros (2, 2))
  ⇒ ans(:,:,1,1) =

       1 1
       1 1

     ans(:,:,1,2) =

       0 0
       0 0
```

See also: [horzcat], page 411, [vertcat], page 411.

`horzcat (array1, array2, ..., arrayN)` [Built-in Function]

Return the horizontal concatenation of N-D array objects, *array1*, *array2*, ..., *arrayN* along dimension 2.

Arrays may also be concatenated horizontally using the syntax for creating new matrices. For example:

```
hcat = [ array1, array2, ... ]
```

See also: [cat], page 411, [vertcat], page 411.

`vertcat (array1, array2, ..., arrayN)` [Built-in Function]

Return the vertical concatenation of N-D array objects, *array1*, *array2*, ..., *arrayN* along dimension 1.

Arrays may also be concatenated vertically using the syntax for creating new matrices. For example:

```
vcat = [ array1; array2; ... ]
```

See also: [cat], page 411, [horzcat], page 411.

`permute (A, perm)` [Built-in Function]

Return the generalized transpose for an N-D array object A.

The permutation vector *perm* must contain the elements `1:ndims (A)` (in any order, but each element must appear only once). The Nth dimension of A gets remapped to dimension `PERM(N)`. For example:

```
x = zeros ([2, 3, 5, 7]);
size (x)
   ⇒ 2   3   5   7

size (permute (x, [2, 1, 3, 4]))
   ⇒ 3   2   5   7

size (permute (x, [1, 3, 4, 2]))
   ⇒ 2   5   7   3

## The identity permutation
size (permute (x, [1, 2, 3, 4]))
   ⇒ 2   3   5   7
```

See also: [ipermute], page 412.

`ipermute (A, iperm)` [Built-in Function]

The inverse of the `permute` function.

The expression

```
ipermute (permute (A, perm), perm)
```

returns the original array A.

See also: [permute], page 411.

`reshape (A, m, n, ...)` [Built-in Function]
`reshape (A, [m n ...])` [Built-in Function]
`reshape (A, ..., [], ...)` [Built-in Function]
`reshape (A, size)` [Built-in Function]

Return a matrix with the specified dimensions (m, n, ...) whose elements are taken from the matrix A.

The elements of the matrix are accessed in column-major order (like Fortran arrays are stored).

The following code demonstrates reshaping a 1x4 row vector into a 2x2 square matrix.

```
reshape ([1, 2, 3, 4], 2, 2)
      ⇒  1  3
         2  4
```

Note that the total number of elements in the original matrix (`prod (size (A))`) must match the total number of elements in the new matrix (`prod ([m n ...])`).

A single dimension of the return matrix may be left unspecified and Octave will determine its size automatically. An empty matrix ([]) is used to flag the unspecified dimension.

See also: [resize], page 413, [vec], page 417, [postpad], page 417, [cat], page 411, [squeeze], page 46.

`resize` (*x*, *m*) [Built-in Function]
`resize` (*x*, *m*, *n*, . . .) [Built-in Function]
`resize` (*x*, [*m n* . . .]) [Built-in Function]

Resize *x* cutting off elements as necessary.

In the result, element with certain indices is equal to the corresponding element of *x* if the indices are within the bounds of *x*; otherwise, the element is set to zero.

In other words, the statement

```
y = resize (x, dv)
```

is equivalent to the following code:

```
y = zeros (dv, class (x));
sz = min (dv, size (x));
for i = 1:length (sz)
  idx{i} = 1:sz(i);
endfor
y(idx{:}) = x(idx{:});
```

but is performed more efficiently.

If only *m* is supplied, and it is a scalar, the dimension of the result is *m*-by-*m*. If *m*, *n*, . . . are all scalars, then the dimensions of the result are *m*-by-*n*-by-. . . . If given a vector as input, then the dimensions of the result are given by the elements of that vector.

An object can be resized to more dimensions than it has; in such case the missing dimensions are assumed to be 1. Resizing an object to fewer dimensions is not possible.

See also: [reshape], page 412, [postpad], page 417, [prepad], page 417, [cat], page 411.

y = `circshift` (*x*, *n*) [Function File]

Circularly shift the values of the array *x*.

n must be a vector of integers no longer than the number of dimensions in *x*. The values of *n* can be either positive or negative, which determines the direction in which the values or *x* are shifted. If an element of *n* is zero, then the corresponding dimension of *x* will not be shifted. For example:

```
x = [1, 2, 3; 4, 5, 6; 7, 8, 9];
circshift (x, 1)
⇒  7, 8, 9
    1, 2, 3
    4, 5, 6
circshift (x, -2)
⇒  7, 8, 9
    1, 2, 3
    4, 5, 6
circshift (x, [0,1])
⇒  3, 1, 2
    6, 4, 5
    9, 7, 8
```

See also: [permute], page 411, [ipermute], page 412, [shiftdim], page 414.

`shift (x, b)` [Function File]
`shift (x, b, dim)` [Function File]

If *x* is a vector, perform a circular shift of length *b* of the elements of *x*.

If *x* is a matrix, do the same for each column of *x*.

If the optional *dim* argument is given, operate along this dimension.

`y = shiftdim (x, n)` [Function File]
`[y, ns] = shiftdim (x)` [Function File]

Shift the dimensions of *x* by *n*, where *n* must be an integer scalar.

When *n* is positive, the dimensions of *x* are shifted to the left, with the leading dimensions circulated to the end. If *n* is negative, then the dimensions of *x* are shifted to the right, with *n* leading singleton dimensions added.

Called with a single argument, `shiftdim`, removes the leading singleton dimensions, returning the number of dimensions removed in the second output argument *ns*.

For example:

```
x = ones (1, 2, 3);
size (shiftdim (x, -1))
   ⇒ [1, 1, 2, 3]
size (shiftdim (x, 1))
   ⇒ [2, 3]
[b, ns] = shiftdim (x)
   ⇒ b = [1, 1, 1; 1, 1, 1]
   ⇒ ns = 1
```

See also: [reshape], page 412, [permute], page 411, [ipermute], page 412, [circshift], page 413, [squeeze], page 46.

`[s, i] = sort (x)` [Built-in Function]
`[s, i] = sort (x, dim)` [Built-in Function]
`[s, i] = sort (x, mode)` [Built-in Function]
`[s, i] = sort (x, dim, mode)` [Built-in Function]

Return a copy of *x* with the elements arranged in increasing order.

For matrices, `sort` orders the elements within columns

For example:

```
sort ([1, 2; 2, 3; 3, 1])
   ⇒  1  1
      2  2
      3  3
```

If the optional argument *dim* is given, then the matrix is sorted along the dimension defined by *dim*. The optional argument `mode` defines the order in which the values will be sorted. Valid values of `mode` are `"ascend"` or `"descend"`.

The `sort` function may also be used to produce a matrix containing the original row indices of the elements in the sorted matrix. For example:

```
[s, i] = sort ([1, 2; 2, 3; 3, 1])
  ⇒ s = 1  1
        2  2
        3  3
  ⇒ i = 1  3
        2  1
        3  2
```

For equal elements, the indices are such that equal elements are listed in the order in which they appeared in the original list.

Sorting of complex entries is done first by magnitude (`abs (z)`) and for any ties by phase angle (`angle (z)`). For example:

```
sort ([1+i; 1; 1-i])
    ⇒ 1 + 0i
      1 - 1i
      1 + 1i
```

NaN values are treated as being greater than any other value and are sorted to the end of the list.

The `sort` function may also be used to sort strings and cell arrays of strings, in which case ASCII dictionary order (uppercase 'A' precedes lowercase 'a') of the strings is used.

The algorithm used in `sort` is optimized for the sorting of partially ordered lists.

See also: [sortrows], page 415, [issorted], page 415.

`[s, i] = sortrows (A)` [Function File]
`[s, i] = sortrows (A, c)` [Function File]

Sort the rows of the matrix A according to the order of the columns specified in c.

By default (c omitted, or a particular column unspecified in c) an ascending sort order is used. However, if elements of c are negative then the corresponding column is sorted in descending order. If the elements of A are strings then a lexicographical sort is used.

Example: sort by column 2 in descending order, then 3 in ascending order

```
x = [ 7, 1, 4;
      8, 3, 5;
      9, 3, 6 ];
sortrows (x, [-2, 3])
   ⇒ 8  3  5
     9  3  6
     7  1  4
```

See also: [sort], page 414.

`issorted (a)` [Built-in Function]
`issorted (a, mode)` [Built-in Function]
`issorted (a, "rows", mode)` [Built-in Function]

Return true if the array is sorted according to *mode*, which may be either `"ascending"`, `"descending"`, or `"either"`.

By default, *mode* is `"ascending"`. NaNs are treated in the same manner as `sort`.

If the optional argument `"rows"` is supplied, check whether the array is sorted by rows as output by the function `sortrows` (with no options).

This function does not support sparse matrices.

See also: [sort], page 414, [sortrows], page 415.

`nth_element` (*x*, *n*) [Built-in Function]
`nth_element` (*x*, *n*, *dim*) [Built-in Function]

Select the n-th smallest element of a vector, using the ordering defined by `sort`.

The result is equivalent to `sort(x)(n)`.

n can also be a contiguous range, either ascending `l:u` or descending `u:-1:l`, in which case a range of elements is returned.

If *x* is an array, `nth_element` operates along the dimension defined by *dim*, or the first non-singleton dimension if *dim* is not given.

Programming Note: nth_element encapsulates the C++ standard library algorithms nth_element and partial_sort. On average, the complexity of the operation is O(M*log(K)), where `M = size (x, dim)` and `K = length (n)`. This function is intended for cases where the ratio K/M is small; otherwise, it may be better to use `sort`.

See also: [sort], page 414, [min], page 445, [max], page 444.

`tril` (*A*) [Function File]
`tril` (*A*, *k*) [Function File]
`tril` (*A*, *k*, *pack*) [Function File]
`triu` (*A*) [Function File]
`triu` (*A*, *k*) [Function File]
`triu` (*A*, *k*, *pack*) [Function File]

Return a new matrix formed by extracting the lower (`tril`) or upper (`triu`) triangular part of the matrix *A*, and setting all other elements to zero.

The second argument is optional, and specifies how many diagonals above or below the main diagonal should also be set to zero.

The default value of *k* is zero, so that `triu` and `tril` normally include the main diagonal as part of the result.

If the value of *k* is nonzero integer, the selection of elements starts at an offset of *k* diagonals above or below the main diagonal; above for positive *k* and below for negative *k*.

The absolute value of *k* must not be greater than the number of subdiagonals or superdiagonals.

For example:

```
tril (ones (3), -1)
     ⇒  0  0  0
        1  0  0
        1  1  0
```

and

```
tril (ones (3), 1)
     ⇒  1  1  0
        1  1  1
        1  1  1
```

If the option `"pack"` is given as third argument, the extracted elements are not inserted into a matrix, but rather stacked column-wise one above other.

See also: [diag], page 418.

v = `vec` (*x*) [Built-in Function]
v = `vec` (*x*, *dim*) [Built-in Function]

Return the vector obtained by stacking the columns of the matrix *x* one above the other.

Without *dim* this is equivalent to `x(:)`.

If *dim* is supplied, the dimensions of *v* are set to *dim* with all elements along the last dimension. This is equivalent to `shiftdim (x(:), 1-dim)`.

See also: [vech], page 417, [resize], page 413, [cat], page 411.

`vech` (*x*) [Function File]

Return the vector obtained by eliminating all superdiagonal elements of the square matrix *x* and stacking the result one column above the other.

This has uses in matrix calculus where the underlying matrix is symmetric and it would be pointless to keep values above the main diagonal.

See also: [vec], page 417.

`prepad` (*x*, *l*) [Function File]
`prepad` (*x*, *l*, *c*) [Function File]
`prepad` (*x*, *l*, *c*, *dim*) [Function File]

Prepend the scalar value *c* to the vector *x* until it is of length *l*. If *c* is not given, a value of 0 is used.

If `length (x) > l`, elements from the beginning of *x* are removed until a vector of length *l* is obtained.

If *x* is a matrix, elements are prepended or removed from each row.

If the optional argument *dim* is given, operate along this dimension.

If *dim* is larger than the dimensions of *x*, the result will have *dim* dimensions.

See also: [postpad], page 417, [cat], page 411, [resize], page 413.

`postpad` (*x*, *l*) [Function File]
`postpad` (*x*, *l*, *c*) [Function File]
`postpad` (*x*, *l*, *c*, *dim*) [Function File]

Append the scalar value *c* to the vector *x* until it is of length *l*. If *c* is not given, a value of 0 is used.

If `length (x) > l`, elements from the end of *x* are removed until a vector of length *l* is obtained.

If *x* is a matrix, elements are appended or removed from each row.

If the optional argument *dim* is given, operate along this dimension.

If *dim* is larger than the dimensions of *x*, the result will have *dim* dimensions.

See also: [prepad], page 417, [cat], page 411, [resize], page 413.

M = `diag` (*v*) [Built-in Function]
M = `diag` (*v*, *k*) [Built-in Function]
M = `diag` (*v*, *m*, *n*) [Built-in Function]
v = `diag` (*M*) [Built-in Function]
v = `diag` (*M*, *k*) [Built-in Function]

Return a diagonal matrix with vector *v* on diagonal *k*.

The second argument is optional. If it is positive, the vector is placed on the *k*-th superdiagonal. If it is negative, it is placed on the -*k*-th subdiagonal. The default value of *k* is 0, and the vector is placed on the main diagonal. For example:

```
diag ([1, 2, 3], 1)
   ⇒  0  1  0  0
      0  0  2  0
      0  0  0  3
      0  0  0  0
```

The 3-input form returns a diagonal matrix with vector *v* on the main diagonal and the resulting matrix being of size *m* rows x *n* columns.

Given a matrix argument, instead of a vector, `diag` extracts the *k*-th diagonal of the matrix.

`blkdiag` (*A*, *B*, *C*, ...) [Function File]

Build a block diagonal matrix from *A*, *B*, *C*, ...

All arguments must be numeric and either two-dimensional matrices or scalars. If any argument is of type sparse, the output will also be sparse.

See also: [diag], page 418, [horzcat], page 411, [vertcat], page 411, [sparse], page 524.

16.3 Special Utility Matrices

`eye` (*n*) [Built-in Function]
`eye` (*m*, *n*) [Built-in Function]
`eye` ([*m* *n*]) [Built-in Function]
`eye` (..., *class*) [Built-in Function]

Return an identity matrix.

If invoked with a single scalar argument *n*, return a square NxN identity matrix.

If supplied two scalar arguments (*m*, *n*), `eye` takes them to be the number of rows and columns. If given a vector with two elements, `eye` uses the values of the elements as the number of rows and columns, respectively. For example:

```
eye (3)
 ⇒  1  0  0
    0  1  0
    0  0  1
```

The following expressions all produce the same result:

```
eye (2)
≡
eye (2, 2)
≡
eye (size ([1, 2; 3, 4]))
```

The optional argument *class*, allows `eye` to return an array of the specified type, like

```
val = zeros (n,m, "uint8")
```

Calling `eye` with no arguments is equivalent to calling it with an argument of 1. Any negative dimensions are treated as zero. These odd definitions are for compatibility with MATLAB.

See also: [speye], page 522, [ones], page 419, [zeros], page 419.

`ones` (*n*) [Built-in Function]
`ones` (*m*, *n*) [Built-in Function]
`ones` (*m*, *n*, *k*, . . .) [Built-in Function]
`ones` ([*m n* . . .]) [Built-in Function]
`ones` (. . ., *class*) [Built-in Function]

Return a matrix or N-dimensional array whose elements are all 1.

If invoked with a single scalar integer argument *n*, return a square NxN matrix.

If invoked with two or more scalar integer arguments, or a vector of integer values, return an array with the given dimensions.

To create a constant matrix whose values are all the same use an expression such as

```
val_matrix = val * ones (m, n)
```

The optional argument *class* specifies the class of the return array and defaults to double. For example:

```
val = ones (m,n, "uint8")
```

See also: [zeros], page 419.

`zeros` (*n*) [Built-in Function]
`zeros` (*m*, *n*) [Built-in Function]
`zeros` (*m*, *n*, *k*, . . .) [Built-in Function]
`zeros` ([*m n* . . .]) [Built-in Function]
`zeros` (. . ., *class*) [Built-in Function]

Return a matrix or N-dimensional array whose elements are all 0.

If invoked with a single scalar integer argument, return a square NxN matrix.

If invoked with two or more scalar integer arguments, or a vector of integer values, return an array with the given dimensions.

The optional argument *class* specifies the class of the return array and defaults to double. For example:

```
val = zeros (m,n, "uint8")
```

See also: [ones], page 419.

`repmat (A, m)` [Function File]
`repmat (A, m, n)` [Function File]
`repmat (A, m, n, p ...)` [Function File]
`repmat (A, [m n])` [Function File]
`repmat (A, [m n p ...])` [Function File]

Form a block matrix of size *m* by *n*, with a copy of matrix *A* as each element.

If *n* is not specified, form an *m* by *m* block matrix. For copying along more than two dimensions, specify the number of times to copy across each dimension *m*, *n*, *p*, ..., in a vector in the second argument.

See also: [repelems], page 420.

`repelems (x, r)` [Built-in Function]

Construct a vector of repeated elements from *x*.

r is a 2x*N* integer matrix specifying which elements to repeat and how often to repeat each element. Entries in the first row, *r*(1,j), select an element to repeat. The corresponding entry in the second row, *r*(2,j), specifies the repeat count. If *x* is a matrix then the columns of *x* are imagined to be stacked on top of each other for purposes of the selection index. A row vector is always returned.

Conceptually the result is calculated as follows:

```
y = [];
for i = 1:columns (r)
  y = [y, x(r(1,i)*ones(1, r(2,i)))];
endfor
```

See also: [repmat], page 420, [cat], page 411.

The functions `linspace` and `logspace` make it very easy to create vectors with evenly or logarithmically spaced elements. See Section 4.2 [Ranges], page 52.

`linspace (base, limit)` [Built-in Function]
`linspace (base, limit, n)` [Built-in Function]

Return a row vector with *n* linearly spaced elements between *base* and *limit*.

If the number of elements is greater than one, then the endpoints *base* and *limit* are always included in the range. If *base* is greater than *limit*, the elements are stored in decreasing order. If the number of points is not specified, a value of 100 is used.

The `linspace` function always returns a row vector if both *base* and *limit* are scalars. If one, or both, of them are column vectors, `linspace` returns a matrix.

For compatibility with MATLAB, return the second argument (*limit*) if fewer than two values are requested.

See also: [logspace], page 420.

`logspace (a, b)` [Function File]
`logspace (a, b, n)` [Function File]
`logspace (a, pi, n)` [Function File]

Return a row vector with *n* elements logarithmically spaced from 10^a to 10^b.

If *n* is unspecified it defaults to 50.

If b is equal to π, the points are between 10^a and π, *not* 10^a and 10^π, in order to be compatible with the corresponding MATLAB function.

Also for compatibility with MATLAB, return the second argument b if fewer than two values are requested.

See also: [linspace], page 420.

`rand` (*n*) [Built-in Function]
`rand` (*m*, *n*, ...) [Built-in Function]
`rand` ([*m* *n* ...]) [Built-in Function]
v = `rand` ("*state*") [Built-in Function]
`rand` ("*state*", *v*) [Built-in Function]
`rand` ("*state*", "*reset*") [Built-in Function]
v = `rand` ("*seed*") [Built-in Function]
`rand` ("*seed*", *v*) [Built-in Function]
`rand` ("*seed*", "*reset*") [Built-in Function]
`rand` (..., "*single*") [Built-in Function]
`rand` (..., "*double*") [Built-in Function]

Return a matrix with random elements uniformly distributed on the interval (0, 1).

The arguments are handled the same as the arguments for `eye`.

You can query the state of the random number generator using the form

```
v = rand ("state")
```

This returns a column vector v of length 625. Later, you can restore the random number generator to the state v using the form

```
rand ("state", v)
```

You may also initialize the state vector from an arbitrary vector of length ≤ 625 for v. This new state will be a hash based on the value of v, not v itself.

By default, the generator is initialized from `/dev/urandom` if it is available, otherwise from CPU time, wall clock time, and the current fraction of a second. Note that this differs from MATLAB, which always initializes the state to the same state at startup. To obtain behavior comparable to MATLAB, initialize with a deterministic state vector in Octave's startup files (see Section 2.1.2 [Startup Files], page 19).

To compute the pseudo-random sequence, `rand` uses the Mersenne Twister with a period of $2^{19937}-1$ (See M. Matsumoto and T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Trans. on Modeling and Computer Simulation Vol. 8, No. 1, pp. 3–30, January 1998, `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`). Do **not** use for cryptography without securely hashing several returned values together, otherwise the generator state can be learned after reading 624 consecutive values.

Older versions of Octave used a different random number generator. The new generator is used by default as it is significantly faster than the old generator, and produces random numbers with a significantly longer cycle time. However, in some circumstances it might be desirable to obtain the same random sequences as produced by the old generators. To do this the keyword `"seed"` is used to specify that the old generators should be used, as in

```
rand ("seed", val)
```

which sets the seed of the generator to *val*. The seed of the generator can be queried with

```
s = rand ("seed")
```

However, it should be noted that querying the seed will not cause `rand` to use the old generators, only setting the seed will. To cause `rand` to once again use the new generators, the keyword `"state"` should be used to reset the state of the `rand`.

The state or seed of the generator can be reset to a new random value using the `"reset"` keyword.

The class of the value returned can be controlled by a trailing `"double"` or `"single"` argument. These are the only valid classes.

See also: [randn], page 422, [rande], page 423, [randg], page 424, [randp], page 423.

`randi` (*imax*) [Function File]
`randi` (*imax*, *n*) [Function File]
`randi` (*imax*, *m*, *n*, . . .) [Function File]
`randi` ([*imin imax*], . . .) [Function File]
`randi` (. . ., "*class*") [Function File]

Return random integers in the range 1:*imax*.

Additional arguments determine the shape of the return matrix. When no arguments are specified a single random integer is returned. If one argument *n* is specified then a square matrix (*n* x *n*) is returned. Two or more arguments will return a multi-dimensional matrix (*m* x *n* x . . .).

The integer range may optionally be described by a two element matrix with a lower and upper bound in which case the returned integers will be on the interval [*imin*, *imax*].

The optional argument *class* will return a matrix of the requested type. The default is `"double"`.

The following example returns 150 integers in the range 1–10.

```
ri = randi (10, 150, 1)
```

Implementation Note: `randi` relies internally on `rand` which uses class `"double"` to represent numbers. This limits the maximum integer (*imax*) and range (*imax* - *imin*) to the value returned by the `flintmax` function. For IEEE floating point numbers this value is $2^{53} - 1$.

See also: [rand], page 421.

`randn` (*n*) [Built-in Function]
`randn` (*m*, *n*, . . .) [Built-in Function]
`randn` ([*m n* . . .]) [Built-in Function]
v = `randn` ("*state*") [Built-in Function]
`randn` ("*state*", *v*) [Built-in Function]
`randn` ("*state*", "*reset*") [Built-in Function]
v = `randn` ("*seed*") [Built-in Function]
`randn` ("*seed*", *v*) [Built-in Function]
`randn` ("*seed*", "*reset*") [Built-in Function]
`randn` (. . ., "*single*") [Built-in Function]

`randn (..., "double")` [Built-in Function]

Return a matrix with normally distributed random elements having zero mean and variance one.

The arguments are handled the same as the arguments for `rand`.

By default, `randn` uses the Marsaglia and Tsang “Ziggurat technique” to transform from a uniform to a normal distribution.

The class of the value returned can be controlled by a trailing `"double"` or `"single"` argument. These are the only valid classes.

Reference: G. Marsaglia and W.W. Tsang, *Ziggurat Method for Generating Random Variables*, J. Statistical Software, vol 5, 2000, `http://www.jstatsoft.org/v05/i08/`

See also: [rand], page 421, [rande], page 423, [randg], page 424, [randp], page 423.

`rande (n)` [Built-in Function]
`rande (m, n, ...)` [Built-in Function]
`rande ([m n ...])` [Built-in Function]
`v = rande ("state")` [Built-in Function]
`rande ("state", v)` [Built-in Function]
`rande ("state", "reset")` [Built-in Function]
`v = rande ("seed")` [Built-in Function]
`rande ("seed", v)` [Built-in Function]
`rande ("seed", "reset")` [Built-in Function]
`rande (..., "single")` [Built-in Function]
`rande (..., "double")` [Built-in Function]

Return a matrix with exponentially distributed random elements.

The arguments are handled the same as the arguments for `rand`.

By default, `rande` uses the Marsaglia and Tsang “Ziggurat technique” to transform from a uniform to an exponential distribution.

The class of the value returned can be controlled by a trailing `"double"` or `"single"` argument. These are the only valid classes.

Reference: G. Marsaglia and W.W. Tsang, *Ziggurat Method for Generating Random Variables*, J. Statistical Software, vol 5, 2000, `http://www.jstatsoft.org/v05/i08/`

See also: [rand], page 421, [randn], page 422, [randg], page 424, [randp], page 423.

`randp (l, n)` [Built-in Function]
`randp (l, m, n, ...)` [Built-in Function]
`randp (l, [m n ...])` [Built-in Function]
`v = randp ("state")` [Built-in Function]
`randp ("state", v)` [Built-in Function]
`randp ("state", "reset")` [Built-in Function]
`v = randp ("seed")` [Built-in Function]
`randp ("seed", v)` [Built-in Function]
`randp ("seed", "reset")` [Built-in Function]
`randp (..., "single")` [Built-in Function]

`randp (..., "double")` [Built-in Function]

Return a matrix with Poisson distributed random elements with mean value parameter given by the first argument, *l*.

The arguments are handled the same as the arguments for `rand`, except for the argument *l*.

Five different algorithms are used depending on the range of *l* and whether or not *l* is a scalar or a matrix.

For scalar $l \leq 12$, use direct method.
: W.H. Press, et al., *Numerical Recipes in C*, Cambridge University Press, 1992.

For scalar *l* > 12, use rejection method.[1]
: W.H. Press, et al., *Numerical Recipes in C*, Cambridge University Press, 1992.

For matrix $l \leq 10$, use inversion method.[2]
: E. Stadlober, et al., WinRand source code, available via FTP.

For matrix *l* > 10, use patchwork rejection method.
: E. Stadlober, et al., WinRand source code, available via FTP, or H. Zechner, *Efficient sampling from continuous and discrete unimodal distributions*, Doctoral Dissertation, 156pp., Technical University Graz, Austria, 1994.

For *l* > 1e8, use normal approximation.
: L. Montanet, et al., *Review of Particle Properties*, Physical Review D 50 p1284, 1994.

The class of the value returned can be controlled by a trailing `"double"` or `"single"` argument. These are the only valid classes.

See also: [rand], page 421, [randn], page 422, [rande], page 423, [randg], page 424.

`randg (n)` [Built-in Function]
`randg (m, n, ...)` [Built-in Function]
`randg ([m n ...])` [Built-in Function]
`v = randg ("state")` [Built-in Function]
`randg ("state", v)` [Built-in Function]
`randg ("state", "reset")` [Built-in Function]
`v = randg ("seed")` [Built-in Function]
`randg ("seed", v)` [Built-in Function]
`randg ("seed", "reset")` [Built-in Function]
`randg (..., "single")` [Built-in Function]
`randg (..., "double")` [Built-in Function]

Return a matrix with `gamma (a,1)` distributed random elements.

The arguments are handled the same as the arguments for `rand`, except for the argument *a*.

This can be used to generate many distributions:

`gamma (a, b)` for `a > -1`, `b > 0`
: `r = b * randg (a)`

`beta (a, b)` for `a > -1, b > -1`

```
r1 = randg (a, 1)
r = r1 / (r1 + randg (b, 1))
```

`Erlang (a, n)`

```
r = a * randg (n)
```

`chisq (df)` for `df > 0`

```
r = 2 * randg (df / 2)
```

`t (df)` for `0 < df < inf` (use randn if df is infinite)

```
r = randn () / sqrt (2 * randg (df / 2) / df)
```

`F (n1, n2)` for `0 < n1, 0 < n2`

```
## r1 equals 1 if n1 is infinite
r1 = 2 * randg (n1 / 2) / n1
## r2 equals 1 if n2 is infinite
r2 = 2 * randg (n2 / 2) / n2
r = r1 / r2
```

negative `binomial (n, p)` for `n > 0, 0 < p <= 1`

```
r = randp ((1 - p) / p * randg (n))
```

non-central `chisq (df, L)`, for `df >= 0` and `L > 0`
(use chisq if `L = 0`)

```
r = randp (L / 2)
r(r > 0) = 2 * randg (r(r > 0))
r(df > 0) += 2 * randg (df(df > 0)/2)
```

`Dirichlet (a1, ... ak)`

```
r = (randg (a1), ..., randg (ak))
r = r / sum (r)
```

The class of the value returned can be controlled by a trailing `"double"` or `"single"` argument. These are the only valid classes.

See also: [rand], page 421, [randn], page 422, [rande], page 423, [randp], page 423.

The generators operate in the new or old style together, it is not possible to mix the two. Initializing any generator with `"state"` or `"seed"` causes the others to switch to the same style for future calls.

The state of each generator is independent and calls to different generators can be interleaved without affecting the final result. For example,

```
rand ("state", [11, 22, 33]);
randn ("state", [44, 55, 66]);
u = rand (100, 1);
n = randn (100, 1);
```

and

```
rand ("state", [11, 22, 33]);
randn ("state", [44, 55, 66]);
u = zeros (100, 1);
n = zeros (100, 1);
for i = 1:100
  u(i) = rand ();
  n(i) = randn ();
end
```

produce equivalent results. When the generators are initialized in the old style with `"seed"` only `rand` and `randn` are independent, because the old `rande`, `randg` and `randp` generators make calls to `rand` and `randn`.

The generators are initialized with random states at start-up, so that the sequences of random numbers are not the same each time you run Octave.[1] If you really do need to reproduce a sequence of numbers exactly, you can set the state or seed to a specific value.

If invoked without arguments, `rand` and `randn` return a single element of a random sequence.

The original `rand` and `randn` functions use Fortran code from RANLIB, a library of Fortran routines for random number generation, compiled by Barry W. Brown and James Lovato of the Department of Biomathematics at The University of Texas, M.D. Anderson Cancer Center, Houston, TX 77030.

`randperm (`*n*`)` [Built-in Function]
`randperm (`*n*, *m*`)` [Built-in Function]

Return a row vector containing a random permutation of `1:`*n*.

If m is supplied, return m unique entries, sampled without replacement from `1:`*n*.

The complexity is O(n) in memory and O(m) in time, unless $m < n/5$, in which case O(m) memory is used as well. The randomization is performed using rand(). All permutations are equally likely.

See also: [perms], page 607.

16.4 Famous Matrices

The following functions return famous matrix forms.

`gallery (`*name*`)` [Function File]
`gallery (`*name*, *args*`)` [Function File]

Create interesting matrices for testing.

c `= gallery ("`*cauchy*`",` *x*`)` [Function File]
c `= gallery ("`*cauchy*`",` *x*, *y*`)` [Function File]

Create a Cauchy matrix.

c `= gallery ("`*chebspec*`",` *n*`)` [Function File]
c `= gallery ("`*chebspec*`",` *n*, *k*`)` [Function File]

Create a Chebyshev spectral differentiation matrix.

---

[1] The old versions of `rand` and `randn` obtain their initial seeds from the system clock.

c = gallery ("*chebvand*", *p*) [Function File]
c = gallery ("*chebvand*", *m*, *p*) [Function File]
Create a Vandermonde-like matrix for the Chebyshev polynomials.

a = gallery ("*chow*", *n*) [Function File]
a = gallery ("*chow*", *n*, *alpha*) [Function File]
a = gallery ("*chow*", *n*, *alpha*, *delta*) [Function File]
Create a Chow matrix – a singular Toeplitz lower Hessenberg matrix.

c = gallery ("*circul*", *v*) [Function File]
Create a circulant matrix.

a = gallery ("*clement*", *n*) [Function File]
a = gallery ("*clement*", *n*, *k*) [Function File]
Create a tridiagonal matrix with zero diagonal entries.

c = gallery ("*compar*", a) [Function File]
c = gallery ("*compar*", a, *k*) [Function File]
Create a comparison matrix.

a = gallery ("*condex*", *n*) [Function File]
a = gallery ("*condex*", *n*, *k*) [Function File]
a = gallery ("*condex*", *n*, *k*, *theta*) [Function File]
Create a 'counterexample' matrix to a condition estimator.

a = gallery ("*cycol*", [*m* *n*]) [Function File]
a = gallery ("*cycol*", *n*) [Function File]
a = gallery (..., *k*) [Function File]
Create a matrix whose columns repeat cyclically.

[*c*, *d*, *e*] = gallery ("*dorr*", *n*) [Function File]
[*c*, *d*, *e*] = gallery ("*dorr*", *n*, *theta*) [Function File]
a = gallery ("*dorr*", ...) [Function File]
Create a diagonally dominant, ill-conditioned, tridiagonal matrix.

a = gallery ("*dramadah*", *n*) [Function File]
a = gallery ("*dramadah*", *n*, *k*) [Function File]
Create a (0, 1) matrix whose inverse has large integer entries.

a = gallery ("*fiedler*", *c*) [Function File]
Create a symmetric Fiedler matrix.

a = gallery ("*forsythe*", *n*) [Function File]
a = gallery ("*forsythe*", *n*, *alpha*) [Function File]
a = gallery ("*forsythe*", *n*, *alpha*, *lambda*) [Function File]
Create a Forsythe matrix (a perturbed Jordan block).

f = gallery ("*frank*", *n*) [Function File]
f = gallery ("*frank*", *n*, *k*) [Function File]
Create a Frank matrix (ill-conditioned eigenvalues).

`c = gallery ("gcdmat", n)` [Function File]

Create a greatest common divisor matrix.

c is an *n*-by-*n* matrix whose values correspond to the greatest common divisor of its coordinate values, i.e., *c*(i,j) correspond `gcd (i, j)`.

`a = gallery ("gearmat", n)` [Function File]
`a = gallery ("gearmat", n, i)` [Function File]
`a = gallery ("gearmat", n, i, j)` [Function File]

Create a Gear matrix.

`g = gallery ("grcar", n)` [Function File]
`g = gallery ("grcar", n, k)` [Function File]

Create a Toeplitz matrix with sensitive eigenvalues.

`a = gallery ("hanowa", n)` [Function File]
`a = gallery ("hanowa", n, d)` [Function File]

Create a matrix whose eigenvalues lie on a vertical line in the complex plane.

`v = gallery ("house", x)` [Function File]
`[v, beta] = gallery ("house", x)` [Function File]

Create a householder matrix.

`a = gallery ("integerdata", imax, [M N ...], j)` [Function File]
`a = gallery ("integerdata", imax, M, N, ..., j)` [Function File]
`a = gallery ("integerdata", [imin, imax], [M N ...], j)` [Function File]
`a = gallery ("integerdata", [imin, imax], M, N, ..., j)` [Function File]
`a = gallery ("integerdata", ..., "class")` [Function File]

Create a matrix with random integers in the range [1, *imax*]. If *imin* is given then the integers are in the range [*imin*, *imax*].

The second input is a matrix of dimensions describing the size of the output. The dimensions can also be input as comma-separated arguments.

The input *j* is an integer index in the range [0, 2^32-1]. The values of the output matrix are always exactly the same (reproducibility) for a given size input and *j* index.

The final optional argument determines the class of the resulting matrix. Possible values for *class*: `"uint8"`, `"uint16"`, `"uint32"`, `"int8"`, `"int16"`, int32", `"single"`, `"double"`. The default is `"double"`.

`a = gallery ("invhess", x)` [Function File]
`a = gallery ("invhess", x, y)` [Function File]

Create the inverse of an upper Hessenberg matrix.

`a = gallery ("invol", n)` [Function File]

Create an involutory matrix.

`a = gallery ("ipjfact", n)` [Function File]
`a = gallery ("ipjfact", n, k)` [Function File]

Create a Hankel matrix with factorial elements.

a = gallery ("*jordbloc*", *n*) [Function File]
a = gallery ("*jordbloc*", *n*, *lambda*) [Function File]
Create a Jordan block.

u = gallery ("*kahan*", *n*) [Function File]
u = gallery ("*kahan*", *n*, *theta*) [Function File]
u = gallery ("*kahan*", *n*, *theta*, *pert*) [Function File]
Create a Kahan matrix (upper trapezoidal).

a = gallery ("*kms*", *n*) [Function File]
a = gallery ("*kms*", *n*, *rho*) [Function File]
Create a Kac-Murdock-Szego Toeplitz matrix.

b = gallery ("*krylov*", a) [Function File]
b = gallery ("*krylov*", a, *x*) [Function File]
b = gallery ("*krylov*", a, *x*, *j*) [Function File]
Create a Krylov matrix.

a = gallery ("*lauchli*", *n*) [Function File]
a = gallery ("*lauchli*", *n*, *mu*) [Function File]
Create a Lauchli matrix (rectangular).

a = gallery ("*lehmer*", *n*) [Function File]
Create a Lehmer matrix (symmetric positive definite).

t = gallery ("*lesp*", *n*) [Function File]
Create a tridiagonal matrix with real, sensitive eigenvalues.

a = gallery ("*lotkin*", *n*) [Function File]
Create a Lotkin matrix.

a = gallery ("*minij*", *n*) [Function File]
Create a symmetric positive definite matrix MIN(i,j).

a = gallery ("*moler*", *n*) [Function File]
a = gallery ("*moler*", *n*, *alpha*) [Function File]
Create a Moler matrix (symmetric positive definite).

[a, *t*] = gallery ("*neumann*", *n*) [Function File]
Create a singular matrix from the discrete Neumann problem (sparse).

a = gallery ("*normaldata*", [*M N* ...], *j*) [Function File]
a = gallery ("*normaldata*", *M*, *N*, ..., *j*) [Function File]
a = gallery ("*normaldata*", ..., "*class*") [Function File]
Create a matrix with random samples from the standard normal distribution (mean = 0, std = 1).

The first input is a matrix of dimensions describing the size of the output. The dimensions can also be input as comma-separated arguments.

The input *j* is an integer index in the range [0, 2^32-1]. The values of the output matrix are always exactly the same (reproducibility) for a given size input and *j* index.

The final optional argument determines the class of the resulting matrix. Possible values for *class*: "single", "double". The default is "double".

q = gallery ("*orthog*", n) [Function File]
q = gallery ("*orthog*", n, k) [Function File]
Create orthogonal and nearly orthogonal matrices.

a = gallery ("*parter*", n) [Function File]
Create a Parter matrix (a Toeplitz matrix with singular values near pi).

p = gallery ("*pei*", n) [Function File]
p = gallery ("*pei*", n, alpha) [Function File]
Create a Pei matrix.

a = gallery ("*Poisson*", n) [Function File]
Create a block tridiagonal matrix from Poisson's equation (sparse).

a = gallery ("*prolate*", n) [Function File]
a = gallery ("*prolate*", n, w) [Function File]
Create a prolate matrix (symmetric, ill-conditioned Toeplitz matrix).

h = gallery ("*randhess*", x) [Function File]
Create a random, orthogonal upper Hessenberg matrix.

a = gallery ("*rando*", n) [Function File]
a = gallery ("*rando*", n, k) [Function File]
Create a random matrix with elements -1, 0 or 1.

a = gallery ("*randsvd*", n) [Function File]
a = gallery ("*randsvd*", n, kappa) [Function File]
a = gallery ("*randsvd*", n, kappa, mode) [Function File]
a = gallery ("*randsvd*", n, kappa, mode, kl) [Function File]
a = gallery ("*randsvd*", n, kappa, mode, kl, ku) [Function File]
Create a random matrix with pre-assigned singular values.

a = gallery ("*redheff*", n) [Function File]
Create a zero and ones matrix of Redheffer associated with the Riemann hypothesis.

a = gallery ("*riemann*", n) [Function File]
Create a matrix associated with the Riemann hypothesis.

a = gallery ("*ris*", n) [Function File]
Create a symmetric Hankel matrix.

a = gallery ("*smoke*", n) [Function File]
a = gallery ("*smoke*", n, k) [Function File]
Create a complex matrix, with a 'smoke ring' pseudospectrum.

t = gallery ("*toeppd*", n) [Function File]
t = gallery ("*toeppd*", n, m) [Function File]
t = gallery ("*toeppd*", n, m, w) [Function File]
t = gallery ("*toeppd*", n, m, w, theta) [Function File]
Create a symmetric positive definite Toeplitz matrix.

p = gallery ("*toeppen*", *n*) [Function File]
p = gallery ("*toeppen*", *n*, *a*) [Function File]
p = gallery ("*toeppen*", *n*, *a*, *b*) [Function File]
p = gallery ("*toeppen*", *n*, *a*, *b*, *c*) [Function File]
p = gallery ("*toeppen*", *n*, *a*, *b*, *c*, *d*) [Function File]
p = gallery ("*toeppen*", *n*, *a*, *b*, *c*, *d*, *e*) [Function File]

Create a pentadiagonal Toeplitz matrix (sparse).

a = gallery ("*tridiag*", *x*, *y*, *z*) [Function File]
a = gallery ("*tridiag*", *n*) [Function File]
a = gallery ("*tridiag*", *n*, *c*, *d*, *e*) [Function File]

Create a tridiagonal matrix (sparse).

t = gallery ("*triw*", *n*) [Function File]
t = gallery ("*triw*", *n*, *alpha*) [Function File]
t = gallery ("*triw*", *n*, *alpha*, *k*) [Function File]

Create an upper triangular matrix discussed by Kahan, Golub, and Wilkinson.

a = gallery ("*uniformdata*", [*M N* ...], *j*) [Function File]
a = gallery ("*uniformdata*", *M*, *N*, ..., *j*) [Function File]
a = gallery ("*uniformdata*", ..., "*class*") [Function File]

Create a matrix with random samples from the standard uniform distribution (range [0,1]).

The first input is a matrix of dimensions describing the size of the output. The dimensions can also be input as comma-separated arguments.

The input *j* is an integer index in the range [0, 2^32-1]. The values of the output matrix are always exactly the same (reproducibility) for a given size input and *j* index.

The final optional argument determines the class of the resulting matrix. Possible values for *class*: `"single"`, `"double"`. The default is `"double"`.

a = gallery ("*wathen*", *nx*, *ny*) [Function File]
a = gallery ("*wathen*", *nx*, *ny*, *k*) [Function File]

Create the Wathen matrix.

[*a*, *b*] = gallery ("*wilk*", *n*) [Function File]

Create various specific matrices devised/discussed by Wilkinson.

hadamard (*n*) [Function File]

Construct a Hadamard matrix (Hn) of size *n*-by-*n*.

The size *n* must be of the form $2^k * p$ in which p is one of 1, 12, 20 or 28. The returned matrix is normalized, meaning `Hn(:,1) == 1` and `Hn(1,:) == 1`.

Some of the properties of Hadamard matrices are:

- `kron (Hm, Hn)` is a Hadamard matrix of size *m*-by-*n*.
- `Hn * Hn' = n * eye (n)`.
- The rows of Hn are orthogonal.
- `det (A) <= abs (det (Hn))` for all *A* with `abs (A(i, j)) <= 1`.

- Multiplying any row or column by -1 and the matrix will remain a Hadamard matrix.

See also: [compan], page 640, [hankel], page 432, [toeplitz], page 433.

`hankel (c)` [Function File]
`hankel (c, r)` [Function File]

Return the Hankel matrix constructed from the first column *c*, and (optionally) the last row *r*.

If the last element of *c* is not the same as the first element of *r*, the last element of *c* is used. If the second argument is omitted, it is assumed to be a vector of zeros with the same size as *c*.

A Hankel matrix formed from an m-vector *c*, and an n-vector *r*, has the elements

$$H(i,j) = \begin{cases} c_{i+j-1}, & i+j-1 \le m; \\ r_{i+j-m}, & \text{otherwise.} \end{cases}$$

See also: [hadamard], page 431, [toeplitz], page 433.

`hilb (n)` [Function File]

Return the Hilbert matrix of order *n*.

The i,j element of a Hilbert matrix is defined as

$$H(i,j) = \frac{1}{(i+j-1)}$$

Hilbert matrices are close to being singular which make them difficult to invert with numerical routines. Comparing the condition number of a random matrix 5x5 matrix with that of a Hilbert matrix of order 5 reveals just how difficult the problem is.

```
cond (rand (5))
   ⇒ 14.392
cond (hilb (5))
   ⇒ 4.7661e+05
```

See also: [invhilb], page 432.

`invhilb (n)` [Function File]

Return the inverse of the Hilbert matrix of order *n*.

This can be computed exactly using

$$\begin{aligned} A_{ij} &= -1^{i+j}(i+j-1)\binom{n+i-1}{n-j}\binom{n+j-1}{n-i}\binom{i+j-2}{i-2}^2 \\ &= \frac{p(i)p(j)}{(i+j-1)} \end{aligned}$$

where

$$p(k) = -1^k \binom{k+n-1}{k-1}\binom{n}{k}$$

The validity of this formula can easily be checked by expanding the binomial coefficients in both formulas as factorials. It can be derived more directly via the theory of Cauchy matrices. See J. W. Demmel, *Applied Numerical Linear Algebra*, p. 92.

Compare this with the numerical calculation of `inverse (hilb (n))`, which suffers from the ill-conditioning of the Hilbert matrix, and the finite precision of your computer's floating point arithmetic.

See also: [hilb], page 432.

`magic (n)` [Function File]

Create an n-by-n magic square.

A magic square is an arrangement of the integers `1:n^2` such that the row sums, column sums, and diagonal sums are all equal to the same value.

Note: n must be greater than 2 for the magic square to exist.

`pascal (n)` [Function File]
`pascal (n, t)` [Function File]

Return the Pascal matrix of order n if `t = 0`.

The default value of t is 0.

When `t = 1`, return the pseudo-lower triangular Cholesky factor of the Pascal matrix (The sign of some columns may be negative). This matrix is its own inverse, that is `pascal (n, 1) ^ 2 == eye (n)`.

If `t = -1`, return the true Cholesky factor with strictly positive values on the diagonal.

If `t = 2`, return a transposed and permuted version of `pascal (n, 1)`, which is the cube root of the identity matrix. That is, `pascal (n, 2) ^ 3 == eye (n)`.

See also: [chol], page 472.

`rosser ()` [Function File]

Return the Rosser matrix.

This is a difficult test case used to evaluate eigenvalue algorithms.

See also: [wilkinson], page 434, [eig], page 467.

`toeplitz (c)` [Function File]
`toeplitz (c, r)` [Function File]

Return the Toeplitz matrix constructed from the first column c, and (optionally) the first row r.

If the first element of r is not the same as the first element of c, the first element of c is used. If the second argument is omitted, the first row is taken to be the same as the first column.

A square Toeplitz matrix has the form:

$$\begin{bmatrix} c_0 & r_1 & r_2 & \cdots & r_n \\ c_1 & c_0 & r_1 & \cdots & r_{n-1} \\ c_2 & c_1 & c_0 & \cdots & r_{n-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_n & c_{n-1} & c_{n-2} & \ldots & c_0 \end{bmatrix}$$

See also: [hankel], page 432.

`vander (c)` [Function File]
`vander (c, n)` [Function File]

Return the Vandermonde matrix whose next to last column is *c*.

If *n* is specified, it determines the number of columns; otherwise, *n* is taken to be equal to the length of *c*.

A Vandermonde matrix has the form:

$$\begin{bmatrix} c_1^{n-1} & \cdots & c_1^2 & c_1 & 1 \\ c_2^{n-1} & \cdots & c_2^2 & c_2 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ c_n^{n-1} & \cdots & c_n^2 & c_n & 1 \end{bmatrix}$$

See also: [polyfit], page 645.

`wilkinson (n)` [Function File]

Return the Wilkinson matrix of order *n*.

Wilkinson matrices are symmetric and tridiagonal with pairs of nearly, but not exactly, equal eigenvalues. They are useful in testing the behavior and performance of eigenvalue solvers.

See also: [rosser], page 433, [eig], page 467.

17 Arithmetic

Unless otherwise noted, all of the functions described in this chapter will work for real and complex scalar, vector, or matrix arguments. Functions described as *mapping functions* apply the given operation individually to each element when given a matrix argument. For example:

```
sin ([1, 2; 3, 4])
     ⇒  0.84147   0.90930
        0.14112  -0.75680
```

17.1 Exponents and Logarithms

`exp (x)` [Mapping Function]

Compute e^x for each element of *x*.

To compute the matrix exponential, see Chapter 18 [Linear Algebra], page 465.

See also: [log], page 435.

`expm1 (x)` [Mapping Function]

Compute $e^x - 1$ accurately in the neighborhood of zero.

See also: [exp], page 435.

`log (x)` [Mapping Function]

Compute the natural logarithm, $\ln(x)$, for each element of *x*.

To compute the matrix logarithm, see Chapter 18 [Linear Algebra], page 465.

See also: [exp], page 435, [log1p], page 435, [log2], page 435, [log10], page 435, [logspace], page 420.

`reallog (x)` [Function File]

Return the real-valued natural logarithm of each element of *x*.

If any element results in a complex return value `reallog` aborts and issues an error.

See also: [log], page 435, [realpow], page 436, [realsqrt], page 436.

`log1p (x)` [Mapping Function]

Compute $\ln(1 + x)$ accurately in the neighborhood of zero.

See also: [log], page 435, [exp], page 435, [expm1], page 435.

`log10 (x)` [Mapping Function]

Compute the base-10 logarithm of each element of *x*.

See also: [log], page 435, [log2], page 435, [logspace], page 420, [exp], page 435.

`log2 (x)` [Mapping Function]
`[f, e] = log2 (x)` [Mapping Function]

Compute the base-2 logarithm of each element of *x*.

If called with two output arguments, split *x* into binary mantissa and exponent so that $\frac{1}{2} \le |f| < 1$ and *e* is an integer. If $x = 0$, $f = e = 0$.

See also: [pow2], page 436, [log], page 435, [log10], page 435, [exp], page 435.

`pow2 (x)` [Function File]
`pow2 (f, e)` [Function File]

With one input argument, compute 2^x for each element of *x*.

With two input arguments, return $f \cdot 2^e$.

See also: [log2], page 435, [nextpow2], page 436, [power], page 144.

`nextpow2 (x)` [Function File]

Compute the exponent for the smallest power of two larger than the input.

For each element in the input array *x*, return the first integer *n* such that $2^n \geq |x|$.

See also: [pow2], page 436, [log2], page 435.

`realpow (x, y)` [Function File]

Compute the real-valued, element-by-element power operator.

This is equivalent to `x .^ y`, except that `realpow` reports an error if any return value is complex.

See also: [power], page 144, [reallog], page 435, [realsqrt], page 436.

`sqrt (x)` [Mapping Function]

Compute the square root of each element of *x*.

If *x* is negative, a complex result is returned.

To compute the matrix square root, see Chapter 18 [Linear Algebra], page 465.

See also: [realsqrt], page 436, [nthroot], page 436.

`realsqrt (x)` [Function File]

Return the real-valued square root of each element of *x*.

If any element results in a complex return value `realsqrt` aborts and issues an error.

See also: [sqrt], page 436, [realpow], page 436, [reallog], page 435.

`cbrt (x)` [Mapping Function]

Compute the real cube root of each element of *x*.

Unlike `x^(1/3)`, the result will be negative if *x* is negative.

See also: [nthroot], page 436.

`nthroot (x, n)` [Function File]

Compute the real (non-complex) *n*-th root of *x*.

x must have all real entries and *n* must be a scalar. If *n* is an even integer and *x* has negative entries then `nthroot` aborts and issues an error.

Example:

```
nthroot (-1, 3)
⇒ -1
(-1) ^ (1 / 3)
⇒ 0.50000 - 0.86603i
```

See also: [realsqrt], page 436, [sqrt], page 436, [cbrt], page 436.

17.2 Complex Arithmetic

In the descriptions of the following functions, z is the complex number $x + iy$, where i is defined as $\sqrt{-1}$.

`abs (z)` [Mapping Function]

Compute the magnitude of z.

The magnitude is defined as $|z| = \sqrt{x^2 + y^2}$.

For example:

```
abs (3 + 4i)
     ⇒ 5
```

See also: [arg], page 437.

`arg (z)` [Mapping Function]
`angle (z)` [Mapping Function]

Compute the argument, i.e., angle of z.

This is defined as, $\theta = atan2(y, x)$, in radians.

For example:

```
arg (3 + 4i)
     ⇒ 0.92730
```

See also: [abs], page 437.

`conj (z)` [Mapping Function]

Return the complex conjugate of z.

The complex conjugate is defined as $\bar{z} = x - iy$.

See also: [real], page 438, [imag], page 437.

`cplxpair (z)` [Function File]
`cplxpair (z, tol)` [Function File]
`cplxpair (z, tol, dim)` [Function File]

Sort the numbers z into complex conjugate pairs ordered by increasing real part.

The negative imaginary complex numbers are placed first within each pair. All real numbers (those with `abs (imag (z) / z) < tol`) are placed after the complex pairs.

If *tol* is unspecified the default value is 100*`eps`.

By default the complex pairs are sorted along the first non-singleton dimension of z. If *dim* is specified, then the complex pairs are sorted along this dimension.

Signal an error if some complex numbers could not be paired. Signal an error if all complex numbers are not exact conjugates (to within *tol*). Note that there is no defined order for pairs with identical real parts but differing imaginary parts.

```
cplxpair (exp(2i*pi*[0:4]'/5)) == exp(2i*pi*[3; 2; 4; 1; 0]/5)
```

`imag (z)` [Mapping Function]

Return the imaginary part of z as a real number.

See also: [real], page 438, [conj], page 437.

`real (z)` [Mapping Function]

Return the real part of *z*.

See also: [imag], page 437, [conj], page 437.

17.3 Trigonometry

Octave provides the following trigonometric functions where angles are specified in radians. To convert from degrees to radians multiply by $\pi/180$ (e.g., `sin (30 * pi/180)` returns the sine of 30 degrees). As an alternative, Octave provides a number of trigonometric functions which work directly on an argument specified in degrees. These functions are named after the base trigonometric function with a '`d`' suffix. For example, `sin` expects an angle in radians while `sind` expects an angle in degrees.

Octave uses the C library trigonometric functions. It is expected that these functions are defined by the ISO/IEC 9899 Standard. This Standard is available at: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`. Section F.9.1 deals with the trigonometric functions. The behavior of most of the functions is relatively straightforward. However, there are some exceptions to the standard behavior. Many of the exceptions involve the behavior for -0. The most complex case is atan2. Octave exactly implements the behavior given in the Standard. Including $atan2(\pm 0, -0)$ returns $\pm\pi$.

It should be noted that MATLAB uses different definitions which apparently do not distinguish -0.

`sin (x)` [Mapping Function]

Compute the sine for each element of *x* in radians.

See also: [asin], page 439, [sind], page 440, [sinh], page 439.

`cos (x)` [Mapping Function]

Compute the cosine for each element of *x* in radians.

See also: [acos], page 439, [cosd], page 441, [cosh], page 439.

`tan (z)` [Mapping Function]

Compute the tangent for each element of *x* in radians.

See also: [atan], page 439, [tand], page 441, [tanh], page 439.

`sec (x)` [Mapping Function]

Compute the secant for each element of *x* in radians.

See also: [asec], page 439, [secd], page 441, [sech], page 439.

`csc (x)` [Mapping Function]

Compute the cosecant for each element of *x* in radians.

See also: [acsc], page 439, [cscd], page 441, [csch], page 439.

`cot (x)` [Mapping Function]

Compute the cotangent for each element of *x* in radians.

See also: [acot], page 439, [cotd], page 441, [coth], page 439.

`asin` (x) [Mapping Function]
Compute the inverse sine in radians for each element of x.

See also: [sin], page 438, [asind], page 441.

`acos` (x) [Mapping Function]
Compute the inverse cosine in radians for each element of x.

See also: [cos], page 438, [acosd], page 441.

`atan` (x) [Mapping Function]
Compute the inverse tangent in radians for each element of x.

See also: [tan], page 438, [atand], page 441.

`asec` (x) [Mapping Function]
Compute the inverse secant in radians for each element of x.

See also: [sec], page 438, [asecd], page 441.

`acsc` (x) [Mapping Function]
Compute the inverse cosecant in radians for each element of x.

See also: [csc], page 438, [acscd], page 441.

`acot` (x) [Mapping Function]
Compute the inverse cotangent in radians for each element of x.

See also: [cot], page 438, [acotd], page 442.

`sinh` (x) [Mapping Function]
Compute the hyperbolic sine for each element of x.

See also: [asinh], page 440, [cosh], page 439, [tanh], page 439.

`cosh` (x) [Mapping Function]
Compute the hyperbolic cosine for each element of x.

See also: [acosh], page 440, [sinh], page 439, [tanh], page 439.

`tanh` (x) [Mapping Function]
Compute hyperbolic tangent for each element of x.

See also: [atanh], page 440, [sinh], page 439, [cosh], page 439.

`sech` (x) [Mapping Function]
Compute the hyperbolic secant of each element of x.

See also: [asech], page 440.

`csch` (x) [Mapping Function]
Compute the hyperbolic cosecant of each element of x.

See also: [acsch], page 440.

`coth` (x) [Mapping Function]
Compute the hyperbolic cotangent of each element of x.

See also: [acoth], page 440.

`asinh (x)` [Mapping Function]

Compute the inverse hyperbolic sine for each element of *x*.

See also: [sinh], page 439.

`acosh (x)` [Mapping Function]

Compute the inverse hyperbolic cosine for each element of *x*.

See also: [cosh], page 439.

`atanh (x)` [Mapping Function]

Compute the inverse hyperbolic tangent for each element of *x*.

See also: [tanh], page 439.

`asech (x)` [Mapping Function]

Compute the inverse hyperbolic secant of each element of *x*.

See also: [sech], page 439.

`acsch (x)` [Mapping Function]

Compute the inverse hyperbolic cosecant of each element of *x*.

See also: [csch], page 439.

`acoth (x)` [Mapping Function]

Compute the inverse hyperbolic cotangent of each element of *x*.

See also: [coth], page 439.

`atan2 (y, x)` [Mapping Function]

Compute atan (*y* / *x*) for corresponding elements of *y* and *x*.

y and *x* must match in size and orientation. The signs of elements of *y* and *x* are used to determine the quadrats of each resulting value.

This function is equivalent to `arg (complex (x, y))`.

See also: [tan], page 438, [tand], page 441, [tanh], page 439, [atanh], page 440.

Octave provides the following trigonometric functions where angles are specified in degrees. These functions produce true zeros at the appropriate intervals rather than the small round-off error that occurs when using radians. For example:

```
cosd (90)
     ⇒ 0
cos (pi/2)
     ⇒ 6.1230e-17
```

`sind (x)` [Function File]

Compute the sine for each element of *x* in degrees.

Returns zero for elements where `x/180` is an integer.

See also: [asind], page 441, [sin], page 438.

`cosd` (*x*) [Function File]

Compute the cosine for each element of *x* in degrees.

Returns zero for elements where `(x-90)/180` is an integer.

See also: [acosd], page 441, [cos], page 438.

`tand` (*x*) [Function File]

Compute the tangent for each element of *x* in degrees.

Returns zero for elements where `x/180` is an integer and `Inf` for elements where `(x-90)/180` is an integer.

See also: [atand], page 441, [tan], page 438.

`secd` (*x*) [Function File]

Compute the secant for each element of *x* in degrees.

See also: [asecd], page 441, [sec], page 438.

`cscd` (*x*) [Function File]

Compute the cosecant for each element of *x* in degrees.

See also: [acscd], page 441, [csc], page 438.

`cotd` (*x*) [Function File]

Compute the cotangent for each element of *x* in degrees.

See also: [acotd], page 442, [cot], page 438.

`asind` (*x*) [Function File]

Compute the inverse sine in degrees for each element of *x*.

See also: [sind], page 440, [asin], page 439.

`acosd` (*x*) [Function File]

Compute the inverse cosine in degrees for each element of *x*.

See also: [cosd], page 441, [acos], page 439.

`atand` (*x*) [Function File]

Compute the inverse tangent in degrees for each element of *x*.

See also: [tand], page 441, [atan], page 439.

`atan2d` (*y*, *x*) [Function File]

Compute atan2 (y / x) in degrees for corresponding elements from *y* and *x*.

See also: [tand], page 441, [atan2], page 440.

`asecd` (*x*) [Function File]

Compute the inverse secant in degrees for each element of *x*.

See also: [secd], page 441, [asec], page 439.

`acscd` (*x*) [Function File]

Compute the inverse cosecant in degrees for each element of *x*.

See also: [cscd], page 441, [acsc], page 439.

`acotd` (*x*) [Function File]

Compute the inverse cotangent in degrees for each element of *x*.

See also: [cotd], page 441, [acot], page 439.

17.4 Sums and Products

`sum` (*x*) [Built-in Function]
`sum` (*x*, *dim*) [Built-in Function]
`sum` (. . . , "*native*") [Built-in Function]
`sum` (. . . , "*double*") [Built-in Function]
`sum` (. . . , "*extra*") [Built-in Function]

Sum of elements along dimension *dim*.

If *dim* is omitted, it defaults to the first non-singleton dimension.

The optional `"type"` input determines the class of the variable used for calculations. If the argument `"native"` is given, then the operation is performed in the same type as the original argument, rather than the default double type.

For example:

```
sum ([true, true])
   ⇒ 2
sum ([true, true], "native")
   ⇒ true
```

On the contrary, if `"double"` is given, the sum is performed in double precision even for single precision inputs.

For double precision inputs, the `"extra"` option will use a more accurate algorithm than straightforward summation. For single precision inputs, `"extra"` is the same as `"double"`. Otherwise, `"extra"` has no effect.

See also: [cumsum], page 443, [sumsq], page 443, [prod], page 442.

`prod` (*x*) [Built-in Function]
`prod` (*x*, *dim*) [Built-in Function]
`prod` (. . . , "*native*") [Built-in Function]
`prod` (. . . , "*double*") [Built-in Function]

Product of elements along dimension *dim*.

If *dim* is omitted, it defaults to the first non-singleton dimension.

The optional `"type"` input determines the class of the variable used for calculations. If the argument `"native"` is given, then the operation is performed in the same type as the original argument, rather than the default double type.

For example:

```
prod ([true, true])
   ⇒ 1
prod ([true, true], "native")
   ⇒ true
```

On the contrary, if `"double"` is given, the operation is performed in double precision even for single precision inputs.

See also: [cumprod], page 443, [sum], page 442.

`cumsum` (*x*) [Built-in Function]
`cumsum` (*x*, *dim*) [Built-in Function]
`cumsum` (. . . , "*native*") [Built-in Function]
`cumsum` (. . . , "*double*") [Built-in Function]
`cumsum` (. . . , "*extra*") [Built-in Function]

Cumulative sum of elements along dimension *dim*.

If *dim* is omitted, it defaults to the first non-singleton dimension.

See `sum` for an explanation of the optional parameters `"native"`, `"double"`, and `"extra"`.

See also: [sum], page 442, [cumprod], page 443.

`cumprod` (*x*) [Built-in Function]
`cumprod` (*x*, *dim*) [Built-in Function]

Cumulative product of elements along dimension *dim*.

If *dim* is omitted, it defaults to the first non-singleton dimension.

See also: [prod], page 442, [cumsum], page 443.

`sumsq` (*x*) [Built-in Function]
`sumsq` (*x*, *dim*) [Built-in Function]

Sum of squares of elements along dimension *dim*.

If *dim* is omitted, it defaults to the first non-singleton dimension.

This function is conceptually equivalent to computing

```
sum (x .* conj (x), dim)
```

but it uses less memory and avoids calling `conj` if *x* is real.

See also: [sum], page 442, [prod], page 442.

17.5 Utility Functions

`ceil` (*x*) [Mapping Function]

Return the smallest integer not less than *x*.

This is equivalent to rounding towards positive infinity.

If *x* is complex, return `ceil (real (x)) + ceil (imag (x)) * I`.

```
ceil ([-2.7, 2.7])
    ⇒ -2    3
```

See also: [floor], page 444, [round], page 444, [fix], page 443.

`fix` (*x*) [Mapping Function]

Truncate fractional portion of *x* and return the integer portion.

This is equivalent to rounding towards zero. If *x* is complex, return `fix (real (x)) + fix (imag (x)) * I`.

```
fix ([-2.7, 2.7])
   ⇒ -2    2
```

See also: [ceil], page 443, [floor], page 444, [round], page 444.

`floor (x)` [Mapping Function]

Return the largest integer not greater than *x*.

This is equivalent to rounding towards negative infinity. If *x* is complex, return `floor (real (x)) + floor (imag (x)) * I`.

```
floor ([-2.7, 2.7])
     ⇒ -3    2
```

See also: [ceil], page 443, [round], page 444, [fix], page 443.

`round (x)` [Mapping Function]

Return the integer nearest to *x*.

If *x* is complex, return `round (real (x)) + round (imag (x)) * I`. If there are two nearest integers, return the one further away from zero.

```
round ([-2.7, 2.7])
     ⇒ -3    3
```

See also: [ceil], page 443, [floor], page 444, [fix], page 443, [roundb], page 444.

`roundb (x)` [Mapping Function]

Return the integer nearest to *x*. If there are two nearest integers, return the even one (banker's rounding).

If *x* is complex, return `roundb (real (x)) + roundb (imag (x)) * I`.

See also: [round], page 444.

`max (x)` [Built-in Function]
`max (x, [], dim)` [Built-in Function]
`[w, iw] = max (x)` [Built-in Function]
`max (x, y)` [Built-in Function]

Find maximum values in the array *x*.

For a vector argument, return the maximum value. For a matrix argument, return a row vector with the maximum value of each column. For a multi-dimensional array, `max` operates along the first non-singleton dimension.

If the optional third argument *dim* is present then operate along this dimension. In this case the second argument is ignored and should be set to the empty matrix.

For two matrices (or a matrix and a scalar), return the pairwise maximum.

Thus,

```
max (max (x))
```

returns the largest element of the 2-D matrix *x*, and

```
max (2:5, pi)
    ⇒  3.1416  3.1416  4.0000  5.0000
```

compares each element of the range `2:5` with `pi`, and returns a row vector of the maximum values.

For complex arguments, the magnitude of the elements are used for comparison. If the magnitudes are identical, then the results are ordered by phase angle in the range (-pi, pi]. Hence,

```
max ([-1 i 1 -i])
    ⇒ -1
```

because all entries have magnitude 1, but -1 has the largest phase angle with value pi.

If called with one input and two output arguments, `max` also returns the first index of the maximum value(s). Thus,

```
[x, ix] = max ([1, 3, 5, 2, 5])
    ⇒  x = 5
        ix = 3
```

See also: [min], page 445, [cummax], page 445, [cummin], page 446.

`min (x)` [Built-in Function]
`min (x, [], dim)` [Built-in Function]
`[w, iw] = min (x)` [Built-in Function]
`min (x, y)` [Built-in Function]

Find minimum values in the array *x*.

For a vector argument, return the minimum value. For a matrix argument, return a row vector with the minimum value of each column. For a multi-dimensional array, `min` operates along the first non-singleton dimension.

If the optional third argument *dim* is present then operate along this dimension. In this case the second argument is ignored and should be set to the empty matrix.

For two matrices (or a matrix and a scalar), return the pairwise minimum.

Thus,

```
min (min (x))
```

returns the smallest element of the 2-D matrix *x*, and

```
min (2:5, pi)
    ⇒  2.0000  3.0000  3.1416  3.1416
```

compares each element of the range `2:5` with `pi`, and returns a row vector of the minimum values.

For complex arguments, the magnitude of the elements are used for comparison. If the magnitudes are identical, then the results are ordered by phase angle in the range (-pi, pi]. Hence,

```
min ([-1 i 1 -i])
    ⇒ -i
```

because all entries have magnitude 1, but -i has the smallest phase angle with value -pi/2.

If called with one input and two output arguments, `min` also returns the first index of the minimum value(s). Thus,

```
[x, ix] = min ([1, 3, 0, 2, 0])
    ⇒  x = 0
        ix = 3
```

See also: [max], page 444, [cummin], page 446, [cummax], page 445.

`cummax (x)` [Built-in Function]
`cummax (x, dim)` [Built-in Function]

`[w, iw] = cummax (...)` [Built-in Function]

Return the cumulative maximum values along dimension *dim*.

If *dim* is unspecified it defaults to column-wise operation. For example:

```
cummax ([1 3 2 6 4 5])
   ⇒  1  3  3  6  6  6
```

If called with two output arguments the index of the maximum value is also returned.

```
[w, iw] = cummax ([1 3 2 6 4 5])
⇒
w =  1  3  3  6  6  6
iw = 1  2  2  4  4  4
```

See also: [cummin], page 446, [max], page 444, [min], page 445.

`cummin (x)` [Built-in Function]
`cummin (x, dim)` [Built-in Function]
`[w, iw] = cummin (x)` [Built-in Function]

Return the cumulative minimum values along dimension *dim*.

If *dim* is unspecified it defaults to column-wise operation. For example:

```
cummin ([5 4 6 2 3 1])
   ⇒  5  4  4  2  2  1
```

If called with two output arguments the index of the minimum value is also returned.

```
[w, iw] = cummin ([5 4 6 2 3 1])
⇒
w =  5  4  4  2  2  1
iw = 1  2  2  4  4  6
```

See also: [cummax], page 445, [min], page 445, [max], page 444.

`hypot (x, y)` [Built-in Function]
`hypot (x, y, z, ...)` [Built-in Function]

Compute the element-by-element square root of the sum of the squares of *x* and *y*.

This is equivalent to `sqrt (x.^2 + y.^2)`, but is calculated in a manner that avoids overflows for large values of *x* or *y*.

`hypot` can also be called with more than 2 arguments; in this case, the arguments are accumulated from left to right:

```
hypot (hypot (x, y), z)
hypot (hypot (hypot (x, y), z), w), etc.
```

`dx = gradient (m)` [Function File]
`[dx, dy, dz, ...] = gradient (m)` [Function File]
`[...] = gradient (m, s)` [Function File]
`[...] = gradient (m, x, y, z, ...)` [Function File]
`[...] = gradient (f, x0)` [Function File]
`[...] = gradient (f, x0, s)` [Function File]
`[...] = gradient (f, x0, x, y, ...)` [Function File]

Calculate the gradient of sampled data or a function.

If *m* is a vector, calculate the one-dimensional gradient of *m*. If *m* is a matrix the gradient is calculated for each dimension.

`[dx, dy] = gradient (m)` calculates the one-dimensional gradient for *x* and *y* direction if *m* is a matrix. Additional return arguments can be use for multi-dimensional matrices.

A constant spacing between two points can be provided by the *s* parameter. If *s* is a scalar, it is assumed to be the spacing for all dimensions. Otherwise, separate values of the spacing can be supplied by the *x*, . . . arguments. Scalar values specify an equidistant spacing. Vector values for the *x*, . . . arguments specify the coordinate for that dimension. The length must match their respective dimension of *m*.

At boundary points a linear extrapolation is applied. Interior points are calculated with the first approximation of the numerical gradient

```
y'(i) = 1/(x(i+1)-x(i-1)) * (y(i-1)-y(i+1)).
```

If the first argument *f* is a function handle, the gradient of the function at the points in *x0* is approximated using central difference. For example, `gradient (@cos, 0)` approximates the gradient of the cosine function in the point $x0 = 0$. As with sampled data, the spacing values between the points from which the gradient is estimated can be set via the *s* or *dx*, *dy*, . . . arguments. By default a spacing of 1 is used.

See also: [diff], page 406, [del2], page 448.

`dot` (*x*, *y*, *dim*) [Built-in Function]

Compute the dot product of two vectors.

If *x* and *y* are matrices, calculate the dot products along the first non-singleton dimension.

If the optional argument *dim* is given, calculate the dot products along this dimension.

This is equivalent to `sum (conj (X) .* Y, dim)`, but avoids forming a temporary array and is faster. When *X* and *Y* are column vectors, the result is equivalent to `X' * Y`.

See also: [cross], page 447, [divergence], page 448.

`cross` (*x*, *y*) [Function File]
`cross` (*x*, *y*, *dim*) [Function File]

Compute the vector cross product of two 3-dimensional vectors *x* and *y*.

If *x* and *y* are matrices, the cross product is applied along the first dimension with three elements.

The optional argument *dim* forces the cross product to be calculated along the specified dimension.

Example Code:

```
cross ([1,1,0], [0,1,1])
     ⇒ [ 1; -1; 1 ]
```

See also: [dot], page 447, [curl], page 448, [divergence], page 448.

div = `divergence` (*x*, *y*, *z*, *fx*, *fy*, *fz*) [Function File]
div = `divergence` (*fx*, *fy*, *fz*) [Function File]
div = `divergence` (*x*, *y*, *fx*, *fy*) [Function File]
div = `divergence` (*fx*, *fy*) [Function File]

Calculate divergence of a vector field given by the arrays *fx*, *fy*, and *fz* or *fx*, *fy* respectively.

$$divF(x,y,z) = \partial_x F + \partial_y F + \partial_z F$$

The coordinates of the vector field can be given by the arguments *x*, *y*, *z* or *x*, *y* respectively.

See also: [curl], page 448, [gradient], page 446, [del2], page 448, [dot], page 447.

[*cx*, *cy*, *cz*, *v*] = `curl` (*x*, *y*, *z*, *fx*, *fy*, *fz*) [Function File]
[*cz*, *v*] = `curl` (*x*, *y*, *fx*, *fy*) [Function File]
[...] = `curl` (*fx*, *fy*, *fz*) [Function File]
[...] = `curl` (*fx*, *fy*) [Function File]
v = `curl` (...) [Function File]

Calculate curl of vector field given by the arrays *fx*, *fy*, and *fz* or *fx*, *fy* respectively.

$$curlF(x,y,z) = \left(\frac{\partial d}{\partial y}F_z - \frac{\partial d}{\partial z}F_y, \frac{\partial d}{\partial z}F_x - \frac{\partial d}{\partial x}F_z, \frac{\partial d}{\partial x}F_y - \frac{\partial d}{\partial y}F_x\right)$$

The coordinates of the vector field can be given by the arguments *x*, *y*, *z* or *x*, *y* respectively. *v* calculates the scalar component of the angular velocity vector in direction of the z-axis for two-dimensional input. For three-dimensional input the scalar rotation is calculated at each grid point in direction of the vector field at that point.

See also: [divergence], page 448, [gradient], page 446, [del2], page 448, [cross], page 447.

d = `del2` (*M*) [Function File]
d = `del2` (*M*, *h*) [Function File]
d = `del2` (*M*, *dx*, *dy*, ...) [Function File]

Calculate the discrete Laplace operator (∇^2).

For a 2-dimensional matrix *M* this is defined as

$$d = \frac{1}{4}\left(\frac{d^2}{dx^2}M(x,y) + \frac{d^2}{dy^2}M(x,y)\right)$$

For N-dimensional arrays the sum in parentheses is expanded to include second derivatives over the additional higher dimensions.

The spacing between evaluation points may be defined by *h*, which is a scalar defining the equidistant spacing in all dimensions. Alternatively, the spacing in each dimension may be defined separately by *dx*, *dy*, etc. A scalar spacing argument defines equidistant spacing, whereas a vector argument can be used to specify variable spacing. The length of the spacing vectors must match the respective dimension of *M*. The default spacing value is 1.

At least 3 data points are needed for each dimension. Boundary points are calculated from the linear extrapolation of interior points.

See also: [gradient], page 446, [diff], page 406.

`factorial (n)` [Function File]

Return the factorial of *n* where *n* is a real non-negative integer.

If *n* is a scalar, this is equivalent to `prod (1:n)`. For vector or matrix arguments, return the factorial of each element in the array.

For non-integers see the generalized factorial function `gamma`. Note that the factorial function grows large quite quickly, and even with double precision values overflow will occur if *n* > 171. For such cases consider `gammaln`.

See also: [prod], page 442, [gamma], page 456, [gammaln], page 458.

`pf = factor (q)` [Function File]
`[pf, n] = factor (q)` [Function File]

Return the prime factorization of *q*.

The prime factorization is defined as `prod (pf) == q` where every element of *pf* is a prime number. If `q == 1`, return 1.

With two output arguments, return the unique prime factors *pf* and their multiplicities. That is, `prod (pf .^ n) == q`.

Implementation Note: The input *q* must be less than `bitmax` (9.0072e+15) in order to factor correctly.

See also: [gcd], page 449, [lcm], page 449, [isprime], page 64, [primes], page 450.

`g = gcd (a1, a2, ...)` [Built-in Function]
`[g, v1, ...] = gcd (a1, a2, ...)` [Built-in Function]

Compute the greatest common divisor of *a1*, *a2*, ...

If more than one argument is given then all arguments must be the same size or scalar. In this case the greatest common divisor is calculated for each element individually. All elements must be ordinary or Gaussian (complex) integers. Note that for Gaussian integers, the gcd is only unique up to a phase factor (multiplication by 1, -1, i, or -i), so an arbitrary greatest common divisor among the four possible is returned.

Optional return arguments *v1*, ..., contain integer vectors such that,

$g = v_1 a_1 + v_2 a_2 + \cdots$

Example code:

```
gcd ([15, 9], [20, 18])
   ⇒  5  9
```

See also: [lcm], page 449, [factor], page 449, [isprime], page 64.

`lcm (x, y)` [Mapping Function]
`lcm (x, y, ...)` [Mapping Function]

Compute the least common multiple of *x* and *y*, or of the list of all arguments.

All elements must be numeric and of the same size or scalar.

See also: [factor], page 449, [gcd], page 449, [isprime], page 64.

`chop` (*x*, *ndigits*, *base*) [Function File]

Truncate elements of *x* to a length of *ndigits* such that the resulting numbers are exactly divisible by *base*.

If *base* is not specified it defaults to 10.

```
chop (-pi, 5, 10)
   ⇒ -3.14200000000000
chop (-pi, 5, 5)
   ⇒ -3.14150000000000
```

`rem` (*x*, *y*) [Mapping Function]

Return the remainder of the division `x / y`.

The remainder is computed using the expression

```
x - y .* fix (x ./ y)
```

An error message is printed if the dimensions of the arguments do not agree, or if either of the arguments is complex.

See also: [mod], page 450.

`mod` (*x*, *y*) [Mapping Function]

Compute the modulo of *x* and *y*.

Conceptually this is given by

```
x - y .* floor (x ./ y)
```

and is written such that the correct modulus is returned for integer types. This function handles negative values correctly. That is, `mod (-1, 3)` is 2, not -1, as `rem (-1, 3)` returns. `mod (x, 0)` returns *x*.

An error results if the dimensions of the arguments do not agree, or if either of the arguments is complex.

See also: [rem], page 450.

`primes` (*n*) [Function File]

Return all primes up to *n*.

The output data class (double, single, uint32, etc.) is the same as the input class of *n*. The algorithm used is the Sieve of Eratosthenes.

Notes: If you need a specific number of primes you can use the fact that the distance from one prime to the next is, on average, proportional to the logarithm of the prime. Integrating, one finds that there are about k primes less than $k \log(5k)$.

See also `list_primes` if you need a specific number *n* of primes.

See also: [list_primes], page 450, [isprime], page 64.

`list_primes` () [Function File]
`list_primes` (*n*) [Function File]

List the first *n* primes.

If *n* is unspecified, the first 25 primes are listed.

See also: [primes], page 450, [isprime], page 64.

sign (*x*) [Mapping Function]

Compute the *signum* function.

This is defined as

$$\operatorname{sign}(x) = \begin{cases} 1, & x > 0; \\ 0, & x = 0; \\ -1, & x < 0. \end{cases}$$

For complex arguments, `sign` returns `x ./ abs (x)`.

Note that `sign (-0.0)` is 0. Although IEEE 754 floating point allows zero to be signed, 0.0 and -0.0 compare equal. If you must test whether zero is signed, use the `signbit` function.

See also: [signbit], page 451.

signbit (*x*) [Mapping Function]

Return logical true if the value of *x* has its sign bit set and false otherwise.

This behavior is consistent with the other logical functions. See Section 4.6 [Logical Values], page 60. The behavior differs from the C language function which returns nonzero if the sign bit is set.

This is not the same as `x < 0.0`, because IEEE 754 floating point allows zero to be signed. The comparison `-0.0 < 0.0` is false, but `signbit (-0.0)` will return a nonzero value.

See also: [sign], page 451.

17.6 Special Functions

[a, ierr] = airy (*k*, *z*, *opt*) [Built-in Function]

Compute Airy functions of the first and second kind, and their derivatives.

```
 K   Function   Scale factor (if "opt" is supplied)
---  --------   ---------------------------------------
 0   Ai (Z)     exp ((2/3) * Z * sqrt (Z))
 1   dAi(Z)/dZ  exp ((2/3) * Z * sqrt (Z))
 2   Bi (Z)     exp (-abs (real ((2/3) * Z * sqrt (Z))))
 3   dBi(Z)/dZ  exp (-abs (real ((2/3) * Z * sqrt (Z))))
```

The function call `airy (z)` is equivalent to `airy (0, z)`.

The result is the same size as *z*.

If requested, *ierr* contains the following status information and is the same size as the result.

0. Normal return.
1. Input error, return `NaN`.
2. Overflow, return `Inf`.
3. Loss of significance by argument reduction results in less than half of machine accuracy.
4. Complete loss of significance by argument reduction, return `NaN`.
5. Error—no computation, algorithm termination condition not met, return `NaN`.

`[j, ierr] = besselj (alpha, x, opt)` [Built-in Function]
`[y, ierr] = bessely (alpha, x, opt)` [Built-in Function]
`[i, ierr] = besseli (alpha, x, opt)` [Built-in Function]
`[k, ierr] = besselk (alpha, x, opt)` [Built-in Function]
`[h, ierr] = besselh (alpha, k, x, opt)` [Built-in Function]

Compute Bessel or Hankel functions of various kinds:

`besselj` Bessel functions of the first kind. If the argument *opt* is 1 or true, the result is multiplied by `exp (-abs (imag (x)))`.

`bessely` Bessel functions of the second kind. If the argument *opt* is 1 or true, the result is multiplied by `exp (-abs (imag (x)))`.

`besseli`

Modified Bessel functions of the first kind. If the argument *opt* is 1 or true, the result is multiplied by `exp (-abs (real (x)))`.

`besselk`

Modified Bessel functions of the second kind. If the argument *opt* is 1 or true, the result is multiplied by `exp (x)`.

`besselh` Compute Hankel functions of the first ($k = 1$) or second ($k = 2$) kind. If the argument *opt* is 1 or true, the result is multiplied by `exp (-I*x)` for $k = 1$ or `exp (I*x)` for $k = 2$.

If *alpha* is a scalar, the result is the same size as *x*. If *x* is a scalar, the result is the same size as *alpha*. If *alpha* is a row vector and *x* is a column vector, the result is a matrix with `length (x)` rows and `length (alpha)` columns. Otherwise, *alpha* and *x* must conform and the result will be the same size.

The value of *alpha* must be real. The value of *x* may be complex.

If requested, *ierr* contains the following status information and is the same size as the result.

0. Normal return.
1. Input error, return `NaN`.
2. Overflow, return `Inf`.
3. Loss of significance by argument reduction results in less than half of machine accuracy.
4. Complete loss of significance by argument reduction, return `NaN`.
5. Error—no computation, algorithm termination condition not met, return `NaN`.

`beta (a, b)` [Mapping Function]

Compute the Beta function for real inputs *a* and *b*.

The Beta function definition is

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

The Beta function can grow quite large and it is often more useful to work with the logarithm of the output rather than the function directly. See [betaln], page 453, for computing the logarithm of the Beta function in an efficient manner.

See also: [betaln], page 453, [betainc], page 453, [betaincinv], page 453.

`betainc (x, a, b)` [Mapping Function]

Compute the regularized incomplete Beta function.

The regularized incomplete Beta function is defined by

$$I(x,a,b) = \frac{1}{B(a,b)} \int_0^x t^{(a-z)}(1-t)^{(b-1)}dt.$$

If x has more than one component, both a and b must be scalars. If x is a scalar, a and b must be of compatible dimensions.

See also: [betaincinv], page 453, [beta], page 452, [betaln], page 453.

`betaincinv (y, a, b)` [Mapping Function]

Compute the inverse of the incomplete Beta function.

The inverse is the value x such that

```
y == betainc (x, a, b)
```

See also: [betainc], page 453, [beta], page 452, [betaln], page 453.

`betaln (a, b)` [Mapping Function]

Compute the natural logarithm of the Beta function for real inputs a and b.

`betaln` is defined as

$$\mathrm{betaln}(a,b) = \ln(B(a,b)) \equiv \ln(\frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}).$$

and is calculated in a way to reduce the occurrence of underflow.

The Beta function can grow quite large and it is often more useful to work with the logarithm of the output rather than the function directly.

See also: [beta], page 452, [betainc], page 453, [betaincinv], page 453, [gammaln], page 458.

`bincoeff (n, k)` [Mapping Function]

Return the binomial coefficient of n and k, defined as

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!}$$

For example:

```
bincoeff (5, 2)
   ⇒ 10
```

In most cases, the `nchoosek` function is faster for small scalar integer arguments. It also warns about loss of precision for big arguments.

See also: [nchoosek], page 607.

`commutation_matrix (m, n)` [Function File]

Return the commutation matrix $K_{m,n}$ which is the unique $mn \times mn$ matrix such that $K_{m,n} \cdot \mathrm{vec}(A) = \mathrm{vec}(A^T)$ for all $m \times n$ matrices A.

If only one argument m is given, $K_{m,m}$ is returned.

See Magnus and Neudecker (1988), *Matrix Differential Calculus with Applications in Statistics and Econometrics.*

`duplication_matrix (n)` [Function File]

Return the duplication matrix D_n which is the unique $n^2 \times n(n+1)/2$ matrix such that $D_n * \mathrm{vech}(A) = \mathrm{vec}(A)$ for all symmetric $n \times n$ matrices A.

See Magnus and Neudecker (1988), *Matrix Differential Calculus with Applications in Statistics and Econometrics.*

`dawson (z)` [Mapping Function]

Compute the Dawson (scaled imaginary error) function.

The Dawson function is defined as

$$\frac{\sqrt{\pi}}{2} e^{-z^2} \mathrm{erfi}(z) \equiv -i \frac{\sqrt{\pi}}{2} e^{-z^2} \mathrm{erf}(iz)$$

See also: [erfc], page 455, [erf], page 455, [erfcx], page 455, [erfi], page 455, [erfinv], page 456, [erfcinv], page 456.

`[sn, cn, dn, err] = ellipj (u, m)` [Built-in Function]
`[sn, cn, dn, err] = ellipj (u, m, tol)` [Built-in Function]

Compute the Jacobi elliptic functions *sn*, *cn*, and *dn* of complex argument *u* and real parameter *m*.

If *m* is a scalar, the results are the same size as *u*. If *u* is a scalar, the results are the same size as *m*. If *u* is a column vector and *m* is a row vector, the results are matrices with `length (u)` rows and `length (m)` columns. Otherwise, *u* and *m* must conform in size and the results will be the same size as the inputs.

The value of *u* may be complex. The value of *m* must be $0 \le m \le 1$.

The optional input *tol* is currently ignored (MATLAB uses this to allow faster, less accurate approximation).

If requested, *err* contains the following status information and is the same size as the result.

0. Normal return.
1. Error—no computation, algorithm termination condition not met, return `NaN`.

Reference: Milton Abramowitz and Irene A Stegun, *Handbook of Mathematical Functions*, Chapter 16 (Sections 16.4, 16.13, and 16.15), Dover, 1965.

See also: [ellipke], page 454.

`k = ellipke (m)` [Function File]
`k = ellipke (m, tol)` [Function File]
`[k, e] = ellipke (...)` [Function File]

Compute complete elliptic integrals of the first K(m) and second E(m) kind.

m must be a scalar or real array with -Inf $\le m \le 1$.

The optional input *tol* controls the stopping tolerance of the algorithm and defaults to `eps (class (m))`. The tolerance can be increased to compute a faster, less accurate approximation.

When called with one output only elliptic integrals of the first kind are returned.

Mathematical Note:

Elliptic integrals of the first kind are defined as

$$\mathrm{K}(m) = \int_0^1 \frac{dt}{\sqrt{(1-t^2)(1-mt^2)}}$$

Elliptic integrals of the second kind are defined as

$$\mathrm{E}(m) = \int_0^1 \frac{\sqrt{1-mt^2}}{\sqrt{1-t^2}} dt$$

Reference: Milton Abramowitz and Irene A. Stegun, *Handbook of Mathematical Functions*, Chapter 17, Dover, 1965.

See also: [ellipj], page 454.

`erf` (*z*) [Mapping Function]

Compute the error function.

The error function is defined as

$$\mathrm{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt$$

See also: [erfc], page 455, [erfcx], page 455, [erfi], page 455, [dawson], page 454, [erfinv], page 456, [erfcinv], page 456.

`erfc` (*z*) [Mapping Function]

Compute the complementary error function.

The complementary error function is defined as $1 - \mathrm{erf}(z)$.

See also: [erfcinv], page 456, [erfcx], page 455, [erfi], page 455, [dawson], page 454, [erf], page 455, [erfinv], page 456.

`erfcx` (*z*) [Mapping Function]

Compute the scaled complementary error function.

The scaled complementary error function is defined as

$$e^{z^2} \mathrm{erfc}(z) \equiv e^{z^2} (1 - \mathrm{erf}(z))$$

See also: [erfc], page 455, [erf], page 455, [erfi], page 455, [dawson], page 454, [erfinv], page 456, [erfcinv], page 456.

`erfi` (*z*) [Mapping Function]

Compute the imaginary error function.

The imaginary error function is defined as

$$-i\mathrm{erf}(iz)$$

See also: [erfc], page 455, [erf], page 455, [erfcx], page 455, [dawson], page 454, [erfinv], page 456, [erfcinv], page 456.

`erfinv (x)` [Mapping Function]

Compute the inverse error function.

The inverse error function is defined such that

```
erf (y) == x
```

See also: [erf], page 455, [erfc], page 455, [erfcx], page 455, [erfi], page 455, [dawson], page 454, [erfcinv], page 456.

`erfcinv (x)` [Mapping Function]

Compute the inverse complementary error function.

The inverse complementary error function is defined such that

```
erfc (y) == x
```

See also: [erfc], page 455, [erf], page 455, [erfcx], page 455, [erfi], page 455, [dawson], page 454, [erfinv], page 456.

`expint (x)` [Function File]

Compute the exponential integral:

$$\mathrm{E}_1(x) = \int_x^\infty \frac{e^{-t}}{t} dt$$

Note: For compatibility, this functions uses the MATLAB definition of the exponential integral. Most other sources refer to this particular value as $E_1(x)$, and the exponential integral as

$$\mathrm{Ei}(x) = -\int_{-x}^\infty \frac{e^{-t}}{t} dt.$$

The two definitions are related, for positive real values of x, by $E_1(-x) = -\mathrm{Ei}(x) - i\pi$.

`gamma (z)` [Mapping Function]

Compute the Gamma function.

The Gamma function is defined as

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt.$$

Programming Note: The gamma function can grow quite large even for small input values. In many cases it may be preferable to use the natural logarithm of the gamma function (`gammaln`) in calculations to minimize loss of precision. The final result is then `exp (result_using_gammaln)`.

See also: [gammainc], page 456, [gammaln], page 458, [factorial], page 449.

`gammainc (x, a)` [Mapping Function]
`gammainc (x, a, "lower")` [Mapping Function]
`gammainc (x, a, "upper")` [Mapping Function]

Compute the normalized incomplete gamma function.

This is defined as

$$\gamma(x, a) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t} dt$$

with the limiting value of 1 as x approaches infinity. The standard notation is $P(a,x)$, e.g., Abramowitz and Stegun (6.5.1).

If a is scalar, then `gammainc (x, a)` is returned for each element of x and vice versa.

If neither x nor a is scalar, the sizes of x and a must agree, and `gammainc` is applied element-by-element.

By default the incomplete gamma function integrated from 0 to x is computed. If `"upper"` is given then the complementary function integrated from x to infinity is calculated. It should be noted that

```
gammainc (x, a) ≡ 1 - gammainc (x, a, "upper")
```

See also: [gamma], page 456, [gammaln], page 458.

l = `legendre` (*n*, *x*) [Function File]
l = `legendre` (*n*, *x*, *normalization*) [Function File]

Compute the Legendre function of degree n and order $m = 0 \ldots n$.

The value n must be a real non-negative integer.

x is a vector with real-valued elements in the range [-1, 1].

The optional argument *normalization* may be one of `"unnorm"`, `"sch"`, or `"norm"`. The default if no normalization is given is `"unnorm"`.

When the optional argument *normalization* is `"unnorm"`, compute the Legendre function of degree n and order m and return all values for $m = 0 \ldots n$. The return value has one dimension more than x.

The Legendre Function of degree n and order m:

$$P_n^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} P_n(x)$$

with Legendre polynomial of degree n:

$$P(x) = \frac{1}{2^n n!} \left(\frac{d^n}{dx^n} (x^2-1)^n \right)$$

`legendre (3, [-1.0, -0.9, -0.8])` returns the matrix:

```
 x  |   -1.0   |   -0.9   |   -0.8
------------------------------------
m=0 | -1.00000 | -0.47250 | -0.08000
m=1 |  0.00000 | -1.99420 | -1.98000
m=2 |  0.00000 | -2.56500 | -4.32000
m=3 |  0.00000 | -1.24229 | -3.24000
```

When the optional argument `normalization` is `"sch"`, compute the Schmidt semi-normalized associated Legendre function. The Schmidt semi-normalized associated Legendre function is related to the unnormalized Legendre functions by the following:

For Legendre functions of degree n and order 0:

$$SP_n^0(x) = P_n^0(x)$$

For Legendre functions of degree n and order m:

$$SP_n^m(x) = P_n^m(x)(-1)^m \left(\frac{2(n-m)!}{(n+m)!}\right)^{0.5}$$

When the optional argument *normalization* is `"norm"`, compute the fully normalized associated Legendre function. The fully normalized associated Legendre function is related to the unnormalized Legendre functions by the following:

For Legendre functions of degree *n* and order *m*

$$NP_n^m(x) = P_n^m(x)(-1)^m \left(\frac{(n+0.5)(n-m)!}{(n+m)!}\right)^{0.5}$$

`gammaln (x)` [Mapping Function]
`lgamma (x)` [Mapping Function]

Return the natural logarithm of the gamma function of *x*.

See also: [gamma], page 456, [gammainc], page 456.

17.7 Rational Approximations

`s = rat (x, tol)` [Function File]
`[n, d] = rat (x, tol)` [Function File]

Find a rational approximation to *x* within the tolerance defined by *tol* using a continued fraction expansion.

For example:

```
rat (pi) = 3 + 1/(7 + 1/16) = 355/113
rat (e) = 3 + 1/(-4 + 1/(2 + 1/(5 + 1/(-2 + 1/(-7)))))
        = 1457/536
```

When called with two output arguments return the numerator and denominator separately as two matrices.

See also: [rats], page 458.

`rats (x, len)` [Built-in Function]

Convert *x* into a rational approximation represented as a string.

The string can be converted back into a matrix as follows:

```
r = rats (hilb (4));
x = str2num (r)
```

The optional second argument defines the maximum length of the string representing the elements of *x*. By default *len* is 9.

If the length of the smallest possible rational approximation exceeds *len*, an asterisk (*) padded with spaces will be returned instead.

See also: [format], page 232, [rat], page 458.

17.8 Coordinate Transformations

`[theta, r] = cart2pol (x, y)` [Function File]
`[theta, r, z] = cart2pol (x, y, z)` [Function File]
`[theta, r] = cart2pol (C)` [Function File]
`[theta, r, z] = cart2pol (C)` [Function File]
`P = cart2pol (...)` [Function File]

Transform Cartesian coordinates to polar or cylindrical coordinates.

The inputs *x*, *y* (, and *z*) must be the same shape, or scalar. If called with a single matrix argument then each row of *C* represents the Cartesian coordinate (*x*, *y* (, *z*)).

theta describes the angle relative to the positive x-axis.

r is the distance to the z-axis (0, 0, z).

If only a single return argument is requested then return a matrix *P* where each row represents one polar/(cylindrical) coordinate (*theta*, *phi* (, *z*)).

See also: [pol2cart], page 459, [cart2sph], page 459, [sph2cart], page 460.

`[x, y] = pol2cart (theta, r)` [Function File]
`[x, y, z] = pol2cart (theta, r, z)` [Function File]
`[x, y] = pol2cart (P)` [Function File]
`[x, y, z] = pol2cart (P)` [Function File]
`C = pol2cart (...)` [Function File]

Transform polar or cylindrical coordinates to Cartesian coordinates.

The inputs *theta*, *r*, (and *z*) must be the same shape, or scalar. If called with a single matrix argument then each row of *P* represents the polar/(cylindrical) coordinate (*theta*, *r* (, *z*)).

theta describes the angle relative to the positive x-axis.

r is the distance to the z-axis (0, 0, z).

If only a single return argument is requested then return a matrix *C* where each row represents one Cartesian coordinate (*x*, *y* (, *z*)).

See also: [cart2pol], page 459, [sph2cart], page 460, [cart2sph], page 459.

`[theta, phi, r] = cart2sph (x, y, z)` [Function File]
`[theta, phi, r] = cart2sph (C)` [Function File]
`S = cart2sph (...)` [Function File]

Transform Cartesian coordinates to spherical coordinates.

The inputs *x*, *y*, and *z* must be the same shape, or scalar. If called with a single matrix argument then each row of *C* represents the Cartesian coordinate (*x*, *y*, *z*).

theta describes the angle relative to the positive x-axis.

phi is the angle relative to the xy-plane.

r is the distance to the origin (0, 0, 0).

If only a single return argument is requested then return a matrix *S* where each row represents one spherical coordinate (*theta*, *phi*, *r*).

See also: [sph2cart], page 460, [cart2pol], page 459, [pol2cart], page 459.

[*x*, *y*, *z*] = sph2cart (*theta*, *phi*, *r*) [Function File]
[*x*, *y*, *z*] = sph2cart (*S*) [Function File]
C = sph2cart (...) [Function File]

Transform spherical coordinates to Cartesian coordinates.

The inputs *theta*, *phi*, and *r* must be the same shape, or scalar. If called with a single matrix argument then each row of *S* represents the spherical coordinate (*theta*, *phi*, *r*).

theta describes the angle relative to the positive x-axis.

phi is the angle relative to the xy-plane.

r is the distance to the origin (0, 0, 0).

If only a single return argument is requested then return a matrix *C* where each row represents one Cartesian coordinate (*x*, *y*, *z*).

See also: [cart2sph], page 459, [pol2cart], page 459, [cart2pol], page 459.

17.9 Mathematical Constants

e [Built-in Function]
e (*n*) [Built-in Function]
e (*n*, *m*) [Built-in Function]
e (*n*, *m*, *k*, ...) [Built-in Function]
e (..., *class*) [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the base of natural logarithms.

The constant e satisfies the equation $\log(e) = 1$.

When called with no arguments, return a scalar with the value e.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [log], page 435, [exp], page 435, [pi], page 460, [I], page 461.

pi [Built-in Function]
pi (*n*) [Built-in Function]
pi (*n*, *m*) [Built-in Function]
pi (*n*, *m*, *k*, ...) [Built-in Function]
pi (..., *class*) [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the ratio of the circumference of a circle to its diameter(π).

Internally, `pi` is computed as '`4.0 * atan (1.0)`'.

When called with no arguments, return a scalar with the value of π.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [e], page 460, [I], page 461.

`I` [Built-in Function]
`I` (*n*) [Built-in Function]
`I` (*n*, *m*) [Built-in Function]
`I` (*n*, *m*, *k*, . . .) [Built-in Function]
`I` (. . ., *class*) [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the pure imaginary unit, defined as $\sqrt{-1}$.

I, and its equivalents i, j, and J, are functions so any of the names may be reused for other purposes (such as i for a counter variable).

When called with no arguments, return a scalar with the value i.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [e], page 460, [pi], page 460, [log], page 435, [exp], page 435.

`Inf` [Built-in Function]
`Inf` (*n*) [Built-in Function]
`Inf` (*n*, *m*) [Built-in Function]
`Inf` (*n*, *m*, *k*, . . .) [Built-in Function]
`Inf` (. . ., *class*) [Built-in Function]

Return a scalar, matrix or N-dimensional array whose elements are all equal to the IEEE representation for positive infinity.

Infinity is produced when results are too large to be represented using the IEEE floating point format for numbers. Two common examples which produce infinity are division by zero and overflow.

```
[ 1/0 e^800 ]
⇒ Inf   Inf
```

When called with no arguments, return a scalar with the value '`Inf`'.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [isinf], page 406, [NaN], page 462.

`NaN` [Built-in Function]
`NaN (`*n*`)` [Built-in Function]
`NaN (`*n*, *m*`)` [Built-in Function]
`NaN (`*n*, *m*, *k*, ...`)` [Built-in Function]
`NaN (..., `*class*`)` [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the IEEE symbol NaN (Not a Number).

NaN is the result of operations which do not produce a well defined numerical result. Common operations which produce a NaN are arithmetic with infinity ($\infty - \infty$), zero divided by zero (0/0), and any operation involving another NaN value (5 + NaN).

Note that NaN always compares not equal to NaN (NaN != NaN). This behavior is specified by the IEEE standard for floating point arithmetic. To find NaN values, use the `isnan` function.

When called with no arguments, return a scalar with the value '`NaN`'.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [isnan], page 406, [Inf], page 461.

`eps` [Built-in Function]
`eps (`*x*`)` [Built-in Function]
`eps (`*n*, *m*`)` [Built-in Function]
`eps (`*n*, *m*, *k*, ...`)` [Built-in Function]
`eps (..., `*class*`)` [Built-in Function]

Return a scalar, matrix or N-dimensional array whose elements are all eps, the machine precision.

More precisely, `eps` is the relative spacing between any two adjacent numbers in the machine's floating point system. This number is obviously system dependent. On machines that support IEEE floating point arithmetic, `eps` is approximately 2.2204×10^{-16} for double precision and 1.1921×10^{-7} for single precision.

When called with no arguments, return a scalar with the value `eps (1.0)`.

Given a single argument *x*, return the distance between *x* and the next largest value.

When called with more than one argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [realmax], page 463, [realmin], page 463, [intmax], page 55, [bitmax], page 58.

`realmax` [Built-in Function]
`realmax` (*n*) [Built-in Function]
`realmax` (*n*, *m*) [Built-in Function]
`realmax` (*n*, *m*, *k*, ...) [Built-in Function]
`realmax` (..., *class*) [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the largest floating point number that is representable.

The actual value is system dependent. On machines that support IEEE floating point arithmetic, `realmax` is approximately 1.7977×10^{308} for double precision and 3.4028×10^{38} for single precision.

When called with no arguments, return a scalar with the value `realmax ("double")`.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [realmin], page 463, [intmax], page 55, [bitmax], page 58, [eps], page 462.

`realmin` [Built-in Function]
`realmin` (*n*) [Built-in Function]
`realmin` (*n*, *m*) [Built-in Function]
`realmin` (*n*, *m*, *k*, ...) [Built-in Function]
`realmin` (..., *class*) [Built-in Function]

Return a scalar, matrix, or N-dimensional array whose elements are all equal to the smallest normalized floating point number that is representable.

The actual value is system dependent. On machines that support IEEE floating point arithmetic, `realmin` is approximately 2.2251×10^{-308} for double precision and 1.1755×10^{-38} for single precision.

When called with no arguments, return a scalar with the value `realmin ("double")`.

When called with a single argument, return a square matrix with the dimension specified.

When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions.

The optional argument *class* specifies the return type and may be either `"double"` or `"single"`.

See also: [realmax], page 463, [intmin], page 56, [eps], page 462.

18 Linear Algebra

This chapter documents the linear algebra functions provided in Octave. Reference material for many of these functions may be found in Golub and Van Loan, *Matrix Computations, 2nd Ed.*, Johns Hopkins, 1989, and in the LAPACK *Users' Guide*, SIAM, 1992. The LAPACK *Users' Guide* is available at: *http://www.netlib.org/lapack/lug/*

A common text for engineering courses is G. Strang, *Linear Algebra and Its Applications, 4th Edition*. It has become a widespread reference for linear algebra. An alternative is P. Lax *Linear Algebra and Its Applications*, and also is a good choice. It claims to be suitable for high school students with substantial mathematical interests as well as first-year undergraduates.

18.1 Techniques Used for Linear Algebra

Octave includes a polymorphic solver that selects an appropriate matrix factorization depending on the properties of the matrix itself. Generally, the cost of determining the matrix type is small relative to the cost of factorizing the matrix itself. In any case the matrix type is cached once it is calculated so that it is not re-determined each time it is used in a linear equation.

The selection tree for how the linear equation is solved or a matrix inverse is formed is given by:

1. If the matrix is upper or lower triangular sparse use a forward or backward substitution using the LAPACK xTRTRS function, and goto 4.
2. If the matrix is square, Hermitian with a real positive diagonal, attempt Cholesky factorization using the LAPACK xPOTRF function.
3. If the Cholesky factorization failed or the matrix is not Hermitian with a real positive diagonal, and the matrix is square, factorize using the LAPACK xGETRF function.
4. If the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, find a least squares solution using the LAPACK xGELSD function.

The user can force the type of the matrix with the `matrix_type` function. This overcomes the cost of discovering the type of the matrix. However, it should be noted that identifying the type of the matrix incorrectly will lead to unpredictable results, and so `matrix_type` should be used with care.

It should be noted that the test for whether a matrix is a candidate for Cholesky factorization, performed above, and by the `matrix_type` function, does not make certain that the matrix is Hermitian. However, the attempt to factorize the matrix will quickly detect a non-Hermitian matrix.

18.2 Basic Matrix Functions

`AA = balance (A)` [Built-in Function]
`AA = balance (A, opt)` [Built-in Function]
`[DD, AA] = balance (A, opt)` [Built-in Function]
`[D, P, AA] = balance (A, opt)` [Built-in Function]

`[CC, DD, AA, BB] = balance (A, B, opt)` [Built-in Function]

Balance the matrix A to reduce numerical errors in future calculations.

Compute `AA = DD \ A * DD` in which AA is a matrix whose row and column norms are roughly equal in magnitude, and `DD = P * D`, in which P is a permutation matrix and D is a diagonal matrix of powers of two. This allows the equilibration to be computed without round-off. Results of eigenvalue calculation are typically improved by balancing first.

If two output values are requested, `balance` returns the diagonal D and the permutation P separately as vectors. In this case, `DD = eye(n)(:,P) * diag (D)`, where n is the matrix size.

If four output values are requested, compute `AA = CC*A*DD` and `BB = CC*B*DD`, in which AA and BB have nonzero elements of approximately the same magnitude and CC and DD are permuted diagonal matrices as in DD for the algebraic eigenvalue problem.

The eigenvalue balancing option *opt* may be one of:

`"noperm"`, `"S"`
: Scale only; do not permute.

`"noscal"`, `"P"`
: Permute only; do not scale.

Algebraic eigenvalue balancing uses standard LAPACK routines.

Generalized eigenvalue problem balancing uses Ward's algorithm (SIAM Journal on Scientific and Statistical Computing, 1981).

`bw = bandwidth (A, type)` [Function File]
`[lower, upper] = bandwidth (A)` [Function File]

Compute the bandwidth of A.

The *type* argument is the string `"lower"` for the lower bandwidth and `"upper"` for the upper bandwidth. If no *type* is specified return both the lower and upper bandwidth of A.

The lower/upper bandwidth of a matrix is the number of subdiagonals/superdiagonals with nonzero entries.

See also: [isbanded], page 64, [isdiag], page 64, [istril], page 64, [istriu], page 64.

`cond (A)` [Function File]
`cond (A, p)` [Function File]

Compute the p-norm condition number of a matrix.

`cond (A)` is defined as $\| A \|_p * \| A^{-1} \|_p$.

By default, `p = 2` is used which implies a (relatively slow) singular value decomposition. Other possible selections are `p = 1, Inf, "fro"` which are generally faster. See `norm` for a full discussion of possible p values.

The condition number of a matrix quantifies the sensitivity of the matrix inversion operation when small changes are made to matrix elements. Ideally the condition number will be close to 1. When the number is large this indicates small changes (such as underflow or round-off error) will produce large changes in the resulting output. In such cases the solution results from numerical computing are not likely to be accurate.

See also: [condest], page 542, [rcond], page 472, [norm], page 470, [svd], page 481.

`det (A)` [Built-in Function]
`[d, rcond] = det (A)` [Built-in Function]

Compute the determinant of A.

Return an estimate of the reciprocal condition number if requested.

Programming Notes: Routines from LAPACK are used for full matrices and code from UMFPACK is used for sparse matrices.

The determinant should not be used to check a matrix for singularity. For that, use any of the condition number functions: `cond`, `condest`, `rcond`.

See also: [cond], page 466, [condest], page 542, [rcond], page 472.

`lambda = eig (A)` [Built-in Function]
`lambda = eig (A, B)` [Built-in Function]
`[V, lambda] = eig (A)` [Built-in Function]
`[V, lambda] = eig (A, B)` [Built-in Function]

Compute the eigenvalues (and optionally the eigenvectors) of a matrix or a pair of matrices

The algorithm used depends on whether there are one or two input matrices, if they are real or complex, and if they are symmetric (Hermitian if complex) or non-symmetric.

The eigenvalues returned by `eig` are not ordered.

See also: [eigs], page 545, [svd], page 481.

`G = givens (x, y)` [Built-in Function]
`[c, s] = givens (x, y)` [Built-in Function]

Compute the Givens rotation matrix G.

The Givens matrix is a 2×2 orthogonal matrix

$$G = \begin{bmatrix} c & s \\ -s' & c \end{bmatrix}$$

such that

$$G \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} * \\ 0 \end{bmatrix}$$

with x and y scalars.

If two output arguments are requested, return the factors c and s rather than the Givens rotation matrix.

For example:

```
givens (1, 1)
   ⇒   0.70711   0.70711
      -0.70711   0.70711
```

See also: [planerot], page 468.

`[G, y] = planerot (x)` [Function File]

Given a two-element column vector, return the 2×2 orthogonal matrix G such that `y = g * x` and `y(2) = 0`.

See also: [givens], page 467.

`x = inv (A)` [Built-in Function]
`[x, rcond] = inv (A)` [Built-in Function]

Compute the inverse of the square matrix A.

Return an estimate of the reciprocal condition number if requested, otherwise warn of an ill-conditioned matrix if the reciprocal condition number is small.

In general it is best to avoid calculating the inverse of a matrix directly. For example, it is both faster and more accurate to solve systems of equations ($A^*x = b$) with `y = A \ b`, rather than `y = inv (A) * b`.

If called with a sparse matrix, then in general x will be a full matrix requiring significantly more storage. Avoid forming the inverse of a sparse matrix if possible.

See also: [ldivide], page 143, [rdivide], page 144.

`x = linsolve (A, b)` [Function File]
`x = linsolve (A, b, opts)` [Function File]
`[x, R] = linsolve (...)` [Function File]

Solve the linear system `A*x = b`.

With no options, this function is equivalent to the left division operator (`x = A \ b`) or the matrix-left-divide function (`x = mldivide (A, b)`).

Octave ordinarily examines the properties of the matrix A and chooses a solver that best matches the matrix. By passing a structure *opts* to `linsolve` you can inform Octave directly about the matrix A. In this case Octave will skip the matrix examination and proceed directly to solving the linear system.

Warning: If the matrix A does not have the properties listed in the *opts* structure then the result will not be accurate AND no warning will be given. When in doubt, let Octave examine the matrix and choose the appropriate solver as this step takes little time and the result is cached so that it is only done once per linear system.

Possible *opts* fields (set value to true/false):

LT — A is lower triangular

UT — A is upper triangular

UHESS — A is upper Hessenberg (currently makes no difference)

SYM — A is symmetric or complex Hermitian (currently makes no difference)

POSDEF — A is positive definite

RECT — A is general rectangular (currently makes no difference)

TRANSA — Solve `A'*x = b` by `transpose (A) \ b`

The optional second output R is the inverse condition number of A (zero if matrix is singular).

See also: [mldivide], page 143, [matrix_type], page 469, [rcond], page 472.

type = `matrix_type` (*A*) [Built-in Function]
type = `matrix_type` (*A*, "*nocompute*") [Built-in Function]
A = `matrix_type` (*A*, *type*) [Built-in Function]
A = `matrix_type` (*A*, "*upper*", *perm*) [Built-in Function]
A = `matrix_type` (*A*, "*lower*", *perm*) [Built-in Function]
A = `matrix_type` (*A*, "*banded*", *nl*, *nu*) [Built-in Function]

Identify the matrix type or mark a matrix as a particular type.

This allows more rapid solutions of linear equations involving A to be performed.

Called with a single argument, `matrix_type` returns the type of the matrix and caches it for future use.

Called with more than one argument, `matrix_type` allows the type of the matrix to be defined.

If the option `"nocompute"` is given, the function will not attempt to guess the type if it is still unknown. This is useful for debugging purposes.

The possible matrix types depend on whether the matrix is full or sparse, and can be one of the following

`"unknown"`
: Remove any previously cached matrix type, and mark type as unknown.

`"full"`
: Mark the matrix as full.

`"positive definite"`
: Probable full positive definite matrix.

`"diagonal"`
: Diagonal matrix. (Sparse matrices only)

`"permuted diagonal"`
: Permuted Diagonal matrix. The permutation does not need to be specifically indicated, as the structure of the matrix explicitly gives this. (Sparse matrices only)

`"upper"`
: Upper triangular. If the optional third argument *perm* is given, the matrix is assumed to be a permuted upper triangular with the permutations defined by the vector *perm*.

`"lower"`
: Lower triangular. If the optional third argument *perm* is given, the matrix is assumed to be a permuted lower triangular with the permutations defined by the vector *perm*.

`"banded"`
`"banded positive definite"`
: Banded matrix with the band size of *nl* below the diagonal and *nu* above it. If *nl* and *nu* are 1, then the matrix is tridiagonal and treated with specialized code. In addition the matrix can be marked as probably a positive definite. (Sparse matrices only)

`"singular"`
: The matrix is assumed to be singular and will be treated with a minimum norm solution.

Note that the matrix type will be discovered automatically on the first attempt to solve a linear equation involving *A*. Therefore `matrix_type` is only useful to give Octave hints of the matrix type. Incorrectly defining the matrix type will result in incorrect results from solutions of linear equations; it is entirely **the responsibility of the user** to correctly identify the matrix type.

Also, the test for positive definiteness is a low-cost test for a Hermitian matrix with a real positive diagonal. This does not guarantee that the matrix is positive definite, but only that it is a probable candidate. When such a matrix is factorized, a Cholesky factorization is first attempted, and if that fails the matrix is then treated with an LU factorization. Once the matrix has been factorized, `matrix_type` will return the correct classification of the matrix.

`norm` (*A*) [Built-in Function]
`norm` (*A*, *p*) [Built-in Function]
`norm` (*A*, *p*, *opt*) [Built-in Function]

Compute the p-norm of the matrix *A*.

If the second argument is missing, `p = 2` is assumed.

If *A* is a matrix (or sparse matrix):

p = `1` 1-norm, the largest column sum of the absolute values of *A*.

p = `2` Largest singular value of *A*.

p = `Inf` or `"inf"`
Infinity norm, the largest row sum of the absolute values of *A*.

p = `"fro"`
Frobenius norm of *A*, `sqrt (sum (diag (A' * A)))`.

other *p*, `p > 1`
maximum `norm (A*x, p)` such that `norm (x, p) == 1`

If *A* is a vector or a scalar:

p = `Inf` or `"inf"`
`max (abs (A))`.

p = `-Inf` `min (abs (A))`.

p = `"fro"`
Frobenius norm of *A*, `sqrt (sumsq (abs (A)))`.

p = 0 Hamming norm - the number of nonzero elements.

other *p*, `p > 1`
p-norm of *A*, `(sum (abs (A) .^ p)) ^ (1/p)`.

other *p* `p < 1`
the p-pseudonorm defined as above.

If *opt* is the value `"rows"`, treat each row as a vector and compute its norm. The result is returned as a column vector. Similarly, if *opt* is `"columns"` or `"cols"` then compute the norms of each column and return a row vector.

See also: [cond], page 466, [svd], page 481.

`null` (*A*) [Function File]
`null` (*A*, *tol*) [Function File]

Return an orthonormal basis of the null space of *A*.

The dimension of the null space is taken as the number of singular values of *A* not greater than *tol*. If the argument *tol* is missing, it is computed as

```
max (size (A)) * max (svd (A)) * eps
```

See also: [orth], page 471.

`orth` (*A*) [Function File]
`orth` (*A*, *tol*) [Function File]

Return an orthonormal basis of the range space of *A*.

The dimension of the range space is taken as the number of singular values of *A* greater than *tol*. If the argument *tol* is missing, it is computed as

```
max (size (A)) * max (svd (A)) * eps
```

See also: [null], page 471.

`[y, h] = mgorth (x, v)` [Built-in Function]

Orthogonalize a given column vector *x* with respect to a set of orthonormal vectors comprising the columns of *v* using the modified Gram-Schmidt method.

On exit, *y* is a unit vector such that:

```
norm (y) = 1
v' * y = 0
x = [v, y]*h'
```

`pinv` (*x*) [Built-in Function]
`pinv` (*x*, *tol*) [Built-in Function]

Return the pseudoinverse of *x*.

Singular values less than *tol* are ignored.

If the second argument is omitted, it is taken to be

```
tol = max (size (x)) * sigma_max (x) * eps,
```

where `sigma_max (x)` is the maximal singular value of *x*.

`rank` (*A*) [Function File]
`rank` (*A*, *tol*) [Function File]

Compute the rank of matrix *A*, using the singular value decomposition.

The rank is taken to be the number of singular values of *A* that are greater than the specified tolerance *tol*. If the second argument is omitted, it is taken to be

```
tol = max (size (A)) * sigma(1) * eps;
```

where `eps` is machine precision and `sigma(1)` is the largest singular value of *A*.

The rank of a matrix is the number of linearly independent rows or columns and determines how many particular solutions exist to a system of equations. Use `null` for finding the remaining homogenous solutions.

Example:

```
x = [1 2 3
     4 5 6
     7 8 9];
rank (x)
  ⇒ 2
```

The number of linearly independent rows is only 2 because the final row is a linear combination of -1*row1 + 2*row2.

See also: [null], page 471, [sprank], page 543, [svd], page 481.

`c = rcond (A)` [Built-in Function]

Compute the 1-norm estimate of the reciprocal condition number as returned by LAPACK.

If the matrix is well-conditioned then *c* will be near 1 and if the matrix is poorly conditioned it will be close to 0.

The matrix *A* must not be sparse. If the matrix is sparse then `condest (A)` or `rcond (full (A))` should be used instead.

See also: [cond], page 466, [condest], page 542.

`trace (A)` [Function File]

Compute the trace of *A*, the sum of the elements along the main diagonal.

The implementation is straightforward: `sum (diag (A))`.

See also: [eig], page 467.

`rref (A)` [Function File]
`rref (A, tol)` [Function File]
`[r, k] = rref (...)` [Function File]

Return the reduced row echelon form of *A*.

tol defaults to `eps * max (size (A)) * norm (A, inf)`.

The optional return argument *k* contains the vector of "bound variables", which are those columns on which elimination has been performed.

18.3 Matrix Factorizations

`R = chol (A)` [Loadable Function]
`[R, p] = chol (A)` [Loadable Function]
`[R, p, Q] = chol (S)` [Loadable Function]
`[R, p, Q] = chol (S, "vector")` [Loadable Function]
`[L, ...] = chol (..., "lower")` [Loadable Function]
`[L, ...] = chol (..., "upper")` [Loadable Function]

Compute the Cholesky factor, *R*, of the symmetric positive definite matrix *A*.

The Cholesky factor is defined by $R^T R = A$.

Called with one output argument `chol` fails if *A* or *S* is not positive definite. With two or more output arguments *p* flags whether the matrix was positive definite and `chol` does not fail. A zero value indicated that the matrix was positive definite and the R gives the factorization, and p will have a positive value otherwise.

If called with 3 outputs then a sparsity preserving row/column permutation is applied to A prior to the factorization. That is R is the factorization of `A(Q,Q)` such that $R^T R = Q^T A Q$.

The sparsity preserving permutation is generally returned as a matrix. However, given the flag `"vector"`, Q will be returned as a vector such that $R^T R = A(Q, Q)$.

Called with either a sparse or full matrix and using the `"lower"` flag, `chol` returns the lower triangular factorization such that $LL^T = A$.

For full matrices, if the `"lower"` flag is set only the lower triangular part of the matrix is used for the factorization, otherwise the upper triangular part is used.

In general the lower triangular factorization is significantly faster for sparse matrices.

See also: [hess], page 474, [lu], page 475, [qr], page 476, [qz], page 479, [schur], page 480, [svd], page 481, [ichol], page 552, [cholinv], page 473, [chol2inv], page 473, [cholupdate], page 473, [cholinsert], page 474, [choldelete], page 474, [cholshift], page 474.

`cholinv (`*A*`)` [Loadable Function]

Compute the inverse of the symmetric positive definite matrix A using the Cholesky factorization.

See also: [chol], page 472, [chol2inv], page 473, [inv], page 468.

`chol2inv (`*U*`)` [Loadable Function]

Invert a symmetric, positive definite square matrix from its Cholesky decomposition, U.

Note that U should be an upper-triangular matrix with positive diagonal elements. `chol2inv (U)` provides `inv (U'*U)` but it is much faster than using `inv`.

See also: [chol], page 472, [cholinv], page 473, [inv], page 468.

`[`*R1*`, `*info*`] = cholupdate (`*R*`, `*u*`, `*op*`)` [Loadable Function]

Update or downdate a Cholesky factorization.

Given an upper triangular matrix R and a column vector u, attempt to determine another upper triangular matrix *R1* such that

- *R1*'**R1* = *R*'**R* + *u***u*' if *op* is `"+"`
- *R1*'**R1* = *R*'**R* - *u***u*' if *op* is `"-"`

If *op* is `"-"`, *info* is set to

- 0 if the downdate was successful,
- 1 if *R*'**R* - *u***u*' is not positive definite,
- 2 if R is singular.

If *info* is not present, an error message is printed in cases 1 and 2.

See also: [chol], page 472, [cholinsert], page 474, [choldelete], page 474, [cholshift], page 474.

`R1 = cholinsert (R, j, u)` [Loadable Function]
`[R1, info] = cholinsert (R, j, u)` [Loadable Function]

Given a Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix $A = R'*R$, R upper triangular, return the Cholesky factorization of *A1*, where A1(p,p) = A, A1(:,j) = A1(j,:)' = u and p = [1:j-1,j+1:n+1]. u(j) should be positive.

On return, *info* is set to

- 0 if the insertion was successful,
- 1 if *A1* is not positive definite,
- 2 if R is singular.

If *info* is not present, an error message is printed in cases 1 and 2.

See also: [chol], page 472, [cholupdate], page 473, [choldelete], page 474, [cholshift], page 474.

`R1 = choldelete (R, j)` [Loadable Function]

Given a Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix $A = R'*R$, R upper triangular, return the Cholesky factorization of A(p,p), where p = [1:j-1,j+1:n+1].

See also: [chol], page 472, [cholupdate], page 473, [cholinsert], page 474, [cholshift], page 474.

`R1 = cholshift (R, i, j)` [Loadable Function]

Given a Cholesky factorization of a real symmetric or complex Hermitian positive definite matrix $A = R'*R$, R upper triangular, return the Cholesky factorization of A(p,p), where p is the permutation
`p = [1:i-1, shift(i:j, 1), j+1:n]` if *i* < *j*
or
`p = [1:j-1, shift(j:i,-1), i+1:n]` if *j* < *i*.

See also: [chol], page 472, [cholupdate], page 473, [cholinsert], page 474, [choldelete], page 474.

`H = hess (A)` [Built-in Function]
`[P, H] = hess (A)` [Built-in Function]

Compute the Hessenberg decomposition of the matrix A.

The Hessenberg decomposition is

$$A = PHP^T$$

where P is a square unitary matrix ($P^T P = I$), and H is upper Hessenberg ($H_{i,j} = 0, \forall i > j + 1$).

The Hessenberg decomposition is usually used as the first step in an eigenvalue computation, but has other applications as well (see Golub, Nash, and Van Loan, IEEE Transactions on Automatic Control, 1979).

See also: [eig], page 467, [chol], page 472, [lu], page 475, [qr], page 476, [qz], page 479, [schur], page 480, [svd], page 481.

`[L, U] = lu (A)` [Built-in Function]
`[L, U, P] = lu (A)` [Built-in Function]
`[L, U, P, Q] = lu (S)` [Built-in Function]
`[L, U, P, Q, R] = lu (S)` [Built-in Function]
`[...] = lu (S, thres)` [Built-in Function]
`y = lu (...)` [Built-in Function]
`[...] = lu (..., "vector")` [Built-in Function]

Compute the LU decomposition of *A*.

If *A* is full subroutines from LAPACK are used and if *A* is sparse then UMFPACK is used.

The result is returned in a permuted form, according to the optional return value *P*. For example, given the matrix `a = [1, 2; 3, 4]`,

```
[l, u, p] = lu (a)
```

returns

```
l =

  1.00000  0.00000
  0.33333  1.00000

u =

  3.00000  4.00000
  0.00000  0.66667

p =

  0  1
  1  0
```

The matrix is not required to be square.

When called with two or three output arguments and a spare input matrix, `lu` does not attempt to perform sparsity preserving column permutations. Called with a fourth output argument, the sparsity preserving column transformation *Q* is returned, such that `P * A * Q = L * U`.

Called with a fifth output argument and a sparse input matrix, `lu` attempts to use a scaling factor *R* on the input matrix such that `P * (R \ A) * Q = L * U`. This typically leads to a sparser and more stable factorization.

An additional input argument *thres*, that defines the pivoting threshold can be given. *thres* can be a scalar, in which case it defines the UMFPACK pivoting tolerance for both symmetric and unsymmetric cases. If *thres* is a 2-element vector, then the first element defines the pivoting tolerance for the unsymmetric UMFPACK pivoting strategy and the second for the symmetric strategy. By default, the values defined by `spparms` are used ([0.1, 0.001]).

Given the string argument `"vector"`, `lu` returns the values of *P* and *Q* as vector values, such that for full matrix, `A (P,:) = L * U`, and `R(P,:) * A (:, Q) = L * U`.

With two output arguments, returns the permuted forms of the upper and lower triangular matrices, such that *A* = *L* * *U*. With one output argument *y*, then the matrix returned by the LAPACK routines is returned. If the input matrix is sparse then the matrix *L* is embedded into *U* to give a return value similar to the full case. For both full and sparse matrices, `lu` loses the permutation information.

See also: [luupdate], page 476, [ilu], page 554, [chol], page 472, [hess], page 474, [qr], page 476, [qz], page 479, [schur], page 480, [svd], page 481.

[*L*, *U*] = **luupdate** (*L*, *U*, *x*, *y*) [Built-in Function]
[*L*, *U*, *P*] = **luupdate** (*L*, *U*, *P*, *x*, *y*) [Built-in Function]

Given an LU factorization of a real or complex matrix $A = L^*U$, *L* lower unit trapezoidal and *U* upper trapezoidal, return the LU factorization of *A* + *x***y*.', where *x* and *y* are column vectors (rank-1 update) or matrices with equal number of columns (rank-k update).

Optionally, row-pivoted updating can be used by supplying a row permutation (pivoting) matrix *P*; in that case, an updated permutation matrix is returned. Note that if *L*, *U*, *P* is a pivoted LU factorization as obtained by `lu`:

```
[L, U, P] = lu (A);
```

then a factorization of `A+x*y.'` can be obtained either as

```
[L1, U1] = lu (L, U, P*x, y)
```

or

```
[L1, U1, P1] = lu (L, U, P, x, y)
```

The first form uses the unpivoted algorithm, which is faster, but less stable. The second form uses a slower pivoted algorithm, which is more stable.

The matrix case is done as a sequence of rank-1 updates; thus, for large enough k, it will be both faster and more accurate to recompute the factorization from scratch.

See also: [lu], page 475, [cholupdate], page 473, [qrupdate], page 477.

[*Q*, *R*, *P*] = **qr** (*A*) [Loadable Function]
[*Q*, *R*, *P*] = **qr** (*A*, '0') [Loadable Function]
[*C*, *R*] = **qr** (*A*, *B*) [Loadable Function]
[*C*, *R*] = **qr** (*A*, *B*, '0') [Loadable Function]

Compute the QR factorization of *A*, using standard LAPACK subroutines.

For example, given the matrix `A = [1, 2; 3, 4]`,

```
[Q, R] = qr (A)
```

returns

```
Q =

  -0.31623  -0.94868
  -0.94868   0.31623

R =

  -3.16228  -4.42719
   0.00000  -0.63246
```

The `qr` factorization has applications in the solution of least squares problems

$$\min_x \|Ax - b\|_2$$

for overdetermined systems of equations (i.e., A is a tall, thin matrix). The QR factorization is $QR = A$ where Q is an orthogonal matrix and R is upper triangular.

If given a second argument of `'0'`, `qr` returns an economy-sized QR factorization, omitting zero rows of R and the corresponding columns of Q.

If the matrix A is full, the permuted QR factorization `[Q, R, P] = qr (A)` forms the QR factorization such that the diagonal entries of R are decreasing in magnitude order. For example, given the matrix `a = [1, 2; 3, 4]`,

```
[Q, R, P] = qr (A)
```

returns

```
Q =

  -0.44721  -0.89443
  -0.89443   0.44721

R =

  -4.47214  -3.13050
   0.00000   0.44721

P =

   0  1
   1  0
```

The permuted `qr` factorization `[Q, R, P] = qr (A)` factorization allows the construction of an orthogonal basis of `span (A)`.

If the matrix A is sparse, then compute the sparse QR factorization of A, using CSparse. As the matrix Q is in general a full matrix, this function returns the Q-less factorization R of A, such that `R = chol (A' * A)`.

If the final argument is the scalar `0` and the number of rows is larger than the number of columns, then an economy factorization is returned. That is R will have only `size (A,1)` rows.

If an additional matrix B is supplied, then `qr` returns C, where `C = Q' * B`. This allows the least squares approximation of `A \ B` to be calculated as

```
[C, R] = qr (A, B)
x = R \ C
```

See also: [chol], page 472, [hess], page 474, [lu], page 475, [qz], page 479, [schur], page 480, [svd], page 481, [qrupdate], page 477, [qrinsert], page 478, [qrdelete], page 478, [qrshift], page 478.

`[Q1, R1] = qrupdate (Q, R, u, v)` [Loadable Function]

Given a QR factorization of a real or complex matrix $A = Q^*R$, Q unitary and R upper trapezoidal, return the QR factorization of $A + u^*v'$, where u and v are column vectors (rank-1 update) or matrices with equal number of columns (rank-k update). Notice that the latter case is done as a sequence of rank-1 updates; thus, for k large enough, it will be both faster and more accurate to recompute the factorization from scratch.

The QR factorization supplied may be either full (Q is square) or economized (R is square).

See also: [qr], page 476, [qrinsert], page 478, [qrdelete], page 478, [qrshift], page 478.

`[Q1, R1] = qrinsert (Q, R, j, x, orient)` [Loadable Function]

Given a QR factorization of a real or complex matrix $A = Q*R$, Q unitary and R upper trapezoidal, return the QR factorization of [A(:,1:j-1) x A(:,j:n)], where *u* is a column vector to be inserted into A (if *orient* is `"col"`), or the QR factorization of [A(1:j-1,:);x;A(:,j:n)], where x is a row vector to be inserted into A (if *orient* is `"row"`).

The default value of *orient* is `"col"`. If *orient* is `"col"`, *u* may be a matrix and *j* an index vector resulting in the QR factorization of a matrix B such that B(:,*j*) gives *u* and B(:,*j*) = [] gives A. Notice that the latter case is done as a sequence of k insertions; thus, for k large enough, it will be both faster and more accurate to recompute the factorization from scratch.

If *orient* is `"col"`, the QR factorization supplied may be either full (Q is square) or economized (R is square).

If *orient* is `"row"`, full factorization is needed.

See also: [qr], page 476, [qrupdate], page 477, [qrdelete], page 478, [qrshift], page 478.

`[Q1, R1] = qrdelete (Q, R, j, orient)` [Loadable Function]

Given a QR factorization of a real or complex matrix $A = Q*R$, Q unitary and R upper trapezoidal, return the QR factorization of [A(:,1:j-1) A(:,j+1:n)], i.e., A with one column deleted (if *orient* is `"col"`), or the QR factorization of [A(1:j-1,:);A(j+1:n,:)], i.e., A with one row deleted (if *orient* is `"row"`).

The default value of *orient* is `"col"`.

If *orient* is `"col"`, *j* may be an index vector resulting in the QR factorization of a matrix B such that A(:,*j*) = [] gives B. Notice that the latter case is done as a sequence of k deletions; thus, for k large enough, it will be both faster and more accurate to recompute the factorization from scratch.

If *orient* is `"col"`, the QR factorization supplied may be either full (Q is square) or economized (R is square).

If *orient* is `"row"`, full factorization is needed.

See also: [qr], page 476, [qrupdate], page 477, [qrinsert], page 478, [qrshift], page 478.

`[Q1, R1] = qrshift (Q, R, i, j)` [Loadable Function]

Given a QR factorization of a real or complex matrix $A = Q*R$, Q unitary and R upper trapezoidal, return the QR factorization of A(:,p), where p is the permutation
`p = [1:i-1, shift(i:j, 1), j+1:n]` if $i < j$
or

`p = [1:j-1, shift(j:i,-1), i+1:n]` if $j < i$.

See also: [qr], page 476, [qrupdate], page 477, [qrinsert], page 478, [qrdelete], page 478.

lambda = qz (*A*, *B*) [Built-in Function]
lambda = qz (*A*, *B*, *opt*) [Built-in Function]

QZ decomposition of the generalized eigenvalue problem ($Ax = sBx$).

There are three ways to call this function:

1. *lambda* = qz (*A*, *B*)

   Computes the generalized eigenvalues λ of $(A - sB)$.

2. [AA, BB, Q, Z, V, W, *lambda*] = qz (*A*, *B*)

   Computes QZ decomposition, generalized eigenvectors, and generalized eigenvalues of $(A - sB)$

$$AV = BV\mathrm{diag}(\lambda)$$

$$W^T A = \mathrm{diag}(\lambda)W^T B$$

$$AA = Q^T AZ, BB = Q^T BZ$$

   with Q and Z orthogonal (unitary)$= I$

3. [AA,BB,Z{, *lambda*}] = qz (*A*, *B*, *opt*)

   As in form [2], but allows ordering of generalized eigenpairs for, e.g., solution of discrete time algebraic Riccati equations. Form 3 is not available for complex matrices, and does not compute the generalized eigenvectors V, W, nor the orthogonal matrix Q.

   opt for ordering eigenvalues of the GEP pencil. The leading block of the revised pencil contains all eigenvalues that satisfy:

   "N" = unordered (default)

   "S" = small: leading block has all |lambda| ≤ 1

   "B" = big: leading block has all |lambda| ≥ 1

   "-" = negative real part: leading block has all eigenvalues in the open left half-plane

   "+" = non-negative real part: leading block has all eigenvalues in the closed right half-plane

Note: `qz` performs permutation balancing, but not scaling (see [XREFbalance], page 465). The order of output arguments was selected for compatibility with MATLAB.

See also: [eig], page 467, [balance], page 465, [lu], page 475, [chol], page 472, [hess], page 474, [qr], page 476, [qzhess], page 480, [schur], page 480, [svd], page 481.

`[aa, bb, q, z] = qzhess (A, B)` [Function File]

Compute the Hessenberg-triangular decomposition of the matrix pencil `(A, B)`, returning `aa = q * A * z`, `bb = q * B * z`, with q and z orthogonal.

For example:

```
[aa, bb, q, z] = qzhess ([1, 2; 3, 4], [5, 6; 7, 8])
     ⇒ aa = [ -3.02244, -4.41741;  0.92998,  0.69749 ]
     ⇒ bb = [ -8.60233, -9.99730;  0.00000, -0.23250 ]
     ⇒  q = [ -0.58124, -0.81373; -0.81373,  0.58124 ]
     ⇒  z = [ 1, 0; 0, 1 ]
```

The Hessenberg-triangular decomposition is the first step in Moler and Stewart's QZ decomposition algorithm.

Algorithm taken from Golub and Van Loan, *Matrix Computations, 2nd edition*.

See also: [lu], page 475, [chol], page 472, [hess], page 474, [qr], page 476, [qz], page 479, [schur], page 480, [svd], page 481.

`S = schur (A)` [Built-in Function]
`S = schur (A, "real")` [Built-in Function]
`S = schur (A, "complex")` [Built-in Function]
`S = schur (A, opt)` [Built-in Function]
`[U, S] = schur (...)` [Built-in Function]

Compute the Schur decomposition of A.

The Schur decomposition is defined as

$$S = U^T A U$$

where U is a unitary matrix ($U^T U$ is identity) and S is upper triangular. The eigenvalues of A (and S) are the diagonal elements of S. If the matrix A is real, then the real Schur decomposition is computed, in which the matrix U is orthogonal and S is block upper triangular with blocks of size at most 2×2 along the diagonal. The diagonal elements of S (or the eigenvalues of the 2×2 blocks, when appropriate) are the eigenvalues of A and S.

The default for real matrices is a real Schur decomposition. A complex decomposition may be forced by passing the flag `"complex"`.

The eigenvalues are optionally ordered along the diagonal according to the value of *opt*. `opt = "a"` indicates that all eigenvalues with negative real parts should be moved to the leading block of S (used in `are`), `opt = "d"` indicates that all eigenvalues with magnitude less than one should be moved to the leading block of S (used in `dare`), and `opt = "u"`, the default, indicates that no ordering of eigenvalues should occur. The leading k columns of U always span the A-invariant subspace corresponding to the k leading eigenvalues of S.

The Schur decomposition is used to compute eigenvalues of a square matrix, and has applications in the solution of algebraic Riccati equations in control (see `are` and `dare`).

See also: [rsf2csf], page 481, [ordschur], page 481, [lu], page 475, [chol], page 472, [hess], page 474, [qr], page 476, [qz], page 479, [svd], page 481.

`[U, T] = rsf2csf (UR, TR)` [Function File]

Convert a real, upper quasi-triangular Schur form *TR* to a complex, upper triangular Schur form *T*.

Note that the following relations hold:

$UR \cdot TR \cdot UR^T = UTU^\dagger$ and $U^\dagger U$ is the identity matrix I.

Note also that *U* and *T* are not unique.

See also: [schur], page 480.

`[UR, SR] = ordschur (U, S, select)` [Loadable Function]

Reorders the real Schur factorization (*U*,*S*) obtained with the `schur` function, so that selected eigenvalues appear in the upper left diagonal blocks of the quasi triangular Schur matrix.

The logical vector *select* specifies the selected eigenvalues as they appear along *S*'s diagonal.

For example, given the matrix `A = [1, 2; 3, 4]`, and its Schur decomposition

```
[U, S] = schur (A)
```

which returns

```
U =

  -0.82456  -0.56577
   0.56577  -0.82456

S =

  -0.37228  -1.00000
   0.00000   5.37228
```

It is possible to reorder the decomposition so that the positive eigenvalue is in the upper left corner, by doing:

```
[U, S] = ordschur (U, S, [0,1])
```

See also: [schur], page 480.

`angle = subspace (A, B)` [Function File]

Determine the largest principal angle between two subspaces spanned by the columns of matrices *A* and *B*.

`s = svd (A)` [Built-in Function]
`[U, S, V] = svd (A)` [Built-in Function]
`[U, S, V] = svd (A, econ)` [Built-in Function]

Compute the singular value decomposition of *A*

$$A = USV^\dagger$$

The function `svd` normally returns only the vector of singular values. When called with three return values, it computes *U*, *S*, and *V*. For example,

```
svd (hilb (3))
```

returns

```
ans =

  1.4083189
  0.1223271
  0.0026873
```

and

```
[u, s, v] = svd (hilb (3))
```

returns

```
u =

  -0.82704   0.54745   0.12766
  -0.45986  -0.52829  -0.71375
  -0.32330  -0.64901   0.68867

s =

  1.40832  0.00000  0.00000
  0.00000  0.12233  0.00000
  0.00000  0.00000  0.00269

v =

  -0.82704   0.54745   0.12766
  -0.45986  -0.52829  -0.71375
  -0.32330  -0.64901   0.68867
```

If given a second argument, `svd` returns an economy-sized decomposition, eliminating the unnecessary rows or columns of *U* or *V*.

See also: [svd_driver], page 482, [svds], page 547, [eig], page 467, [lu], page 475, [chol], page 472, [hess], page 474, [qr], page 476, [qz], page 479.

val = svd_driver () [Built-in Function]
old_val = svd_driver (*new_val*) [Built-in Function]
svd_driver (*new_val*, "*local*") [Built-in Function]

Query or set the underlying LAPACK driver used by `svd`.

Currently recognized values are `"gesvd"` and `"gesdd"`. The default is `"gesvd"`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [svd], page 481.

[*housv*, *beta*, *zer*] = housh (*x*, *j*, *z*) [Function File]

Compute Householder reflection vector *housv* to reflect *x* to be the j-th column of identity, i.e.,

```
(I - beta*housv*housv')x =  norm (x)*e(j) if x(j) < 0,
(I - beta*housv*housv')x = -norm (x)*e(j) if x(j) >= 0
```

Inputs

x vector

j index into vector

z threshold for zero (usually should be the number 0)

Outputs (see Golub and Van Loan):

beta If beta = 0, then no reflection need be applied (zer set to 0)

housv householder vector

`[u, h, nu] = krylov (A, V, k, eps1, pflg)` [Function File]

Construct an orthogonal basis *u* of block Krylov subspace

```
[v a*v a^2*v ... a^(k+1)*v]
```

using Householder reflections to guard against loss of orthogonality.

If *V* is a vector, then *h* contains the Hessenberg matrix such that `a*u == u*h+rk*ek'`, in which `rk = a*u(:,k)-u*h(:,k)`, and `ek'` is the vector `[0, 0, ..., 1]` of length `k`. Otherwise, *h* is meaningless.

If *V* is a vector and *k* is greater than `length (A) - 1`, then *h* contains the Hessenberg matrix such that `a*u == u*h`.

The value of *nu* is the dimension of the span of the Krylov subspace (based on *eps1*).

If *b* is a vector and *k* is greater than *m-1*, then *h* contains the Hessenberg decomposition of *A*.

The optional parameter *eps1* is the threshold for zero. The default value is 1e-12.

If the optional parameter *pflg* is nonzero, row pivoting is used to improve numerical behavior. The default value is 0.

Reference: A. Hodel, P. Misra, *Partial Pivoting in the Computation of Krylov Subspaces of Large Sparse Systems*, Proceedings of the 42nd IEEE Conference on Decision and Control, December 2003.

18.4 Functions of a Matrix

`expm (A)` [Function File]

Return the exponential of a matrix.

The matrix exponential is defined as the infinite Taylor series

$$\exp(A) = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots$$

However, the Taylor series is *not* the way to compute the matrix exponential; see Moler and Van Loan, *Nineteen Dubious Ways to Compute the Exponential of a Matrix*, SIAM Review, 1978. This routine uses Ward's diagonal Padé approximation method with three step preconditioning (SIAM Journal on Numerical Analysis, 1977). Diagonal Padé approximations are rational polynomials of matrices $D_q(A)^{-1}N_q(A)$ whose Taylor series matches the first $2q+1$ terms of the Taylor series above; direct evaluation of the Taylor series (with the same preconditioning steps) may be desirable in lieu of the Padé approximation when $D_q(A)$ is ill-conditioned.

See also: [logm], page 484, [sqrtm], page 484.

`s = logm (A)` [Function File]
`s = logm (A, opt_iters)` [Function File]
`[s, iters] = logm (...)` [Function File]

Compute the matrix logarithm of the square matrix *A*.

The implementation utilizes a Padé approximant and the identity

```
logm (A) = 2^k * logm (A^(1 / 2^k))
```

The optional input *opt_iters* is the maximum number of square roots to compute and defaults to 100.

The optional output *iters* is the number of square roots actually computed.

See also: [expm], page 483, [sqrtm], page 484.

`s = sqrtm (A)` [Built-in Function]
`[s, error_estimate] = sqrtm (A)` [Built-in Function]

Compute the matrix square root of the square matrix *A*.

Ref: N.J. Higham. *A New sqrtm for* MATLAB. Numerical Analysis Report No. 336, Manchester Centre for Computational Mathematics, Manchester, England, January 1999.

See also: [expm], page 483, [logm], page 484.

`kron (A, B)` [Built-in Function]
`kron (A1, A2, ...)` [Built-in Function]

Form the Kronecker product of two or more matrices.

This is defined block by block as

```
x = [ a(i,j)*b ]
```

For example:

```
kron (1:4, ones (3, 1))
     ⇒  1  2  3  4
        1  2  3  4
        1  2  3  4
```

If there are more than two input arguments *A1*, *A2*, ..., *An* the Kronecker product is computed as

```
kron (kron (A1, A2), ..., An)
```

Since the Kronecker product is associative, this is well-defined.

`blkmm (A, B)` [Built-in Function]

Compute products of matrix blocks.

The blocks are given as 2-dimensional subarrays of the arrays *A*, *B*. The size of *A* must have the form `[m,k,...]` and size of *B* must be `[k,n,...]`. The result is then of size `[m,n,...]` and is computed as follows:

```
for i = 1:prod (size (A)(3:end))
  C(:,:,i) = A(:,:,i) * B(:,:,i)
endfor
```

`X = syl (A, B, C)` [Built-in Function]

Solve the Sylvester equation

$$AX + XB = C$$

using standard LAPACK subroutines.

For example:

```
sylvester ([1, 2; 3, 4], [5, 6; 7, 8], [9, 10; 11, 12])
   ⇒ [ 0.50000, 0.66667; 0.66667, 0.50000 ]
```

18.5 Specialized Solvers

`x = bicg (A, b, rtol, maxit, M1, M2, x0)` [Function File]
`x = bicg (A, b, rtol, maxit, P)` [Function File]
`[x, flag, relres, iter, resvec] = bicg (A, b, ...)` [Function File]

Solve `A x = b` using the Bi-conjugate gradient iterative method.

- *rtol* is the relative tolerance, if not given or set to [] the default value 1e-6 is used.
- *maxit* the maximum number of outer iterations, if not given or set to [] the default value `min (20, numel (b))` is used.
- *x0* the initial guess, if not given or set to [] the default value `zeros (size (b))` is used.

A can be passed as a matrix or as a function handle or inline function `f` such that `f(x, "notransp") = A*x` and `f(x, "transp") = A'*x`.

The preconditioner *P* is given as `P = M1 * M2`. Both *M1* and *M2* can be passed as a matrix or as a function handle or inline function `g` such that `g(x, "notransp") = M1 \ x` or `g(x, "notransp") = M2 \ x` and `g(x, "transp") = M1' \ x` or `g(x, "transp") = M2' \ x`.

If called with more than one output parameter

- *flag* indicates the exit status:
  - 0: iteration converged to the within the chosen tolerance
  - 1: the maximum number of iterations was reached before convergence
  - 3: the algorithm reached stagnation

  (the value 2 is unused but skipped for compatibility).
- *relres* is the final value of the relative residual.
- *iter* is the number of iterations performed.
- *resvec* is a vector containing the relative residual at each iteration.

See also: [bicgstab], page 485, [cgs], page 486, [gmres], page 487, [pcg], page 548, [qmr], page 487.

`x = bicgstab (A, b, rtol, maxit, M1, M2, x0)` [Function File]
`x = bicgstab (A, b, rtol, maxit, P)` [Function File]
`[x, flag, relres, iter, resvec] = bicgstab (A, b, ...)` [Function File]

Solve `A x = b` using the stabilizied Bi-conjugate gradient iterative method.

- *rtol* is the relative tolerance, if not given or set to [] the default value 1e-6 is used.

- *maxit* the maximum number of outer iterations, if not given or set to [] the default value `min (20, numel (b))` is used.
- *x0* the initial guess, if not given or set to [] the default value `zeros (size (b))` is used.

A can be passed as a matrix or as a function handle or inline function `f` such that `f(x) = A*x`.

The preconditioner *P* is given as `P = M1 * M2`. Both *M1* and *M2* can be passed as a matrix or as a function handle or inline function `g` such that `g(x) = M1 \ x` or `g(x) = M2 \ x`.

If called with more than one output parameter

- *flag* indicates the exit status:
  - 0: iteration converged to the within the chosen tolerance
  - 1: the maximum number of iterations was reached before convergence
  - 3: the algorithm reached stagnation

  (the value 2 is unused but skipped for compatibility).
- *relres* is the final value of the relative residual.
- *iter* is the number of iterations performed.
- *resvec* is a vector containing the relative residual at each iteration.

See also: [bicg], page 485, [cgs], page 486, [gmres], page 487, [pcg], page 548, [qmr], page 487.

`x = cgs (A, b, rtol, maxit, M1, M2, x0)` [Function File]
`x = cgs (A, b, rtol, maxit, P)` [Function File]
`[x, flag, relres, iter, resvec] = cgs (A, b, ...)` [Function File]

Solve `A x = b`, where *A* is a square matrix, using the Conjugate Gradients Squared method.

- *rtol* is the relative tolerance, if not given or set to [] the default value 1e-6 is used.
- *maxit* the maximum number of outer iterations, if not given or set to [] the default value `min (20, numel (b))` is used.
- *x0* the initial guess, if not given or set to [] the default value `zeros (size (b))` is used.

A can be passed as a matrix or as a function handle or inline function `f` such that `f(x) = A*x`.

The preconditioner *P* is given as `P = M1 * M2`. Both *M1* and *M2* can be passed as a matrix or as a function handle or inline function `g` such that `g(x) = M1 \ x` or `g(x) = M2 \ x`.

If called with more than one output parameter

- *flag* indicates the exit status:
  - 0: iteration converged to the within the chosen tolerance
  - 1: the maximum number of iterations was reached before convergence
  - 3: the algorithm reached stagnation

(the value 2 is unused but skipped for compatibility).

- *relres* is the final value of the relative residual.
- *iter* is the number of iterations performed.
- *resvec* is a vector containing the relative residual at each iteration.

See also: [pcg], page 548, [bicgstab], page 485, [bicg], page 485, [gmres], page 487, [qmr], page 487.

`x = gmres (A, b, m, rtol, maxit, M1, M2, x0)` [Function File]
`x = gmres (A, b, m, rtol, maxit, P)` [Function File]
`[x, flag, relres, iter, resvec] = gmres (...)` [Function File]

Solve `A x = b` using the Preconditioned GMRES iterative method with restart, a.k.a. PGMRES(m).

- *rtol* is the relative tolerance, if not given or set to [] the default value 1e-6 is used.
- *maxit* is the maximum number of outer iterations, if not given or set to [] the default value `min (10, numel (b) / restart)` is used.
- *x0* is the initial guess, if not given or set to [] the default value `zeros (size (b))` is used.
- *m* is the restart parameter, if not given or set to [] the default value `numel (b)` is used.

Argument *A* can be passed as a matrix, function handle, or inline function `f` such that `f(x) = A*x`.

The preconditioner *P* is given as `P = M1 * M2`. Both *M1* and *M2* can be passed as a matrix, function handle, or inline function `g` such that `g(x) = M1\x` or `g(x) = M2\x`.

Besides the vector *x*, additional outputs are:

- *flag* indicates the exit status:

  0 : iteration converged to within the specified tolerance
  1 : maximum number of iterations exceeded
  2 : unused, but skipped for compatibility
  3 : algorithm reached stagnation (no change between iterations)
- *relres* is the final value of the relative residual.
- *iter* is a vector containing the number of outer iterations and total iterations performed.
- *resvec* is a vector containing the relative residual at each iteration.

See also: [bicg], page 485, [bicgstab], page 485, [cgs], page 486, [pcg], page 548, [pcr], page 550, [qmr], page 487.

`x = qmr (A, b, rtol, maxit, M1, M2, x0)` [Function File]
`x = qmr (A, b, rtol, maxit, P)` [Function File]
`[x, flag, relres, iter, resvec] = qmr (A, b, ...)` [Function File]

Solve `A x = b` using the Quasi-Minimal Residual iterative method (without look-ahead).

- *rtol* is the relative tolerance, if not given or set to [] the default value 1e-6 is used.

- *maxit* the maximum number of outer iterations, if not given or set to [] the default value `min (20, numel (b))` is used.
- *x0* the initial guess, if not given or set to [] the default value `zeros (size (b))` is used.

A can be passed as a matrix or as a function handle or inline function `f` such that `f(x, "notransp") = A*x` and `f(x, "transp") = A'*x`.

The preconditioner *P* is given as `P = M1 * M2`. Both *M1* and *M2* can be passed as a matrix or as a function handle or inline function `g` such that `g(x, "notransp") = M1 \ x` or `g(x, "notransp") = M2 \ x` and `g(x, "transp") = M1' \ x` or `g(x, "transp") = M2' \ x`.

If called with more than one output parameter

- *flag* indicates the exit status:
  - 0: iteration converged to the within the chosen tolerance
  - 1: the maximum number of iterations was reached before convergence
  - 3: the algorithm reached stagnation

  (the value 2 is unused but skipped for compatibility).
- *relres* is the final value of the relative residual.
- *iter* is the number of iterations performed.
- *resvec* is a vector containing the residual norms at each iteration.

References:

1. R. Freund and N. Nachtigal, *QMR: a quasi-minimal residual method for non-Hermitian linear systems*, Numerische Mathematik, 1991, 60, pp. 315-339.
2. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhour, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the solution of linear systems: Building blocks for iterative methods*, SIAM, 2nd ed., 1994.

See also: [bicg], page 485, [bicgstab], page 485, [cgs], page 486, [gmres], page 487, [pcg], page 548.

19 Vectorization and Faster Code Execution

Vectorization is a programming technique that uses vector operations instead of element-by-element loop-based operations. Besides frequently producing more succinct Octave code, vectorization also allows for better optimization in the subsequent implementation. The optimizations may occur either in Octave's own Fortran, C, or C++ internal implementation, or even at a lower level depending on the compiler and external numerical libraries used to build Octave. The ultimate goal is to make use of your hardware's vector instructions if possible or to perform other optimizations in software.

Vectorization is not a concept unique to Octave, but it is particularly important because Octave is a matrix-oriented language. Vectorized Octave code will see a dramatic speed up (10X–100X) in most cases.

This chapter discusses vectorization and other techniques for writing faster code.

19.1 Basic Vectorization

To a very good first approximation, the goal in vectorization is to write code that avoids loops and uses whole-array operations. As a trivial example, consider

```
for i = 1:n
  for j = 1:m
    c(i,j) = a(i,j) + b(i,j);
  endfor
endfor
```

compared to the much simpler

```
c = a + b;
```

This isn't merely easier to write; it is also internally much easier to optimize. Octave delegates this operation to an underlying implementation which, among other optimizations, may use special vector hardware instructions or could conceivably even perform the additions in parallel. In general, if the code is vectorized, the underlying implementation has more freedom about the assumptions it can make in order to achieve faster execution.

This is especially important for loops with "cheap" bodies. Often it suffices to vectorize just the innermost loop to get acceptable performance. A general rule of thumb is that the "order" of the vectorized body should be greater or equal to the "order" of the enclosing loop.

As a less trivial example, instead of

```
for i = 1:n-1
  a(i) = b(i+1) - b(i);
endfor
```

write

```
a = b(2:n) - b(1:n-1);
```

This shows an important general concept about using arrays for indexing instead of looping over an index variable. See Section 8.1 [Index Expressions], page 135. Also use boolean indexing generously. If a condition needs to be tested, this condition can also be written as a boolean index. For instance, instead of

```
for i = 1:n
  if (a(i) > 5)
    a(i) -= 20
  endif
endfor
```

write

```
a(a>5) -= 20;
```

which exploits the fact that `a > 5` produces a boolean index.

Use elementwise vector operators whenever possible to avoid looping (operators like `.*` and `.^`). See Section 8.3 [Arithmetic Ops], page 141. For simple inline functions, the `vectorize` function can do this automatically.

`vectorize (`*`fun`*`)` [Built-in Function]

Create a vectorized version of the inline function *fun* by replacing all occurrences of `*`, `/`, etc., with `.*`, `./`, etc.

This may be useful, for example, when using inline functions with numerical integration or optimization where a vector-valued function is expected.

```
fcn = vectorize (inline ("x^2 - 1"))
   ⇒ fcn = f(x) = x.^2 - 1
quadv (fcn, 0, 3)
   ⇒ 6
```

See also: [inline], page 202, [formula], page 202, [argnames], page 202.

Also exploit broadcasting in these elementwise operators both to avoid looping and unnecessary intermediate memory allocations. See Section 19.2 [Broadcasting], page 491.

Use built-in and library functions if possible. Built-in and compiled functions are very fast. Even with an m-file library function, chances are good that it is already optimized, or will be optimized more in a future release.

For instance, even better than

```
a = b(2:n) - b(1:n-1);
```

is

```
a = diff (b);
```

Most Octave functions are written with vector and array arguments in mind. If you find yourself writing a loop with a very simple operation, chances are that such a function already exists. The following functions occur frequently in vectorized code:

- Index manipulation
  - find
  - sub2ind
  - ind2sub
  - sort
  - unique

- lookup
- ifelse / merge
- Repetition
  - repmat
  - repelems
- Vectorized arithmetic
  - sum
  - prod
  - cumsum
  - cumprod
  - sumsq
  - diff
  - dot
  - cummax
  - cummin
- Shape of higher dimensional arrays
  - reshape
  - resize
  - permute
  - squeeze
  - deal

19.2 Broadcasting

Broadcasting refers to how Octave binary operators and functions behave when their matrix or array operands or arguments differ in size. Since version 3.6.0, Octave now automatically broadcasts vectors, matrices, and arrays when using elementwise binary operators and functions. Broadly speaking, smaller arrays are "broadcast" across the larger one, until they have a compatible shape. The rule is that corresponding array dimensions must either

1. be equal, or
2. one of them must be 1.

In case all dimensions are equal, no broadcasting occurs and ordinary element-by-element arithmetic takes place. For arrays of higher dimensions, if the number of dimensions isn't the same, then missing trailing dimensions are treated as 1. When one of the dimensions is 1, the array with that singleton dimension gets copied along that dimension until it matches the dimension of the other array. For example, consider

```
x = [1 2 3;
     4 5 6;
     7 8 9];

y = [10 20 30];

x + y
```

Without broadcasting, `x + y` would be an error because the dimensions do not agree. However, with broadcasting it is as if the following operation were performed:

```
x = [1 2 3
     4 5 6
     7 8 9];

y = [10 20 30
     10 20 30
     10 20 30];

x + y
⇒    11   22   33
     14   25   36
     17   28   39
```

That is, the smaller array of size `[1 3]` gets copied along the singleton dimension (the number of rows) until it is `[3 3]`. No actual copying takes place, however. The internal implementation reuses elements along the necessary dimension in order to achieve the desired effect without copying in memory.

Both arrays can be broadcast across each other, for example, all pairwise differences of the elements of a vector with itself:

```
y - y'
⇒     0   10   20
    -10    0   10
    -20  -10    0
```

Here the vectors of size `[1 3]` and `[3 1]` both get broadcast into matrices of size `[3 3]` before ordinary matrix subtraction takes place.

A special case of broadcasting that may be familiar is when all dimensions of the array being broadcast are 1, i.e., the array is a scalar. Thus for example, operations like `x - 42` and `max (x, 2)` are basic examples of broadcasting.

For a higher-dimensional example, suppose `img` is an RGB image of size `[m n 3]` and we wish to multiply each color by a different scalar. The following code accomplishes this with broadcasting,

```
img .*= permute ([0.8, 0.9, 1.2], [1, 3, 2]);
```

Note the usage of permute to match the dimensions of the `[0.8, 0.9, 1.2]` vector with `img`.

For functions that are not written with broadcasting semantics, `bsxfun` can be useful for coercing them to broadcast.

bsxfun (*f*, *A*, *B*) [Built-in Function]

The binary singleton expansion function performs broadcasting, that is, it applies a binary function *f* element-by-element to two array arguments *A* and *B*, and expands as necessary singleton dimensions in either input argument.

f is a function handle, inline function, or string containing the name of the function to evaluate. The function *f* must be capable of accepting two column-vector arguments of equal length, or one column vector argument and a scalar.

The dimensions of A and B must be equal or singleton. The singleton dimensions of the arrays will be expanded to the same dimensionality as the other array.

See also: [arrayfun], page 494, [cellfun], page 496.

Broadcasting is only applied if either of the two broadcasting conditions hold. As usual, however, broadcasting does not apply when two dimensions differ and neither is 1:

```
x = [1 2 3
     4 5 6];
y = [10 20
     30 40];
x + y
```

This will produce an error about nonconformant arguments.

Besides common arithmetic operations, several functions of two arguments also broadcast. The full list of functions and operators that broadcast is

```
plus      +  .+
minus     -  .-
times     .*
rdivide   ./
ldivide   .\
power     .^  .**
lt        <
le        <=
eq        ==
gt        >
ge        >=
ne        !=  ~=
and       &
or        |
atan2
hypot
max
min
mod
rem
xor

+=  -=  .+=  .-=  .*=  ./=  .\=  .^=  .**=  &=  |=
```

Beware of resorting to broadcasting if a simpler operation will suffice. For matrices a and b, consider the following:

```
c = sum (permute (a, [1, 3, 2]) .* permute (b, [3, 2, 1]), 3);
```

This operation broadcasts the two matrices with permuted dimensions across each other during elementwise multiplication in order to obtain a larger 3-D array, and this array is then summed along the third dimension. A moment of thought will prove that this operation is simply the much faster ordinary matrix multiplication, `c = a*b;`.

A note on terminology: "broadcasting" is the term popularized by the Numpy numerical environment in the Python programming language. In other programming languages and environments, broadcasting may also be known as *binary singleton expansion* (BSX, in MATLAB, and the origin of the name of the `bsxfun` function), *recycling* (R programming language), *single-instruction multiple data* (SIMD), or *replication*.

19.2.1 Broadcasting and Legacy Code

The new broadcasting semantics almost never affect code that worked in previous versions of Octave. Consequently, all code inherited from MATLAB that worked in previous versions of Octave should still work without change in Octave. The only exception is code such as

```
try
  c = a.*b;
catch
  c = a.*a;
end_try_catch
```

that may have relied on matrices of different size producing an error. Because such operation is now valid Octave syntax, this will no longer produce an error. Instead, the following code should be used:

```
if (isequal (size (a), size (b)))
  c = a .* b;
else
  c = a .* a;
endif
```

19.3 Function Application

As a general rule, functions should already be written with matrix arguments in mind and should consider whole matrix operations in a vectorized manner. Sometimes, writing functions in this way appears difficult or impossible for various reasons. For those situations, Octave provides facilities for applying a function to each element of an array, cell, or struct.

`arrayfun (`*func*`,` *A*`)` [Function File]
x `= arrayfun (`*func*`,` *A*`)` [Function File]
x `= arrayfun (`*func*`,` *A*`,` *b*`, ...)` [Function File]
`[`*x*`,` *y*`, ...] = arrayfun (`*func*`,` *A*`, ...)` [Function File]
`arrayfun (..., "`*UniformOutput*`",` *val*`)` [Function File]
`arrayfun (..., "`*ErrorHandler*`",` *errfunc*`)` [Function File]

Execute a function on each element of an array.

This is useful for functions that do not accept array arguments. If the function does accept array arguments it is better to call the function directly.

The first input argument *func* can be a string, a function handle, an inline function, or an anonymous function. The input argument *A* can be a logic array, a numeric array, a string array, a structure array, or a cell array. By a call of the function `arrayfun` all elements of *A* are passed on to the named function *func* individually.

The named function can also take more than two input arguments, with the input arguments given as third input argument *b*, fourth input argument *c*, ... If given more than one array input argument then all input arguments must have the same sizes, for example:

```
arrayfun (@atan2, [1, 0], [0, 1])
     ⇒ [ 1.5708   0.0000 ]
```

If the parameter *val* after a further string input argument `"UniformOutput"` is set `true` (the default), then the named function *func* must return a single element which then will be concatenated into the return value and is of type matrix. Otherwise, if that parameter is set to `false`, then the outputs are concatenated in a cell array. For example:

```
arrayfun (@(x,y) x:y, "abc", "def", "UniformOutput", false)
⇒
   {
     [1,1] = abcd
     [1,2] = bcde
     [1,3] = cdef
   }
```

If more than one output arguments are given then the named function must return the number of return values that also are expected, for example:

```
[A, B, C] = arrayfun (@find, [10; 0], "UniformOutput", false)
⇒
A =
{
   [1,1] =  1
   [2,1] = [](0x0)
}
B =
{
   [1,1] =  1
   [2,1] = [](0x0)
}
C =
{
   [1,1] =  10
   [2,1] = [](0x0)
}
```

If the parameter *errfunc* after a further string input argument `"ErrorHandler"` is another string, a function handle, an inline function, or an anonymous function, then *errfunc* defines a function to call in the case that *func* generates an error. The definition of the function must be of the form

```
function [...] = errfunc (s, ...)
```

where there is an additional input argument to *errfunc* relative to *func*, given by *s*. This is a structure with the elements `"identifier"`, `"message"`, and `"index"` giving, respectively, the error identifier, the error message, and the index of the array elements that caused the error. The size of the output argument of *errfunc* must have the same size as the output argument of *func*, otherwise a real error is thrown. For example:

```
function y = ferr (s, x), y = "MyString"; endfunction
arrayfun (@str2num, [1234],
          "UniformOutput", false, "ErrorHandler", @ferr)
⇒
   {
     [1,1] = MyString
   }
```

See also: [spfun], page 496, [cellfun], page 496, [structfun], page 498.

y = `spfun` (*f*, *S*) [Function File]

Compute `f(S)` for the nonzero values of *S*.

This results in a sparse matrix with the same structure as *S*. The function *f* can be passed as a string, a function handle, or an inline function.

See also: [arrayfun], page 494, [cellfun], page 496, [structfun], page 498.

`cellfun` (*name*, *C*) [Built-in Function]
`cellfun` ("*size*", *C*, *k*) [Built-in Function]
`cellfun` ("*isclass*", *C*, *class*) [Built-in Function]
`cellfun` (*func*, *C*) [Built-in Function]
`cellfun` (*func*, *C*, *D*) [Built-in Function]
`[a, ...] = cellfun (...)` [Built-in Function]
`cellfun` (. . . , "*ErrorHandler*", *errfunc*) [Built-in Function]
`cellfun` (. . . , "*UniformOutput*", *val*) [Built-in Function]

Evaluate the function named *name* on the elements of the cell array *C*.

Elements in *C* are passed on to the named function individually. The function *name* can be one of the functions

`isempty` Return 1 for empty elements.

`islogical`
Return 1 for logical elements.

`isnumeric`
Return 1 for numeric elements.

`isreal` Return 1 for real elements.

`length` Return a vector of the lengths of cell elements.

`ndims` Return the number of dimensions of each element.

`numel`
`prodofsize`
Return the number of elements contained within each cell element. The number is the product of the dimensions of the object at each cell element.

`size` Return the size along the *k*-th dimension.

`isclass` Return 1 for elements of *class*.

Additionally, `cellfun` accepts an arbitrary function *func* in the form of an inline function, function handle, or the name of a function (in a character string). The function can take one or more arguments, with the inputs arguments given by *C*, *D*, etc. Equally the function can return one or more output arguments. For example:

```
cellfun ("atan2", {1, 0}, {0, 1})
     ⇒ [ 1.57080   0.00000 ]
```

The number of output arguments of `cellfun` matches the number of output arguments of the function. The outputs of the function will be collected into the output arguments of `cellfun` like this:

```
function [a, b] = twoouts (x)
  a = x;
  b = x*x;
endfunction
[aa, bb] = cellfun (@twoouts, {1, 2, 3})
     ⇒
        aa =
           1 2 3
        bb =
           1 4 9
```

Note that per default the output argument(s) are arrays of the same size as the input arguments. Input arguments that are singleton (1x1) cells will be automatically expanded to the size of the other arguments.

If the parameter `"UniformOutput"` is set to true (the default), then the function must return scalars which will be concatenated into the return array(s). If `"UniformOutput"` is false, the outputs are concatenated into a cell array (or cell arrays). For example:

```
cellfun ("tolower", {"Foo", "Bar", "FooBar"},
         "UniformOutput", false)
⇒ {"foo", "bar", "foobar"}
```

Given the parameter `"ErrorHandler"`, then *errfunc* defines a function to call in case *func* generates an error. The form of the function is

```
function [...] = errfunc (s, ...)
```

where there is an additional input argument to *errfunc* relative to *func*, given by *s*. This is a structure with the elements `"identifier"`, `"message"` and `"index"`, giving respectively the error identifier, the error message, and the index into the input arguments of the element that caused the error. For example:

```
function y = foo (s, x), y = NaN; endfunction
cellfun ("factorial", {-1,2}, "ErrorHandler", @foo)
⇒ [NaN 2]
```

Use `cellfun` intelligently. The `cellfun` function is a useful tool for avoiding loops. It is often used with anonymous function handles; however, calling an anonymous function involves an overhead quite comparable to the overhead of an m-file function. Passing a handle to a built-in function is faster, because the interpreter is not involved in the internal loop. For example:

```
a = {...}
v = cellfun (@(x) det (x), a); # compute determinants
v = cellfun (@det, a); # faster
```

See also: [arrayfun], page 494, [structfun], page 498, [spfun], page 496.

`structfun (func, S)` [Function File]
`[A, ...] = structfun (...)` [Function File]
`structfun (..., "ErrorHandler", errfunc)` [Function File]
`structfun (..., "UniformOutput", val)` [Function File]

Evaluate the function named *name* on the fields of the structure *S*. The fields of *S* are passed to the function *func* individually.

`structfun` accepts an arbitrary function *func* in the form of an inline function, function handle, or the name of a function (in a character string). In the case of a character string argument, the function must accept a single argument named *x*, and it must return a string value. If the function returns more than one argument, they are returned as separate output variables.

If the parameter `"UniformOutput"` is set to true (the default), then the function must return a single element which will be concatenated into the return value. If `"UniformOutput"` is false, the outputs are placed into a structure with the same fieldnames as the input structure.

```
s.name1 = "John Smith";
s.name2 = "Jill Jones";
structfun (@(x) regexp (x, '(\w+)$', "matches"){1}, s,
           "UniformOutput", false)
⇒
   {
     name1 = Smith
     name2 = Jones
   }
```

Given the parameter `"ErrorHandler"`, *errfunc* defines a function to call in case *func* generates an error. The form of the function is

```
function [...] = errfunc (se, ...)
```

where there is an additional input argument to *errfunc* relative to *func*, given by *se*. This is a structure with the elements `"identifier"`, `"message"` and `"index"`, giving respectively the error identifier, the error message, and the index into the input arguments of the element that caused the error. For an example on how to use an error handler, see [cellfun], page 496.

See also: [cellfun], page 496, [arrayfun], page 494, [spfun], page 496.

Consistent with earlier advice, seek to use Octave built-in functions whenever possible for the best performance. This advice applies especially to the four functions above. For example, when adding two arrays together element-by-element one could use a handle to the built-in addition function `@plus` or define an anonymous function `@(x,y) x + y`. But, the anonymous function is 60% slower than the first method. See Section 34.4.2 [Operator Overloading], page 731, for a list of basic functions which might be used in place of anonymous ones.

19.4 Accumulation

Whenever it's possible to categorize according to indices the elements of an array when performing a computation, accumulation functions can be useful.

`accumarray (subs, vals, sz, func, fillval, issparse)` [Function File]
`accumarray (subs, vals, ...)` [Function File]

Create an array by accumulating the elements of a vector into the positions defined by their subscripts.

The subscripts are defined by the rows of the matrix *subs* and the values by *vals*. Each row of *subs* corresponds to one of the values in *vals*. If *vals* is a scalar, it will be used for each of the row of *subs*. If *subs* is a cell array of vectors, all vectors must be of the same length, and the subscripts in the kth vector must correspond to the kth dimension of the result.

The size of the matrix will be determined by the subscripts themselves. However, if *sz* is defined it determines the matrix size. The length of *sz* must correspond to the number of columns in *subs*. An exception is if *subs* has only one column, in which case *sz* may be the dimensions of a vector and the subscripts of *subs* are taken as the indices into it.

The default action of `accumarray` is to sum the elements with the same subscripts. This behavior can be modified by defining the *func* function. This should be a function or function handle that accepts a column vector and returns a scalar. The result of the function should not depend on the order of the subscripts.

The elements of the returned array that have no subscripts associated with them are set to zero. Defining *fillval* to some other value allows these values to be defined. This behavior changes, however, for certain values of *func*. If *func* is `min` (respectively, `max`) then the result will be filled with the minimum (respectively, maximum) integer if *vals* is of integral type, logical false (respectively, logical true) if *vals* is of logical type, zero if *fillval* is zero and all values are non-positive (respectively, non-negative), and NaN otherwise.

By default `accumarray` returns a full matrix. If *issparse* is logically true, then a sparse matrix is returned instead.

The following `accumarray` example constructs a frequency table that in the first column counts how many occurrences each number in the second column has, taken from the vector *x*. Note the usage of `unique` for assigning to all repeated elements of *x* the same index (see [unique], page 635).

```
x = [91, 92, 90, 92, 90, 89, 91, 89, 90, 100, 100, 100];
[u, ~, j] = unique (x);
[accumarray(j', 1), u']
  ⇒  2    89
     3    90
     2    91
     2    92
     3   100
```

Another example, where the result is a multi-dimensional 3-D array and the default value (zero) appears in the output:

```
accumarray ([1, 1, 1;
             2, 1, 2;
             2, 3, 2;
             2, 1, 2;
             2, 3, 2], 101:105)
⇒ ans(:,:,1) = [101, 0, 0; 0, 0, 0]
⇒ ans(:,:,2) = [0, 0, 0; 206, 0, 208]
```

The sparse option can be used as an alternative to the `sparse` constructor (see [sparse], page 524). Thus

```
sparse (i, j, sv)
```

can be written with `accumarray` as

```
accumarray ([i, j], sv', [], [], 0, true)
```

For repeated indices, `sparse` adds the corresponding value. To take the minimum instead, use `min` as an accumulator function:

```
accumarray ([i, j], sv', [], @min, 0, true)
```

The complexity of accumarray in general for the non-sparse case is generally O(M+N), where N is the number of subscripts and M is the maximum subscript (linearized in multi-dimensional case). If *func* is one of `@sum` (default), `@max`, `@min` or `@(x) {x}`, an optimized code path is used. Note that for general reduction function the interpreter overhead can play a major part and it may be more efficient to do multiple accumarray calls and compute the results in a vectorized manner.

See also: [accumdim], page 500, [unique], page 635, [sparse], page 524.

`accumdim (subs, vals, dim, n, func, fillval)` [Function File]

Create an array by accumulating the slices of an array into the positions defined by their subscripts along a specified dimension.

The subscripts are defined by the index vector *subs*. The dimension is specified by *dim*. If not given, it defaults to the first non-singleton dimension. The length of *subs* must be equal to `size (vals, dim)`.

The extent of the result matrix in the working dimension will be determined by the subscripts themselves. However, if *n* is defined it determines this extent.

The default action of `accumdim` is to sum the subarrays with the same subscripts. This behavior can be modified by defining the *func* function. This should be a function or function handle that accepts an array and a dimension, and reduces the array along this dimension. As a special exception, the built-in `min` and `max` functions can be used directly, and `accumdim` accounts for the middle empty argument that is used in their calling.

The slices of the returned array that have no subscripts associated with them are set to zero. Defining *fillval* to some other value allows these values to be defined.

An example of the use of `accumdim` is:

```
accumdim ([1, 2, 1, 2, 1], [ 7, -10,   4;
                             -5, -12,   8;
                            -12,   2,   8;
                            -10,   9,  -3;
                             -5,  -3, -13])
⇒ [-10,-11,-1;-15,-3,5]
```

See also: [accumarray], page 499.

19.5 JIT Compiler

Vectorization is the preferred technique for eliminating loops and speeding up code. Nevertheless, it is not always possible to replace every loop. In such situations it may be worth trying Octave's **experimental** Just-In-Time (JIT) compiler.

A JIT compiler works by analyzing the body of a loop, translating the Octave statements into another language, compiling the new code segment into an executable, and then running the executable and collecting any results. The process is not simple and there is a significant amount of work to perform for each step. It can still make sense, however, if the number of loop iterations is large. Because Octave is an interpreted language every time through a loop Octave must parse the statements in the loop body before executing them. With a JIT compiler this is done just once when the body is translated to another language.

The JIT compiler is a very new feature in Octave and not all valid Octave statements can currently be accelerated. However, if no other technique is available it may be worth benchmarking the code with JIT enabled. The function `jit_enable` is used to turn compilation on or off. The function `jit_startcnt` sets the threshold for acceleration. Loops with iteration counts above `jit_startcnt` will be accelerated. The functions `jit_failcnt` and `debug_jit` are not likely to be of use to anyone not working directly on the implementation of the JIT compiler.

val = `jit_enable` () [Built-in Function]
old_val = `jit_enable` (*new_val*) [Built-in Function]
`jit_enable` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that enables Octave's JIT compiler.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [jit_startcnt], page 501, [debug_jit], page 502.

val = `jit_startcnt` () [Built-in Function]
old_val = `jit_startcnt` (*new_val*) [Built-in Function]
`jit_startcnt` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that determines whether JIT compilation will take place for a specific loop.

Because compilation is a costly operation it does not make sense to employ JIT when the loop count is low. By default only loops with greater than 1000 iterations will be accelerated.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [jit_enable], page 501, [jit_failcnt], page 502, [debug_jit], page 502.

val = jit_failcnt () [Built-in Function]
old_val = jit_failcnt (*new_val*) [Built-in Function]
jit_failcnt (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that counts the number of JIT fail exceptions for Octave's JIT compiler.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [jit_enable], page 501, [jit_startcnt], page 501, [debug_jit], page 502.

val = debug_jit () [Built-in Function]
old_val = debug_jit (*new_val*) [Built-in Function]
debug_jit (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that determines whether debugging/tracing is enabled for Octave's JIT compiler.

When called from inside a function with the "local" option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [jit_enable], page 501, [jit_startcnt], page 501.

19.6 Miscellaneous Techniques

Here are some other ways of improving the execution speed of Octave programs.

- Avoid computing costly intermediate results multiple times. Octave currently does not eliminate common subexpressions. Also, certain internal computation results are cached for variables. For instance, if a matrix variable is used multiple times as an index, checking the indices (and internal conversion to integers) is only done once.
- Be aware of lazy copies (copy-on-write). When a copy of an object is created, the data is not immediately copied, but rather shared. The actual copying is postponed until the copied data needs to be modified. For example:

  ```
  a = zeros (1000); # create a 1000x1000 matrix
  b = a; # no copying done here
  b(1) = 1; # copying done here
  ```

  Lazy copying applies to whole Octave objects such as matrices, cells, struct, and also individual cell or struct elements (not array elements).

  Additionally, index expressions also use lazy copying when Octave can determine that the indexed portion is contiguous in memory. For example:

  ```
  a = zeros (1000); # create a 1000x1000 matrix
  b = a(:,10:100);  # no copying done here
  b = a(10:100,:);  # copying done here
  ```

This applies to arrays (matrices), cell arrays, and structs indexed using '()'. Index expressions generating comma-separated lists can also benefit from shallow copying in some cases. In particular, when *a* is a struct array, expressions like `{a.x}, {a(:,2).x}` will use lazy copying, so that data can be shared between a struct array and a cell array.

Most indexing expressions do not live longer than their parent objects. In rare cases, however, a lazily copied slice outlasts its parent, in which case it becomes orphaned, still occupying unnecessarily more memory than needed. To provide a remedy working in most real cases, Octave checks for orphaned lazy slices at certain situations, when a value is stored into a "permanent" location, such as a named variable or cell or struct element, and possibly economizes them. For example:

```
a = zeros (1000); # create a 1000x1000 matrix
b = a(:,10:100);  # lazy slice
a = []; # the original "a" array is still allocated
c{1} = b; # b is reallocated at this point
```

- Avoid deep recursion. Function calls to m-file functions carry a relatively significant overhead, so rewriting a recursion as a loop often helps. Also, note that the maximum level of recursion is limited.
- Avoid resizing matrices unnecessarily. When building a single result matrix from a series of calculations, set the size of the result matrix first, then insert values into it. Write

```
result = zeros (big_n, big_m)
for i = over:and_over
  ridx = ...
  cidx = ...
  result(ridx, cidx) = new_value ();
endfor
```

instead of

```
result = [];
for i = ever:and_ever
  result = [ result, new_value() ];
endfor
```

Sometimes the number of items can not be computed in advance, and stack-like operations are needed. When elements are being repeatedly inserted or removed from the end of an array, Octave detects it as stack usage and attempts to use a smarter memory management strategy by pre-allocating the array in bigger chunks. This strategy is also applied to cell and struct arrays.

```
a = [];
while (condition)
  ...
  a(end+1) = value; # "push" operation
  ...
  a(end) = []; # "pop" operation
  ...
endwhile
```

- Avoid calling `eval` or `feval` excessively. Parsing input or looking up the name of a function in the symbol table are relatively expensive operations.

  If you are using `eval` merely as an exception handling mechanism, and not because you need to execute some arbitrary text, use the `try` statement instead. See Section 10.9 [The try Statement], page 168.

- Use `ignore_function_time_stamp` when appropriate. If you are calling lots of functions, and none of them will need to change during your run, set the variable `ignore_function_time_stamp` to `"all"`. This will stop Octave from checking the time stamp of a function file to see if it has been updated while the program is being run.

19.7 Examples

The following are examples of vectorization questions asked by actual users of Octave and their solutions.

- For a vector `A`, the following loop

  ```
  n = length (A);
  B = zeros (n, 2);
  for i = 1:length (A)
    ## this will be two columns, the first is the difference and
    ## the second the mean of the two elements used for the diff.
    B(i,:) = [A(i+1)-A(i), (A(i+1) + A(i))/2];
  endfor
  ```

  can be turned into the following one-liner:

  ```
  B = [diff(A)(:), 0.5*(A(1:end-1)+A(2:end))(:)]
  ```

  Note the usage of colon indexing to flatten an intermediate result into a column vector. This is a common vectorization trick.

20 Nonlinear Equations

20.1 Solvers

Octave can solve sets of nonlinear equations of the form

$$f(x) = 0$$

using the function `fsolve`, which is based on the MINPACK subroutine `hybrd`. This is an iterative technique so a starting point must be provided. This also has the consequence that convergence is not guaranteed even if a solution exists.

`fsolve (`*fcn*`,` *x0*`,` *options*`)` [Function File]
`[`*x*`,` *fvec*`,` *info*`,` *output*`,` *fjac*`] = fsolve (`*fcn*`, ...)` [Function File]

Solve a system of nonlinear equations defined by the function *fcn*.

fcn should accept a vector (array) defining the unknown variables, and return a vector of left-hand sides of the equations. Right-hand sides are defined to be zeros. In other words, this function attempts to determine a vector *x* such that `fcn (x)` gives (approximately) all zeros.

x0 determines a starting guess. The shape of *x0* is preserved in all calls to *fcn*, but otherwise it is treated as a column vector.

options is a structure specifying additional options. Currently, `fsolve` recognizes these options: `"FunValCheck"`, `"OutputFcn"`, `"TolX"`, `"TolFun"`, `"MaxIter"`, `"MaxFunEvals"`, `"Jacobian"`, `"Updating"`, `"ComplexEqn"` `"TypicalX"`, `"AutoScaling"` and `"FinDiffType"`.

If `"Jacobian"` is `"on"`, it specifies that *fcn*, called with 2 output arguments also returns the Jacobian matrix of right-hand sides at the requested point. `"TolX"` specifies the termination tolerance in the unknown variables, while `"TolFun"` is a tolerance for equations. Default is `1e-7` for both `"TolX"` and `"TolFun"`.

If `"AutoScaling"` is on, the variables will be automatically scaled according to the column norms of the (estimated) Jacobian. As a result, TolF becomes scaling-independent. By default, this option is off because it may sometimes deliver unexpected (though mathematically correct) results.

If `"Updating"` is `"on"`, the function will attempt to use Broyden updates to update the Jacobian, in order to reduce the amount of Jacobian calculations. If your user function always calculates the Jacobian (regardless of number of output arguments) then this option provides no advantage and should be set to false.

`"ComplexEqn"` is `"on"`, `fsolve` will attempt to solve complex equations in complex variables, assuming that the equations possess a complex derivative (i.e., are holomorphic). If this is not what you want, you should unpack the real and imaginary parts of the system to get a real system.

For description of the other options, see `optimset`.

On return, *fval* contains the value of the function *fcn* evaluated at *x*.

info may be one of the following values:

1 Converged to a solution point. Relative residual error is less than specified by TolFun.

2 Last relative step size was less that TolX.

3 Last relative decrease in residual was less than TolF.

0 Iteration limit exceeded.

-3 The trust region radius became excessively small.

Note: If you only have a single nonlinear equation of one variable, using `fzero` is usually a much better idea.

Note about user-supplied Jacobians: As an inherent property of the algorithm, a Jacobian is always requested for a solution vector whose residual vector is already known, and it is the last accepted successful step. Often this will be one of the last two calls, but not always. If the savings by reusing intermediate results from residual calculation in Jacobian calculation are significant, the best strategy is to employ OutputFcn: After a vector is evaluated for residuals, if OutputFcn is called with that vector, then the intermediate results should be saved for future Jacobian evaluation, and should be kept until a Jacobian evaluation is requested or until OutputFcn is called with a different vector, in which case they should be dropped in favor of this most recent vector. A short example how this can be achieved follows:

```
function [fvec, fjac] = user_func (x, optimvalues, state)
persistent sav = [], sav0 = [];
if (nargin == 1)
  ## evaluation call
  if (nargout == 1)
    sav0.x = x; # mark saved vector
    ## calculate fvec, save results to sav0.
  elseif (nargout == 2)
    ## calculate fjac using sav.
  endif
else
  ## outputfcn call.
  if (all (x == sav0.x))
    sav = sav0;
  endif
  ## maybe output iteration status, etc.
endif
endfunction

## ...

fsolve (@user_func, x0, optimset ("OutputFcn", @user_func, ...))
```

See also: [fzero], page 507, [optimset], page 597.

The following is a complete example. To solve the set of equations

$$-2x^2 + 3xy + 4\sin(y) - 6 = 0$$
$$3x^2 - 2xy^2 + 3\cos(x) + 4 = 0$$

you first need to write a function to compute the value of the given function. For example:

```
function y = f (x)
  y = zeros (2, 1);
  y(1) = -2*x(1)^2 + 3*x(1)*x(2)   + 4*sin(x(2)) - 6;
  y(2) =  3*x(1)^2 - 2*x(1)*x(2)^2 + 3*cos(x(1)) + 4;
endfunction
```

Then, call `fsolve` with a specified initial condition to find the roots of the system of equations. For example, given the function `f` defined above,

```
[x, fval, info] = fsolve (@f, [1; 2])
```

results in the solution

```
x =

  0.57983
  2.54621

fval =

  -5.7184e-10
   5.5460e-10

info = 1
```

A value of `info = 1` indicates that the solution has converged.

When no Jacobian is supplied (as in the example above) it is approximated numerically. This requires more function evaluations, and hence is less efficient. In the example above we could compute the Jacobian analytically as

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 3x_2 - 4x_1 & 4\cos(x_2) + 3x_1 \\ -2x_2^2 - 3\sin(x_1) + 6x_1 & -4x_1 x_2 \end{bmatrix}$$

and compute it with the following Octave function

```
function [y, jac] = f (x)
  y = zeros (2, 1);
  y(1) = -2*x(1)^2 + 3*x(1)*x(2)   + 4*sin(x(2)) - 6;
  y(2) =  3*x(1)^2 - 2*x(1)*x(2)^2 + 3*cos(x(1)) + 4;
  if (nargout == 2)
    jac = zeros (2, 2);
    jac(1,1) =  3*x(2) - 4*x(1);
    jac(1,2) =  4*cos(x(2)) + 3*x(1);
    jac(2,1) = -2*x(2)^2 - 3*sin(x(1)) + 6*x(1);
    jac(2,2) = -4*x(1)*x(2);
  endif
endfunction
```

The Jacobian can then be used with the following call to `fsolve`:

```
[x, fval, info] = fsolve (@f, [1; 2], optimset ("jacobian", "on"));
```

which gives the same solution as before.

`fzero (fun, x0)` [Function File]
`fzero (fun, x0, options)` [Function File]
`[x, fval, info, output] = fzero (...)` [Function File]

Find a zero of a univariate function.

fun is a function handle, inline function, or string containing the name of the function to evaluate.

x0 should be a two-element vector specifying two points which bracket a zero. In other words, there must be a change in sign of the function between *x0*(1) and *x0*(2). More mathematically, the following must hold

```
sign (fun(x0(1))) * sign (fun(x0(2))) <= 0
```

If *x0* is a single scalar then several nearby and distant values are probed in an attempt to obtain a valid bracketing. If this is not successful, the function fails.

options is a structure specifying additional options. Currently, `fzero` recognizes these options: `"FunValCheck"`, `"OutputFcn"`, `"TolX"`, `"MaxIter"`, `"MaxFunEvals"`. For a description of these options, see [optimset], page 597.

On exit, the function returns *x*, the approximate zero point and *fval*, the function value thereof.

info is an exit flag that can have these values:

- 1 The algorithm converged to a solution.
- 0 Maximum number of iterations or function evaluations has been reached.
- -1 The algorithm has been terminated from user output function.
- -5 The algorithm may have converged to a singular point.

output is a structure containing runtime information about the `fzero` algorithm. Fields in the structure are:

- iterations Number of iterations through loop.
- nfev Number of function evaluations.
- bracketx A two-element vector with the final bracketing of the zero along the x-axis.
- brackety A two-element vector with the final bracketing of the zero along the y-axis.

See also: [optimset], page 597, [fsolve], page 505.

20.2 Minimizers

Often it is useful to find the minimum value of a function rather than just the zeroes where it crosses the x-axis. `fminbnd` is designed for the simpler, but very common, case of a univariate function where the interval to search is bounded. For unbounded minimization of a function with potentially many variables use `fminunc` or `fminsearch`. The two functions use different internal algorithms and some knowledge of the objective function is required. For functions which can be differentiated, `fminunc` is appropriate. For functions with discontinuities, or for which a gradient search would fail, use `fminsearch`. See Chapter 25 [Optimization], page 585, for minimization with the presence of constraint functions. Note that searches can be made for maxima by simply inverting the objective function ($F_{max} = -F_{min}$).

`[x, fval, info, output] = fminbnd (fun, a, b, options)` [Function File]

Find a minimum point of a univariate function.

fun should be a function handle or name. *a*, *b* specify a starting interval. *options* is a structure specifying additional options. Currently, `fminbnd` recognizes these options: `"FunValCheck"`, `"OutputFcn"`, `"TolX"`, `"MaxIter"`, `"MaxFunEvals"`. For a description of these options, see [optimset], page 597.

On exit, the function returns *x*, the approximate minimum point and *fval*, the function value thereof.

info is an exit flag that can have these values:

- 1 The algorithm converged to a solution.
- 0 Maximum number of iterations or function evaluations has been exhausted.
- -1 The algorithm has been terminated from user output function.

Notes: The search for a minimum is restricted to be in the interval bound by *a* and *b*. If you only have an initial point to begin searching from you will need to use an unconstrained minimization algorithm such as `fminunc` or `fminsearch`. `fminbnd` internally uses a Golden Section search strategy.

See also: [fzero], page 507, [fminunc], page 509, [fminsearch], page 510, [optimset], page 597.

`fminunc (fcn, x0)` [Function File]
`fminunc (fcn, x0, options)` [Function File]
`[x, fval, info, output, grad, hess] = fminunc (fcn, ...)` [Function File]

Solve an unconstrained optimization problem defined by the function *fcn*.

fcn should accept a vector (array) defining the unknown variables, and return the objective function value, optionally with gradient. `fminunc` attempts to determine a vector *x* such that `fcn (x)` is a local minimum.

x0 determines a starting guess. The shape of *x0* is preserved in all calls to *fcn*, but otherwise is treated as a column vector.

options is a structure specifying additional options. Currently, `fminunc` recognizes these options: `"FunValCheck"`, `"OutputFcn"`, `"TolX"`, `"TolFun"`, `"MaxIter"`, `"MaxFunEvals"`, `"GradObj"`, `"FinDiffType"`, `"TypicalX"`, `"AutoScaling"`.

If `"GradObj"` is `"on"`, it specifies that *fcn*, when called with 2 output arguments, also returns the Jacobian matrix of partial first derivatives at the requested point. `TolX` specifies the termination tolerance for the unknown variables *x*, while `TolFun` is a tolerance for the objective function value *fval*. The default is `1e-7` for both options.

For a description of the other options, see `optimset`.

On return, *x* is the location of the minimum and *fval* contains the value of the objective function at *x*.

info may be one of the following values:

1 Converged to a solution point. Relative gradient error is less than specified by `TolFun`.

2 Last relative step size was less than `TolX`.

3 Last relative change in function value was less than `TolFun`.

0 Iteration limit exceeded—either maximum number of algorithm iterations `MaxIter` or maximum number of function evaluations `MaxFunEvals`.

-1 Algorithm terminated by `OutputFcn`.

-3 The trust region radius became excessively small.

Optionally, `fminunc` can return a structure with convergence statistics (*output*), the output gradient (*grad*) at the solution *x*, and approximate Hessian (*hess*) at the solution *x*.

Application Notes: If have only a single nonlinear equation of one variable then using `fminbnd` is usually a better choice.

The algorithm used by `fminsearch` is a gradient search which depends on the objective function being differentiable. If the function has discontinuities it may be better to use a derivative-free algorithm such as `fminsearch`.

See also: [fminbnd], page 508, [fminsearch], page 510, [optimset], page 597.

x = **fminsearch** (*fun*, *x0*) [Function File]
x = **fminsearch** (*fun*, *x0*, *options*) [Function File]
[*x*, *fval*] = **fminsearch** (...) [Function File]

Find a value of *x* which minimizes the function *fun*.

The search begins at the point *x0* and iterates using the Nelder & Mead Simplex algorithm (a derivative-free method). This algorithm is better-suited to functions which have discontinuities or for which a gradient-based search such as `fminunc` fails.

Options for the search are provided in the parameter *options* using the function `optimset`. Currently, `fminsearch` accepts the options: `"TolX"`, `"MaxFunEvals"`, `"MaxIter"`, `"Display"`. For a description of these options, see `optimset`.

On exit, the function returns *x*, the minimum point, and *fval*, the function value thereof.

Example usages:

```
fminsearch (@(x) (x(1)-5).^2+(x(2)-8).^4, [0;0])

fminsearch (inline ("(x(1)-5).^2+(x(2)-8).^4", "x"), [0;0])
```

See also: [fminbnd], page 508, [fminunc], page 509, [optimset], page 597.

21 Diagonal and Permutation Matrices

21.1 Creating and Manipulating Diagonal/Permutation Matrices

A diagonal matrix is defined as a matrix that has zero entries outside the main diagonal; that is, $D_{ij} = 0$ if $i \neq j$ Most often, square diagonal matrices are considered; however, the definition can equally be applied to non-square matrices, in which case we usually speak of a rectangular diagonal matrix.

A permutation matrix is defined as a square matrix that has a single element equal to unity in each row and each column; all other elements are zero. That is, there exists a permutation (vector) p such that $P_{ij} = 1$ if $j = p_i$ and $P_{ij} = 0$ otherwise.

Octave provides special treatment of real and complex rectangular diagonal matrices, as well as permutation matrices. They are stored as special objects, using efficient storage and algorithms, facilitating writing both readable and efficient matrix algebra expressions in the Octave language. The special treatment may be disabled by using the functions *disable_diagonal_matrix* and *disable_permutation_matrix*.

`val = disable_diagonal_matrix ()` [Built-in Function]
`old_val = disable_diagonal_matrix (new_val)` [Built-in Function]
`disable_diagonal_matrix (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether diagonal matrices are stored in a special space-efficient format.

The default value is true. If this option is disabled Octave will store diagonal matrices as full matrices.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [disable_range], page 52, [disable_permutation_matrix], page 511.

`val = disable_permutation_matrix ()` [Built-in Function]
`old_val = disable_permutation_matrix (new_val)` [Built-in Function]
`disable_permutation_matrix (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether permutation matrices are stored in a special space-efficient format.

The default value is true. If this option is disabled Octave will store permutation matrices as full matrices.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [disable_range], page 52, [disable_diagonal_matrix], page 511.

The space savings are significant as demonstrated by the following code.

```
x = diag (rand (10, 1));
xf = full (x);
sizeof (x)
⇒ 80
sizeof (xf)
⇒ 800
```

21.1.1 Creating Diagonal Matrices

The most common and easiest way to create a diagonal matrix is using the built-in function *diag*. The expression `diag (v)`, with *v* a vector, will create a square diagonal matrix with elements on the main diagonal given by the elements of *v*, and size equal to the length of *v*. `diag (v, m, n)` can be used to construct a rectangular diagonal matrix. The result of these expressions will be a special diagonal matrix object, rather than a general matrix object.

Diagonal matrix with unit elements can be created using *eye*. Some other built-in functions can also return diagonal matrices. Examples include *balance* or *inv*.

Example:

```
  diag (1:4)
⇒
Diagonal Matrix

   1   0   0   0
   0   2   0   0
   0   0   3   0
   0   0   0   4

  diag (1:3,5,3)

⇒
Diagonal Matrix

   1   0   0
   0   2   0
   0   0   3
   0   0   0
   0   0   0
```

21.1.2 Creating Permutation Matrices

For creating permutation matrices, Octave does not introduce a new function, but rather overrides an existing syntax: permutation matrices can be conveniently created by indexing an identity matrix by permutation vectors. That is, if *q* is a permutation vector of length *n*, the expression

```
  P = eye (n) (:, q);
```

will create a permutation matrix - a special matrix object.

```
eye (n) (q, :)
```

will also work (and create a row permutation matrix), as well as

```
eye (n) (q1, q2).
```

For example:

```
  eye (4) ([1,3,2,4],:)
⇒
Permutation Matrix

   1   0   0   0
   0   0   1   0
   0   1   0   0
   0   0   0   1

  eye (4) (:,[1,3,2,4])
⇒
Permutation Matrix

   1   0   0   0
   0   0   1   0
   0   1   0   0
   0   0   0   1
```

Mathematically, an identity matrix is both diagonal and permutation matrix. In Octave, `eye (n)` returns a diagonal matrix, because a matrix can only have one class. You can convert this diagonal matrix to a permutation matrix by indexing it by an identity permutation, as shown below. This is a special property of the identity matrix; indexing other diagonal matrices generally produces a full matrix.

```
  eye (3)
⇒
Diagonal Matrix

   1   0   0
   0   1   0
   0   0   1

  eye(3)(1:3,:)
⇒
Permutation Matrix

   1   0   0
   0   1   0
   0   0   1
```

Some other built-in functions can also return permutation matrices. Examples include *inv* or *lu*.

21.1.3 Explicit and Implicit Conversions

The diagonal and permutation matrices are special objects in their own right. A number of operations and built-in functions are defined for these matrices to use special, more efficient code than would be used for a full matrix in the same place. Examples are given in further sections.

To facilitate smooth mixing with full matrices, backward compatibility, and compatibility with MATLAB, the diagonal and permutation matrices should allow any operation that works on full matrices, and will either treat it specially, or implicitly convert themselves to full matrices.

Instances include matrix indexing, except for extracting a single element or a leading submatrix, indexed assignment, or applying most mapper functions, such as *exp*.

An explicit conversion to a full matrix can be requested using the built-in function *full*. It should also be noted that the diagonal and permutation matrix objects will cache the result of the conversion after it is first requested (explicitly or implicitly), so that subsequent conversions will be very cheap.

21.2 Linear Algebra with Diagonal/Permutation Matrices

As has been already said, diagonal and permutation matrices make it possible to use efficient algorithms while preserving natural linear algebra syntax. This section describes in detail the operations that are treated specially when performed on these special matrix objects.

21.2.1 Expressions Involving Diagonal Matrices

Assume D is a diagonal matrix. If M is a full matrix, then `D*M` will scale the rows of M. That means, if `S = D*M`, then for each pair of indices i,j it holds

$$S_{ij} = D_{ii} M_{ij}$$

Similarly, `M*D` will do a column scaling.

The matrix D may also be rectangular, m-by-n where `m != n`. If `m < n`, then the expression `D*M` is equivalent to

```
D(:,1:m) * M(1:m,:),
```

i.e., trailing `n-m` rows of M are ignored. If `m > n`, then `D*M` is equivalent to

```
[D(1:n,n) * M; zeros(m-n, columns (M))],
```

i.e., null rows are appended to the result. The situation for right-multiplication `M*D` is analogous.

The expressions `D \ M` and `M / D` perform inverse scaling. They are equivalent to solving a diagonal (or rectangular diagonal) in a least-squares minimum-norm sense. In exact arithmetic, this is equivalent to multiplying by a pseudoinverse. The pseudoinverse of a rectangular diagonal matrix is again a rectangular diagonal matrix with swapped dimensions, where each nonzero diagonal element is replaced by its reciprocal. The matrix division algorithms do, in fact, use division rather than multiplication by reciprocals for better numerical accuracy; otherwise, they honor the above definition. Note that a diagonal matrix is never truncated due to ill-conditioning; otherwise, it would not be of much use for scaling. This is typically consistent with linear algebra needs. A full matrix that only happens to be diagonal (and is thus not a special object) is of course treated normally.

Multiplication and division by diagonal matrices work efficiently also when combined with sparse matrices, i.e., `D*S`, where D is a diagonal matrix and S is a sparse matrix scales the rows of the sparse matrix and returns a sparse matrix. The expressions `S*D`, `D\S`, `S/D` work analogically.

If *D1* and *D2* are both diagonal matrices, then the expressions

```
D1 + D2
D1 - D2
D1 * D2
D1 / D2
D1 \ D2
```

again produce diagonal matrices, provided that normal dimension matching rules are obeyed. The relations used are same as described above.

Also, a diagonal matrix D can be multiplied or divided by a scalar, or raised to a scalar power if it is square, producing diagonal matrix result in all cases.

A diagonal matrix can also be transposed or conjugate-transposed, giving the expected result. Extracting a leading submatrix of a diagonal matrix, i.e., `D(1:m,1:n)`, will produce a diagonal matrix, other indexing expressions will implicitly convert to full matrix.

Adding a diagonal matrix to a full matrix only operates on the diagonal elements. Thus,

```
A = A + eps * eye (n)
```

is an efficient method of augmenting the diagonal of a matrix. Subtraction works analogically.

When involved in expressions with other element-by-element operators, `.*`, `./`, `.\` or `.^`, an implicit conversion to full matrix will take place. This is not always strictly necessary but chosen to facilitate better consistency with MATLAB.

21.2.2 Expressions Involving Permutation Matrices

If P is a permutation matrix and M a matrix, the expression `P*M` will permute the rows of M. Similarly, `M*P` will yield a column permutation. Matrix division `P\M` and `M/P` can be used to do inverse permutation.

The previously described syntax for creating permutation matrices can actually help an user to understand the connection between a permutation matrix and a permuting vector. Namely, the following holds, where `I = eye (n)` is an identity matrix:

```
I(p,:) * M = (I*M) (p,:) = M(p,:)
```

Similarly,

```
M * I(:,p) = (M*I) (:,p) = M(:,p)
```

The expressions `I(p,:)` and `I(:,p)` are permutation matrices.

A permutation matrix can be transposed (or conjugate-transposed, which is the same, because a permutation matrix is never complex), inverting the permutation, or equivalently, turning a row-permutation matrix into a column-permutation one. For permutation matrices, transpose is equivalent to inversion, thus `P\M` is equivalent to `P'*M`. Transpose of a permutation matrix (or inverse) is a constant-time operation, flipping only a flag internally, and thus the choice between the two above equivalent expressions for inverse permuting is completely up to the user's taste.

Multiplication and division by permutation matrices works efficiently also when combined with sparse matrices, i.e., `P*S`, where P is a permutation matrix and S is a sparse matrix permutes the rows of the sparse matrix and returns a sparse matrix. The expressions `S*P`, `P\S`, `S/P` work analogically.

Two permutation matrices can be multiplied or divided (if their sizes match), performing a composition of permutations. Also a permutation matrix can be indexed by a permutation vector (or two vectors), giving again a permutation matrix. Any other operations do not generally yield a permutation matrix and will thus trigger the implicit conversion.

21.3 Functions That Are Aware of These Matrices

This section lists the built-in functions that are aware of diagonal and permutation matrices on input, or can return them as output. Passed to other functions, these matrices will in general trigger an implicit conversion. (Of course, user-defined dynamically linked functions may also work with diagonal or permutation matrices).

21.3.1 Diagonal Matrix Functions

inv and *pinv* can be applied to a diagonal matrix, yielding again a diagonal matrix. *det* will use an efficient straightforward calculation when given a diagonal matrix, as well as *cond*. The following mapper functions can be applied to a diagonal matrix without converting it to a full one: *abs*, *real*, *imag*, *conj*, *sqrt*. A diagonal matrix can also be returned from the *balance* and *svd* functions. The *sparse* function will convert a diagonal matrix efficiently to a sparse matrix.

21.3.2 Permutation Matrix Functions

inv and *pinv* will invert a permutation matrix, preserving its specialness. *det* can be applied to a permutation matrix, efficiently calculating the sign of the permutation (which is equal to the determinant).

A permutation matrix can also be returned from the built-in functions *lu* and *qr*, if a pivoted factorization is requested.

The *sparse* function will convert a permutation matrix efficiently to a sparse matrix. The *find* function will also work efficiently with a permutation matrix, making it possible to conveniently obtain the permutation indices.

21.4 Examples of Usage

The following can be used to solve a linear system `A*x = b` using the pivoted LU factorization:

```
[L, U, P] = lu (A); ## now L*U = P*A
x = U \ L \ P*b;
```

This is one way to normalize columns of a matrix X to unit norm:

```
s = norm (X, "columns");
X /= diag (s);
```

The same can also be accomplished with broadcasting (see Section 19.2 [Broadcasting], page 491):

```
s = norm (X, "columns");
X ./= s;
```

The following expression is a way to efficiently calculate the sign of a permutation, given by a permutation vector *p*. It will also work in earlier versions of Octave, but slowly.

```
det (eye (length (p))(p, :))
```

Finally, here's how to solve a linear system `A*x = b` with Tikhonov regularization (ridge regression) using SVD (a skeleton only):

```
m = rows (A); n = columns (A);
[U, S, V] = svd (A);
## determine the regularization factor alpha
## alpha = ...
## transform to orthogonal basis
b = U'*b;
## Use the standard formula, replacing A with S.
## S is diagonal, so the following will be very fast and accurate.
x = (S'*S + alpha^2 * eye (n)) \ (S' * b);
## transform to solution basis
x = V*x;
```

21.5 Differences in Treatment of Zero Elements

Making diagonal and permutation matrices special matrix objects in their own right and the consequent usage of smarter algorithms for certain operations implies, as a side effect, small differences in treating zeros. The contents of this section apply also to sparse matrices, discussed in the following chapter. (see Chapter 22 [Sparse Matrices], page 519)

The IEEE floating point standard defines the result of the expressions `0*Inf` and `0*NaN` as `NaN`. This is widely agreed to be a good compromise. Numerical software dealing with structured and sparse matrices (including Octave) however, almost always makes a distinction between a "numerical zero" and an "assumed zero". A "numerical zero" is a zero value occurring in a place where any floating-point value could occur. It is normally stored somewhere in memory as an explicit value. An "assumed zero", on the contrary, is a zero matrix element implied by the matrix structure (diagonal, triangular) or a sparsity pattern; its value is usually not stored explicitly anywhere, but is implied by the underlying data structure.

The primary distinction is that an assumed zero, when multiplied by any number, or divided by any nonzero number, yields *always* a zero, even when, e.g., multiplied by `Inf` or divided by `NaN`. The reason for this behavior is that the numerical multiplication is not actually performed anywhere by the underlying algorithm; the result is just assumed to be zero. Equivalently, one can say that the part of the computation involving assumed zeros is performed symbolically, not numerically.

This behavior not only facilitates the most straightforward and efficient implementation of algorithms, but also preserves certain useful invariants, like:

- scalar * diagonal matrix is a diagonal matrix
- sparse matrix / scalar preserves the sparsity pattern
- permutation matrix * matrix is equivalent to permuting rows

all of these natural mathematical truths would be invalidated by treating assumed zeros as numerical ones.

Note that MATLAB does not strictly follow this principle and converts assumed zeros to numerical zeros in certain cases, while not doing so in other cases. As of today, there are no intentions to mimic such behavior in Octave.

Examples of effects of assumed zeros vs. numerical zeros:

```
Inf * eye (3)
⇒
   Inf     0     0
     0   Inf     0
     0     0   Inf

Inf * speye (3)
⇒
Compressed Column Sparse (rows = 3, cols = 3, nnz = 3 [33%])

  (1, 1) -> Inf
  (2, 2) -> Inf
  (3, 3) -> Inf

Inf * full (eye (3))
⇒
   Inf   NaN   NaN
   NaN   Inf   NaN
   NaN   NaN   Inf

diag (1:3) * [NaN; 1; 1]
⇒
   NaN
     2
     3

sparse (1:3,1:3,1:3) * [NaN; 1; 1]
⇒
   NaN
     2
     3
[1,0,0;0,2,0;0,0,3] * [NaN; 1; 1]
⇒
   NaN
   NaN
   NaN
```

22 Sparse Matrices

22.1 Creation and Manipulation of Sparse Matrices

The size of mathematical problems that can be treated at any particular time is generally limited by the available computing resources. Both, the speed of the computer and its available memory place limitation on the problem size.

There are many classes of mathematical problems which give rise to matrices, where a large number of the elements are zero. In this case it makes sense to have a special matrix type to handle this class of problems where only the nonzero elements of the matrix are stored. Not only does this reduce the amount of memory to store the matrix, but it also means that operations on this type of matrix can take advantage of the a priori knowledge of the positions of the nonzero elements to accelerate their calculations.

A matrix type that stores only the nonzero elements is generally called sparse. It is the purpose of this document to discuss the basics of the storage and creation of sparse matrices and the fundamental operations on them.

22.1.1 Storage of Sparse Matrices

It is not strictly speaking necessary for the user to understand how sparse matrices are stored. However, such an understanding will help to get an understanding of the size of sparse matrices. Understanding the storage technique is also necessary for those users wishing to create their own oct-files.

There are many different means of storing sparse matrix data. What all of the methods have in common is that they attempt to reduce the complexity and storage given a priori knowledge of the particular class of problems that will be solved. A good summary of the available techniques for storing sparse matrix is given by Saad[1]. With full matrices, knowledge of the point of an element of the matrix within the matrix is implied by its position in the computers memory. However, this is not the case for sparse matrices, and so the positions of the nonzero elements of the matrix must equally be stored.

An obvious way to do this is by storing the elements of the matrix as triplets, with two elements being their position in the array (rows and column) and the third being the data itself. This is conceptually easy to grasp, but requires more storage than is strictly needed.

The storage technique used within Octave is the compressed column format. It is similar to the Yale format.[2] In this format the position of each element in a row and the data are stored as previously. However, if we assume that all elements in the same column are stored adjacent in the computers memory, then we only need to store information on the number of nonzero elements in each column, rather than their positions. Thus assuming that the matrix has more nonzero elements than there are columns in the matrix, we win in terms of the amount of memory used.

In fact, the column index contains one more element than the number of columns, with the first element always being zero. The advantage of this is a simplification in the code, in

[1] Y. Saad "SPARSKIT: A basic toolkit for sparse matrix computation", 1994, `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/paper.ps`

[2] `http://en.wikipedia.org/wiki/Sparse_matrix#Yale_format`

that there is no special case for the first or last columns. A short example, demonstrating this in C is.

```
for (j = 0; j < nc; j++)
  for (i = cidx(j); i < cidx(j+1); i++)
     printf ("nonzero element (%i,%i) is %d\n",
         ridx(i), j, data(i));
```

A clear understanding might be had by considering an example of how the above applies to an example matrix. Consider the matrix

```
1   2   0  0
0   0   0  3
0   0   0  4
```

The nonzero elements of this matrix are

```
(1, 1)  ⇒ 1
(1, 2)  ⇒ 2
(2, 4)  ⇒ 3
(3, 4)  ⇒ 4
```

This will be stored as three vectors *cidx*, *ridx* and *data*, representing the column indexing, row indexing and data respectively. The contents of these three vectors for the above matrix will be

```
cidx = [0, 1, 2, 2, 4]
ridx = [0, 0, 1, 2]
data = [1, 2, 3, 4]
```

Note that this is the representation of these elements with the first row and column assumed to start at zero, while in Octave itself the row and column indexing starts at one. Thus the number of elements in the *i*-th column is given by `cidx (i + 1) - cidx (i)`.

Although Octave uses a compressed column format, it should be noted that compressed row formats are equally possible. However, in the context of mixed operations between mixed sparse and dense matrices, it makes sense that the elements of the sparse matrices are in the same order as the dense matrices. Octave stores dense matrices in column major ordering, and so sparse matrices are equally stored in this manner.

A further constraint on the sparse matrix storage used by Octave is that all elements in the rows are stored in increasing order of their row index, which makes certain operations faster. However, it imposes the need to sort the elements on the creation of sparse matrices. Having disordered elements is potentially an advantage in that it makes operations such as concatenating two sparse matrices together easier and faster, however it adds complexity and speed problems elsewhere.

22.1.2 Creating Sparse Matrices

There are several means to create sparse matrix.

Returned from a function
: There are many functions that directly return sparse matrices. These include *speye*, *sprand*, *diag*, etc.

Constructed from matrices or vectors
: The function *sparse* allows a sparse matrix to be constructed from three vectors representing the row, column and data. Alternatively, the function *spconvert* uses a three column matrix format to allow easy importation of data from elsewhere.

Created and then filled
: The function *sparse* or *spalloc* can be used to create an empty matrix that is then filled by the user

From a user binary program
: The user can directly create the sparse matrix within an oct-file.

There are several basic functions to return specific sparse matrices. For example the sparse identity matrix, is a matrix that is often needed. It therefore has its own function to create it as `speye (n)` or `speye (r, c)`, which creates an n-by-n or r-by-c sparse identity matrix.

Another typical sparse matrix that is often needed is a random distribution of random elements. The functions *sprand* and *sprandn* perform this for uniform and normal random distributions of elements. They have exactly the same calling convention, where `sprand (r, c, d)`, creates an r-by-c sparse matrix with a density of filled elements of d.

Other functions of interest that directly create sparse matrices, are *diag* or its generalization *spdiags*, that can take the definition of the diagonals of the matrix and create the sparse matrix that corresponds to this. For example,

```
s = diag (sparse (randn (1,n)), -1);
```

creates a sparse (n+1)-by-(n+1) sparse matrix with a single diagonal defined.

`B = spdiags (A)` [Function File]
`[B, d] = spdiags (A)` [Function File]
`B = spdiags (A, d)` [Function File]
`A = spdiags (v, d, A)` [Function File]
`A = spdiags (v, d, m, n)` [Function File]

A generalization of the function `diag`.

Called with a single input argument, the nonzero diagonals d of A are extracted.

With two arguments the diagonals to extract are given by the vector d.

The other two forms of `spdiags` modify the input matrix by replacing the diagonals. They use the columns of v to replace the diagonals represented by the vector d. If the sparse matrix A is defined then the diagonals of this matrix are replaced. Otherwise a matrix of m by n is created with the diagonals given by the columns of v.

Negative values of d represent diagonals below the main diagonal, and positive values of d diagonals above the main diagonal.

For example:

```
spdiags (reshape (1:12, 4, 3), [-1 0 1], 5, 4)
   ⇒ 5 10  0  0
      1  6 11  0
      0  2  7 12
      0  0  3  8
      0  0  0  4
```

See also: [diag], page 418.

s = speye (*m*, *n*) [Function File]
s = speye (*m*) [Function File]
s = speye (*sz*) [Function File]

Return a sparse identity matrix of size *m*x*n*.

The implementation is significantly more efficient than `sparse (eye (m))` as the full matrix is not constructed.

Called with a single argument a square matrix of size *m*-by-*m* is created. If called with a single vector argument *sz*, this argument is taken to be the size of the matrix to create.

See also: [sparse], page 524, [spdiags], page 521, [eye], page 418.

r = spones (*S*) [Function File]

Replace the nonzero entries of *S* with ones.

This creates a sparse matrix with the same structure as *S*.

See also: [sparse], page 524, [sprand], page 522, [sprandn], page 522, [sprandsym], page 523, [spfun], page 496, [spy], page 528.

sprand (*m*, *n*, *d*) [Function File]
sprand (*m*, *n*, *d*, *rc*) [Function File]
sprand (*s*) [Function File]

Generate a sparse matrix with uniformly distributed random values.

The size of the matrix is *m*x*n* with a density of values *d*. *d* must be between 0 and 1. Values will be uniformly distributed on the interval (0, 1).

If called with a single matrix argument, a sparse matrix is generated with random values wherever the matrix *s* is nonzero.

If called with a scalar fourth argument *rc*, a random sparse matrix with reciprocal condition number *rc* is generated. If *rc* is a vector, then it specifies the first singular values of the generated matrix (`length (rc) <= min (m, n)`).

See also: [sprandn], page 522, [sprandsym], page 523, [rand], page 421.

sprandn (*m*, *n*, *d*) [Function File]
sprandn (*m*, *n*, *d*, *rc*) [Function File]
sprandn (*s*) [Function File]

Generate a sparse matrix with normally distributed random values.

The size of the matrix is *m*x*n* with a density of values *d*. *d* must be between 0 and 1. Values will be normally distributed with a mean of 0 and a variance of 1.

If called with a single matrix argument, a sparse matrix is generated with random values wherever the matrix *s* is nonzero.

If called with a scalar fourth argument *rc*, a random sparse matrix with reciprocal condition number *rc* is generated. If *rc* is a vector, then it specifies the first singular values of the generated matrix (`length (rc) <= min (m, n)`).

See also: [sprand], page 522, [sprandsym], page 523, [randn], page 422.

`sprandsym` (*n*, *d*) [Function File]
`sprandsym` (*s*) [Function File]

Generate a symmetric random sparse matrix.

The size of the matrix will be *n*x*n*, with a density of values given by *d*. *d* must be between 0 and 1 inclusive. Values will be normally distributed with a mean of zero and a variance of 1.

If called with a single matrix argument, a random sparse matrix is generated wherever the matrix *s* is nonzero in its lower triangular part.

See also: [sprand], page 522, [sprandn], page 522, [spones], page 522, [sparse], page 524.

The recommended way for the user to create a sparse matrix, is to create two vectors containing the row and column index of the data and a third vector of the same size containing the data to be stored. For example,

```
ri = ci = d = [];
for j = 1:c
  ri = [ri; randperm(r,n)'];
  ci = [ci; j*ones(n,1)];
  d = [d; rand(n,1)];
endfor
s = sparse (ri, ci, d, r, c);
```

creates an *r*-by-*c* sparse matrix with a random distribution of *n* (<*r*) elements per column. The elements of the vectors do not need to be sorted in any particular order as Octave will sort them prior to storing the data. However, pre-sorting the data will make the creation of the sparse matrix faster.

The function *spconvert* takes a three or four column real matrix. The first two columns represent the row and column index respectively and the third and four columns, the real and imaginary parts of the sparse matrix. The matrix can contain zero elements and the elements can be sorted in any order. Adding zero elements is a convenient way to define the size of the sparse matrix. For example:

```
s = spconvert ([1 2 3 4; 1 3 4 4; 1 2 3 0]')
⇒ Compressed Column Sparse (rows=4, cols=4, nnz=3)
      (1 , 1) -> 1
      (2 , 3) -> 2
      (3 , 4) -> 3
```

An example of creating and filling a matrix might be

```
k = 5;
nz = r * k;
s = spalloc (r, c, nz)
for j = 1:c
  idx = randperm (r);
  s (:, j) = [zeros(r - k, 1); ...
        rand(k, 1)] (idx);
endfor
```

It should be noted, that due to the way that the Octave assignment functions are written that the assignment will reallocate the memory used by the sparse matrix at each iteration of the above loop. Therefore the *spalloc* function ignores the *nz* argument and does not pre-assign the memory for the matrix. Therefore, it is vitally important that code using to above structure should be vectorized as much as possible to minimize the number of assignments and reduce the number of memory allocations.

FM = `full` (*SM*) [Built-in Function]

Return a full storage matrix from a sparse, diagonal, or permutation matrix, or a range.

See also: [sparse], page 524, [issparse], page 526.

s = `spalloc` (*m*, *n*, *nz*) [Built-in Function]

Create an *m*-by-*n* sparse matrix with pre-allocated space for at most *nz* nonzero elements.

This is useful for building a matrix incrementally by a sequence of indexed assignments. Subsequent indexed assignments after `spalloc` will reuse the pre-allocated memory, provided they are of one of the simple forms

- `s(I:J) = x`
- `s(:,I:J) = x`
- `s(K:L,I:J) = x`

and that the following conditions are met:

- the assignment does not decrease nnz (*S*).
- after the assignment, nnz (*S*) does not exceed *nz*.
- no index is out of bounds.

Partial movement of data may still occur, but in general the assignment will be more memory and time efficient under these circumstances. In particular, it is possible to efficiently build a pre-allocated sparse matrix from a contiguous block of columns.

The amount of pre-allocated memory for a given matrix may be queried using the function `nzmax`.

See also: [nzmax], page 526, [sparse], page 524.

s = `sparse` (*a*) [Built-in Function]
s = `sparse` (*i*, *j*, *sv*, *m*, *n*) [Built-in Function]
s = `sparse` (*i*, *j*, *sv*) [Built-in Function]
s = `sparse` (*m*, *n*) [Built-in Function]
s = `sparse` (*i*, *j*, *s*, *m*, *n*, "*unique*") [Built-in Function]
s = `sparse` (*i*, *j*, *sv*, *m*, *n*, *nzmax*) [Built-in Function]

Create a sparse matrix from a full matrix, or row, column, value triplets.

If *a* is a full matrix, convert it to a sparse matrix representation, removing all zero values in the process.

Given the integer index vectors *i* and *j*, and a 1-by-`nnz` vector of real or complex values *sv*, construct the sparse matrix `S(i(k),j(k)) = sv(k)` with overall dimensions *m* and *n*. If any of *sv*, *i* or *j* are scalars, they are expanded to have a common size.

If *m* or *n* are not specified their values are derived from the maximum index in the vectors *i* and *j* as given by `m = max (i)`, `n = max (j)`.

Note: if multiple values are specified with the same *i*, *j* indices, the corresponding value in *s* will be the sum of the values at the repeated location. See `accumarray` for an example of how to produce different behavior, such as taking the minimum instead.

If the option `"unique"` is given, and more than one value is specified at the same *i*, *j* indices, then the last specified value will be used.

`sparse (m, n)` will create an empty *m*x*n* sparse matrix and is equivalent to `sparse ([], [], [], m, n)`

The argument `nzmax` is ignored but accepted for compatibility with MATLAB.

Example 1 (sum at repeated indices):

```
i = [1 1 2]; j = [1 1 2]; sv = [3 4 5];
sparse (i, j, sv, 3, 4)
⇒
Compressed Column Sparse (rows = 3, cols = 4, nnz = 2 [17%])

  (1, 1) ->  7
  (2, 2) ->  5
```

Example 2 ("unique" option):

```
i = [1 1 2]; j = [1 1 2]; sv = [3 4 5];
sparse (i, j, sv, 3, 4, "unique")
⇒
Compressed Column Sparse (rows = 3, cols = 4, nnz = 2 [17%])

  (1, 1) ->  4
  (2, 2) ->  5
```

See also: [full], page 524, [accumarray], page 499, [spalloc], page 524, [spdiags], page 521, [speye], page 522, [spones], page 522, [sprand], page 522, [sprandn], page 522, [sprandsym], page 523, [spconvert], page 525, [spfun], page 496.

`x = spconvert (m)` [Function File]

Convert a simple sparse matrix format easily generated by other programs into Octave's internal sparse format.

The input *m* is either a 3 or 4 column real matrix, containing the row, column, real, and imaginary parts of the elements of the sparse matrix. An element with a zero real and imaginary part can be used to force a particular matrix size.

See also: [sparse], page 524.

The above problem of memory reallocation can be avoided in oct-files. However, the construction of a sparse matrix from an oct-file is more complex than can be discussed here. See Appendix A [External Code Interface], page 817, for a full description of the techniques involved.

22.1.3 Finding Information about Sparse Matrices

There are a number of functions that allow information concerning sparse matrices to be obtained. The most basic of these is *issparse* that identifies whether a particular Octave object is in fact a sparse matrix.

Another very basic function is *nnz* that returns the number of nonzero entries there are in a sparse matrix, while the function *nzmax* returns the amount of storage allocated to the sparse matrix. Note that Octave tends to crop unused memory at the first opportunity for sparse objects. There are some cases of user created sparse objects where the value returned by *nzmax* will not be the same as *nnz*, but in general they will give the same result. The function *spstats* returns some basic statistics on the columns of a sparse matrix including the number of elements, the mean and the variance of each column.

`issparse (x)` [Built-in Function]

Return true if *x* is a sparse matrix.

See also: [ismatrix], page 63.

`n = nnz (a)` [Built-in Function]

Return the number of nonzero elements in *a*.

See also: [nzmax], page 526, [nonzeros], page 526, [find], page 407.

`nonzeros (s)` [Function File]

Return a vector of the nonzero values of the sparse matrix *s*.

See also: [find], page 407, [nnz], page 526.

`n = nzmax (SM)` [Built-in Function]

Return the amount of storage allocated to the sparse matrix *SM*.

Note that Octave tends to crop unused memory at the first opportunity for sparse objects. Thus, in general the value of `nzmax` will be the same as `nnz` except for some cases of user-created sparse objects.

See also: [nnz], page 526, [spalloc], page 524, [sparse], page 524.

`[count, mean, var] = spstats (S)` [Function File]
`[count, mean, var] = spstats (S, j)` [Function File]

Return the stats for the nonzero elements of the sparse matrix *S*.

count is the number of nonzeros in each column, *mean* is the mean of the nonzeros in each column, and *var* is the variance of the nonzeros in each column.

Called with two input arguments, if *S* is the data and *j* is the bin number for the data, compute the stats for each bin. In this case, bins can contain data values of zero, whereas with `spstats (S)` the zeros may disappear.

When solving linear equations involving sparse matrices Octave determines the means to solve the equation based on the type of the matrix (see Section 22.2 [Sparse Linear Algebra], page 540). Octave probes the matrix type when the div (/) or ldiv (\) operator is first used with the matrix and then caches the type. However the *matrix_type* function can be used to determine the type of the sparse matrix prior to use of the div or ldiv operators. For example,

```
a = tril (sprandn (1024, 1024, 0.02), -1) ...
    + speye (1024);
matrix_type (a);
ans = Lower
```

shows that Octave correctly determines the matrix type for lower triangular matrices. *matrix_type* can also be used to force the type of a matrix to be a particular type. For example:

```
a = matrix_type (tril (sprandn (1024, ...
   1024, 0.02), -1) + speye (1024), "Lower");
```

This allows the cost of determining the matrix type to be avoided. However, incorrectly defining the matrix type will result in incorrect results from solutions of linear equations, and so it is entirely the responsibility of the user to correctly identify the matrix type

There are several graphical means of finding out information about sparse matrices. The first is the *spy* command, which displays the structure of the nonzero elements of the matrix. See Figure 22.1, for an example of the use of *spy*. More advanced graphical information can be obtained with the *treeplot*, *etreeplot* and *gplot* commands.

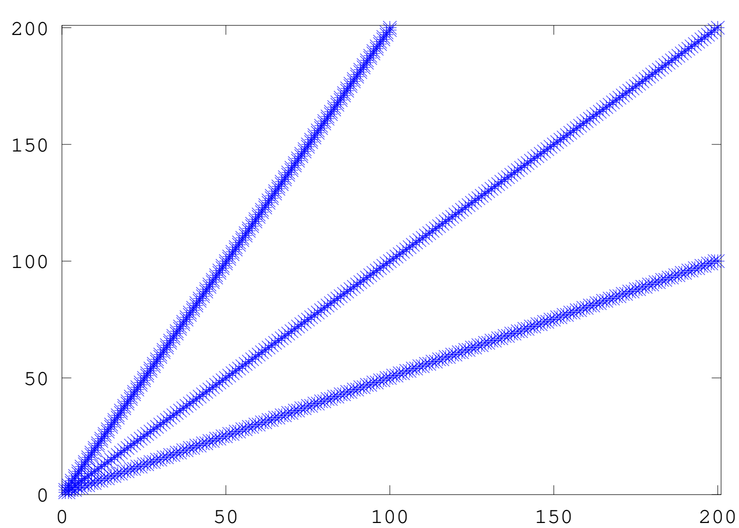

Figure 22.1: Structure of simple sparse matrix.

One use of sparse matrices is in graph theory, where the interconnections between nodes are represented as an adjacency matrix. That is, if the i-th node in a graph is connected to the j-th node. Then the ij-th node (and in the case of undirected graphs the ji-th node) of the sparse adjacency matrix is nonzero. If each node is then associated with a set of coordinates, then the *gplot* command can be used to graphically display the interconnections between nodes.

As a trivial example of the use of *gplot* consider the example,

```
A = sparse ([2,6,1,3,2,4,3,5,4,6,1,5],
    [1,1,2,2,3,3,4,4,5,5,6,6],1,6,6);
xy = [0,4,8,6,4,2;5,0,5,7,5,7]';
gplot (A,xy)
```

which creates an adjacency matrix `A` where node 1 is connected to nodes 2 and 6, node 2 with nodes 1 and 3, etc. The coordinates of the nodes are given in the n-by-2 matrix `xy`. See Figure 22.2.

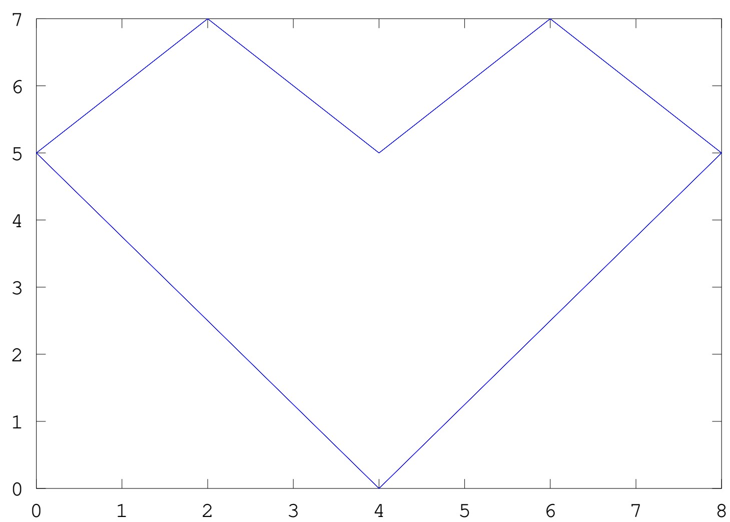

Figure 22.2: Simple use of the *gplot* command.

The dependencies between the nodes of a Cholesky factorization can be calculated in linear time without explicitly needing to calculate the Cholesky factorization by the `etree` command. This command returns the elimination tree of the matrix and can be displayed graphically by the command `treeplot (etree (A))` if `A` is symmetric or `treeplot (etree (A+A'))` otherwise.

`spy` (*x*) [Function File]
`spy` (..., *markersize*) [Function File]
`spy` (..., *line_spec*) [Function File]

Plot the sparsity pattern of the sparse matrix *x*.

If the argument *markersize* is given as a scalar value, it is used to determine the point size in the plot.

If the string *line_spec* is given it is passed to `plot` and determines the appearance of the plot.

See also: [plot], page 272, [gplot], page 529.

p = `etree` (*S*) [Loadable Function]
p = `etree` (*S*, *typ*) [Loadable Function]
[*p*, *q*] = `etree` (*S*, *typ*) [Loadable Function]

Return the elimination tree for the matrix *S*.

By default *S* is assumed to be symmetric and the symmetric elimination tree is returned. The argument *typ* controls whether a symmetric or column elimination tree is returned. Valid values of *typ* are `"sym"` or `"col"`, for symmetric or column elimination tree respectively.

Called with a second argument, `etree` also returns the postorder permutations on the tree.

`etreeplot (A)` [Function File]
`etreeplot (A, node_style, edge_style)` [Function File]

Plot the elimination tree of the matrix *A* or `A+A'` if *A* in not symmetric.

The optional parameters *node_style* and *edge_style* define the output style.

See also: [treeplot], page 529, [gplot], page 529.

`gplot (A, xy)` [Function File]
`gplot (A, xy, line_style)` [Function File]
`[x, y] = gplot (A, xy)` [Function File]

Plot a graph defined by *A* and *xy* in the graph theory sense.

A is the adjacency matrix of the array to be plotted and *xy* is an *n*-by-2 matrix containing the coordinates of the nodes of the graph.

The optional parameter *line_style* defines the output style for the plot. Called with no output arguments the graph is plotted directly. Otherwise, return the coordinates of the plot in *x* and *y*.

See also: [treeplot], page 529, [etreeplot], page 529, [spy], page 528.

`treeplot (tree)` [Function File]
`treeplot (tree, node_style, edge_style)` [Function File]

Produce a graph of tree or forest.

The first argument is vector of predecessors.

The optional parameters *node_style* and *edge_style* define the output plot style.

The complexity of the algorithm is O(n) in terms of is time and memory requirements.

See also: [etreeplot], page 529, [gplot], page 529.

`treelayout (tree)` [Function File]
`treelayout (tree, permutation)` [Function File]

treelayout lays out a tree or a forest.

The first argument *tree* is a vector of predecessors.

The parameter *permutation* is an optional postorder permutation.

The complexity of the algorithm is O(n) in terms of time and memory requirements.

See also: [etreeplot], page 529, [gplot], page 529, [treeplot], page 529.

22.1.4 Basic Operators and Functions on Sparse Matrices

22.1.4.1 Sparse Functions

Many Octave functions have been overloaded to work with either sparse or full matrices. There is no difference in calling convention when using an overloaded function with a sparse matrix, however, there is also no access to potentially sparse-specific features. At any time the sparse matrix specific version of a function can be used by explicitly calling its function name.

The table below lists all of the sparse functions of Octave. Note that the names of the specific sparse forms of the functions are typically the same as the general versions with a *sp* prefix. In the table below, and in the rest of this article, the specific sparse versions of functions are used.

Generate sparse matrices:
: *spalloc*, *spdiags*, *speye*, *sprand*, *sprandn*, *sprandsym*

Sparse matrix conversion:
: *full*, *sparse*, *spconvert*

Manipulate sparse matrices
: *issparse*, *nnz*, *nonzeros*, *nzmax*, *spfun*, *spones*, *spy*

Graph Theory:
: *etree*, *etreeplot*, *gplot*, *treeplot*

Sparse matrix reordering:
: *amd*, *ccolamd*, *colamd*, *colperm*, *csymamd*, *dmperm*, *symamd*, *randperm*, *symrcm*

Linear algebra:
: *condest*, *eigs*, *matrix_type*, *normest*, *sprank*, *spaugment*, *svds*

Iterative techniques:
: *ichol*, *ilu*, *pcg*, *pcr*

Miscellaneous:
: *spparms*, *symbfact*, *spstats*

In addition all of the standard Octave mapper functions (i.e., basic math functions that take a single argument) such as *abs*, etc. can accept sparse matrices. The reader is referred to the documentation supplied with these functions within Octave itself for further details.

22.1.4.2 Return Types of Operators and Functions

The two basic reasons to use sparse matrices are to reduce the memory usage and to not have to do calculations on zero elements. The two are closely related in that the computation time on a sparse matrix operator or function is roughly linear with the number of nonzero elements.

Therefore, there is a certain density of nonzero elements of a matrix where it no longer makes sense to store it as a sparse matrix, but rather as a full matrix. For this reason operators and functions that have a high probability of returning a full matrix will always return one. For example adding a scalar constant to a sparse matrix will almost always make it a full matrix, and so the example,

```
speye (3) + 0
⇒   1  0  0
  0  1  0
  0  0  1
```

returns a full matrix as can be seen.

Additionally, if `sparse_auto_mutate` is true, all sparse functions test the amount of memory occupied by the sparse matrix to see if the amount of storage used is larger than the amount used by the full equivalent. Therefore `speye (2) * 1` will return a full matrix as the memory used is smaller for the full version than the sparse version.

As all of the mixed operators and functions between full and sparse matrices exist, in general this does not cause any problems. However, one area where it does cause a problem is where a sparse matrix is promoted to a full matrix, where subsequent operations would resparsify the matrix. Such cases are rare, but can be artificially created, for example `(fliplr (speye (3)) + speye (3)) - speye (3)` gives a full matrix when it should give a sparse one. In general, where such cases occur, they impose only a small memory penalty.

There is however one known case where this behavior of Octave's sparse matrices will cause a problem. That is in the handling of the *diag* function. Whether *diag* returns a sparse or full matrix depending on the type of its input arguments. So

```
a = diag (sparse ([1,2,3]), -1);
```

should return a sparse matrix. To ensure this actually happens, the *sparse* function, and other functions based on it like *speye*, always returns a sparse matrix, even if the memory used will be larger than its full representation.

val = `sparse_auto_mutate` () [Built-in Function]
old_val = `sparse_auto_mutate` (*new_val*) [Built-in Function]
`sparse_auto_mutate` (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that controls whether Octave will automatically mutate sparse matrices to full matrices to save memory.

For example:

```
s = speye (3);
sparse_auto_mutate (false);
s(:, 1) = 1;
typeinfo (s)
⇒ sparse matrix
sparse_auto_mutate (true);
s(1, :) = 1;
typeinfo (s)
⇒ matrix
```

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

Note that the `sparse_auto_mutate` option is incompatible with MATLAB, and so it is off by default.

22.1.4.3 Mathematical Considerations

The attempt has been made to make sparse matrices behave in exactly the same manner as there full counterparts. However, there are certain differences and especially differences with other products sparse implementations.

First, the "./" and ".^" operators must be used with care. Consider what the examples

```
s = speye (4);
a1 = s .^ 2;
a2 = s .^ s;
a3 = s .^ -2;
a4 = s ./ 2;
a5 = 2 ./ s;
a6 = s ./ s;
```

will give. The first example of *s* raised to the power of 2 causes no problems. However *s* raised element-wise to itself involves a large number of terms `0 .^ 0` which is 1. There `s .^ s` is a full matrix.

Likewise `s .^ -2` involves terms like `0 .^ -2` which is infinity, and so `s .^ -2` is equally a full matrix.

For the "./" operator `s ./ 2` has no problems, but `2 ./ s` involves a large number of infinity terms as well and is equally a full matrix. The case of `s ./ s` involves terms like `0 ./ 0` which is a `NaN` and so this is equally a full matrix with the zero elements of *s* filled with `NaN` values.

The above behavior is consistent with full matrices, but is not consistent with sparse implementations in other products.

A particular problem of sparse matrices comes about due to the fact that as the zeros are not stored, the sign-bit of these zeros is equally not stored. In certain cases the sign-bit of zero is important. For example:

```
a = 0 ./ [-1, 1; 1, -1];
b = 1 ./ a
⇒ -Inf            Inf
    Inf           -Inf
c = 1 ./ sparse (a)
⇒  Inf            Inf
    Inf            Inf
```

To correct this behavior would mean that zero elements with a negative sign-bit would need to be stored in the matrix to ensure that their sign-bit was respected. This is not done at this time, for reasons of efficiency, and so the user is warned that calculations where the sign-bit of zero is important must not be done using sparse matrices.

In general any function or operator used on a sparse matrix will result in a sparse matrix with the same or a larger number of nonzero elements than the original matrix. This is particularly true for the important case of sparse matrix factorizations. The usual way to address this is to reorder the matrix, such that its factorization is sparser than the factorization of the original matrix. That is the factorization of `L * U = P * S * Q` has sparser terms `L` and `U` than the equivalent factorization `L * U = S`.

Several functions are available to reorder depending on the type of the matrix to be factorized. If the matrix is symmetric positive-definite, then *symamd* or *csymamd* should be used. Otherwise *amd*, *colamd* or *ccolamd* should be used. For completeness the reordering functions *colperm* and *randperm* are also available.

See Figure 22.3, for an example of the structure of a simple positive definite matrix.

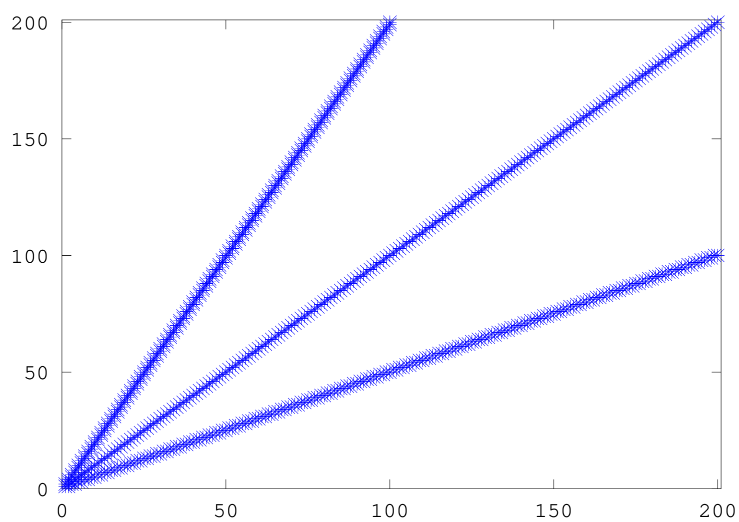

Figure 22.3: Structure of simple sparse matrix.

The standard Cholesky factorization of this matrix can be obtained by the same command that would be used for a full matrix. This can be visualized with the command `r = chol (A); spy (r);`. See Figure 22.4. The original matrix had 598 nonzero terms, while this Cholesky factorization has 10200, with only half of the symmetric matrix being stored. This is a significant level of fill in, and although not an issue for such a small test case, can represents a large overhead in working with other sparse matrices.

The appropriate sparsity preserving permutation of the original matrix is given by *symamd* and the factorization using this reordering can be visualized using the command `q = symamd (A); r = chol (A(q,q)); spy (r)`. This gives 399 nonzero terms which is a significant improvement.

The Cholesky factorization itself can be used to determine the appropriate sparsity preserving reordering of the matrix during the factorization, In that case this might be obtained with three return arguments as `[r, p, q] = chol (A); spy (r)`.

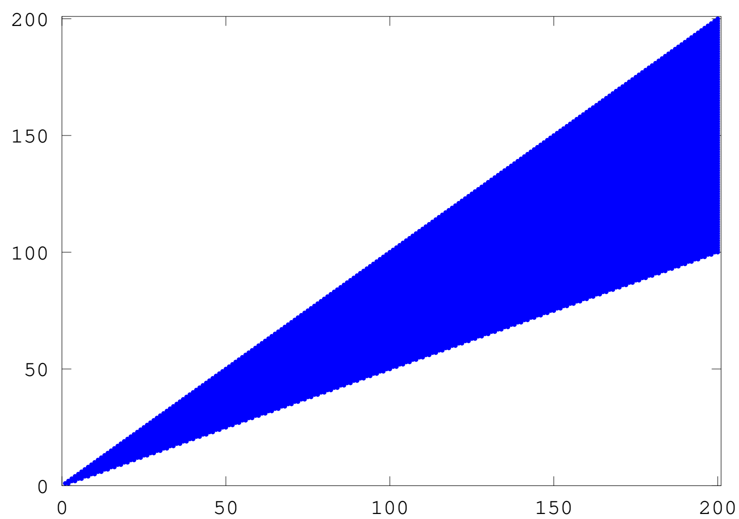

Figure 22.4: Structure of the unpermuted Cholesky factorization of the above matrix.

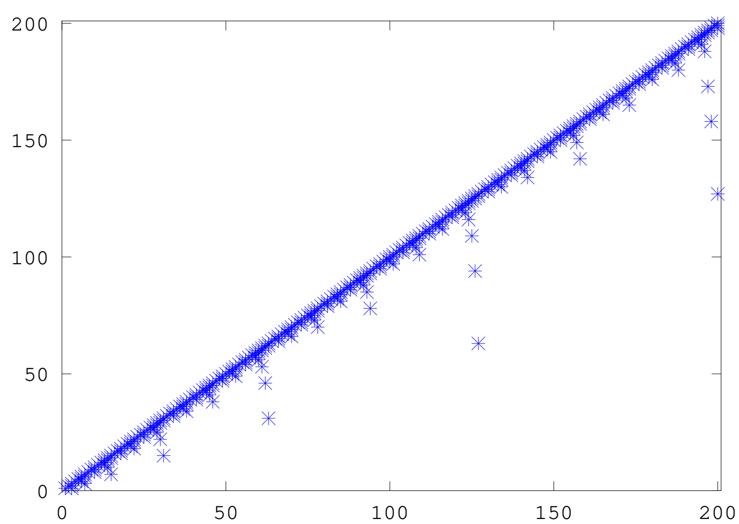

Figure 22.5: Structure of the permuted Cholesky factorization of the above matrix.

In the case of an asymmetric matrix, the appropriate sparsity preserving permutation is *colamd* and the factorization using this reordering can be visualized using the command `q = colamd (A); [l, u, p] = lu (A(:,q)); spy (l+u)`.

Finally, Octave implicitly reorders the matrix when using the div (/) and ldiv (\) operators, and so no the user does not need to explicitly reorder the matrix to maximize performance.

p = amd (*S*) [Loadable Function]
p = amd (*S*, *opts*) [Loadable Function]

Return the approximate minimum degree permutation of a matrix.

This is a permutation such that the Cholesky factorization of `S (p, p)` tends to be sparser than the Cholesky factorization of *S* itself. `amd` is typically faster than `symamd` but serves a similar purpose.

The optional parameter *opts* is a structure that controls the behavior of `amd`. The fields of the structure are

opts.dense
: Determines what `amd` considers to be a dense row or column of the input matrix. Rows or columns with more than `max (16, (dense * sqrt (n)))` entries, where *n* is the order of the matrix *S*, are ignored by `amd` during the calculation of the permutation. The value of dense must be a positive scalar and the default value is 10.0

opts.aggressive
: If this value is a nonzero scalar, then `amd` performs aggressive absorption. The default is not to perform aggressive absorption.

The author of the code itself is Timothy A. Davis `davis@cise.ufl.edu`, University of Florida (see `http://www.cise.ufl.edu/research/sparse/amd`).

See also: [symamd], page 538, [colamd], page 536.

p = ccolamd (*S*) [Loadable Function]
p = ccolamd (*S*, *knobs*) [Loadable Function]
p = ccolamd (*S*, *knobs*, *cmember*) [Loadable Function]
[*p*, *stats*] = ccolamd (...) [Loadable Function]

Constrained column approximate minimum degree permutation.

`p = ccolamd (S)` returns the column approximate minimum degree permutation vector for the sparse matrix *S*. For a non-symmetric matrix *S*, `S(:, p)` tends to have sparser LU factors than *S*. `chol (S(:, p)' * S(:, p))` also tends to be sparser than `chol (S' * S)`. `p = ccolamd (S, 1)` optimizes the ordering for `lu (S(:, p))`. The ordering is followed by a column elimination tree post-ordering.

knobs is an optional 1-element to 5-element input vector, with a default value of `[0 10 10 1 0]` if not present or empty. Entries not present are set to their defaults.

`knobs(1)`
: if nonzero, the ordering is optimized for `lu (S(:, p))`. It will be a poor ordering for `chol (S(:, p)' * S(:, p))`. This is the most important knob for ccolamd.

`knobs(2)`
: if *S* is m-by-n, rows with more than `max (16, knobs(2) * sqrt (n))` entries are ignored.

`knobs(3)`
: columns with more than `max (16, knobs(3) * sqrt (min (m, n)))` entries are ignored and ordered last in the output permutation (subject to the cmember constraints).

`knobs(4)`
: if nonzero, aggressive absorption is performed.

`knobs(5)`
: if nonzero, statistics and knobs are printed.

cmember is an optional vector of length n. It defines the constraints on the column ordering. If `cmember(j) = c`, then column j is in constraint set c (c must be in the range 1 to n). In the output permutation p, all columns in set 1 appear first, followed by all columns in set 2, and so on. `cmember = ones (1,n)` if not present or empty. `ccolamd (S, [], 1 : n)` returns `1 : n`

`p = ccolamd (S)` is about the same as `p = colamd (S)`. *knobs* and its default values differ. `colamd` always does aggressive absorption, and it finds an ordering suitable for both `lu (S(:, p))` and `chol (S(:, p)' * S(:, p))`; it cannot optimize its ordering for `lu (S(:, p))` to the extent that `ccolamd (S, 1)` can.

stats is an optional 20-element output vector that provides data about the ordering and the validity of the input matrix *S*. Ordering statistics are in `stats(1 : 3)`. `stats(1)` and `stats(2)` are the number of dense or empty rows and columns ignored by CCOLAMD and `stats(3)` is the number of garbage collections performed on the internal data structure used by CCOLAMD (roughly of size `2.2 * nnz (S) + 4 * m + 7 * n` integers).

`stats(4 : 7)` provide information if CCOLAMD was able to continue. The matrix is OK if `stats(4)` is zero, or 1 if invalid. `stats(5)` is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. `stats(6)` is the last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or zero if no such row index exists. `stats(7)` is the number of duplicate or out-of-order row indices. `stats(8 : 20)` is always zero in the current version of CCOLAMD (reserved for future use).

The authors of the code itself are S. Larimore, T. Davis (Univ. of Florida) and S. Rajamanickam in collaboration with J. Bilbert and E. Ng. Supported by the National Science Foundation (DMS-9504974, DMS-9803599, CCR-0203270), and a grant from Sandia National Lab. See `http://www.cise.ufl.edu/research/sparse` for ccolamd, csymamd, amd, colamd, symamd, and other related orderings.

See also: [colamd], page 536, [csymamd], page 537.

`p = colamd (S)` [Loadable Function]
`p = colamd (S, knobs)` [Loadable Function]
`[p, stats] = colamd (S)` [Loadable Function]
`[p, stats] = colamd (S, knobs)` [Loadable Function]

Compute the column approximate minimum degree permutation.

`p = colamd (S)` returns the column approximate minimum degree permutation vector for the sparse matrix *S*. For a non-symmetric matrix *S*, `S(:,p)` tends to have sparser LU factors than *S*. The Cholesky factorization of `S(:,p)' * S(:,p)` also tends to be sparser than that of `S' * S`.

knobs is an optional one- to three-element input vector. If *S* is m-by-n, then rows with more than `max(16,knobs(1)*sqrt(n))` entries are ignored. Columns with more than `max (16,knobs(2)*sqrt(min(m,n)))` entries are removed prior to ordering, and ordered last in the output permutation p. Only completely dense rows or columns are removed if `knobs(1)` and `knobs(2)` are < 0, respectively. If `knobs(3)` is nonzero, *stats* and *knobs* are printed. The default is `knobs = [10 10 0]`. Note that *knobs* differs from earlier versions of colamd.

stats is an optional 20-element output vector that provides data about the ordering and the validity of the input matrix *S*. Ordering statistics are in `stats(1:3)`. `stats(1)` and `stats(2)` are the number of dense or empty rows and columns ignored by COLAMD and `stats(3)` is the number of garbage collections performed on the internal data structure used by COLAMD (roughly of size `2.2 * nnz(S) + 4 * m + 7 * n` integers).

Octave built-in functions are intended to generate valid sparse matrices, with no duplicate entries, with ascending row indices of the nonzeros in each column, with a non-negative number of entries in each column (!) and so on. If a matrix is invalid, then COLAMD may or may not be able to continue. If there are duplicate entries (a row index appears two or more times in the same column) or if the row indices in a column are out of order, then COLAMD can correct these errors by ignoring the duplicate entries and sorting each column of its internal copy of the matrix *S* (the input matrix *S* is not repaired, however). If a matrix is invalid in other ways then COLAMD cannot continue, an error message is printed, and no output arguments (*p* or *stats*) are returned. COLAMD is thus a simple way to check a sparse matrix to see if it's valid.

`stats(4:7)` provide information if COLAMD was able to continue. The matrix is OK if `stats(4)` is zero, or 1 if invalid. `stats(5)` is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. `stats(6)` is the last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or zero if no such row index exists. `stats(7)` is the number of duplicate or out-of-order row indices. `stats(8:20)` is always zero in the current version of COLAMD (reserved for future use).

The ordering is followed by a column elimination tree post-ordering.

The authors of the code itself are Stefan I. Larimore and Timothy A. Davis `davis@cise.ufl.edu`, University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. (see `http://www.cise.ufl.edu/research/sparse/colamd`)

See also: [colperm], page 537, [symamd], page 538, [ccolamd], page 535.

`p = colperm (s)` [Function File]

Return the column permutations such that the columns of `s (:, p)` are ordered in terms of increasing number of nonzero elements.

If *s* is symmetric, then *p* is chosen such that `s (p, p)` orders the rows and columns with increasing number of nonzeros elements.

`p = csymamd (S)` [Loadable Function]
`p = csymamd (S, knobs)` [Loadable Function]
`p = csymamd (S, knobs, cmember)` [Loadable Function]
`[p, stats] = csymamd (...)` [Loadable Function]

For a symmetric positive definite matrix *S*, return the permutation vector *p* such that `S(p,p)` tends to have a sparser Cholesky factor than *S*.

Sometimes `csymamd` works well for symmetric indefinite matrices too. The matrix *S* is assumed to be symmetric; only the strictly lower triangular part is referenced. *S* must be square. The ordering is followed by an elimination tree post-ordering.

knobs is an optional 1-element to 3-element input vector, with a default value of `[10 1 0]`. Entries not present are set to their defaults.

knobs(1) If *S* is n-by-n, then rows and columns with more than `max(16,knobs(1)*sqrt(n))` entries are ignored, and ordered last in the output permutation (subject to the cmember constraints).

knobs(2) If nonzero, aggressive absorption is performed.

knobs(3) If nonzero, statistics and knobs are printed.

cmember is an optional vector of length n. It defines the constraints on the ordering. If `cmember(j) = S`, then row/column j is in constraint set *c* (*c* must be in the range 1 to n). In the output permutation *p*, rows/columns in set 1 appear first, followed by all rows/columns in set 2, and so on. `cmember = ones (1,n)` if not present or empty. `csymamd (S,[],1:n)` returns `1:n`.

`p = csymamd (S)` is about the same as `p = symamd (S)`. *knobs* and its default values differ.

`stats(4:7)` provide information if CCOLAMD was able to continue. The matrix is OK if `stats(4)` is zero, or 1 if invalid. `stats(5)` is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. `stats(6)` is the last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or zero if no such row index exists. `stats(7)` is the number of duplicate or out-of-order row indices. `stats(8:20)` is always zero in the current version of CCOLAMD (reserved for future use).

The authors of the code itself are S. Larimore, T. Davis (Univ. of Florida) and S. Rajamanickam in collaboration with J. Bilbert and E. Ng. Supported by the National Science Foundation (DMS-9504974, DMS-9803599, CCR-0203270), and a grant from Sandia National Lab. See `http://www.cise.ufl.edu/research/sparse` for ccolamd, csymamd, amd, colamd, symamd, and other related orderings.

See also: [symamd], page 538, [ccolamd], page 535.

`p = dmperm (S)` [Loadable Function]
`[p, q, r, S] = dmperm (S)` [Loadable Function]

Perform a Dulmage-Mendelsohn permutation of the sparse matrix *S*.

With a single output argument `dmperm` performs the row permutations *p* such that `S(p,:)` has no zero elements on the diagonal.

Called with two or more output arguments, returns the row and column permutations, such that `S(p, q)` is in block triangular form. The values of *r* and *S* define the boundaries of the blocks. If *S* is square then `r == S`.

The method used is described in: A. Pothen & C.-J. Fan. *Computing the Block Triangular Form of a Sparse Matrix*. ACM Trans. Math. Software, 16(4):303-324, 1990.

See also: [colamd], page 536, [ccolamd], page 535.

`p = symamd (S)` [Loadable Function]
`p = symamd (S, knobs)` [Loadable Function]
`[p, stats] = symamd (S)` [Loadable Function]

`[p, stats] = symamd (S, knobs)` [Loadable Function]

For a symmetric positive definite matrix *S*, returns the permutation vector p such that `S(p, p)` tends to have a sparser Cholesky factor than *S*.

Sometimes `symamd` works well for symmetric indefinite matrices too. The matrix *S* is assumed to be symmetric; only the strictly lower triangular part is referenced. *S* must be square.

knobs is an optional one- to two-element input vector. If *S* is n-by-n, then rows and columns with more than `max (16,knobs(1)*sqrt(n))` entries are removed prior to ordering, and ordered last in the output permutation *p*. No rows/columns are removed if `knobs(1) < 0`. If `knobs (2)` is nonzero, `stats` and *knobs* are printed. The default is `knobs = [10 0]`. Note that *knobs* differs from earlier versions of `symamd`.

stats is an optional 20-element output vector that provides data about the ordering and the validity of the input matrix *S*. Ordering statistics are in `stats(1:3)`. `stats(1) = stats(2)` is the number of dense or empty rows and columns ignored by SYMAMD and `stats(3)` is the number of garbage collections performed on the internal data structure used by SYMAMD (roughly of size `8.4 * nnz (tril (S, -1)) + 9 * n` integers).

Octave built-in functions are intended to generate valid sparse matrices, with no duplicate entries, with ascending row indices of the nonzeros in each column, with a non-negative number of entries in each column (!) and so on. If a matrix is invalid, then SYMAMD may or may not be able to continue. If there are duplicate entries (a row index appears two or more times in the same column) or if the row indices in a column are out of order, then SYMAMD can correct these errors by ignoring the duplicate entries and sorting each column of its internal copy of the matrix S (the input matrix S is not repaired, however). If a matrix is invalid in other ways then SYMAMD cannot continue, an error message is printed, and no output arguments (*p* or *stats*) are returned. SYMAMD is thus a simple way to check a sparse matrix to see if it's valid.

`stats(4:7)` provide information if SYMAMD was able to continue. The matrix is OK if `stats (4)` is zero, or 1 if invalid. `stats(5)` is the rightmost column index that is unsorted or contains duplicate entries, or zero if no such column exists. `stats(6)` is the last seen duplicate or out-of-order row index in the column index given by `stats(5)`, or zero if no such row index exists. `stats(7)` is the number of duplicate or out-of-order row indices. `stats(8:20)` is always zero in the current version of SYMAMD (reserved for future use).

The ordering is followed by a column elimination tree post-ordering.

The authors of the code itself are Stefan I. Larimore and Timothy A. Davis `davis@cise.ufl.edu`, University of Florida. The algorithm was developed in collaboration with John Gilbert, Xerox PARC, and Esmond Ng, Oak Ridge National Laboratory. (see `http://www.cise.ufl.edu/research/sparse/colamd`)

See also: [colperm], page 537, [colamd], page 536.

`p = symrcm (S)` [Loadable Function]

Return the symmetric reverse Cuthill-McKee permutation of *S*.

p is a permutation vector such that `S(p, p)` tends to have its diagonal elements closer to the diagonal than *S*. This is a good preordering for LU or Cholesky factorization of matrices that come from "long, skinny" problems. It works for both symmetric and asymmetric *S*.

The algorithm represents a heuristic approach to the NP-complete bandwidth minimization problem. The implementation is based in the descriptions found in

E. Cuthill, J. McKee. *Reducing the Bandwidth of Sparse Symmetric Matrices*. Proceedings of the 24th ACM National Conference, 157–172 1969, Brandon Press, New Jersey.

A. George, J.W.H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall Series in Computational Mathematics, ISBN 0-13-165274-5, 1981.

See also: [colperm], page 537, [colamd], page 536, [symamd], page 538.

22.2 Linear Algebra on Sparse Matrices

Octave includes a polymorphic solver for sparse matrices, where the exact solver used to factorize the matrix, depends on the properties of the sparse matrix itself. Generally, the cost of determining the matrix type is small relative to the cost of factorizing the matrix itself, but in any case the matrix type is cached once it is calculated, so that it is not re-determined each time it is used in a linear equation.

The selection tree for how the linear equation is solve is

1. If the matrix is diagonal, solve directly and goto 8
2. If the matrix is a permuted diagonal, solve directly taking into account the permutations. Goto 8
3. If the matrix is square, banded and if the band density is less than that given by `spparms ("bandden")` continue, else goto 4.
   a. If the matrix is tridiagonal and the right-hand side is not sparse continue, else goto 3b.
      1. If the matrix is Hermitian, with a positive real diagonal, attempt Cholesky factorization using LAPACK xPTSV.
      2. If the above failed or the matrix is not Hermitian with a positive real diagonal use Gaussian elimination with pivoting using LAPACK xGTSV, and goto 8.

   b. If the matrix is Hermitian with a positive real diagonal, attempt Cholesky factorization using LAPACK xPBTRF.
   c. if the above failed or the matrix is not Hermitian with a positive real diagonal use Gaussian elimination with pivoting using LAPACK xGBTRF, and goto 8.
4. If the matrix is upper or lower triangular perform a sparse forward or backward substitution, and goto 8
5. If the matrix is an upper triangular matrix with column permutations or lower triangular matrix with row permutations, perform a sparse forward or backward substitution, and goto 8
6. If the matrix is square, Hermitian with a real positive diagonal, attempt sparse Cholesky factorization using CHOLMOD.

7. If the sparse Cholesky factorization failed or the matrix is not Hermitian with a real positive diagonal, and the matrix is square, factorize using UMFPACK.
8. If the matrix is not square, or any of the previous solvers flags a singular or near singular matrix, find a minimum norm solution using CXSPARSE[3].

The band density is defined as the number of nonzero values in the band divided by the total number of values in the full band. The banded matrix solvers can be entirely disabled by using *spparms* to set `bandden` to 1 (i.e., `spparms ("bandden", 1)`).

The QR solver factorizes the problem with a Dulmage-Mendelsohn decomposition, to separate the problem into blocks that can be treated as over-determined, multiple well determined blocks, and a final over-determined block. For matrices with blocks of strongly connected nodes this is a big win as LU decomposition can be used for many blocks. It also significantly improves the chance of finding a solution to over-determined problems rather than just returning a vector of *NaN*'s.

All of the solvers above, can calculate an estimate of the condition number. This can be used to detect numerical stability problems in the solution and force a minimum norm solution to be used. However, for narrow banded, triangular or diagonal matrices, the cost of calculating the condition number is significant, and can in fact exceed the cost of factoring the matrix. Therefore the condition number is not calculated in these cases, and Octave relies on simpler techniques to detect singular matrices or the underlying LAPACK code in the case of banded matrices.

The user can force the type of the matrix with the `matrix_type` function. This overcomes the cost of discovering the type of the matrix. However, it should be noted that identifying the type of the matrix incorrectly will lead to unpredictable results, and so `matrix_type` should be used with care.

`n = normest (A)` [Function File]
`n = normest (A, tol)` [Function File]
`[n, c] = normest (...)` [Function File]

Estimate the 2-norm of the matrix *A* using a power series analysis.

This is typically used for large matrices, where the cost of calculating `norm (A)` is prohibitive and an approximation to the 2-norm is acceptable.

tol is the tolerance to which the 2-norm is calculated. By default *tol* is 1e-6.

The optional output *c* returns the number of iterations needed for `normest` to converge.

`[est, v, w, iter] = onenormest (A, t)` [Function File]
`[est, v, w, iter] = onenormest (apply, apply_t, n, t)` [Function File]

Apply Higham and Tisseur's randomized block 1-norm estimator to matrix *A* using *t* test vectors.

If *t* exceeds 5, then only 5 test vectors are used.

If the matrix is not explicit, e.g., when estimating the norm of `inv (A)` given an LU factorization, `onenormest` applies *A* and its conjugate transpose through a pair

[3] The CHOLMOD, UMFPACK and CXSPARSE packages were written by Tim Davis and are available at `http://www.cise.ufl.edu/research/sparse/`

of functions *apply* and *apply_t*, respectively, to a dense matrix of size *n* by *t*. The implicit version requires an explicit dimension *n*.

Returns the norm estimate *est*, two vectors *v* and *w* related by `norm (w, 1) = est * norm (v, 1)`, and the number of iterations *iter*. The number of iterations is limited to 10 and is at least 2.

References:

- N.J. Higham and F. Tisseur, *A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra*. SIMAX vol 21, no 4, pp 1185-1201. `http://dx.doi.org/10.1137/S0895479899356080`
- N.J. Higham and F. Tisseur, *A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra*. `http://citeseer.ist.psu.edu/223007.html`

See also: [condest], page 542, [norm], page 470, [cond], page 466.

`condest (A)` [Function File]
`condest (A, t)` [Function File]
`[est, v] = condest (...)` [Function File]
`[est, v] = condest (A, solve, solve_t, t)` [Function File]
`[est, v] = condest (apply, apply_t, solve, solve_t, n, t)` [Function File]

Estimate the 1-norm condition number of a matrix *A* using *t* test vectors using a randomized 1-norm estimator.

If *t* exceeds 5, then only 5 test vectors are used.

If the matrix is not explicit, e.g., when estimating the condition number of *A* given an LU factorization, `condest` uses the following functions:

apply `A*x` for a matrix `x` of size *n* by *t*.

apply_t `A'*x` for a matrix `x` of size *n* by *t*.

solve `A \ b` for a matrix `b` of size *n* by *t*.

solve_t `A' \ b` for a matrix `b` of size *n* by *t*.

The implicit version requires an explicit dimension *n*.

`condest` uses a randomized algorithm to approximate the 1-norms.

`condest` returns the 1-norm condition estimate *est* and a vector *v* satisfying `norm (A*v, 1) == norm (A, 1) * norm (v, 1) / est`. When *est* is large, *v* is an approximate null vector.

References:

- N.J. Higham and F. Tisseur, *A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra*. SIMAX vol 21, no 4, pp 1185-1201. `http://dx.doi.org/10.1137/S0895479899356080`
- N.J. Higham and F. Tisseur, *A Block Algorithm for Matrix 1-Norm Estimation, with an Application to 1-Norm Pseudospectra*. `http://citeseer.ist.psu.edu/223007.html`

See also: [cond], page 466, [norm], page 470, [onenormest], page 541.

spparms () [Built-in Function]
vals = spparms () [Built-in Function]
[*keys*, *vals*] = spparms () [Built-in Function]
val = spparms (*key*) [Built-in Function]
spparms (*vals*) [Built-in Function]
spparms ("*default*") [Built-in Function]
spparms ("*tight*") [Built-in Function]
spparms (*key*, *val*) [Built-in Function]

Query or set the parameters used by the sparse solvers and factorization functions.

The first four calls above get information about the current settings, while the others change the current settings. The parameters are stored as pairs of keys and values, where the values are all floats and the keys are one of the following strings:

'spumoni' Printing level of debugging information of the solvers (default 0)

'ths_rel' Included for compatibility. Not used. (default 1)

'ths_abs' Included for compatibility. Not used. (default 1)

'exact_d' Included for compatibility. Not used. (default 0)

'supernd' Included for compatibility. Not used. (default 3)

'rreduce' Included for compatibility. Not used. (default 3)

'wh_frac' Included for compatibility. Not used. (default 0.5)

'autommd' Flag whether the LU/QR and the '\' and '/' operators will automatically use the sparsity preserving mmd functions (default 1)

'autoamd' Flag whether the LU and the '\' and '/' operators will automatically use the sparsity preserving amd functions (default 1)

'piv_tol' The pivot tolerance of the UMFPACK solvers (default 0.1)

'sym_tol' The pivot tolerance of the UMFPACK symmetric solvers (default 0.001)

'bandden' The density of nonzero elements in a banded matrix before it is treated by the LAPACK banded solvers (default 0.5)

'umfpack' Flag whether the UMFPACK or mmd solvers are used for the LU, '\' and '/' operations (default 1)

The value of individual keys can be set with `spparms (key, val)`. The default values can be restored with the special keyword `"default"`. The special keyword `"tight"` can be used to set the mmd solvers to attempt a sparser solution at the potential cost of longer running time.

See also: [chol], page 472, [colamd], page 536, [lu], page 475, [qr], page 476, [symamd], page 538.

p = sprank (*S*) [Loadable Function]

Calculate the structural rank of the sparse matrix *S*.

Note that only the structure of the matrix is used in this calculation based on a Dulmage-Mendelsohn permutation to block triangular form. As such the numerical rank of the matrix S is bounded by `sprank (S) >= rank (S)`. Ignoring floating point errors `sprank (S) == rank (S)`.

See also: [dmperm], page 538.

`[count, h, parent, post, r] = symbfact (S)` [Loadable Function]
`[...] = symbfact (S, typ)` [Loadable Function]
`[...] = symbfact (S, typ, mode)` [Loadable Function]

Perform a symbolic factorization analysis on the sparse matrix S.

The input variables are

S S is a complex or real sparse matrix.

typ Is the type of the factorization and can be one of

- 'sym' Factorize S. This is the default.
- 'col' Factorize `S' * S`.
- 'row' Factorize `S * S'`.
- 'lo' Factorize `S'`

mode The default is to return the Cholesky factorization for *r*, and if *mode* is `'L'`, the conjugate transpose of the Cholesky factorization is returned. The conjugate transpose version is faster and uses less memory, but returns the same values for *count*, *h*, *parent* and *post* outputs.

The output variables are

count The row counts of the Cholesky factorization as determined by *typ*.

h The height of the elimination tree.

parent The elimination tree itself.

post A sparse boolean matrix whose structure is that of the Cholesky factorization as determined by *typ*.

For non square matrices, the user can also utilize the `spaugment` function to find a least squares solution to a linear equation.

`s = spaugment (A, c)` [Function File]

Create the augmented matrix of A.

This is given by

```
[c * eye(m, m), A;
            A', zeros(n, n)]
```

This is related to the least squares solution of `A \ b`, by

```
s * [ r / c; x] = [ b, zeros(n, columns(b)) ]
```

where r is the residual error

```
r = b - A * x
```

As the matrix s is symmetric indefinite it can be factorized with `lu`, and the minimum norm solution can therefore be found without the need for a `qr` factorization. As the residual error will be `zeros (m, m)` for underdetermined problems, and example can be

```
m = 11; n = 10; mn = max (m, n);
A = spdiags ([ones(mn,1), 10*ones(mn,1), -ones(mn,1)],
             [-1, 0, 1], m, n);
x0 = A \ ones (m,1);
s = spaugment (A);
[L, U, P, Q] = lu (s);
x1 = Q * (U \ (L \ (P  * [ones(m,1); zeros(n,1)])));
x1 = x1(end - n + 1 : end);
```

To find the solution of an overdetermined problem needs an estimate of the residual error r and so it is more complex to formulate a minimum norm solution using the `spaugment` function.

In general the left division operator is more stable and faster than using the `spaugment` function.

See also: [mldivide], page 143.

Finally, the function `eigs` can be used to calculate a limited number of eigenvalues and eigenvectors based on a selection criteria and likewise for `svds` which calculates a limited number of singular values and vectors.

d = `eigs` (*A*) [Function File]
d = `eigs` (*A*, *k*) [Function File]
d = `eigs` (*A*, *k*, *sigma*) [Function File]
d = `eigs` (*A*, *k*, *sigma*, *opts*) [Function File]
d = `eigs` (*A*, *B*) [Function File]
d = `eigs` (*A*, *B*, *k*) [Function File]
d = `eigs` (*A*, *B*, *k*, *sigma*) [Function File]
d = `eigs` (*A*, *B*, *k*, *sigma*, *opts*) [Function File]
d = `eigs` (*af*, *n*) [Function File]
d = `eigs` (*af*, *n*, *B*) [Function File]
d = `eigs` (*af*, *n*, *k*) [Function File]
d = `eigs` (*af*, *n*, *B*, *k*) [Function File]
d = `eigs` (*af*, *n*, *k*, *sigma*) [Function File]
d = `eigs` (*af*, *n*, *B*, *k*, *sigma*) [Function File]
d = `eigs` (*af*, *n*, *k*, *sigma*, *opts*) [Function File]
d = `eigs` (*af*, *n*, *B*, *k*, *sigma*, *opts*) [Function File]
[*V*, *d*] = `eigs` (*A*, ...) [Function File]
[*V*, *d*] = `eigs` (*af*, *n*, ...) [Function File]
[*V*, *d*, *flag*] = `eigs` (*A*, ...) [Function File]
[*V*, *d*, *flag*] = `eigs` (*af*, *n*, ...) [Function File]

Calculate a limited number of eigenvalues and eigenvectors of A, based on a selection criteria.

The number of eigenvalues and eigenvectors to calculate is given by k and defaults to 6.

By default, `eigs` solve the equation $A\nu = \lambda\nu$, where λ is a scalar representing one of the eigenvalues, and ν is the corresponding eigenvector. If given the positive definite matrix B then `eigs` solves the general eigenvalue equation $A\nu = \lambda B\nu$.

The argument *sigma* determines which eigenvalues are returned. *sigma* can be either a scalar or a string. When *sigma* is a scalar, the k eigenvalues closest to *sigma* are returned. If *sigma* is a string, it must have one of the following values.

`"lm"` Largest Magnitude (default).

`"sm"` Smallest Magnitude.

`"la"` Largest Algebraic (valid only for real symmetric problems).

`"sa"` Smallest Algebraic (valid only for real symmetric problems).

`"be"` Both Ends, with one more from the high-end if k is odd (valid only for real symmetric problems).

`"lr"` Largest Real part (valid only for complex or unsymmetric problems).

`"sr"` Smallest Real part (valid only for complex or unsymmetric problems).

`"li"` Largest Imaginary part (valid only for complex or unsymmetric problems).

`"si"` Smallest Imaginary part (valid only for complex or unsymmetric problems).

If *opts* is given, it is a structure defining possible options that `eigs` should use. The fields of the *opts* structure are:

`issym` If *af* is given, then flags whether the function *af* defines a symmetric problem. It is ignored if A is given. The default is false.

`isreal` If *af* is given, then flags whether the function *af* defines a real problem. It is ignored if A is given. The default is true.

`tol` Defines the required convergence tolerance, calculated as `tol * norm (A)`. The default is `eps`.

`maxit` The maximum number of iterations. The default is 300.

`p` The number of Lanzcos basis vectors to use. More vectors will result in faster convergence, but a greater use of memory. The optimal value of `p` is problem dependent and should be in the range k to n. The default value is `2 * k`.

`v0` The starting vector for the algorithm. An initial vector close to the final vector will speed up convergence. The default is for ARPACK to randomly generate a starting vector. If specified, `v0` must be an n-by-1 vector where `n = rows (A)`

`disp` The level of diagnostic printout (0|1|2). If `disp` is 0 then diagnostics are disabled. The default value is 0.

`cholB` Flag if `chol (B)` is passed rather than B. The default is false.

`permB` The permutation vector of the Cholesky factorization of B if `cholB` is true. That is `chol (B(permB, permB))`. The default is `1:n`.

It is also possible to represent A by a function denoted *af*. *af* must be followed by a scalar argument n defining the length of the vector argument accepted by *af*. *af* can be a function handle, an inline function, or a string. When *af* is a string it holds the name of the function to use.

af is a function of the form `y = af (x)` where the required return value of *af* is determined by the value of *sigma*. The four possible forms are

`A * x` if *sigma* is not given or is a string other than "sm".

`A \ x` if *sigma* is 0 or "sm".

`(A - sigma * I) \ x`
for the standard eigenvalue problem, where `I` is the identity matrix of the same size as A.

`(A - sigma * B) \ x`
for the general eigenvalue problem.

The return arguments of `eigs` depend on the number of return arguments requested. With a single return argument, a vector d of length k is returned containing the k eigenvalues that have been found. With two return arguments, V is a n-by-k matrix whose columns are the k eigenvectors corresponding to the returned eigenvalues. The eigenvalues themselves are returned in d in the form of a n-by-k matrix, where the elements on the diagonal are the eigenvalues.

Given a third return argument *flag*, `eigs` returns the status of the convergence. If *flag* is 0 then all eigenvalues have converged. Any other value indicates a failure to converge.

This function is based on the ARPACK package, written by R. Lehoucq, K. Maschhoff, D. Sorensen, and C. Yang. For more information see `http://www.caam.rice.edu/software/ARPACK/`.

See also: [eig], page 467, [svds], page 547.

`s = svds (A)` [Function File]
`s = svds (A, k)` [Function File]
`s = svds (A, k, sigma)` [Function File]
`s = svds (A, k, sigma, opts)` [Function File]
`[u, s, v] = svds (...)` [Function File]
`[u, s, v, flag] = svds (...)` [Function File]

Find a few singular values of the matrix A.

The singular values are calculated using

```
[m, n] = size (A);
s = eigs ([sparse(m, m), A;
                    A', sparse(n, n)])
```

The eigenvalues returned by `eigs` correspond to the singular values of A. The number of singular values to calculate is given by k and defaults to 6.

The argument *sigma* specifies which singular values to find. When *sigma* is the string `'L'`, the default, the largest singular values of A are found. Otherwise, *sigma* must be a real scalar and the singular values closest to *sigma* are found. As a corollary, `sigma = 0` finds the smallest singular values. Note that for relatively small values of *sigma*, there is a chance that the requested number of singular values will not be found. In that case *sigma* should be increased.

opts is a structure defining options that `svds` will pass to `eigs`. The possible fields of this structure are documented in `eigs`. By default, `svds` sets the following three fields:

`tol` The required convergence tolerance for the singular values. The default value is 1e-10. `eigs` is passed `tol / sqrt(2)`.

`maxit` The maximum number of iterations. The default is 300.

`disp` The level of diagnostic printout (0|1|2). If `disp` is 0 then diagnostics are disabled. The default value is 0.

If more than one output is requested then `svds` will return an approximation of the singular value decomposition of A

```
A_approx = u*s*v'
```

where A_approx is a matrix of size A but only rank k.

flag returns 0 if the algorithm has succesfully converged, and 1 otherwise. The test for convergence is

```
norm (A*v - u*s, 1) <= tol * norm (A, 1)
```

`svds` is best for finding only a few singular values from a large sparse matrix. Otherwise, `svd (full (A))` will likely be more efficient.

See also: [svd], page 481, [eigs], page 545.

22.3 Iterative Techniques Applied to Sparse Matrices

The left division `\` and right division `/` operators, discussed in the previous section, use direct solvers to resolve a linear equation of the form `x = A \ b` or `x = b / A`. Octave also includes a number of functions to solve sparse linear equations using iterative techniques.

`x = pcg (A, b, tol, maxit, m1, m2, x0, ...)` [Function File]
`[x, flag, relres, iter, resvec, eigest] = pcg (...)` [Function File]

Solve the linear system of equations `A * x = b` by means of the Preconditioned Conjugate Gradient iterative method.

The input arguments are

- A can be either a square (preferably sparse) matrix or a function handle, inline function or string containing the name of a function which computes `A * x`. In principle, A should be symmetric and positive definite; if `pcg` finds A not to be positive definite, a warning is printed and the *flag* output will be set.
- b is the right-hand side vector.
- *tol* is the required relative tolerance for the residual error, `b - A * x`. The iteration stops if `norm (b - A * x)` $\leq$ *tol* * norm (b). If *tol* is omitted or empty then a tolerance of 1e-6 is used.

- *maxit* is the maximum allowable number of iterations; if *maxit* is omitted or empty then a value of 20 is used.
- $m = m1 * m2$ is the (left) preconditioning matrix, so that the iteration is (theoretically) equivalent to solving by `pcg` `P * x = m \ b`, with `P = m \ A`. Note that a proper choice of the preconditioner may dramatically improve the overall performance of the method. Instead of matrices *m1* and *m2*, the user may pass two functions which return the results of applying the inverse of *m1* and *m2* to a vector (usually this is the preferred way of using the preconditioner). If *m1* is omitted or empty `[]` then no preconditioning is applied. If *m2* is omitted, $m = m1$ will be used as a preconditioner.
- *x0* is the initial guess. If *x0* is omitted or empty then the function sets *x0* to a zero vector by default.

The arguments which follow *x0* are treated as parameters, and passed in a proper way to any of the functions (A or m) which are passed to `pcg`. See the examples below for further details. The output arguments are

- *x* is the computed approximation to the solution of `A * x = b`.
- *flag* reports on the convergence. A value of 0 means the solution converged and the tolerance criterion given by *tol* is satisfied. A value of 1 means that the *maxit* limit for the iteration count was reached. A value of 3 indicates that the (preconditioned) matrix was found not to be positive definite.
- *relres* is the ratio of the final residual to its initial value, measured in the Euclidean norm.
- *iter* is the actual number of iterations performed.
- *resvec* describes the convergence history of the method. `resvec(i,1)` is the Euclidean norm of the residual, and `resvec(i,2)` is the preconditioned residual norm, after the (*i*-1)-th iteration, `i = 1, 2, ..., iter+1`. The preconditioned residual norm is defined as `norm (r) ^ 2 = r' * (m \ r)` where `r = b - A * x`, see also the description of *m*. If *eigest* is not required, only `resvec(:,1)` is returned.
- *eigest* returns the estimate for the smallest `eigest(1)` and largest `eigest(2)` eigenvalues of the preconditioned matrix `P = m \ A`. In particular, if no preconditioning is used, the estimates for the extreme eigenvalues of A are returned. `eigest(1)` is an overestimate and `eigest(2)` is an underestimate, so that `eigest(2) / eigest(1)` is a lower bound for `cond (P, 2)`, which nevertheless in the limit should theoretically be equal to the actual value of the condition number. The method which computes *eigest* works only for symmetric positive definite A and m, and the user is responsible for verifying this assumption.

Let us consider a trivial problem with a diagonal matrix (we exploit the sparsity of A)

```
n = 10;
A = diag (sparse (1:n));
b = rand (n, 1);
[l, u, p] = ilu (A, struct ("droptol", 1.e-3));
```

Example 1: Simplest use of `pcg`

```
x = pcg (A, b)
```

Example 2: pcg with a function which computes $A * x$

```
function y = apply_a (x)
  y = [1:N]' .* x;
endfunction

x = pcg ("apply_a", b)
```

Example 3: pcg with a preconditioner: $l * u$

```
x = pcg (A, b, 1.e-6, 500, l*u)
```

Example 4: pcg with a preconditioner: $l * u$. Faster than Example 3 since lower and upper triangular matrices are easier to invert

```
x = pcg (A, b, 1.e-6, 500, l, u)
```

Example 5: Preconditioned iteration, with full diagnostics. The preconditioner (quite strange, because even the original matrix A is trivial) is defined as a function

```
function y = apply_m (x)
  k = floor (length (x) - 2);
  y = x;
  y(1:k) = x(1:k) ./ [1:k]';
endfunction

[x, flag, relres, iter, resvec, eigest] = ...
                   pcg (A, b, [], [], "apply_m");
semilogy (1:iter+1, resvec);
```

Example 6: Finally, a preconditioner which depends on a parameter k.

```
function y = apply_M (x, varargin)
  K = varargin{1};
  y = x;
  y(1:K) = x(1:K) ./ [1:K]';
endfunction

[x, flag, relres, iter, resvec, eigest] = ...
     pcg (A, b, [], [], "apply_m", [], [], 3)
```

References:

1. C.T. Kelley, *Iterative Methods for Linear and Nonlinear Equations*, SIAM, 1995. (the base PCG algorithm)
2. Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS 1996. (condition number estimate from PCG) Revised version of this book is available online at http://www-users.cs.umn.edu/~saad/books.html

See also: [sparse], page 524, [pcr], page 550.

x = pcr (*A*, *b*, *tol*, *maxit*, *m*, *x0*, ...) [Function File]
[*x*, *flag*, *relres*, *iter*, *resvec*] = pcr (...) [Function File]

Solve the linear system of equations $A * x = b$ by means of the Preconditioned Conjugate Residuals iterative method.

The input arguments are

- *A* can be either a square (preferably sparse) matrix or a function handle, inline function or string containing the name of a function which computes `A * x`. In principle *A* should be symmetric and non-singular; if `pcr` finds *A* to be numerically singular, you will get a warning message and the *flag* output parameter will be set.
- *b* is the right hand side vector.
- *tol* is the required relative tolerance for the residual error, `b - A * x`. The iteration stops if `norm (b - A * x) <= tol * norm (b - A * x0)`. If *tol* is empty or is omitted, the function sets `tol = 1e-6` by default.
- *maxit* is the maximum allowable number of iterations; if `[]` is supplied for `maxit`, or `pcr` has less arguments, a default value equal to 20 is used.
- *m* is the (left) preconditioning matrix, so that the iteration is (theoretically) equivalent to solving by `pcr` `P * x = m \ b`, with `P = m \ A`. Note that a proper choice of the preconditioner may dramatically improve the overall performance of the method. Instead of matrix *m*, the user may pass a function which returns the results of applying the inverse of *m* to a vector (usually this is the preferred way of using the preconditioner). If `[]` is supplied for *m*, or *m* is omitted, no preconditioning is applied.
- *x0* is the initial guess. If *x0* is empty or omitted, the function sets *x0* to a zero vector by default.

The arguments which follow *x0* are treated as parameters, and passed in a proper way to any of the functions (*A* or *m*) which are passed to `pcr`. See the examples below for further details.

The output arguments are

- *x* is the computed approximation to the solution of `A * x = b`.
- *flag* reports on the convergence. `flag = 0` means the solution converged and the tolerance criterion given by *tol* is satisfied. `flag = 1` means that the *maxit* limit for the iteration count was reached. `flag = 3` reports a `pcr` breakdown, see [1] for details.
- *relres* is the ratio of the final residual to its initial value, measured in the Euclidean norm.
- *iter* is the actual number of iterations performed.
- *resvec* describes the convergence history of the method, so that `resvec (i)` contains the Euclidean norms of the residual after the (*i*-1)-th iteration, `i = 1,2, ..., iter+1`.

Let us consider a trivial problem with a diagonal matrix (we exploit the sparsity of A)

```
n = 10;
A = sparse (diag (1:n));
b = rand (N, 1);
```

EXAMPLE 1: Simplest use of `pcr`

```
x = pcr (A, b)
```

Example 2: pcr with a function which computes $A * x$.

```
function y = apply_a (x)
  y = [1:10]' .* x;
endfunction

x = pcr ("apply_a", b)
```

Example 3: Preconditioned iteration, with full diagnostics. The preconditioner (quite strange, because even the original matrix A is trivial) is defined as a function

```
function y = apply_m (x)
  k = floor (length (x) - 2);
  y = x;
  y(1:k) = x(1:k) ./ [1:k]';
endfunction

[x, flag, relres, iter, resvec] = ...
                   pcr (A, b, [], [], "apply_m")
semilogy ([1:iter+1], resvec);
```

Example 4: Finally, a preconditioner which depends on a parameter k.

```
function y = apply_m (x, varargin)
  k = varargin{1};
  y = x;
  y(1:k) = x(1:k) ./ [1:k]';
endfunction

[x, flag, relres, iter, resvec] = ...
                   pcr (A, b, [], [], "apply_m"', [], 3)
```

References:

[1] W. Hackbusch, *Iterative Solution of Large Sparse Systems of Equations*, section 9.5.4; Springer, 1994

See also: [sparse], page 524, [pcg], page 548.

The speed with which an iterative solver converges to a solution can be accelerated with the use of a pre-conditioning matrix M. In this case the linear equation `M^-1 * x = M^-1 * A \ b` is solved instead. Typical pre-conditioning matrices are partial factorizations of the original matrix.

L = ichol (*A*) [Function File]
L = ichol (*A*, *opts*) [Function File]

Compute the incomplete Cholesky factorization of the sparse square matrix A.

By default, `ichol` uses only the lower triangle of A and produces a lower triangular factor L such that `L*L'` approximates A.

The factor given by this routine may be useful as a preconditioner for a system of linear equations being solved by iterative methods such as PCG (Preconditioned Conjugate Gradient).

The factorization may be modified by passing options in a structure *opts*. The option name is a field of the structure and the setting is the value of field. Names and specifiers are case sensitive.

type
: Type of factorization.

  `"nofill"` (default)
  : Incomplete Cholesky factorization with no fill-in (IC(0)).

  `"ict"`
  : Incomplete Cholesky factorization with threshold dropping (ICT).

diagcomp
: A non-negative scalar *alpha* for incomplete Cholesky factorization of `A + alpha * diag (diag (A))` instead of *A*. This can be useful when *A* is not positive definite. The default value is 0.

droptol
: A non-negative scalar specifying the drop tolerance for factorization if performing ICT. The default value is 0 which produces the complete Cholesky factorization.

  Non-diagonal entries of *L* are set to 0 unless

  `abs (L(i,j)) >= droptol * norm (A(j:end, j), 1)`.

michol
: Modified incomplete Cholesky factorization:

  `"off"` (default)
  : Row and column sums are not necessarily preserved.

  `"on"`
  : The diagonal of *L* is modified so that row (and column) sums are preserved even when elements have been dropped during the factorization. The relationship preserved is: `A * e = L * L' * e`, where e is a vector of ones.

shape
:

  `"lower"` (default)
  : Use only the lower triangle of *A* and return a lower triangular factor *L* such that `L*L'` approximates *A*.

  `"upper"`
  : Use only the upper triangle of *A* and return an upper triangular factor *U* such that `U'*U` approximates *A*.

EXAMPLES

The following problem demonstrates how to factorize a sample symmetric positive definite matrix with the full Cholesky decomposition and with the incomplete one.

```
A = [ 0.37, -0.05,  -0.05,  -0.07;
     -0.05,  0.116,  0.0,   -0.05;
     -0.05,  0.0,    0.116, -0.05;
     -0.07, -0.05,  -0.05,   0.202];
A = sparse (A);
nnz (tril (A))
ans =  9
L = chol (A, "lower");
nnz (L)
ans =  10
norm (A - L * L', "fro") / norm (A, "fro")
ans =  1.1993e-16
opts.type = "nofill";
L = ichol (A, opts);
nnz (L)
ans =  9
norm (A - L * L', "fro") / norm (A, "fro")
ans =  0.019736
```

Another example for decomposition is a finite difference matrix used to solve a boundary value problem on the unit square.

```
nx = 400; ny = 200;
hx = 1 / (nx + 1); hy = 1 / (ny + 1);
Dxx = spdiags ([ones(nx, 1), -2*ones(nx, 1), ones(nx, 1)],
               [-1 0 1 ], nx, nx) / (hx ^ 2);
Dyy = spdiags ([ones(ny, 1), -2*ones(ny, 1), ones(ny, 1)],
               [-1 0 1 ], ny, ny) / (hy ^ 2);
A = -kron (Dxx, speye (ny)) - kron (speye (nx), Dyy);
nnz (tril (A))
ans =  239400
opts.type = "nofill";
L = ichol (A, opts);
nnz (tril (A))
ans =  239400
norm (A - L * L', "fro") / norm (A, "fro")
ans =  0.062327
```

References for implemented algorithms:

[1] Y. Saad. "Preconditioning Techniques." *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, 1996.

[2] M. Jones, P. Plassmann: *An Improved Incomplete Cholesky Factorization*, 1992.

See also: [chol], page 472, [ilu], page 554, [pcg], page 548.

`ilu (`*A*`)` [Function File]
`ilu (`*A*, *opts*`)` [Function File]
`[`*L*, *U*`] = ilu (...)` [Function File]
`[`*L*, *U*, *P*`] = ilu (...)` [Function File]

Compute the incomplete LU factorization of the sparse square matrix *A*.

`ilu` returns a unit lower triangular matrix L, an upper triangular matrix U, and optionally a permutation matrix P, such that `L*U` approximates `P*A`.

The factors given by this routine may be useful as preconditioners for a system of linear equations being solved by iterative methods such as BICG (BiConjugate Gradients) or GMRES (Generalized Minimum Residual Method).

The factorization may be modified by passing options in a structure *opts*. The option name is a field of the structure and the setting is the value of field. Names and specifiers are case sensitive.

`type` Type of factorization.

- `"nofill"` ILU factorization with no fill-in (ILU(0)).

  Additional supported options: `milu`.

- `"crout"` Crout version of ILU factorization (ILUC).

  Additional supported options: `milu`, `droptol`.

- `"ilutp"` (default)

  ILU factorization with threshold and pivoting.

  Additional supported options: `milu`, `droptol`, `udiag`, `thresh`.

`droptol` A non-negative scalar specifying the drop tolerance for factorization. The default value is 0 which produces the complete LU factorization.

Non-diagonal entries of U are set to 0 unless

`abs (U(i,j)) >= droptol * norm (A(:,j))`.

Non-diagonal entries of L are set to 0 unless

`abs (L(i,j)) >= droptol * norm (A(:,j))/U(j,j)`.

`milu` Modified incomplete LU factorization:

- `"row"` Row-sum modified incomplete LU factorization. The factorization preserves row sums: `A * e = L * U * e`, where e is a vector of ones.

- `"col"` Column-sum modified incomplete LU factorization. The factorization preserves column sums: `e' * A = e' * L * U`.

- `"off"` (default)

  Row and column sums are not necessarily preserved.

`udiag` If true, any zeros on the diagonal of the upper triangular factor are replaced by the local drop tolerance `droptol * norm (A(:,j))/U(j,j)`. The default is false.

`thresh` Pivot threshold for factorization. It can range between 0 (diagonal pivoting) and 1 (default), where the maximum magnitude entry in the column is chosen to be the pivot.

If `ilu` is called with just one output, the returned matrix is `L + U - speye (size (A))`, where L is unit lower triangular and U is upper triangular.

With two outputs, `ilu` returns a unit lower triangular matrix L and an upper triangular matrix U. For *opts*.type == `"ilutp"`, one of the factors is permuted based on the value of *opts*.milu. When *opts*.milu == `"row"`, U is a column permuted upper triangular factor. Otherwise, L is a row-permuted unit lower triangular factor.

If there are three named outputs and *opts*.milu != `"row"`, P is returned such that L and U are incomplete factors of `P*A`. When *opts*.milu == `"row"`, P is returned such that L and U are incomplete factors of `A*P`.

EXAMPLES

```
A = gallery ("neumann", 1600) + speye (1600);
opts.type = "nofill";
nnz (A)
ans = 7840

nnz (lu (A))
ans = 126478

nnz (ilu (A, opts))
ans = 7840
```

This shows that A has 7,840 nonzeros, the complete LU factorization has 126,478 nonzeros, and the incomplete LU factorization, with 0 level of fill-in, has 7,840 nonzeros, the same amount as A. Taken from: http://www.mathworks.com/help/matlab/ref/ilu.html

```
A = gallery ("wathen", 10, 10);
b = sum (A, 2);
tol = 1e-8;
maxit = 50;
opts.type = "crout";
opts.droptol = 1e-4;
[L, U] = ilu (A, opts);
x = bicg (A, b, tol, maxit, L, U);
norm (A * x - b, inf)
```

This example uses ILU as preconditioner for a random FEM-Matrix, which has a large condition number. Without L and U BICG would not converge.

See also: [lu], page 475, [ichol], page 552, [bicg], page 485, [gmres], page 487.

22.4 Real Life Example using Sparse Matrices

A common application for sparse matrices is in the solution of Finite Element Models. Finite element models allow numerical solution of partial differential equations that do not have closed form solutions, typically because of the complex shape of the domain.

In order to motivate this application, we consider the boundary value Laplace equation. This system can model scalar potential fields, such as heat or electrical potential. Given a medium Ω with boundary $\partial\Omega$. At all points on the $\partial\Omega$ the boundary conditions are known, and we wish to calculate the potential in Ω. Boundary conditions may specify the potential (Dirichlet boundary condition), its normal derivative across the boundary (Neumann boundary condition), or a weighted sum of the potential and its derivative (Cauchy boundary condition).

In a thermal model, we want to calculate the temperature in Ω and know the boundary temperature (Dirichlet condition) or heat flux (from which we can calculate the Neumann condition by dividing by the thermal conductivity at the boundary). Similarly, in an electrical model, we want to calculate the voltage in Ω and know the boundary voltage (Dirichlet) or current (Neumann condition after diving by the electrical conductivity). In an electrical model, it is common for much of the boundary to be electrically isolated; this is a Neumann boundary condition with the current equal to zero.

The simplest finite element models will divide Ω into simplexes (triangles in 2D, pyramids in 3D). We take as a 3-D example a cylindrical liquid filled tank with a small non-conductive ball from the EIDORS project[4]. This is model is designed to reflect an application of electrical impedance tomography, where current patterns are applied to such a tank in order to image the internal conductivity distribution. In order to describe the FEM geometry, we have a matrix of vertices `nodes` and simplices `elems`.

The following example creates a simple rectangular 2-D electrically conductive medium with 10 V and 20 V imposed on opposite sides (Dirichlet boundary conditions). All other edges are electrically isolated.

```
node_y = [1;1.2;1.5;1.8;2]*ones(1,11);
node_x = ones(5,1)*[1,1.05,1.1,1.2, ...
          1.3,1.5,1.7,1.8,1.9,1.95,2];
nodes = [node_x(:), node_y(:)];

[h,w] = size (node_x);
elems = [];
for idx = 1:w-1
  widx = (idx-1)*h;
  elems = [elems; ...
    widx+[(1:h-1);(2:h);h+(1:h-1)]'; ...
    widx+[(2:h);h+(2:h);h+(1:h-1)]' ];
endfor

E = size (elems,1); # No. of simplices
N = size (nodes,1); # No. of vertices
D = size (elems,2); # dimensions+1
```

This creates a N-by-2 matrix `nodes` and a E-by-3 matrix `elems` with values, which define finite element triangles:

---

[4] EIDORS - Electrical Impedance Tomography and Diffuse optical Tomography Reconstruction Software `http://eidors3d.sourceforge.net`

```
nodes(1:7,:)'
  1.00 1.00 1.00 1.00 1.00 1.05 1.05 ...
  1.00 1.20 1.50 1.80 2.00 1.00 1.20 ...

elems(1:7,:)'
  1    2    3    4    2    3    4 ...
  2    3    4    5    7    8    9 ...
  6    7    8    9    6    7    8 ...
```

Using a first order FEM, we approximate the electrical conductivity distribution in Ω as constant on each simplex (represented by the vector `conductivity`). Based on the finite element geometry, we first calculate a system (or stiffness) matrix for each simplex (represented as 3-by-3 elements on the diagonal of the element-wise system matrix `SE`). Based on `SE` and a N-by-DE connectivity matrix `C`, representing the connections between simplices and vertices, the global connectivity matrix `S` is calculated.

```
## Element conductivity
conductivity = [1*ones(1,16), ...
       2*ones(1,48), 1*ones(1,16)];

## Connectivity matrix
C = sparse ((1:D*E), reshape (elems', ...
       D*E, 1), 1, D*E, N);

## Calculate system matrix
Siidx = floor ([0:D*E-1]'/D) * D * ...
       ones(1,D) + ones(D*E,1)*(1:D) ;
Sjidx = [1:D*E]'*ones (1,D);
Sdata = zeros (D*E,D);
dfact = factorial (D-1);
for j = 1:E
   a = inv ([ones(D,1), ...
       nodes(elems(j,:), :)]);
   const = conductivity(j) * 2 / ...
       dfact / abs (det (a));
   Sdata(D*(j-1)+(1:D),:) = const * ...
       a(2:D,:)' * a(2:D,:);
endfor
## Element-wise system matrix
SE = sparse(Siidx,Sjidx,Sdata);
## Global system matrix
S = C'* SE *C;
```

The system matrix acts like the conductivity S in Ohm's law $SV = I$. Based on the Dirichlet and Neumann boundary conditions, we are able to solve for the voltages at each vertex `V`.

```
## Dirichlet boundary conditions
D_nodes = [1:5, 51:55];
D_value = [10*ones(1,5), 20*ones(1,5)];
```

```
V = zeros (N,1);
V(D_nodes) = D_value;
idx = 1:N; # vertices without Dirichlet
           # boundary condns
idx(D_nodes) = [];

## Neumann boundary conditions.  Note that
## N_value must be normalized by the
## boundary length and element conductivity
N_nodes = [];
N_value = [];

Q = zeros (N,1);
Q(N_nodes) = N_value;

V(idx) = S(idx,idx) \ ( Q(idx) - ...
          S(idx,D_nodes) * V(D_nodes));
```

Finally, in order to display the solution, we show each solved voltage value in the z-axis for each simplex vertex. See Figure 22.6.

```
elemx = elems(:,[1,2,3,1])';
xelems = reshape (nodes(elemx, 1), 4, E);
yelems = reshape (nodes(elemx, 2), 4, E);
velems = reshape (V(elemx), 4, E);
plot3 (xelems,yelems,velems,"k");
print "grid.eps";
```

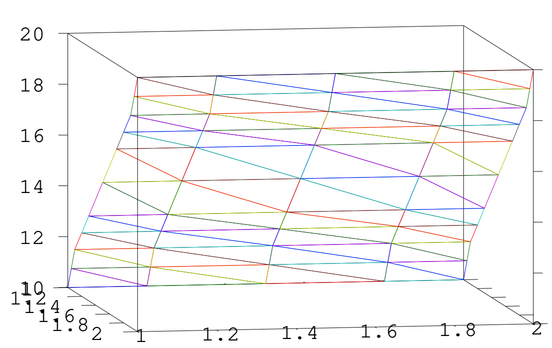

Figure 22.6: Example finite element model the showing triangular elements. The height of each vertex corresponds to the solution value.

23 Numerical Integration

Octave comes with several built-in functions for computing the integral of a function numerically (termed quadrature). These functions all solve 1-dimensional integration problems.

23.1 Functions of One Variable

Octave supports five different algorithms for computing the integral

$$\int_a^b f(x)dx$$

of a function f over the interval from a to b. These are

`quad` Numerical integration based on Gaussian quadrature.

`quadv` Numerical integration using an adaptive vectorized Simpson's rule.

`quadl` Numerical integration using an adaptive Lobatto rule.

`quadgk` Numerical integration using an adaptive Gauss-Konrod rule.

`quadcc` Numerical integration using adaptive Clenshaw-Curtis rules.

`trapz, cumtrapz`
Numerical integration of data using the trapezoidal method.

The best quadrature algorithm to use depends on the integrand. If you have empirical data, rather than a function, the choice is `trapz` or `cumtrapz`. If you are uncertain about the characteristics of the integrand, `quadcc` will be the most robust as it can handle discontinuities, singularities, oscillatory functions, and infinite intervals. When the integrand is smooth `quadgk` may be the fastest of the algorithms.

Function	Characteristics
quad	Low accuracy with nonsmooth integrands
quadv	Medium accuracy with smooth integrands
quadl	Medium accuracy with smooth integrands. Slower than quadgk.
quadgk	Medium accuracy ($1e^{-6}$–$1e^{-9}$) with smooth integrands.
	Handles oscillatory functions and infinite bounds
quadcc	Low to High accuracy with nonsmooth/smooth integrands
	Handles oscillatory functions, singularities, and infinite bounds

Here is an example of using `quad` to integrate the function

$$f(x) = x\sin(1/x)\sqrt{|1-x|}$$

from $x = 0$ to $x = 3$.

This is a fairly difficult integration (plot the function over the range of integration to see why).

The first step is to define the function:

```
function y = f (x)
  y = x .* sin (1./x) .* sqrt (abs (1 - x));
endfunction
```

Note the use of the 'dot' forms of the operators. This is not necessary for the `quad` integrator, but is required by the other integrators. In any case, it makes it much easier to generate a set of points for plotting because it is possible to call the function with a vector argument to produce a vector result.

The second step is to call quad with the limits of integration:

```
[q, ier, nfun, err] = quad ("f", 0, 3)
     ⇒ 1.9819
     ⇒ 1
     ⇒ 5061
     ⇒ 1.1522e-07
```

Although `quad` returns a nonzero value for *ier*, the result is reasonably accurate (to see why, examine what happens to the result if you move the lower bound to 0.1, then 0.01, then 0.001, etc.).

The function `"f"` can be the string name of a function, a function handle, or an inline function. These options make it quite easy to do integration without having to fully define a function in an m-file. For example:

```
# Verify integral (x^3) = x^4/4
f = inline ("x.^3");
quadgk (f, 0, 1)
     ⇒ 0.25000

# Verify gamma function = (n-1)! for n = 4
f = @(x) x.^3 .* exp (-x);
quadcc (f, 0, Inf)
     ⇒ 6.0000
```

q = **quad** (*f*, *a*, *b*) [Built-in Function]
q = **quad** (*f*, *a*, *b*, *tol*) [Built-in Function]
q = **quad** (*f*, *a*, *b*, *tol*, *sing*) [Built-in Function]
[*q*, *ier*, *nfun*, *err*] = **quad** (...) [Built-in Function]

Numerically evaluate the integral of *f* from *a* to *b* using Fortran routines from QUADPACK.

f is a function handle, inline function, or a string containing the name of the function to evaluate. The function must have the form `y = f (x)` where *y* and *x* are scalars.

a and *b* are the lower and upper limits of integration. Either or both may be infinite.

The optional argument *tol* is a vector that specifies the desired accuracy of the result. The first element of the vector is the desired absolute tolerance, and the second element is the desired relative tolerance. To choose a relative test only, set the absolute tolerance to zero. To choose an absolute test only, set the relative tolerance to zero. Both tolerances default to `sqrt (eps)` or approximately $1.5e^{-8}$.

The optional argument *sing* is a vector of values at which the integrand is known to be singular.

The result of the integration is returned in *q*.

ier contains an integer error code (0 indicates a successful integration).

nfun indicates the number of function evaluations that were made.

err contains an estimate of the error in the solution.

The function `quad_options` can set other optional parameters for `quad`.

Note: because `quad` is written in Fortran it cannot be called recursively. This prevents its use in integrating over more than one variable by routines `dblquad` and `triplequad`.

See also: [quad_options], page 563, [quadv], page 563, [quadl], page 564, [quadgk], page 564, [quadcc], page 566, [trapz], page 567, [dblquad], page 569, [triplequad], page 570.

`quad_options ()` [Built-in Function]
`val = quad_options (`*opt*`)` [Built-in Function]
`quad_options (`*opt*`, `*val*`)` [Built-in Function]

Query or set options for the function `quad`.

When called with no arguments, the names of all available options and their current values are displayed.

Given one argument, return the value of the option *opt*.

When called with two arguments, `quad_options` sets the option *opt* to value *val*.

Options include

`"absolute tolerance"`
: Absolute tolerance; may be zero for pure relative error test.

`"relative tolerance"`
: Non-negative relative tolerance. If the absolute tolerance is zero, the relative tolerance must be greater than or equal to `max (50*eps, 0.5e-28)`.

`"single precision absolute tolerance"`
: Absolute tolerance for single precision; may be zero for pure relative error test.

`"single precision relative tolerance"`
: Non-negative relative tolerance for single precision. If the absolute tolerance is zero, the relative tolerance must be greater than or equal to `max (50*eps, 0.5e-28)`.

q `= quadv (`*f*`, `*a*`, `*b*`)` [Function File]
q `= quadv (`*f*`, `*a*`, `*b*`, `*tol*`)` [Function File]
q `= quadv (`*f*`, `*a*`, `*b*`, `*tol*`, `*trace*`)` [Function File]
q `= quadv (`*f*`, `*a*`, `*b*`, `*tol*`, `*trace*`, `*p1*`, `*p2*`, ...)` [Function File]
`[`*q*`, `*nfun*`] = quadv (...)` [Function File]

Numerically evaluate the integral of *f* from *a* to *b* using an adaptive Simpson's rule.

f is a function handle, inline function, or string containing the name of the function to evaluate. `quadv` is a vectorized version of `quad` and the function defined by *f* must accept a scalar or vector as input and return a scalar, vector, or array as output.

a and *b* are the lower and upper limits of integration. Both limits must be finite.

The optional argument *tol* defines the absolute tolerance used to stop the adaptation procedure. The default value is 1e-6.

The algorithm used by `quadv` involves recursively subdividing the integration interval and applying Simpson's rule on each subinterval. If *trace* is true then after computing each of these partial integrals display: (1) the total number of function evaluations, (2) the left end of the subinterval, (3) the length of the subinterval, (4) the approximation of the integral over the subinterval.

Additional arguments *p1*, etc., are passed directly to the function *f*. To use default values for *tol* and *trace*, one may pass empty matrices ([]).

The result of the integration is returned in *q*

nfun indicates the number of function evaluations that were made.

Note: `quadv` is written in Octave's scripting language and can be used recursively in `dblquad` and `triplequad`, unlike the `quad` function.

See also: [quad], page 562, [quadl], page 564, [quadgk], page 564, [quadcc], page 566, [trapz], page 567, [dblquad], page 569, [triplequad], page 570.

q = `quadl` (*f*, *a*, *b*) [Function File]
q = `quadl` (*f*, *a*, *b*, *tol*) [Function File]
q = `quadl` (*f*, *a*, *b*, *tol*, *trace*) [Function File]
q = `quadl` (*f*, *a*, *b*, *tol*, *trace*, *p1*, *p2*, . . .) [Function File]

Numerically evaluate the integral of *f* from *a* to *b* using an adaptive Lobatto rule.

f is a function handle, inline function, or string containing the name of the function to evaluate. The function *f* must be vectorized and return a vector of output values when given a vector of input values.

a and *b* are the lower and upper limits of integration. Both limits must be finite.

The optional argument *tol* defines the relative tolerance with which to perform the integration. The default value is `eps`.

The algorithm used by `quadl` involves recursively subdividing the integration interval. If *trace* is defined then for each subinterval display: (1) the left end of the subinterval, (2) the length of the subinterval, (3) the approximation of the integral over the subinterval.

Additional arguments *p1*, etc., are passed directly to the function *f*. To use default values for *tol* and *trace*, one may pass empty matrices ([]).

Reference: W. Gander and W. Gautschi, *Adaptive Quadrature - Revisited*, BIT Vol. 40, No. 1, March 2000, pp. 84–101. `http://www.inf.ethz.ch/personal/gander/`

See also: [quad], page 562, [quadv], page 563, [quadgk], page 564, [quadcc], page 566, [trapz], page 567, [dblquad], page 569, [triplequad], page 570.

q = `quadgk` (*f*, *a*, *b*) [Function File]
q = `quadgk` (*f*, *a*, *b*, *abstol*) [Function File]
q = `quadgk` (*f*, *a*, *b*, *abstol*, *trace*) [Function File]
q = `quadgk` (*f*, *a*, *b*, *prop*, *val*, . . .) [Function File]
[*q*, *err*] = `quadgk` (. . .) [Function File]

Numerically evaluate the integral of *f* from *a* to *b* using adaptive Gauss-Konrod quadrature.

f is a function handle, inline function, or string containing the name of the function to evaluate. The function *f* must be vectorized and return a vector of output values when given a vector of input values.

a and *b* are the lower and upper limits of integration. Either or both limits may be infinite or contain weak end singularities. Variable transformation will be used to treat any infinite intervals and weaken the singularities. For example:

```
quadgk (@(x) 1 ./ (sqrt (x) .* (x + 1)), 0, Inf)
```

Note that the formulation of the integrand uses the element-by-element operator `./` and all user functions to `quadgk` should do the same.

The optional argument *tol* defines the absolute tolerance used to stop the integration procedure. The default value is 1e-10.

The algorithm used by `quadgk` involves subdividing the integration interval and evaluating each subinterval. If *trace* is true then after computing each of these partial integrals display: (1) the number of subintervals at this step, (2) the current estimate of the error *err*, (3) the current estimate for the integral *q*.

Alternatively, properties of `quadgk` can be passed to the function as pairs `"prop"`, `val`. Valid properties are

`AbsTol` Define the absolute error tolerance for the quadrature. The default absolute tolerance is 1e-10.

`RelTol` Define the relative error tolerance for the quadrature. The default relative tolerance is 1e-5.

`MaxIntervalCount`
: `quadgk` initially subdivides the interval on which to perform the quadrature into 10 intervals. Subintervals that have an unacceptable error are subdivided and re-evaluated. If the number of subintervals exceeds 650 subintervals at any point then a poor convergence is signaled and the current estimate of the integral is returned. The property `"MaxIntervalCount"` can be used to alter the number of subintervals that can exist before exiting.

`WayPoints`
: Discontinuities in the first derivative of the function to integrate can be flagged with the `"WayPoints"` property. This forces the ends of a subinterval to fall on the breakpoints of the function and can result in significantly improved estimation of the error in the integral, faster computation, or both. For example,

```
quadgk (@(x) abs (1 - x.^2), 0, 2, "Waypoints", 1)
```

signals the breakpoint in the integrand at `x = 1`.

`Trace` If logically true `quadgk` prints information on the convergence of the quadrature at each iteration.

If any of *a*, *b*, or *waypoints* is complex then the quadrature is treated as a contour integral along a piecewise continuous path defined by the above. In this case the integral is assumed to have no edge singularities. For example,

```
quadgk (@(z) log (z), 1+1i, 1+1i, "WayPoints",
        [1-1i, -1,-1i, -1+1i])
```

integrates `log (z)` along the square defined by `[1+1i, 1-1i, -1-1i, -1+1i]`.

The result of the integration is returned in *q*.

err is an approximate bound on the error in the integral `abs (q - I)`, where *I* is the exact value of the integral.

Reference: L.F. Shampine, "*Vectorized adaptive quadrature in* MATLAB", Journal of Computational and Applied Mathematics, pp. 131–140, Vol 211, Issue 2, Feb 2008.

See also: [quad], page 562, [quadv], page 563, [quadl], page 564, [quadcc], page 566, [trapz], page 567, [dblquad], page 569, [triplequad], page 570.

`q = quadcc (f, a, b)` [Function File]
`q = quadcc (f, a, b, tol)` [Function File]
`q = quadcc (f, a, b, tol, sing)` [Function File]
`[q, err, nr_points] = quadcc (...)` [Function File]

Numerically evaluate the integral of *f* from *a* to *b* using doubly-adaptive Clenshaw-Curtis quadrature.

f is a function handle, inline function, or string containing the name of the function to evaluate. The function *f* must be vectorized and must return a vector of output values if given a vector of input values. For example,

```
f = @(x) x .* sin (1./x) .* sqrt (abs (1 - x));
```

which uses the element-by-element "dot" form for all operators.

a and *b* are the lower and upper limits of integration. Either or both limits may be infinite. `quadcc` handles an inifinite limit by substituting the variable of integration with `x = tan (pi/2*u)`.

The optional argument *tol* defines the relative tolerance used to stop the integration procedure. The default value is $1e^{-6}$.

The optional argument *sing* contains a list of points where the integrand has known singularities, or discontinuities in any of its derivatives, inside the integration interval. For the example above, which has a discontinuity at x=1, the call to `quadcc` would be as follows

```
int = quadcc (f, a, b, 1.0e-6, [ 1 ]);
```

The result of the integration is returned in *q*.

err is an estimate of the absolute integration error.

nr_points is the number of points at which the integrand was evaluated.

If the adaptive integration did not converge, the value of *err* will be larger than the requested tolerance. Therefore, it is recommended to verify this value for difficult integrands.

`quadcc` is capable of dealing with non-numeric values of the integrand such as `NaN` or `Inf`. If the integral diverges, and `quadcc` detects this, then a warning is issued and `Inf` or `-Inf` is returned.

Note: `quadcc` is a general purpose quadrature algorithm and, as such, may be less efficient for a smooth or otherwise well-behaved integrand than other methods such as `quadgk`.

The algorithm uses Clenshaw-Curtis quadrature rules of increasing degree in each interval and bisects the interval if either the function does not appear to be smooth or a rule of maximum degree has been reached. The error estimate is computed from the L2-norm of the difference between two successive interpolations of the integrand over the nodes of the respective quadrature rules.

Reference: P. Gonnet, *Increasing the Reliability of Adaptive Quadrature Using Explicit Interpolants*, ACM Transactions on Mathematical Software, Vol. 37, Issue 3, Article No. 3, 2010.

See also: [quad], page 562, [quadv], page 563, [quadl], page 564, [quadgk], page 564, [trapz], page 567, [dblquad], page 569, [triplequad], page 570.

Sometimes one does not have the function, but only the raw (x, y) points from which to perform an integration. This can occur when collecting data in an experiment. The `trapz` function can integrate these values as shown in the following example where "data" has been collected on the cosine function over the range [0, pi/2).

```
x = 0:0.1:pi/2;  # Uniformly spaced points
y = cos (x);
trapz (x, y)
     ⇒ 0.99666
```

The answer is reasonably close to the exact value of 1. Ordinary quadrature is sensitive to the characteristics of the integrand. Empirical integration depends not just on the integrand, but also on the particular points chosen to represent the function. Repeating the example above with the sine function over the range [0, pi/2) produces far inferior results.

```
x = 0:0.1:pi/2;  # Uniformly spaced points
y = sin (x);
trapz (x, y)
     ⇒ 0.92849
```

However, a slightly different choice of data points can change the result significantly. The same integration, with the same number of points, but spaced differently produces a more accurate answer.

```
x = linspace (0, pi/2, 16);  # Uniformly spaced, but including endpoint
y = sin (x);
trapz (x, y)
     ⇒ 0.99909
```

In general there may be no way of knowing the best distribution of points ahead of time. Or the points may come from an experiment where there is no freedom to select the best distribution. In any case, one must remain aware of this issue when using `trapz`.

q = `trapz` (*y*) [Function File]
q = `trapz` (*x*, *y*) [Function File]
q = `trapz` (. . ., *dim*) [Function File]

Numerically evaluate the integral of points *y* using the trapezoidal method.

`trapz (y)` computes the integral of *y* along the first non-singleton dimension. When the argument *x* is omitted an equally spaced *x* vector with unit spacing (1) is assumed. `trapz (x, y)` evaluates the integral with respect to the spacing in *x* and the values in *y*. This is useful if the points in *y* have been sampled unevenly.

If the optional *dim* argument is given, operate along this dimension.

Application Note: If *x* is not specified then unit spacing will be used. To scale the integral to the correct value you must multiply by the actual spacing value (deltaX). As an example, the integral of x^3 over the range [0, 1] is $x^4/4$ or 0.25. The following code uses `trapz` to calculate the integral in three different ways.

```
x = 0:0.1:1;
y = x.^3;
q = trapz (y)
  ⇒ q = 2.525   # No scaling
q * 0.1
  ⇒ q = 0.2525  # Approximation to integral by scaling
trapz (x, y)
  ⇒ q = 0.2525  # Same result by specifying x
```

See also: [cumtrapz], page 568.

q = `cumtrapz` (*y*) [Function File]
q = `cumtrapz` (*x*, *y*) [Function File]
q = `cumtrapz` (..., *dim*) [Function File]

Cumulative numerical integration of points *y* using the trapezoidal method.

`cumtrapz (y)` computes the cumulative integral of *y* along the first non-singleton dimension. Where `trapz` reports only the overall integral sum, `cumtrapz` reports the current partial sum value at each point of *y*.

When the argument *x* is omitted an equally spaced *x* vector with unit spacing (1) is assumed. `cumtrapz (x, y)` evaluates the integral with respect to the spacing in *x* and the values in *y*. This is useful if the points in *y* have been sampled unevenly.

If the optional *dim* argument is given, operate along this dimension.

Application Note: If *x* is not specified then unit spacing will be used. To scale the integral to the correct value you must multiply by the actual spacing value (deltaX).

See also: [trapz], page 567, [cumsum], page 443.

23.2 Orthogonal Collocation

[*r*, *amat*, *bmat*, *q*] = `colloc` (*n*, "*left*", "*right*") [Built-in Function]

Compute derivative and integral weight matrices for orthogonal collocation.

Reference: J. Villadsen, M. L. Michelsen, *Solution of Differential Equation Models by Polynomial Approximation*.

Here is an example of using `colloc` to generate weight matrices for solving the second order differential equation $u' - \alpha u'' = 0$ with the boundary conditions $u(0) = 0$ and $u(1) = 1$.

First, we can generate the weight matrices for *n* points (including the endpoints of the interval), and incorporate the boundary conditions in the right hand side (for a specific value of α).

```
n = 7;
alpha = 0.1;
[r, a, b] = colloc (n-2, "left", "right");
at = a(2:n-1,2:n-1);
bt = b(2:n-1,2:n-1);
rhs = alpha * b(2:n-1,n) - a(2:n-1,n);
```

Then the solution at the roots r is

```
u = [ 0; (at - alpha * bt) \ rhs; 1]
     ⇒ [ 0.00; 0.004; 0.01 0.00; 0.12; 0.62; 1.00 ]
```

23.3 Functions of Multiple Variables

Octave does not have built-in functions for computing the integral of functions of multiple variables directly. It is possible, however, to compute the integral of a function of multiple variables using the existing functions for one-dimensional integrals.

To illustrate how the integration can be performed, we will integrate the function

$$f(x,y) = \sin(\pi xy)\sqrt{xy}$$

for x and y between 0 and 1.

The first approach creates a function that integrates f with respect to x, and then integrates that function with respect to y. Because `quad` is written in Fortran it cannot be called recursively. This means that `quad` cannot integrate a function that calls `quad`, and hence cannot be used to perform the double integration. Any of the other integrators, however, can be used which is what the following code demonstrates.

```
function q = g(y)
  q = ones (size (y));
  for i = 1:length (y)
    f = @(x) sin (pi*x.*y(i)) .* sqrt (x.*y(i));
    q(i) = quadgk (f, 0, 1);
  endfor
endfunction

I = quadgk ("g", 0, 1)
      ⇒ 0.30022
```

The above process can be simplified with the `dblquad` and `triplequad` functions for integrals over two and three variables. For example:

```
I = dblquad (@(x, y) sin (pi*x.*y) .* sqrt (x.*y), 0, 1, 0, 1)
      ⇒ 0.30022
```

`dblquad` (*f*, *xa*, *xb*, *ya*, *yb*) [Function File]
`dblquad` (*f*, *xa*, *xb*, *ya*, *yb*, *tol*) [Function File]
`dblquad` (*f*, *xa*, *xb*, *ya*, *yb*, *tol*, *quadf*) [Function File]
`dblquad` (*f*, *xa*, *xb*, *ya*, *yb*, *tol*, *quadf*, ...) [Function File]

Numerically evaluate the double integral of *f*.

f is a function handle, inline function, or string containing the name of the function to evaluate. The function *f* must have the form $z = f(x, y)$ where x is a vector and y is a scalar. It should return a vector of the same length and orientation as x.

xa, *ya* and *xb*, *yb* are the lower and upper limits of integration for x and y respectively. The underlying integrator determines whether infinite bounds are accepted.

The optional argument *tol* defines the absolute tolerance used to integrate each sub-integral. The default value is $1e^{-6}$.

The optional argument *quadf* specifies which underlying integrator function to use. Any choice but `quad` is available and the default is `quadcc`.

Additional arguments, are passed directly to *f*. To use the default value for *tol* or *quadf* one may pass `':'` or an empty matrix ([]).

See also: [triplequad], page 570, [quad], page 562, [quadv], page 563, [quadl], page 564, [quadgk], page 564, [quadcc], page 566, [trapz], page 567.

`triplequad (f, xa, xb, ya, yb, za, zb)` [Function File]
`triplequad (f, xa, xb, ya, yb, za, zb, tol)` [Function File]
`triplequad (f, xa, xb, ya, yb, za, zb, tol, quadf)` [Function File]
`triplequad (f, xa, xb, ya, yb, za, zb, tol, quadf, ...)` [Function File]

Numerically evaluate the triple integral of *f*.

f is a function handle, inline function, or string containing the name of the function to evaluate. The function *f* must have the form $w = f(x, y, z)$ where either x or y is a vector and the remaining inputs are scalars. It should return a vector of the same length and orientation as x or y.

xa, *ya*, *za* and *xb*, *yb*, *zb* are the lower and upper limits of integration for x, y, and z respectively. The underlying integrator determines whether infinite bounds are accepted.

The optional argument *tol* defines the absolute tolerance used to integrate each sub-integral. The default value is 1e-6.

The optional argument *quadf* specifies which underlying integrator function to use. Any choice but `quad` is available and the default is `quadcc`.

Additional arguments, are passed directly to *f*. To use the default value for *tol* or *quadf* one may pass `':'` or an empty matrix ([]).

See also: [dblquad], page 569, [quad], page 562, [quadv], page 563, [quadl], page 564, [quadgk], page 564, [quadcc], page 566, [trapz], page 567.

The above mentioned approach works, but is fairly slow, and that problem increases exponentially with the dimensionality of the integral. Another possible solution is to use Orthogonal Collocation as described in the previous section (see Section 23.2 [Orthogonal Collocation], page 568). The integral of a function $f(x, y)$ for x and y between 0 and 1 can be approximated using n points by

$$\int_0^1 \int_0^1 f(x, y) dx dy \approx \sum_{i=1}^{n} \sum_{j=1}^{n} q_i q_j f(r_i, r_j),$$

where q and r is as returned by `colloc (n)`. The generalization to more than two variables is straight forward. The following code computes the studied integral using $n = 8$ points.

```
f = @(x,y) sin (pi*x*y') .* sqrt (x*y');
n = 8;
[t, ~, ~, q] = colloc (n);
I = q'*f(t,t)*q;
      ⇒ 0.30022
```

It should be noted that the number of points determines the quality of the approximation. If the integration needs to be performed between a and b, instead of 0 and 1, then a change of variables is needed.

24 Differential Equations

Octave has built-in functions for solving ordinary differential equations, and differential-algebraic equations. All solvers are based on reliable ODE routines written in Fortran.

24.1 Ordinary Differential Equations

The function `lsode` can be used to solve ODEs of the form

$$\frac{dx}{dt} = f(x,t)$$

using Hindmarsh's ODE solver LSODE.

`[x, istate, msg] = lsode (fcn, x_0, t)` [Built-in Function]
`[x, istate, msg] = lsode (fcn, x_0, t, t_crit)` [Built-in Function]

Ordinary Differential Equation (ODE) solver.

The set of differential equations to solve is

$$\frac{dx}{dt} = f(x,t)$$

with

$$x(t_0) = x_0$$

The solution is returned in the matrix *x*, with each row corresponding to an element of the vector *t*. The first element of *t* should be t_0 and should correspond to the initial state of the system *x_0*, so that the first row of the output is *x_0*.

The first argument, *fcn*, is a string, inline, or function handle that names the function f to call to compute the vector of right hand sides for the set of equations. The function must have the form

```
xdot = f (x, t)
```

in which *xdot* and *x* are vectors and *t* is a scalar.

If *fcn* is a two-element string array or a two-element cell array of strings, inline functions, or function handles, the first element names the function f described above, and the second element names a function to compute the Jacobian of f. The Jacobian function must have the form

```
jac = j (x, t)
```

in which *jac* is the matrix of partial derivatives

$$J = \frac{\partial f_i}{\partial x_j} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \cdots & \frac{\partial f_3}{\partial x_N} \end{bmatrix}$$

The second and third arguments specify the initial state of the system, x_0, and the initial value of the independent variable t_0.

The fourth argument is optional, and may be used to specify a set of times that the ODE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of *istate* will be 2 (consistent with the Fortran version of LSODE).

If the computation is not successful, *istate* will be something other than 2 and *msg* will contain additional information.

You can use the function `lsode_options` to set optional parameters for `lsode`.

See also: [daspk], page 576, [dassl], page 579, [dasrt], page 581.

`lsode_options ()` [Built-in Function]
`val = lsode_options (`*opt*`)` [Built-in Function]
`lsode_options (`*opt*`, `*val*`)` [Built-in Function]

Query or set options for the function `lsode`.

When called with no arguments, the names of all available options and their current values are displayed.

Given one argument, return the value of the option *opt*.

When called with two arguments, `lsode_options` sets the option *opt* to value *val*.

Options include

`"absolute tolerance"`
: Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector.

`"relative tolerance"`
: Relative tolerance parameter. Unlike the absolute tolerance, this parameter may only be a scalar.

  The local error test applied at each integration step is

```
abs (local error in x(i)) <= ...
    rtol * abs (y(i)) + atol(i)
```

`"integration method"`
: A string specifying the method of integration to use to solve the ODE system. Valid values are

  `"adams"`
  `"non-stiff"`
  : No Jacobian used (even if it is available).

  `"bdf"`
  `"stiff"`
  : Use stiff backward differentiation formula (BDF) method. If a function to compute the Jacobian is not supplied, `lsode` will compute a finite difference approximation of the Jacobian matrix.

`"initial step size"`
: The step size to be attempted on the first step (default is determined automatically).

"maximum order"
: Restrict the maximum order of the solution method. If using the Adams method, this option must be between 1 and 12. Otherwise, it must be between 1 and 5, inclusive.

"maximum step size"
: Setting the maximum stepsize will avoid passing over very large regions (default is not specified).

"minimum step size"
: The minimum absolute step size allowed (default is 0).

"step limit"
: Maximum number of steps allowed (default is 100000).

Here is an example of solving a set of three differential equations using `lsode`. Given the function

```
## oregonator differential equation
function xdot = f (x, t)

  xdot = zeros (3,1);

  xdot(1) = 77.27 * (x(2) - x(1)*x(2) + x(1) \
            - 8.375e-06*x(1)^2);
  xdot(2) = (x(3) - x(1)*x(2) - x(2)) / 77.27;
  xdot(3) = 0.161*(x(1) - x(3));

endfunction
```

and the initial condition `x0 = [ 4; 1.1; 4 ]`, the set of equations can be integrated using the command

```
t = linspace (0, 500, 1000);

y = lsode ("f", x0, t);
```

If you try this, you will see that the value of the result changes dramatically between $t = 0$ and 5, and again around $t = 305$. A more efficient set of output points might be

```
t = [0, logspace(-1, log10(303), 150), \
        logspace(log10(304), log10(500), 150)];
```

See Alan C. Hindmarsh, *ODEPACK, A Systematized Collection of ODE Solvers*, in Scientific Computing, R. S. Stepleman, editor, (1983) for more information about the inner workings of `lsode`.

An m-file for the differential equation used above is included with the Octave distribution in the examples directory under the name `oregonator.m`.

24.2 Differential-Algebraic Equations

The function `daspk` can be used to solve DAEs of the form

$$0 = f(\dot{x}, x, t), \qquad x(t = 0) = x_0, \dot{x}(t = 0) = \dot{x}_0$$

where $\dot{x} = \frac{dx}{dt}$ is the derivative of x. The equation is solved using Petzold's DAE solver DASPK.

`[x, xdot, istate, msg] = daspk (fcn, x_0, xdot_0, t, t_crit)` [Built-in Function]

Solve the set of differential-algebraic equations

$$0 = f(x, \dot{x}, t)$$

with

$$x(t_0) = x_0, \dot{x}(t_0) = \dot{x}_0$$

The solution is returned in the matrices *x* and *xdot*, with each row in the result matrices corresponding to one of the elements in the vector *t*. The first element of *t* should be t_0 and correspond to the initial state of the system *x_0* and its derivative *xdot_0*, so that the first row of the output *x* is *x_0* and the first row of the output *xdot* is *xdot_0*.

The first argument, *fcn*, is a string, inline, or function handle that names the function f to call to compute the vector of residuals for the set of equations. It must have the form

```
res = f (x, xdot, t)
```

in which *x*, *xdot*, and *res* are vectors, and *t* is a scalar.

If *fcn* is a two-element string array or a two-element cell array of strings, inline functions, or function handles, the first element names the function f described above, and the second element names a function to compute the modified Jacobian

$$J = \frac{\partial f}{\partial x} + c\frac{\partial f}{\partial \dot{x}}$$

The modified Jacobian function must have the form

```
jac = j (x, xdot, t, c)
```

The second and third arguments to `daspk` specify the initial condition of the states and their derivatives, and the fourth argument specifies a vector of output times at which the solution is desired, including the time corresponding to the initial condition.

The set of initial states and derivatives are not strictly required to be consistent. If they are not consistent, you must use the `daspk_options` function to provide additional information so that `daspk` can compute a consistent starting point.

The fifth argument is optional, and may be used to specify a set of times that the DAE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of *istate* will be greater than zero (consistent with the Fortran version of DASPK).

If the computation is not successful, the value of *istate* will be less than zero and *msg* will contain additional information.

You can use the function `daspk_options` to set optional parameters for `daspk`.

See also: [dassl], page 579.

`daspk_options ()` [Built-in Function]
`val = daspk_options (`*opt*`)` [Built-in Function]
`daspk_options (`*opt*, *val*`)` [Built-in Function]

Query or set options for the function `daspk`.

When called with no arguments, the names of all available options and their current values are displayed.

Given one argument, return the value of the option *opt*.

When called with two arguments, `daspk_options` sets the option *opt* to value *val*.

Options include

`"absolute tolerance"`
: Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the relative tolerance must also be a vector of the same length.

`"relative tolerance"`
: Relative tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the absolute tolerance must also be a vector of the same length.

The local error test applied at each integration step is

```
abs (local error in x(i))
     <= rtol(i) * abs (Y(i)) + atol(i)
```

`"compute consistent initial condition"`
: Denoting the differential variables in the state vector by '`Y_d`' and the algebraic variables by '`Y_a`', `ddaspk` can solve one of two initialization problems:

1. Given Y_d, calculate Y_a and Y'_d
2. Given Y', calculate Y.

In either case, initial values for the given components are input, and initial guesses for the unknown components must also be provided as input. Set this option to 1 to solve the first problem, or 2 to solve the second (the default is 0, so you must provide a set of initial conditions that are consistent).

If this option is set to a nonzero value, you must also set the `"algebraic variables"` option to declare which variables in the problem are algebraic.

`"use initial condition heuristics"`
: Set to a nonzero value to use the initial condition heuristics options described below.

`"initial condition heuristics"`
: A vector of the following parameters that can be used to control the initial condition calculation.

`MXNIT` Maximum number of Newton iterations (default is 5).

`MXNJ` Maximum number of Jacobian evaluations (default is 6).

`MXNH` Maximum number of values of the artificial stepsize parameter to be tried if the `"compute consistent initial condition"` option has been set to 1 (default is 5).

Note that the maximum total number of Newton iterations allowed is `MXNIT*MXNJ*MXNH` if the `"compute consistent initial condition"` option has been set to 1 and `MXNIT*MXNJ` if it is set to 2.

`LSOFF` Set to a nonzero value to disable the linesearch algorithm (default is 0).

`STPTOL` Minimum scaled step in linesearch algorithm (default is eps^(2/3)).

`EPINIT` Swing factor in the Newton iteration convergence test. The test is applied to the residual vector, premultiplied by the approximate Jacobian. For convergence, the weighted RMS norm of this vector (scaled by the error weights) must be less than `EPINIT*EPCON`, where `EPCON` = 0.33 is the analogous test constant used in the time steps. The default is `EPINIT` = 0.01.

`"print initial condition info"`
Set this option to a nonzero value to display detailed information about the initial condition calculation (default is 0).

`"exclude algebraic variables from error test"`
Set to a nonzero value to exclude algebraic variables from the error test. You must also set the `"algebraic variables"` option to declare which variables in the problem are algebraic (default is 0).

`"algebraic variables"`
A vector of the same length as the state vector. A nonzero element indicates that the corresponding element of the state vector is an algebraic variable (i.e., its derivative does not appear explicitly in the equation set).

This option is required by the `"compute consistent initial condition"` and `"exclude algebraic variables from error test"` options.

`"enforce inequality constraints"`
Set to one of the following values to enforce the inequality constraints specified by the `"inequality constraint types"` option (default is 0).

1. To have constraint checking only in the initial condition calculation.
2. To enforce constraint checking during the integration.
3. To enforce both options 1 and 2.

`"inequality constraint types"`
A vector of the same length as the state specifying the type of inequality constraint. Each element of the vector corresponds to an element of the state and should be assigned one of the following codes

-2 Less than zero.

-1	Less than or equal to zero.
0	Not constrained.
1	Greater than or equal to zero.
2	Greater than zero.

This option only has an effect if the `"enforce inequality constraints"` option is nonzero.

`"initial step size"`
: Differential-algebraic problems may occasionally suffer from severe scaling difficulties on the first step. If you know a great deal about the scaling of your problem, you can help to alleviate this problem by specifying an initial stepsize (default is computed automatically).

`"maximum order"`
: Restrict the maximum order of the solution method. This option must be between 1 and 5, inclusive (default is 5).

`"maximum step size"`
: Setting the maximum stepsize will avoid passing over very large regions (default is not specified).

Octave also includes DASSL, an earlier version of DASPK, and DASRT, which can be used to solve DAEs with constraints (stopping conditions).

`[x, xdot, istate, msg] = dassl (fcn, x_0, xdot_0, t, t_crit)` [Built-in Function]

Solve the set of differential-algebraic equations

$$0 = f(x, \dot{x}, t)$$

with

$$x(t_0) = x_0, \dot{x}(t_0) = \dot{x}_0$$

The solution is returned in the matrices *x* and *xdot*, with each row in the result matrices corresponding to one of the elements in the vector *t*. The first element of *t* should be t_0 and correspond to the initial state of the system *x_0* and its derivative *xdot_0*, so that the first row of the output *x* is *x_0* and the first row of the output *xdot* is *xdot_0*.

The first argument, *fcn*, is a string, inline, or function handle that names the function f to call to compute the vector of residuals for the set of equations. It must have the form

```
res = f (x, xdot, t)
```

in which *x*, *xdot*, and *res* are vectors, and *t* is a scalar.

If *fcn* is a two-element string array or a two-element cell array of strings, inline functions, or function handles, the first element names the function f described above, and the second element names a function to compute the modified Jacobian

$$J = \frac{\partial f}{\partial x} + c\frac{\partial f}{\partial \dot{x}}$$

The modified Jacobian function must have the form

```
jac = j (x, xdot, t, c)
```

The second and third arguments to `dassl` specify the initial condition of the states and their derivatives, and the fourth argument specifies a vector of output times at which the solution is desired, including the time corresponding to the initial condition.

The set of initial states and derivatives are not strictly required to be consistent. In practice, however, DASSL is not very good at determining a consistent set for you, so it is best if you ensure that the initial values result in the function evaluating to zero.

The fifth argument is optional, and may be used to specify a set of times that the DAE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of *istate* will be greater than zero (consistent with the Fortran version of DASSL).

If the computation is not successful, the value of *istate* will be less than zero and *msg* will contain additional information.

You can use the function `dassl_options` to set optional parameters for `dassl`.

See also: [daspk], page 576, [dasrt], page 581, [lsode], page 573.

`dassl_options ()` [Built-in Function]
`val = dassl_options (`*opt*`)` [Built-in Function]
`dassl_options (`*opt*, *val*`)` [Built-in Function]

Query or set options for the function `dassl`.

When called with no arguments, the names of all available options and their current values are displayed.

Given one argument, return the value of the option *opt*.

When called with two arguments, `dassl_options` sets the option *opt* to value *val*.

Options include

`"absolute tolerance"`
: Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the relative tolerance must also be a vector of the same length.

`"relative tolerance"`
: Relative tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the absolute tolerance must also be a vector of the same length.

  The local error test applied at each integration step is

```
abs (local error in x(i))
     <= rtol(i) * abs (Y(i)) + atol(i)
```

`"compute consistent initial condition"`
: If nonzero, `dassl` will attempt to compute a consistent set of initial conditions. This is generally not reliable, so it is best to provide a consistent set and leave this option set to zero.

`"enforce nonnegativity constraints"`
: If you know that the solutions to your equations will always be nonnegative, it may help to set this parameter to a nonzero value. However, it is probably best to try leaving this option set to zero first, and only setting it to a nonzero value if that doesn't work very well.

`"initial step size"`
: Differential-algebraic problems may occasionally suffer from severe scaling difficulties on the first step. If you know a great deal about the scaling of your problem, you can help to alleviate this problem by specifying an initial stepsize.

`"maximum order"`
: Restrict the maximum order of the solution method. This option must be between 1 and 5, inclusive.

`"maximum step size"`
: Setting the maximum stepsize will avoid passing over very large regions (default is not specified).

`"step limit"`
: Maximum number of integration steps to attempt on a single call to the underlying Fortran code.

`[`*x*`,` *xdot*`,` *t_out*`,` *istat*`,` *msg*`] = dasrt (`*fcn*, [], *x_0*, *xdot_0*, *t*`)` [Built-in Function]
`... = dasrt (`*fcn*, *g*, *x_0*, *xdot_0*, *t*`)` [Built-in Function]
`... = dasrt (`*fcn*, [], *x_0*, *xdot_0*, *t*, *t_crit*`)` [Built-in Function]
`... = dasrt (`*fcn*, *g*, *x_0*, *xdot_0*, *t*, *t_crit*`)` [Built-in Function]

Solve the set of differential-algebraic equations

$$0 = f(x, \dot{x}, t)$$

with

$$x(t_0) = x_0, \dot{x}(t_0) = \dot{x}_0$$

with functional stopping criteria (root solving).

The solution is returned in the matrices *x* and *xdot*, with each row in the result matrices corresponding to one of the elements in the vector *t_out*. The first element of *t* should be t_0 and correspond to the initial state of the system *x_0* and its derivative *xdot_0*, so that the first row of the output *x* is *x_0* and the first row of the output *xdot* is *xdot_0*.

The vector *t* provides an upper limit on the length of the integration. If the stopping condition is met, the vector *t_out* will be shorter than *t*, and the final element of *t_out* will be the point at which the stopping condition was met, and may not correspond to any element of the vector *t*.

The first argument, *fcn*, is a string, inline, or function handle that names the function f to call to compute the vector of residuals for the set of equations. It must have the form

```
res = f (x, xdot, t)
```

in which *x*, *xdot*, and *res* are vectors, and *t* is a scalar.

If *fcn* is a two-element string array or a two-element cell array of strings, inline functions, or function handles, the first element names the function f described above, and the second element names a function to compute the modified Jacobian

$$J = \frac{\partial f}{\partial x} + c\frac{\partial f}{\partial \dot{x}}$$

The modified Jacobian function must have the form

```
jac = j (x, xdot, t, c)
```

The optional second argument names a function that defines the constraint functions whose roots are desired during the integration. This function must have the form

```
g_out = g (x, t)
```

and return a vector of the constraint function values. If the value of any of the constraint functions changes sign, DASRT will attempt to stop the integration at the point of the sign change.

If the name of the constraint function is omitted, `dasrt` solves the same problem as `daspk` or `dassl`.

Note that because of numerical errors in the constraint functions due to round-off and integration error, DASRT may return false roots, or return the same root at two or more nearly equal values of T. If such false roots are suspected, the user should consider smaller error tolerances or higher precision in the evaluation of the constraint functions.

If a root of some constraint function defines the end of the problem, the input to DASRT should nevertheless allow integration to a point slightly past that root, so that DASRT can locate the root by interpolation.

The third and fourth arguments to `dasrt` specify the initial condition of the states and their derivatives, and the fourth argument specifies a vector of output times at which the solution is desired, including the time corresponding to the initial condition.

The set of initial states and derivatives are not strictly required to be consistent. In practice, however, DASSL is not very good at determining a consistent set for you, so it is best if you ensure that the initial values result in the function evaluating to zero.

The sixth argument is optional, and may be used to specify a set of times that the DAE solver should not integrate past. It is useful for avoiding difficulties with singularities and points where there is a discontinuity in the derivative.

After a successful computation, the value of *istate* will be greater than zero (consistent with the Fortran version of DASSL).

If the computation is not successful, the value of *istate* will be less than zero and *msg* will contain additional information.

You can use the function `dasrt_options` to set optional parameters for `dasrt`.

See also: [dasrt_options], page 583, [daspk], page 576, [dasrt], page 581, [lsode], page 573.

`dasrt_options ()` [Built-in Function]
`val = dasrt_options (`*opt*`)` [Built-in Function]
`dasrt_options (`*opt*`, `*val*`)` [Built-in Function]

Query or set options for the function `dasrt`.

When called with no arguments, the names of all available options and their current values are displayed.

Given one argument, return the value of the option *opt*.

When called with two arguments, `dasrt_options` sets the option *opt* to value *val*.

Options include

`"absolute tolerance"`
: Absolute tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the relative tolerance must also be a vector of the same length.

`"relative tolerance"`
: Relative tolerance. May be either vector or scalar. If a vector, it must match the dimension of the state vector, and the absolute tolerance must also be a vector of the same length.

  The local error test applied at each integration step is

```
abs (local error in x(i)) <= ...
    rtol(i) * abs (Y(i)) + atol(i)
```

`"initial step size"`
: Differential-algebraic problems may occasionally suffer from severe scaling difficulties on the first step. If you know a great deal about the scaling of your problem, you can help to alleviate this problem by specifying an initial stepsize.

`"maximum order"`
: Restrict the maximum order of the solution method. This option must be between 1 and 5, inclusive.

`"maximum step size"`
: Setting the maximum stepsize will avoid passing over very large regions.

`"step limit"`
: Maximum number of integration steps to attempt on a single call to the underlying Fortran code.

See K. E. Brenan, et al., *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, North-Holland (1989) for more information about the implementation of DASSL.

25 Optimization

Octave comes with support for solving various kinds of optimization problems. Specifically Octave can solve problems in Linear Programming, Quadratic Programming, Nonlinear Programming, and Linear Least Squares Minimization.

25.1 Linear Programming

Octave can solve Linear Programming problems using the `glpk` function. That is, Octave can solve

$$\min_x c^T x$$

subject to the linear constraints $Ax = b$ where $x \geq 0$.

The `glpk` function also supports variations of this problem.

[*xopt*, *fmin*, *errnum*, *extra*] = glpk (*c*, *A*, *b*, *lb*, *ub*, *ctype*, *vartype*, *sense*, *param*) [Function File]

Solve a linear program using the GNU GLPK library.

Given three arguments, `glpk` solves the following standard LP:

$$\min_x C^T x$$

subject to

$$Ax = b \qquad x \geq 0$$

but may also solve problems of the form

$$[\min_x \mid \max_x] C^T x$$

subject to

$$Ax[= \mid \leq \mid \geq]b \qquad LB \leq x \leq UB$$

Input arguments:

c A column array containing the objective function coefficients.

A A matrix containing the constraints coefficients.

b A column array containing the right-hand side value for each constraint in the constraint matrix.

lb An array containing the lower bound on each of the variables. If *lb* is not supplied, the default lower bound for the variables is zero.

ub An array containing the upper bound on each of the variables. If *ub* is not supplied, the default upper bound is assumed to be infinite.

ctype An array of characters containing the sense of each constraint in the constraint matrix. Each element of the array may be one of the following values

"F" A free (unbounded) constraint (the constraint is ignored).

"U" An inequality constraint with an upper bound (`A(i,:)*x <= b(i)`).

"S" An equality constraint (`A(i,:)*x = b(i)`).

"L" An inequality with a lower bound (`A(i,:)*x >= b(i)`).

"D" An inequality constraint with both upper and lower bounds (`A(i,:)*x >= -b(i)`) *and* (`A(i,:)*x <= b(i)`).

vartype A column array containing the types of the variables.

"C" A continuous variable.

"I" An integer variable.

sense If *sense* is 1, the problem is a minimization. If *sense* is -1, the problem is a maximization. The default value is 1.

param A structure containing the following parameters used to define the behavior of solver. Missing elements in the structure take on default values, so you only need to set the elements that you wish to change from the default.

Integer parameters:

`msglev (default: 1)`
Level of messages output by solver routines:

0 (`GLP_MSG_OFF`)
No output.

1 (`GLP_MSG_ERR`)
Error and warning messages only.

2 (`GLP_MSG_ON`)
Normal output.

3 (`GLP_MSG_ALL`)
Full output (includes informational messages).

`scale (default: 16)`
Scaling option. The values can be combined with the bitwise OR operator and may be the following:

1 (`GLP_SF_GM`)
Geometric mean scaling.

16 (`GLP_SF_EQ`)
Equilibration scaling.

32 (`GLP_SF_2N`)
Round scale factors to power of two.

64 (`GLP_SF_SKIP`)
Skip if problem is well scaled.

Alternatively, a value of 128 (`GLP_SF_AUTO`) may be also specified, in which case the routine chooses the scaling options automatically.

`dual (default: 1)`
: Simplex method option:

  1 (`GLP_PRIMAL`)
  : Use two-phase primal simplex.

  2 (`GLP_DUALP`)
  : Use two-phase dual simplex, and if it fails, switch to the primal simplex.

  3 (`GLP_DUAL`)
  : Use two-phase dual simplex.

`price (default: 34)`
: Pricing option (for both primal and dual simplex):

  17 (`GLP_PT_STD`)
  : Textbook pricing.

  34 (`GLP_PT_PSE`)
  : Steepest edge pricing.

`itlim (default: intmax)`
: Simplex iterations limit. It is decreased by one each time when one simplex iteration has been performed, and reaching zero value signals the solver to stop the search.

`outfrq (default: 200)`
: Output frequency, in iterations. This parameter specifies how frequently the solver sends information about the solution to the standard output.

`branch (default: 4)`
: Branching technique option (for MIP only):

  1 (`GLP_BR_FFV`)
  : First fractional variable.

  2 (`GLP_BR_LFV`)
  : Last fractional variable.

  3 (`GLP_BR_MFV`)
  : Most fractional variable.

  4 (`GLP_BR_DTH`)
  : Heuristic by Driebeck and Tomlin.

  5 (`GLP_BR_PCH`)
  : Hybrid pseudocost heuristic.

`btrack (default: 4)`
: Backtracking technique option (for MIP only):

  `1 (GLP_BT_DFS)`
  : Depth first search.

  `2 (GLP_BT_BFS)`
  : Breadth first search.

  `3 (GLP_BT_BLB)`
  : Best local bound.

  `4 (GLP_BT_BPH)`
  : Best projection heuristic.

`presol (default: 1)`
: If this flag is set, the simplex solver uses the built-in LP presolver. Otherwise the LP presolver is not used.

`lpsolver (default: 1)`
: Select which solver to use. If the problem is a MIP problem this flag will be ignored.

  1 Revised simplex method.

  2 Interior point method.

`rtest (default: 34)`
: Ratio test technique:

  `17 (GLP_RT_STD)`
  : Standard ("textbook").

  `34 (GLP_RT_HAR)`
  : Harris' two-pass ratio test.

`tmlim (default: intmax)`
: Searching time limit, in milliseconds.

`outdly (default: 0)`
: Output delay, in seconds. This parameter specifies how long the solver should delay sending information about the solution to the standard output.

`save (default: 0)`
: If this parameter is nonzero, save a copy of the problem in CPLEX LP format to the file `"outpb.lp"`. There is currently no way to change the name of the output file.

Real parameters:

`tolbnd (default: 1e-7)`
: Relative tolerance used to check if the current basic solution is primal feasible. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`toldj (default: 1e-7)`
: Absolute tolerance used to check if the current basic solution is dual feasible. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`tolpiv (default: 1e-10)`
: Relative tolerance used to choose eligible pivotal elements of the simplex table. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`objll (default: -DBL_MAX)`
: Lower limit of the objective function. If the objective function reaches this limit and continues decreasing, the solver stops the search. This parameter is used in the dual simplex method only.

`objul (default: +DBL_MAX)`
: Upper limit of the objective function. If the objective function reaches this limit and continues increasing, the solver stops the search. This parameter is used in the dual simplex only.

`tolint (default: 1e-5)`
: Relative tolerance used to check if the current basic solution is integer feasible. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

`tolobj (default: 1e-7)`
: Relative tolerance used to check if the value of the objective function is not better than in the best known integer feasible solution. It is not recommended that you change this parameter unless you have a detailed understanding of its purpose.

Output values:

xopt The optimizer (the value of the decision variables at the optimum).

fopt The optimum value of the objective function.

errnum Error code.

0
: No error.

1 (`GLP_EBADB`)
: Invalid basis.

2 (`GLP_ESING`)
: Singular matrix.

3 (`GLP_ECOND`)
: Ill-conditioned matrix.

4 (`GLP_EBOUND`)
Invalid bounds.

5 (`GLP_EFAIL`)
Solver failed.

6 (`GLP_EOBJLL`)
Objective function lower limit reached.

7 (`GLP_EOBJUL`)
Objective function upper limit reached.

8 (`GLP_EITLIM`)
Iterations limit exhausted.

9 (`GLP_ETMLIM`)
Time limit exhausted.

10 (`GLP_ENOPFS`)
No primal feasible solution.

11 (`GLP_ENODFS`)
No dual feasible solution.

12 (`GLP_EROOT`)
Root LP optimum not provided.

13 (`GLP_ESTOP`)
Search terminated by application.

14 (`GLP_EMIPGAP`)
Relative MIP gap tolerance reached.

15 (`GLP_ENOFEAS`)
No primal/dual feasible solution.

16 (`GLP_ENOCVG`)
No convergence.

17 (`GLP_EINSTAB`)
Numerical instability.

18 (`GLP_EDATA`)
Invalid data.

19 (`GLP_ERANGE`)
Result out of range.

extra A data structure containing the following fields:

`lambda` Dual variables.

`redcosts` Reduced Costs.

`time` Time (in seconds) used for solving LP/MIP problem.

`status` Status of the optimization.

1 (`GLP_UNDEF`)
Solution status is undefined.

2 (GLP_FEAS)
: Solution is feasible.

3 (GLP_INFEAS)
: Solution is infeasible.

4 (GLP_NOFEAS)
: Problem has no feasible solution.

5 (GLP_OPT)
: Solution is optimal.

6 (GLP_UNBND)
: Problem has no unbounded solution.

Example:

```
c = [10, 6, 4]';
A = [ 1, 1, 1;
     10, 4, 5;
      2, 2, 6];
b = [100, 600, 300]';
lb = [0, 0, 0]';
ub = [];
ctype = "UUU";
vartype = "CCC";
s = -1;

param.msglev = 1;
param.itlim = 100;

[xmin, fmin, status, extra] = ...
   glpk (c, A, b, lb, ub, ctype, vartype, s, param);
```

25.2 Quadratic Programming

Octave can also solve Quadratic Programming problems, this is

$$\min_x \frac{1}{2} x^T H x + x^T q$$

subject to

$$Ax = b \qquad lb \leq x \leq ub \qquad A_{lb} \leq A_{in} \leq A_{ub}$$

[*x*, *obj*, *info*, *lambda*] = qp (*x0*, *H*) [Function File]
[*x*, *obj*, *info*, *lambda*] = qp (*x0*, *H*, *q*) [Function File]
[*x*, *obj*, *info*, *lambda*] = qp (*x0*, *H*, *q*, *A*, *b*) [Function File]
[*x*, *obj*, *info*, *lambda*] = qp (*x0*, *H*, *q*, *A*, *b*, *lb*, *ub*) [Function File]
[*x*, *obj*, *info*, *lambda*] = qp (*x0*, *H*, *q*, *A*, *b*, *lb*, *ub*, *A_lb*, *A_in*, *A_ub*) [Function File]
[*x*, *obj*, *info*, *lambda*] = qp (..., *options*) [Function File]

Solve a quadratic program (QP).

Solve the quadratic program defined by

$$\min_x \frac{1}{2} x^T H x + x^T q$$

subject to

$$Ax = b \qquad lb \leq x \leq ub \qquad A_{lb} \leq A_{in} \leq A_{ub}$$

using a null-space active-set method.

Any bound (*A*, *b*, *lb*, *ub*, *A_lb*, *A_ub*) may be set to the empty matrix (`[]`) if not present. If the initial guess is feasible the algorithm is faster.

options An optional structure containing the following parameter(s) used to define the behavior of the solver. Missing elements in the structure take on default values, so you only need to set the elements that you wish to change from the default.

`MaxIter (default: 200)`
Maximum number of iterations.

info Structure containing run-time information about the algorithm. The following fields are defined:

`solveiter`
The number of iterations required to find the solution.

`info` An integer indicating the status of the solution.

0 The problem is feasible and convex. Global solution found.

1 The problem is not convex. Local solution found.

2 The problem is not convex and unbounded.

3 Maximum number of iterations reached.

6 The problem is infeasible.

x = **pqpnonneg** (*c*, *d*) [Function File]
x = **pqpnonneg** (*c*, *d*, *x0*) [Function File]
[*x*, *minval*] = **pqpnonneg** (...) [Function File]
[*x*, *minval*, *exitflag*] = **pqpnonneg** (...) [Function File]
[*x*, *minval*, *exitflag*, *output*] = **pqpnonneg** (...) [Function File]
[*x*, *minval*, *exitflag*, *output*, *lambda*] = **pqpnonneg** (...) [Function File]

Minimize `1/2*x'*c*x + d'*x` subject to `x >= 0`.

c ## and *d* must be real, and *c* must be symmetric and positive definite.

x0 is an optional initial guess for *x*.

Outputs:

- minval

  The minimum attained model value, 1/2*xmin'*c*xmin + d'*xmin

- exitflag

  An indicator of convergence. 0 indicates that the iteration count was exceeded, and therefore convergence was not reached; >0 indicates that the algorithm converged. (The algorithm is stable and will converge given enough iterations.)

- output

  A structure with two fields:

  - `"algorithm"`: The algorithm used (`"nnls"`)
  - `"iterations"`: The number of iterations taken.

- lambda

  Not implemented.

See also: [optimset], page 597, [lsqnonneg], page 596, [qp], page 591.

25.3 Nonlinear Programming

Octave can also perform general nonlinear minimization using a successive quadratic programming solver.

`[x, obj, info, iter, nf, lambda] = sqp (x0, phi)` [Function File]
`[...] = sqp (x0, phi, g)` [Function File]
`[...] = sqp (x0, phi, g, h)` [Function File]
`[...] = sqp (x0, phi, g, h, lb, ub)` [Function File]
`[...] = sqp (x0, phi, g, h, lb, ub, maxiter)` [Function File]
`[...] = sqp (x0, phi, g, h, lb, ub, maxiter, tol)` [Function File]

Minimize an objective function using sequential quadratic programming (SQP).

Solve the nonlinear program

$$\min_x \phi(x)$$

subject to

$$g(x) = 0 \qquad h(x) \geq 0 \qquad lb \leq x \leq ub$$

using a sequential quadratic programming method.

The first argument is the initial guess for the vector *x0*.

The second argument is a function handle pointing to the objective function *phi*. The objective function must accept one vector argument and return a scalar.

The second argument may also be a 2- or 3-element cell array of function handles. The first element should point to the objective function, the second should point to a function that computes the gradient of the objective function, and the third should point to a function that computes the Hessian of the objective function. If the gradient function is not supplied, the gradient is computed by finite differences. If the Hessian function is not supplied, a BFGS update formula is used to approximate the Hessian.

When supplied, the gradient function `phi{2}` must accept one vector argument and return a vector. When supplied, the Hessian function `phi{3}` must accept one vector argument and return a matrix.

The third and fourth arguments g and h are function handles pointing to functions that compute the equality constraints and the inequality constraints, respectively. If the problem does not have equality (or inequality) constraints, then use an empty matrix ([]) for g (or h). When supplied, these equality and inequality constraint functions must accept one vector argument and return a vector.

The third and fourth arguments may also be 2-element cell arrays of function handles. The first element should point to the constraint function and the second should point to a function that computes the gradient of the constraint function:

$$\left(\frac{\partial f(x)}{\partial x_1}, \frac{\partial f(x)}{\partial x_2}, \ldots, \frac{\partial f(x)}{\partial x_N}\right)^T$$

The fifth and sixth arguments, *lb* and *ub*, contain lower and upper bounds on *x*. These must be consistent with the equality and inequality constraints *g* and *h*. If the arguments are vectors then *x*(i) is bound by *lb*(i) and *ub*(i). A bound can also be a scalar in which case all elements of *x* will share the same bound. If only one bound (lb, ub) is specified then the other will default to (-*realmax*, +*realmax*).

The seventh argument *maxiter* specifies the maximum number of iterations. The default value is 100.

The eighth argument *tol* specifies the tolerance for the stopping criteria. The default value is `sqrt (eps)`.

The value returned in *info* may be one of the following:

101 The algorithm terminated normally. All constraints meet the specified tolerance.

102 The BFGS update failed.

103 The maximum number of iterations was reached.

104 The stepsize has become too small, i.e., Δx, is less than `tol * norm (x)`.

An example of calling `sqp`:

```
function r = g (x)
  r = [ sumsq(x)-10;
        x(2)*x(3)-5*x(4)*x(5);
        x(1)^3+x(2)^3+1 ];
endfunction

function obj = phi (x)
  obj = exp (prod (x)) - 0.5*(x(1)^3+x(2)^3+1)^2;
endfunction

x0 = [-1.8; 1.7; 1.9; -0.8; -0.8];

[x, obj, info, iter, nf, lambda] = sqp (x0, @phi, @g, [])

x =
```

```
  -1.71714
   1.59571
   1.82725
  -0.76364
  -0.76364

obj = 0.053950
info = 101
iter = 8
nf = 10
lambda =

  -0.0401627
   0.0379578
  -0.0052227
```

See also: [qp], page 591.

25.4 Linear Least Squares

Octave also supports linear least squares minimization. That is, Octave can find the parameter b such that the model $y = xb$ fits data (x, y) as well as possible, assuming zero-mean Gaussian noise. If the noise is assumed to be isotropic the problem can be solved using the '\' or '/' operators, or the `ols` function. In the general case where the noise is assumed to be anisotropic the `gls` is needed.

`[beta, sigma, r] = ols (y, x)` [Function File]

Ordinary least squares estimation.

OLS applies to the multivariate model $y = xb + e$ with $\bar{e} = 0$, and $\mathrm{cov}(\mathrm{vec}(e)) = \mathrm{kron}(s, I)$ where y is a $t \times p$ matrix, x is a $t \times k$ matrix, b is a $k \times p$ matrix, and e is a $t \times p$ matrix.

Each row of y and x is an observation and each column a variable.

The return values *beta*, *sigma*, and *r* are defined as follows.

beta The OLS estimator for b. *beta* is calculated directly via $(x^T x)^{-1} x^T y$ if the matrix $x^T x$ is of full rank. Otherwise, `beta = pinv (x) * y` where `pinv (x)` denotes the pseudoinverse of *x*.

sigma The OLS estimator for the matrix *s*,

```
sigma = (y-x*beta)'
  * (y-x*beta)
  / (t-rank(x))
```

r The matrix of OLS residuals, `r = y - x*beta`.

See also: [gls], page 595, [pinv], page 471.

`[beta, v, r] = gls (y, x, o)` [Function File]

Generalized least squares model.

Perform a generalized least squares estimation for the multivariate model $y = xb + e$ with $\bar{e} = 0$ and $\mathrm{cov}(\mathrm{vec}(e)) = (s^2)o$, where y is a $t \times p$ matrix, x is a $t \times k$ matrix, b is a $k \times p$ matrix, e is a $t \times p$ matrix, and o is a $tp \times tp$ matrix.

Each row of y and x is an observation and each column a variable. The return values *beta*, *v*, and *r* are defined as follows.

beta The GLS estimator for b.

v The GLS estimator for s^2.

r The matrix of GLS residuals, $r = y - x * beta$.

See also: [ols], page 595.

`x = lsqnonneg (c, d)` [Function File]
`x = lsqnonneg (c, d, x0)` [Function File]
`x = lsqnonneg (c, d, x0, options)` [Function File]
`[x, resnorm] = lsqnonneg (...)` [Function File]
`[x, resnorm, residual] = lsqnonneg (...)` [Function File]
`[x, resnorm, residual, exitflag] = lsqnonneg (...)` [Function File]
`[x, resnorm, residual, exitflag, output] = lsqnonneg (...)` [Function File]
`[x, resnorm, residual, exitflag, output, lambda] = lsqnonneg (...)` [Function File]

Minimize `norm (c*x - d)` subject to `x >= 0`.

c and *d* must be real.

x0 is an optional initial guess for *x*.

Currently, `lsqnonneg` recognizes these options: `"MaxIter"`, `"TolX"`. For a description of these options, see [optimset], page 597.

Outputs:

- resnorm

  The squared 2-norm of the residual: norm (*c***x*-*d*)^2
- residual

  The residual: *d*-*c***x*
- exitflag

  An indicator of convergence. 0 indicates that the iteration count was exceeded, and therefore convergence was not reached; >0 indicates that the algorithm converged. (The algorithm is stable and will converge given enough iterations.)
- output

  A structure with two fields:
  - `"algorithm"`: The algorithm used (`"nnls"`)
  - `"iterations"`: The number of iterations taken.
- lambda

  Not implemented.

See also: [optimset], page 597, [pqpnonneg], page 592, [lscov], page 597.

`x = lscov (A, b)` [Function File]
`x = lscov (A, b, V)` [Function File]
`x = lscov (A, b, V, alg)` [Function File]
`[x, stdx, mse, S] = lscov (...)` [Function File]

Compute a generalized linear least squares fit.

Estimate *x* under the model $b = Ax + w$, where the noise *w* is assumed to follow a normal distribution with covariance matrix $\sigma^2 V$.

If the size of the coefficient matrix *A* is n-by-p, the size of the vector/array of constant terms *b* must be n-by-k.

The optional input argument *V* may be a n-by-1 vector of positive weights (inverse variances), or a n-by-n symmetric positive semidefinite matrix representing the covariance of *b*. If *V* is not supplied, the ordinary least squares solution is returned.

The *alg* input argument, a guidance on solution method to use, is currently ignored.

Besides the least-squares estimate matrix *x* (p-by-k), the function also returns *stdx* (p-by-k), the error standard deviation of estimated *x*; *mse* (k-by-1), the estimated data error covariance scale factors (σ^2); and *S* (p-by-p, or p-by-p-by-k if k > 1), the error covariance of *x*.

Reference: Golub and Van Loan (1996), *Matrix Computations (3rd Ed.)*, Johns Hopkins, Section 5.6.3

See also: [ols], page 595, [gls], page 595, [lsqnonneg], page 596.

`optimset ()` [Function File]
`options = optimset ()` [Function File]
`options = optimset (par, val, ...)` [Function File]
`options = optimset (old, par, val, ...)` [Function File]
`options = optimset (old, new)` [Function File]

Create options structure for optimization functions.

When called without any input or output arguments, `optimset` prints a list of all valid optimization parameters.

When called with one output and no inputs, return an options structure with all valid option parameters initialized to `[]`.

When called with a list of parameter/value pairs, return an options structure with only the named parameters initialized.

When the first input is an existing options structure *old*, the values are updated from either the *par*/*val* list or from the options structure *new*.

Valid parameters are:

AutoScaling

ComplexEqn

Display — Request verbose display of results from optimizations. Values are:

- `"off"` [default]
  No display.
- `"iter"` Display intermediate results for every loop iteration.
- `"final"` Display the result of the final loop iteration.

"notify" Display the result of the final loop iteration if the function has failed to converge.

FinDiffType

FunValCheck
: When enabled, display an error if the objective function returns an invalid value (a complex number, NaN, or Inf). Must be set to `"on"` or `"off"` [default]. Note: the functions `fzero` and `fminbnd` correctly handle Inf values and only complex values or NaN will cause an error in this case.

GradObj
: When set to `"on"`, the function to be minimized must return a second argument which is the gradient, or first derivative, of the function at the point *x*. If set to `"off"` [default], the gradient is computed via finite differences.

Jacobian
: When set to `"on"`, the function to be minimized must return a second argument which is the Jacobian, or first derivative, of the function at the point *x*. If set to `"off"` [default], the Jacobian is computed via finite differences.

MaxFunEvals
: Maximum number of function evaluations before optimization stops. Must be a positive integer.

MaxIter
: Maximum number of algorithm iterations before optimization stops. Must be a positive integer.

OutputFcn
: A user-defined function executed once per algorithm iteration.

TolFun
: Termination criterion for the function output. If the difference in the calculated objective function between one algorithm iteration and the next is less than `TolFun` the optimization stops. Must be a positive scalar.

TolX
: Termination criterion for the function input. If the difference in *x*, the current search point, between one algorithm iteration and the next is less than `TolX` the optimization stops. Must be a positive scalar.

TypicalX

Updating

See also: [optimget], page 598.

`optimget (`*`options`*`, `*`parname`*`)` [Function File]

`optimget (`*`options`*`, `*`parname`*`, `*`default`*`)` [Function File]

Return the specific option *parname* from the optimization options structure *options* created by `optimset`.

If *parname* is not defined then return *default* if supplied, otherwise return an empty matrix.

See also: [optimset], page 597.

26 Statistics

Octave has support for various statistical methods. This includes basic descriptive statistics, probability distributions, statistical tests, random number generation, and much more.

The functions that analyze data all assume that multi-dimensional data is arranged in a matrix where each row is an observation, and each column is a variable. Thus, the matrix defined by

```
a = [ 0.9, 0.7;
      0.1, 0.1;
      0.5, 0.4 ];
```

contains three observations from a two-dimensional distribution. While this is the default data arrangement, most functions support different arrangements.

It should be noted that the statistics functions don't test for data containing NaN, NA, or Inf. These values need to be detected and dealt with explicitly. See [isnan], page 406, [isna], page 43, [isinf], page 406, [isfinite], page 407.

26.1 Descriptive Statistics

One principal goal of descriptive statistics is to represent the essence of a large data set concisely. Octave provides the mean, median, and mode functions which all summarize a data set with just a single number corresponding to the central tendency of the data.

`mean` (*x*) [Function File]
`mean` (*x*, *dim*) [Function File]
`mean` (*x*, *opt*) [Function File]
`mean` (*x*, *dim*, *opt*) [Function File]

Compute the mean of the elements of the vector *x*.

The mean is defined as

$$\text{mean}(x) = \bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

If *x* is a matrix, compute the mean for each column and return them in a row vector.

If the optional argument *dim* is given, operate along this dimension.

The optional argument *opt* selects the type of mean to compute. The following options are recognized:

`"a"` Compute the (ordinary) arithmetic mean. [default]

`"g"` Compute the geometric mean.

`"h"` Compute the harmonic mean.

Both *dim* and *opt* are optional. If both are supplied, either may appear first.

See also: [median], page 600, [mode], page 600.

`median (x)` [Function File]
`median (x, dim)` [Function File]

Compute the median value of the elements of the vector *x*.

When the elements of *x* are sorted, the median is defined as

$$\operatorname{median}(x) = \begin{cases} x(\lceil N/2 \rceil), & N \text{ odd;} \\ (x(N/2) + x(N/2+1))/2, & N \text{ even.} \end{cases}$$

If *x* is a matrix, compute the median value for each column and return them in a row vector.

If the optional *dim* argument is given, operate along this dimension.

See also: [mean], page 599, [mode], page 600.

`mode (x)` [Function File]
`mode (x, dim)` [Function File]
`[m, f, c] = mode (...)` [Function File]

Compute the most frequently occurring value in a dataset (mode).

`mode` determines the frequency of values along the first non-singleton dimension and returns the value with the highest frequency. If two, or more, values have the same frequency `mode` returns the smallest.

If the optional argument *dim* is given, operate along this dimension.

The return variable *f* is the number of occurrences of the mode in the dataset.

The cell array *c* contains all of the elements with the maximum frequency.

See also: [mean], page 599, [median], page 600.

Using just one number, such as the mean, to represent an entire data set may not give an accurate picture of the data. One way to characterize the fit is to measure the dispersion of the data. Octave provides several functions for measuring dispersion.

`range (x)` [Function File]
`range (x, dim)` [Function File]

Return the range, i.e., the difference between the maximum and the minimum of the input data.

If *x* is a vector, the range is calculated over the elements of *x*. If *x* is a matrix, the range is calculated over each column of *x*.

If the optional argument *dim* is given, operate along this dimension.

The range is a quickly computed measure of the dispersion of a data set, but is less accurate than `iqr` if there are outlying data points.

See also: [iqr], page 600, [std], page 601.

`iqr (x)` [Function File]
`iqr (x, dim)` [Function File]

Return the interquartile range, i.e., the difference between the upper and lower quartile of the input data.

If *x* is a matrix, do the above for first non-singleton dimension of *x*.

If the optional argument *dim* is given, operate along this dimension.

As a measure of dispersion, the interquartile range is less affected by outliers than either `range` or `std`.

See also: [range], page 600, [std], page 601.

`meansq` (*x*) [Function File]
`meansq` (*x*, *dim*) [Function File]

Compute the mean square of the elements of the vector *x*.

The mean square is defined as

$$\text{meansq}(x) = \frac{\sum_{i=1}^{N} {x_i}^2}{N}$$

where $\bar{x}$ is the mean value of x. For matrix arguments, return a row vector containing the mean square of each column.

If the optional argument *dim* is given, operate along this dimension.

See also: [var], page 601, [std], page 601, [moment], page 603.

`std` (*x*) [Function File]
`std` (*x*, *opt*) [Function File]
`std` (*x*, *opt*, *dim*) [Function File]

Compute the standard deviation of the elements of the vector *x*.

The standard deviation is defined as

$$\text{std}(x) = \sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \bar{x})^2}{N - 1}}$$

where $\bar{x}$ is the mean value of x and N is the number of elements.

If *x* is a matrix, compute the standard deviation for each column and return them in a row vector.

The argument *opt* determines the type of normalization to use. Valid values are

0: normalize with $N - 1$, provides the square root of the best unbiased estimator of the variance [default]

1: normalize with N, this provides the square root of the second moment around the mean

If the optional argument *dim* is given, operate along this dimension.

See also: [var], page 601, [range], page 600, [iqr], page 600, [mean], page 599, [median], page 600.

In addition to knowing the size of a dispersion it is useful to know the shape of the data set. For example, are data points massed to the left or right of the mean? Octave provides several common measures to describe the shape of the data set. Octave can also calculate moments allowing arbitrary shape measures to be developed.

`var` (*x*) [Function File]
`var` (*x*, *opt*) [Function File]
`var` (*x*, *opt*, *dim*) [Function File]

Compute the variance of the elements of the vector *x*.

The variance is defined as

$$\mathrm{var}(x) = \sigma^2 = \frac{\sum_{i=1}^{N} (x_i - \bar{x})^2}{N - 1}$$

where $\bar{x}$ is the mean value of x. If *x* is a matrix, compute the variance for each column and return them in a row vector.

The argument *opt* determines the type of normalization to use. Valid values are

0: normalize with $N-1$, provides the best unbiased estimator of the variance [default]

1: normalizes with N, this provides the second moment around the mean

If $N == 1$ the value of *opt* is ignored and normalization by N is used.

If the optional argument *dim* is given, operate along this dimension.

See also: [cov], page 610, [std], page 601, [skewness], page 602, [kurtosis], page 603, [moment], page 603.

`skewness` (*x*) [Function File]
`skewness` (*x*, *flag*) [Function File]
`skewness` (*x*, *flag*, *dim*) [Function File]

Compute the sample skewness of the elements of *x*.

The sample skewness is defined as

$$\mathrm{skewness}(x) = \frac{\frac{1}{N} \sum_{i=1}^{N} (\mathrm{x}_i - \bar{\mathrm{x}})^3}{\sigma^3},$$

where N is the length of x, $\bar{\mathrm{x}}$ its mean and σ its (uncorrected) standard deviation.

The optional argument *flag* controls which normalization is used. If *flag* is equal to 1 (default value, used when *flag* is omitted or empty), return the sample skewness as defined above. If *flag* is equal to 0, return the adjusted skewness coefficient instead:

$$\mathrm{skewness}(x) = \frac{\sqrt{N(N-1)}}{N-2} \times \frac{\frac{1}{N} \sum_{i=1}^{N} (\mathrm{x}_i - \bar{\mathrm{x}})^3}{\sigma^3}$$

The adjusted skewness coefficient is obtained by replacing the sample second and third central moments by their bias-corrected versions.

If *x* is a matrix, or more generally a multi-dimensional array, return the skewness along the first non-singleton dimension. If the optional *dim* argument is given, operate along this dimension.

See also: [var], page 601, [kurtosis], page 603, [moment], page 603.

`kurtosis (x)` [Function File]
`kurtosis (x, flag)` [Function File]
`kurtosis (x, flag, dim)` [Function File]

Compute the sample kurtosis of the elements of *x*.

The sample kurtosis is defined as

$$\kappa_1 = \frac{\frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^4}{\sigma^4},$$

where N is the length of x, $\bar{x}$ its mean, and σ its (uncorrected) standard deviation.

The optional argument *flag* controls which normalization is used. If *flag* is equal to 1 (default value, used when *flag* is omitted or empty), return the sample kurtosis as defined above. If *flag* is equal to 0, return the "bias-corrected" kurtosis coefficient instead:

$$\kappa_0 = 3 + \frac{N-1}{(N-2)(N-3)} \left((N+1)\,\kappa_1 - 3(N-1) \right)$$

The bias-corrected kurtosis coefficient is obtained by replacing the sample second and fourth central moments by their unbiased versions. It is an unbiased estimate of the population kurtosis for normal populations.

If *x* is a matrix, or more generally a multi-dimensional array, return the kurtosis along the first non-singleton dimension. If the optional *dim* argument is given, operate along this dimension.

See also: [var], page 601, [skewness], page 602, [moment], page 603.

`moment (x, p)` [Function File]
`moment (x, p, type)` [Function File]
`moment (x, p, dim)` [Function File]
`moment (x, p, type, dim)` [Function File]
`moment (x, p, dim, type)` [Function File]

Compute the *p*-th central moment of the vector *x*.

$$\frac{\sum_{i=1}^{N} (x_i - \bar{x})^p}{N}$$

If *x* is a matrix, return the row vector containing the *p*-th central moment of each column.

If the optional argument *dim* is given, operate along this dimension.

The optional string *type* specifies the type of moment to be computed. Valid options are:

`"c"` Central Moment (default).

`"a"`
`"ac"` Absolute Central Moment. The moment about the mean ignoring sign defined as

$$\frac{\sum_{i=1}^{N} |x_i - \bar{x}|^p}{N}$$

`"r"` Raw Moment. The moment about zero defined as

$$\text{moment}(x) = \frac{\sum_{i=1}^{N} {x_i}^p}{N}$$

"ar" Absolute Raw Moment. The moment about zero ignoring sign defined as

$$\frac{\sum_{i=1}^{N} |x_i|^p}{N}$$

If both *type* and *dim* are given they may appear in any order.

See also: [var], page 601, [skewness], page 602, [kurtosis], page 603.

q = quantile (*x*) [Function File]
q = quantile (*x*, *p*) [Function File]
q = quantile (*x*, *p*, *dim*) [Function File]
q = quantile (*x*, *p*, *dim*, *method*) [Function File]

For a sample, *x*, calculate the quantiles, *q*, corresponding to the cumulative probability values in *p*. All non-numeric values (NaNs) of *x* are ignored.

If *x* is a matrix, compute the quantiles for each column and return them in a matrix, such that the i-th row of *q* contains the *p*(i)th quantiles of each column of *x*.

If *p* is unspecified, return the quantiles for `[0.00 0.25 0.50 0.75 1.00]`. The optional argument *dim* determines the dimension along which the quantiles are calculated. If *dim* is omitted it defaults to the first non-singleton dimension.

The methods available to calculate sample quantiles are the nine methods used by R (`http://www.r-project.org/`). The default value is METHOD = 5.

Discontinuous sample quantile methods 1, 2, and 3

1. Method 1: Inverse of empirical distribution function.
2. Method 2: Similar to method 1 but with averaging at discontinuities.
3. Method 3: SAS definition: nearest even order statistic.

Continuous sample quantile methods 4 through 9, where p(k) is the linear interpolation function respecting each methods' representative cdf.

4. Method 4: p(k) = k / n. That is, linear interpolation of the empirical cdf.
5. Method 5: p(k) = (k - 0.5) / n. That is a piecewise linear function where the knots are the values midway through the steps of the empirical cdf.
6. Method 6: p(k) = k / (n + 1).
7. Method 7: p(k) = (k - 1) / (n - 1).
8. Method 8: p(k) = (k - 1/3) / (n + 1/3). The resulting quantile estimates are approximately median-unbiased regardless of the distribution of *x*.
9. Method 9: p(k) = (k - 3/8) / (n + 1/4). The resulting quantile estimates are approximately unbiased for the expected order statistics if *x* is normally distributed.

Hyndman and Fan (1996) recommend method 8. Maxima, S, and R (versions prior to 2.0.0) use 7 as their default. Minitab and SPSS use method 6. MATLAB uses method 5.

References:

- Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. Wadsworth & Brooks/Cole.
- Hyndman, R. J. and Fan, Y. (1996) Sample quantiles in statistical packages, American Statistician, 50, 361–365.
- R: A Language and Environment for Statistical Computing; `http://cran.r-project.org/doc/manuals/fullrefman.pdf`.

Examples:

```
x = randi (1000, [10, 1]);  # Create empirical data in range 1-1000
q = quantile (x, [0, 1]);   # Return minimum, maximum of distribution
q = quantile (x, [0.25 0.5 0.75]); # Return quartiles of distribution
```

See also: [prctile], page 605.

q = `prctile` (*x*) [Function File]
q = `prctile` (*x*, *p*) [Function File]
q = `prctile` (*x*, *p*, *dim*) [Function File]

For a sample *x*, compute the quantiles, *q*, corresponding to the cumulative probability values, *p*, in percent.

If *x* is a matrix, compute the percentiles for each column and return them in a matrix, such that the i-th row of *y* contains the *p*(i)th percentiles of each column of *x*.

If *p* is unspecified, return the quantiles for `[0 25 50 75 100]`.

The optional argument *dim* determines the dimension along which the percentiles are calculated. If *dim* is omitted it defaults to the first non-singleton dimension.

Programming Note: All non-numeric values (NaNs) of *x* are ignored.

See also: [quantile], page 604.

A summary view of a data set can be generated quickly with the `statistics` function.

`statistics` (*x*) [Function File]
`statistics` (*x*, *dim*) [Function File]

Return a vector with the minimum, first quartile, median, third quartile, maximum, mean, standard deviation, skewness, and kurtosis of the elements of the vector *x*.

If *x* is a matrix, calculate statistics over the first non-singleton dimension.

If the optional argument *dim* is given, operate along this dimension.

See also: [min], page 445, [max], page 444, [median], page 600, [mean], page 599, [std], page 601, [skewness], page 602, [kurtosis], page 603.

26.2 Basic Statistical Functions

Octave supports various helpful statistical functions. Many are useful as initial steps to prepare a data set for further analysis. Others provide different measures from those of the basic descriptive statistics.

`center (x)` [Function File]
`center (x, dim)` [Function File]

Center data by subtracting its mean.

If *x* is a vector, subtract its mean.

If *x* is a matrix, do the above for each column.

If the optional argument *dim* is given, operate along this dimension.

Programming Note: `center` has obvious application for normalizing statistical data. It is also useful for improving the precision of general numerical calculations. Whenever there is a large value that is common to a batch of data, the mean can be subtracted off, the calculation performed, and then the mean added back to obtain the final answer.

See also: [zscore], page 606.

`z = zscore (x)` [Function File]
`z = zscore (x, opt)` [Function File]
`z = zscore (x, opt, dim)` [Function File]
`[z, mu, sigma] = zscore (...)` [Function File]

Compute the Z score of *x*

If *x* is a vector, subtract its mean and divide by its standard deviation. If the standard deviation is zero, divide by 1 instead.

The optional parameter *opt* determines the normalization to use when computing the standard deviation and has the same definition as the corresponding parameter for `std`.

If *x* is a matrix, calculate along the first non-singleton dimension. If the third optional argument *dim* is given, operate along this dimension.

The optional outputs *mu* and *sigma* contain the mean and standard deviation.

See also: [mean], page 599, [std], page 601, [center], page 605.

`n = histc (x, edges)` [Function File]
`n = histc (x, edges, dim)` [Function File]
`[n, idx] = histc (...)` [Function File]

Compute histogram counts.

When *x* is a vector, the function counts the number of elements of *x* that fall in the histogram bins defined by *edges*. This must be a vector of monotonically increasing values that define the edges of the histogram bins. `n(k)` contains the number of elements in *x* for which `edges(k) <= x < edges(k+1)`. The final element of *n* contains the number of elements of *x* exactly equal to the last element of *edges*.

When *x* is an *N*-dimensional array, the computation is carried out along dimension *dim*. If not specified *dim* defaults to the first non-singleton dimension.

When a second output argument is requested an index matrix is also returned. The *idx* matrix has the same size as *x*. Each element of *idx* contains the index of the histogram bin in which the corresponding element of *x* was counted.

See also: [hist], page 279.

`c = nchoosek (n, k)` [Function File]
`c = nchoosek (set, k)` [Function File]

Compute the binomial coefficient of *n* or list all possible combinations of a *set* of items.

If *n* is a scalar then calculate the binomial coefficient of *n* and *k* which is defined as

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

This is the number of combinations of *n* items taken in groups of size *k*.

If the first argument is a vector, *set*, then generate all combinations of the elements of *set*, taken *k* at a time, with one row per combination. The result *c* has *k* columns and `nchoosek (length (set), k)` rows.

For example:

How many ways can three items be grouped into pairs?

```
nchoosek (3, 2)
   ⇒ 3
```

What are the possible pairs?

```
nchoosek (1:3, 2)
   ⇒  1   2
      1   3
      2   3
```

Programming Note: When calculating the binomial coefficient `nchoosek` works only for non-negative, integer arguments. Use `bincoeff` for non-integer and negative scalar arguments, or for computing many binomial coefficients at once with vector inputs for *n* or *k*.

See also: [bincoeff], page 453, [perms], page 607.

`perms (v)` [Function File]

Generate all permutations of *v* with one row per permutation.

The result has size `factorial (n) * n`, where *n* is the length of *v*.

Example

```
perms ([1, 2, 3])
⇒
  1   2   3
  2   1   3
  1   3   2
  2   3   1
  3   1   2
  3   2   1
```

Programming Note: The maximum length of *v* should be less than or equal to 10 to limit memory consumption.

See also: [permute], page 411, [randperm], page 426, [nchoosek], page 607.

`ranks (x, dim)` [Function File]

Return the ranks of *x* along the first non-singleton dimension adjusted for ties.

If the optional argument *dim* is given, operate along this dimension.

See also: [spearman], page 610, [kendall], page 611.

`run_count (x, n)` [Function File]
`run_count (x, n, dim)` [Function File]

Count the upward runs along the first non-singleton dimension of *x* of length 1, 2, ..., *n*-1 and greater than or equal to *n*.

If the optional argument *dim* is given then operate along this dimension.

See also: [runlength], page 608.

`count = runlength (x)` [Function File]
`[count, value] = runlength (x)` [Function File]

Find the lengths of all sequences of common values.

count is a vector with the lengths of each repeated value.

The optional output *value* contains the value that was repeated in the sequence.

```
runlength ([2, 2, 0, 4, 4, 4, 0, 1, 1, 1, 1])
⇒  [2, 1, 3, 1, 4]
```

See also: [run_count], page 608.

`probit (p)` [Function File]

Return the probit (the quantile of the standard normal distribution) for each element of *p*.

See also: [logit], page 608.

`logit (p)` [Function File]

Compute the logit for each value of *p*

The logit is defined as

$$\mathrm{logit}(p) = \log\left(\frac{p}{1-p}\right)$$

See also: [probit], page 608, [logistic_cdf], page 616.

`cloglog (x)` [Function File]

Return the complementary log-log function of *x*.

The complementary log-log function is defined as

$$\mathrm{cloglog}(x) = -\log(-\log(x))$$

`mahalanobis (x, y)` [Function File]

Return the Mahalanobis' D-square distance between the multivariate samples *x* and *y*.

The data *x* and *y* must have the same number of components (columns), but may have a different number of observations (rows).

`[t, l_x] = table (x)` [Function File]
`[t, l_x, l_y] = table (x, y)` [Function File]

Create a contingency table *t* from data vectors.

The *l_x* and *l_y* vectors are the corresponding levels.

Currently, only 1- and 2-dimensional tables are supported.

26.3 Statistical Plots

Octave can create Quantile Plots (QQ-Plots), and Probability Plots (PP-Plots). These are simple graphical tests for determining if a data set comes from a certain distribution.

Note that Octave can also show histograms of data using the `hist` function as described in Section 15.2.1 [Two-Dimensional Plots], page 271.

`[q, s] = qqplot (x)` [Function File]
`[q, s] = qqplot (x, y)` [Function File]
`[q, s] = qqplot (x, dist)` [Function File]
`[q, s] = qqplot (x, y, params)` [Function File]
`qqplot (...)` [Function File]

Perform a QQ-plot (quantile plot).

If F is the CDF of the distribution *dist* with parameters *params* and G its inverse, and *x* a sample vector of length *n*, the QQ-plot graphs ordinate $s(i)$ = *i*-th largest element of x versus abscissa $q(if) = \mathrm{G}((i - 0.5)/n)$.

If the sample comes from F, except for a transformation of location and scale, the pairs will approximately follow a straight line.

If the second argument is a vector *y* the empirical CDF of *y* is used as *dist*.

The default for *dist* is the standard normal distribution. The optional argument *params* contains a list of parameters of *dist*. For example, for a quantile plot of the uniform distribution on [2,4] and *x*, use

```
qqplot (x, "unif", 2, 4)
```

dist can be any string for which a function *distinv* or *dist_inv* exists that calculates the inverse CDF of distribution *dist*.

If no output arguments are given, the data are plotted directly.

`[p, y] = ppplot (x, dist, params)` [Function File]

Perform a PP-plot (probability plot).

If F is the CDF of the distribution *dist* with parameters *params* and *x* a sample vector of length *n*, the PP-plot graphs ordinate $y(i)$ = F (*i*-th largest element of *x*) versus abscissa $p(i) = (i - 0.5)/n$. If the sample comes from F, the pairs will approximately follow a straight line.

The default for *dist* is the standard normal distribution.

The optional argument *params* contains a list of parameters of *dist*.

For example, for a probability plot of the uniform distribution on [2,4] and *x*, use

```
ppplot (x, "uniform", 2, 4)
```

dist can be any string for which a function *dist_cdf* that calculates the CDF of distribution *dist* exists.

If no output is requested then the data are plotted immediately.

26.4 Correlation and Regression Analysis

`cov (x)` [Function File]
`cov (x, opt)` [Function File]
`cov (x, y)` [Function File]
`cov (x, y, opt)` [Function File]

Compute the covariance matrix.

If each row of *x* and *y* is an observation, and each column is a variable, then the (*i*, *j*)-th entry of `cov (x, y)` is the covariance between the *i*-th variable in *x* and the *j*-th variable in *y*.

$$\sigma_{ij} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})$$

where $\bar{x}$ and $\bar{y}$ are the mean values of x and y.

If called with one argument, compute `cov (x, x)`, the covariance between the columns of *x*.

The argument *opt* determines the type of normalization to use. Valid values are

0: normalize with $N-1$, provides the best unbiased estimator of the covariance [default]

1: normalize with N, this provides the second moment around the mean

Compatibility Note:: Octave always computes the covariance matrix. For two inputs, however, MATLAB will calculate `cov (x(:), y(:))` whenever the number of elements in *x* and *y* are equal. This will result in a scalar rather than a matrix output. Code relying on this odd definition will need to be changed when running in Octave.

See also: [corr], page 610.

`corr (x)` [Function File]
`corr (x, y)` [Function File]

Compute matrix of correlation coefficients.

If each row of *x* and *y* is an observation and each column is a variable, then the (*i*, *j*)-th entry of `corr (x, y)` is the correlation between the *i*-th variable in *x* and the *j*-th variable in *y*.

$$\mathrm{corr}(x,y) = \frac{\mathrm{cov}(x,y)}{\mathrm{std}(x)\mathrm{std}(y)}$$

If called with one argument, compute `corr (x, x)`, the correlation between the columns of *x*.

See also: [cov], page 610.

`spearman (x)` [Function File]
`spearman (x, y)` [Function File]

Compute Spearman's rank correlation coefficient *rho*.

For two data vectors *x* and *y*, Spearman's *rho* is the correlation coefficient of the ranks of *x* and *y*.

If *x* and *y* are drawn from independent distributions, *rho* has zero mean and variance `1 / (n - 1)`, and is asymptotically normally distributed.

`spearman (x)` is equivalent to `spearman (x, x)`.

See also: [ranks], page 608, [kendall], page 611.

`kendall (x)` [Function File]
`kendall (x, y)` [Function File]

Compute Kendall's *tau*.

For two data vectors *x*, *y* of common length *n*, Kendall's *tau* is the correlation of the signs of all rank differences of *x* and *y*; i.e., if both *x* and *y* have distinct entries, then

$$\tau = \frac{1}{n(n-1)} \sum_{i,j} \mathrm{sign}(q_i - q_j)\mathrm{sign}(r_i - r_j)$$

in which the q_i and r_i are the ranks of *x* and *y*, respectively.

If *x* and *y* are drawn from independent distributions, Kendall's *tau* is asymptotically normal with mean 0 and variance $\frac{2(2n+5)}{9n(n-1)}$.

`kendall (x)` is equivalent to `kendall (x, x)`.

See also: [ranks], page 608, [spearman], page 610.

`[theta, beta, dev, dl, d2l, p] = logistic_regression (y, x, print, theta, beta)` [Function File]

Perform ordinal logistic regression.

Suppose *y* takes values in *k* ordered categories, and let `gamma_i (x)` be the cumulative probability that *y* falls in one of the first *i* categories given the covariate *x*. Then

```
[theta, beta] = logistic_regression (y, x)
```

fits the model

```
logit (gamma_i (x)) = theta_i - beta' * x,   i = 1 ... k-1
```

The number of ordinal categories, *k*, is taken to be the number of distinct values of `round (y)`. If *k* equals 2, *y* is binary and the model is ordinary logistic regression. The matrix *x* is assumed to have full column rank.

Given *y* only, `theta = logistic_regression (y)` fits the model with baseline logit odds only.

The full form is

```
[theta, beta, dev, dl, d2l, gamma]
   = logistic_regression (y, x, print, theta, beta)
```

in which all output arguments and all input arguments except *y* are optional.

Setting *print* to 1 requests summary information about the fitted model to be displayed. Setting *print* to 2 requests information about convergence at each iteration. Other values request no information to be displayed. The input arguments *theta* and *beta* give initial estimates for *theta* and *beta*.

The returned value *dev* holds minus twice the log-likelihood.

The returned values *dl* and *d2l* are the vector of first and the matrix of second derivatives of the log-likelihood with respect to *theta* and *beta*.

p holds estimates for the conditional distribution of *y* given *x*.

26.5 Distributions

Octave has functions for computing the Probability Density Function (PDF), the Cumulative Distribution function (CDF), and the quantile (the inverse of the CDF) for a large number of distributions.

The following table summarizes the supported distributions (in alphabetical order).

Distribution	**PDF**	**CDF**	**Quantile**
Beta	betapdf	betacdf	betainv
Binomial	binopdf	binocdf	binoinv
Cauchy	cauchy_pdf	cauchy_cdf	cauchy_inv
Chi-Square	chi2pdf	chi2cdf	chi2inv
Univariate Discrete	discrete_pdf	discrete_cdf	discrete_inv
Empirical	empirical_pdf	empirical_cdf	empirical_inv
Exponential	exppdf	expcdf	expinv
F	fpdf	fcdf	finv
Gamma	gampdf	gamcdf	gaminv
Geometric	geopdf	geocdf	geoinv
Hypergeometric	hygepdf	hygecdf	hygeinv
Kolmogorov Smirnov	*Not Available*	kolmogorov_smirnov_cdf	*Not Available*
Laplace	laplace_pdf	laplace_cdf	laplace_inv
Logistic	logistic_pdf	logistic_cdf	logistic_inv
Log-Normal	lognpdf	logncdf	logninv
Univariate Normal	normpdf	normcdf	norminv
Pascal	nbinpdf	nbincdf	nbininv
Poisson	poisspdf	poisscdf	poissinv
Standard Normal	stdnormal_pdf	stdnormal_cdf	stdnormal_inv
t (Student)	tpdf	tcdf	tinv
Uniform Discrete	unidpdf	unidcdf	unidinv
Uniform	unifpdf	unifcdf	unifinv
Weibull	wblpdf	wblcdf	wblinv

`betapdf` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the Beta distribution with parameters *a* and *b*.

`betacdf` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Beta distribution with parameters *a* and *b*.

`betainv` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Beta distribution with parameters *a* and *b*.

`binopdf (x, n, p)` [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the binomial distribution with parameters *n* and *p*, where *n* is the number of trials and *p* is the probability of success.

`binocdf (x, n, p)` [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the binomial distribution with parameters *n* and *p*, where *n* is the number of trials and *p* is the probability of success.

`binoinv (x, n, p)` [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the binomial distribution with parameters *n* and *p*, where *n* is the number of trials and *p* is the probability of success.

`cauchy_pdf (x)` [Function File]
`cauchy_pdf (x, location, scale)` [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the Cauchy distribution with location parameter *location* and scale parameter *scale* > 0.

Default values are *location* = 0, *scale* = 1.

`cauchy_cdf (x)` [Function File]
`cauchy_cdf (x, location, scale)` [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Cauchy distribution with location parameter *location* and scale parameter *scale*.

Default values are *location* = 0, *scale* = 1.

`cauchy_inv (x)` [Function File]
`cauchy_inv (x, location, scale)` [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Cauchy distribution with location parameter *location* and scale parameter *scale*.

Default values are *location* = 0, *scale* = 1.

`chi2pdf (x, n)` [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the chi-square distribution with *n* degrees of freedom.

`chi2cdf (x, n)` [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the chi-square distribution with *n* degrees of freedom.

`chi2inv (x, n)` [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the chi-square distribution with *n* degrees of freedom.

`discrete_pdf (x, v, p)` [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of a univariate discrete distribution which assumes the values in *v* with probabilities *p*.

`discrete_cdf` (*x*, *v*, *p*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of a univariate discrete distribution which assumes the values in *v* with probabilities *p*.

`discrete_inv` (*x*, *v*, *p*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the univariate distribution which assumes the values in *v* with probabilities *p*.

`empirical_pdf` (*x*, *data*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the empirical distribution obtained from the univariate sample *data*.

`empirical_cdf` (*x*, *data*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the empirical distribution obtained from the univariate sample *data*.

`empirical_inv` (*x*, *data*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the empirical distribution obtained from the univariate sample *data*.

`exppdf` (*x*, *lambda*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the exponential distribution with mean *lambda*.

`expcdf` (*x*, *lambda*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the exponential distribution with mean *lambda*.

The arguments can be of common size or scalars.

`expinv` (*x*, *lambda*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the exponential distribution with mean *lambda*.

`fpdf` (*x*, *m*, *n*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the F distribution with *m* and *n* degrees of freedom.

`fcdf` (*x*, *m*, *n*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the F distribution with *m* and *n* degrees of freedom.

`finv` (*x*, *m*, *n*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the F distribution with *m* and *n* degrees of freedom.

`gampdf` (*x*, *a*, *b*) [Function File]

For each element of *x*, return the probability density function (PDF) at *x* of the Gamma distribution with shape parameter *a* and scale *b*.

`gamcdf` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Gamma distribution with shape parameter *a* and scale *b*.

`gaminv` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Gamma distribution with shape parameter *a* and scale *b*.

`geopdf` (*x*, *p*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the geometric distribution with parameter *p*.

The geometric distribution models the number of failures (*x*-1) of a Bernoulli trial with probability *p* before the first success (*x*).

`geocdf` (*x*, *p*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the geometric distribution with parameter *p*.

The geometric distribution models the number of failures (*x*-1) of a Bernoulli trial with probability *p* before the first success (*x*).

`geoinv` (*x*, *p*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the geometric distribution with parameter *p*.

The geometric distribution models the number of failures (*x*-1) of a Bernoulli trial with probability *p* before the first success (*x*).

`hygepdf` (*x*, *t*, *m*, *n*) [Function File]

Compute the probability density function (PDF) at *x* of the hypergeometric distribution with parameters *t*, *m*, and *n*.

This is the probability of obtaining *x* marked items when randomly drawing a sample of size *n* without replacement from a population of total size *t* containing *m* marked items.

The parameters *t*, *m*, and *n* must be positive integers with *m* and *n* not greater than *t*.

`hygecdf` (*x*, *t*, *m*, *n*) [Function File]

Compute the cumulative distribution function (CDF) at *x* of the hypergeometric distribution with parameters *t*, *m*, and *n*.

This is the probability of obtaining not more than *x* marked items when randomly drawing a sample of size *n* without replacement from a population of total size *t* containing *m* marked items.

The parameters *t*, *m*, and *n* must be positive integers with *m* and *n* not greater than *t*.

`hygeinv` (*x*, *t*, *m*, *n*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the hypergeometric distribution with parameters *t*, *m*, and *n*.

This is the probability of obtaining *x* marked items when randomly drawing a sample of size *n* without replacement from a population of total size *t* containing *m* marked items.

The parameters *t*, *m*, and *n* must be positive integers with *m* and *n* not greater than *t*.

`kolmogorov_smirnov_cdf` (*x*, *tol*) [Function File]

Return the cumulative distribution function (CDF) at *x* of the Kolmogorov-Smirnov distribution.

This is defined as

$$Q(x) = \sum_{k=-\infty}^{\infty} (-1)^k \exp(-2k^2x^2)$$

for $x > 0$.

The optional parameter *tol* specifies the precision up to which the series should be evaluated; the default is *tol* = `eps`.

`laplace_pdf` (*x*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the Laplace distribution.

`laplace_cdf` (*x*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Laplace distribution.

`laplace_inv` (*x*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Laplace distribution.

`logistic_pdf` (*x*) [Function File]

For each element of *x*, compute the PDF at *x* of the logistic distribution.

`logistic_cdf` (*x*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the logistic distribution.

`logistic_inv` (*x*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the logistic distribution.

`lognpdf` (*x*) [Function File]
`lognpdf` (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the lognormal distribution with parameters *mu* and *sigma*.

If a random variable follows this distribution, its logarithm is normally distributed with mean *mu* and standard deviation *sigma*.

Default values are *mu* = 0, *sigma* = 1.

`logncdf` (*x*) [Function File]
`logncdf` (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the lognormal distribution with parameters *mu* and *sigma*.

If a random variable follows this distribution, its logarithm is normally distributed with mean *mu* and standard deviation *sigma*.

Default values are *mu* = 0, *sigma* = 1.

`logninv` (*x*) [Function File]
`logninv` (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the lognormal distribution with parameters *mu* and *sigma*.

If a random variable follows this distribution, its logarithm is normally distributed with mean *mu* and standard deviation *sigma*.

Default values are $mu = 0$, $sigma = 1$.

`nbinpdf` (*x*, *n*, *p*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the negative binomial distribution with parameters *n* and *p*.

When *n* is integer this is the Pascal distribution. When *n* is extended to real numbers this is the Polya distribution.

The number of failures in a Bernoulli experiment with success probability *p* before the *n*-th success follows this distribution.

`nbincdf` (*x*, *n*, *p*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the negative binomial distribution with parameters *n* and *p*.

When *n* is integer this is the Pascal distribution. When *n* is extended to real numbers this is the Polya distribution.

The number of failures in a Bernoulli experiment with success probability *p* before the *n*-th success follows this distribution.

`nbininv` (*x*, *n*, *p*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the negative binomial distribution with parameters *n* and *p*.

When *n* is integer this is the Pascal distribution. When *n* is extended to real numbers this is the Polya distribution.

The number of failures in a Bernoulli experiment with success probability *p* before the *n*-th success follows this distribution.

`normpdf` (*x*) [Function File]
`normpdf` (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the normal distribution with mean *mu* and standard deviation *sigma*.

Default values are $mu = 0$, $sigma = 1$.

`normcdf` (*x*) [Function File]
`normcdf` (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the normal distribution with mean *mu* and standard deviation *sigma*.

Default values are $mu = 0$, $sigma = 1$.

`norminv` (*x*) [Function File]
`norminv` (*x*, *mu*, *sigma*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the normal distribution with mean *mu* and standard deviation *sigma*.

Default values are *mu* = 0, *sigma* = 1.

`poisspdf` (*x*, *lambda*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of the Poisson distribution with parameter *lambda*.

`poisscdf` (*x*, *lambda*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the Poisson distribution with parameter *lambda*.

`poissinv` (*x*, *lambda*) [Function File]
For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the Poisson distribution with parameter *lambda*.

`stdnormal_pdf` (*x*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of the standard normal distribution (mean = 0, standard deviation = 1).

`stdnormal_cdf` (*x*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the standard normal distribution (mean = 0, standard deviation = 1).

`stdnormal_inv` (*x*) [Function File]
For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the standard normal distribution (mean = 0, standard deviation = 1).

`tpdf` (*x*, *n*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of the *t* (Student) distribution with *n* degrees of freedom.

`tcdf` (*x*, *n*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the t (Student) distribution with *n* degrees of freedom.

`tinv` (*x*, *n*) [Function File]
For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the t (Student) distribution with *n* degrees of freedom.

This function is analogous to looking in a table for the t-value of a single-tailed distribution.

`unidpdf` (*x*, *n*) [Function File]
For each element of *x*, compute the probability density function (PDF) at *x* of a discrete uniform distribution which assumes the integer values 1–*n* with equal probability.

Warning: The underlying implementation uses the double class and will only be accurate for $n \leq$ `bitmax` ($2^{53} - 1$ on IEEE 754 compatible systems).

`unidcdf` (*x*, *n*) [Function File]
For each element of *x*, compute the cumulative distribution function (CDF) at *x* of a discrete uniform distribution which assumes the integer values 1–*n* with equal probability.

`unidinv` (*x*, *n*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the discrete uniform distribution which assumes the integer values 1–*n* with equal probability.

`unifpdf` (*x*) [Function File]
`unifpdf` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the probability density function (PDF) at *x* of the uniform distribution on the interval [*a*, *b*].

Default values are $a = 0$, $b = 1$.

`unifcdf` (*x*) [Function File]
`unifcdf` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the cumulative distribution function (CDF) at *x* of the uniform distribution on the interval [*a*, *b*].

Default values are $a = 0$, $b = 1$.

`unifinv` (*x*) [Function File]
`unifinv` (*x*, *a*, *b*) [Function File]

For each element of *x*, compute the quantile (the inverse of the CDF) at *x* of the uniform distribution on the interval [*a*, *b*].

Default values are $a = 0$, $b = 1$.

`wblpdf` (*x*) [Function File]
`wblpdf` (*x*, *scale*) [Function File]
`wblpdf` (*x*, *scale*, *shape*) [Function File]

Compute the probability density function (PDF) at *x* of the Weibull distribution with scale parameter *scale* and shape parameter *shape*.

This is given by

$$\frac{shape}{scale^{shape}} \cdot x^{shape-1} \cdot e^{-(\frac{x}{scale})^{shape}}$$

for $x \geq 0$.

Default values are $scale = 1$, $shape = 1$.

`wblcdf` (*x*) [Function File]
`wblcdf` (*x*, *scale*) [Function File]
`wblcdf` (*x*, *scale*, *shape*) [Function File]

Compute the cumulative distribution function (CDF) at *x* of the Weibull distribution with scale parameter *scale* and shape parameter *shape*.

This is defined as

$$1 - e^{-(\frac{x}{scale})^{shape}}$$

for $x \geq 0$.

`wblinv` (*x*) [Function File]
`wblinv` (*x*, *scale*) [Function File]
`wblinv` (*x*, *scale*, *shape*) [Function File]

Compute the quantile (the inverse of the CDF) at *x* of the Weibull distribution with scale parameter *scale* and shape parameter *shape*.

Default values are $scale = 1$, $shape = 1$.

26.6 Tests

Octave can perform many different statistical tests. The following table summarizes the available tests.

Hypothesis	**Test Functions**
Equal mean values	anova, hotelling_test2, t_test_2, welch_test, wilcoxon_test, z_test_2
Equal medians	kruskal_wallis_test, sign_test
Equal variances	bartlett_test, manova, var_test
Equal distributions	chisquare_test_homogeneity, kolmogorov_smirnov_test_2, u_test
Equal marginal frequencies	mcnemar_test
Equal success probabilities	prop_test_2
Independent observations	chisquare_test_independence, run_test
Uncorrelated observations	cor_test
Given mean value	hotelling_test, t_test, z_test
Observations from distribution	kolmogorov_smirnov_test
Regression	f_test_regression, t_test_regression

The tests return a p-value that describes the outcome of the test. Assuming that the test hypothesis is true, the p-value is the probability of obtaining a worse result than the observed one. So large p-values corresponds to a successful test. Usually a test hypothesis is accepted if the p-value exceeds 0.05.

`[pval, f, df_b, df_w] = anova (y, g)` [Function File]

Perform a one-way analysis of variance (ANOVA).

The goal is to test whether the population means of data taken from *k* different groups are all equal.

Data may be given in a single vector *y* with groups specified by a corresponding vector of group labels *g* (e.g., numbers from 1 to *k*). This is the general form which does not impose any restriction on the number of data in each group or the group labels.

If *y* is a matrix and *g* is omitted, each column of *y* is treated as a group. This form is only appropriate for balanced ANOVA in which the numbers of samples from each group are all equal.

Under the null of constant means, the statistic *f* follows an F distribution with *df_b* and *df_w* degrees of freedom.

The p-value (1 minus the CDF of this distribution at *f*) is returned in *pval*.

If no output argument is given, the standard one-way ANOVA table is printed.

See also: [manova], page 623.

`[pval, chisq, df] = bartlett_test (x1, ...)` [Function File]

Perform a Bartlett test for the homogeneity of variances in the data vectors *x1*, *x2*, ..., *xk*, where *k* > 1.

Under the null of equal variances, the test statistic *chisq* approximately follows a chi-square distribution with *df* degrees of freedom.

The p-value (1 minus the CDF of this distribution at *chisq*) is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[`*pval*`, `*chisq*`, `*df*`] = chisquare_test_homogeneity (`*x*, *y*, *c*`)` [Function File]

Given two samples *x* and *y*, perform a chisquare test for homogeneity of the null hypothesis that *x* and *y* come from the same distribution, based on the partition induced by the (strictly increasing) entries of *c*.

For large samples, the test statistic *chisq* approximately follows a chisquare distribution with *df* = `length (`*c*`)` degrees of freedom.

The p-value (1 minus the CDF of this distribution at *chisq*) is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[`*pval*`, `*chisq*`, `*df*`] = chisquare_test_independence (`*x*`)` [Function File]

Perform a chi-square test for independence based on the contingency table *x*.

Under the null hypothesis of independence, *chisq* approximately has a chi-square distribution with *df* degrees of freedom.

The p-value (1 minus the CDF of this distribution at chisq) of the test is returned in *pval*.

If no output argument is given, the p-value is displayed.

`cor_test (`*x*, *y*, *alt*, *method*`)` [Function File]

Test whether two samples *x* and *y* come from uncorrelated populations.

The optional argument string *alt* describes the alternative hypothesis, and can be `"!="` or `"<>"` (nonzero), `">"` (greater than 0), or `"<"` (less than 0). The default is the two-sided case.

The optional argument string *method* specifies which correlation coefficient to use for testing. If *method* is `"pearson"` (default), the (usual) Pearson's produt moment correlation coefficient is used. In this case, the data should come from a bivariate normal distribution. Otherwise, the other two methods offer nonparametric alternatives. If *method* is `"kendall"`, then Kendall's rank correlation tau is used. If *method* is `"spearman"`, then Spearman's rank correlation rho is used. Only the first character is necessary.

The output is a structure with the following elements:

pval The p-value of the test.

stat The value of the test statistic.

dist The distribution of the test statistic.

params The parameters of the null distribution of the test statistic.

alternative
The alternative hypothesis.

method The method used for testing.

If no output argument is given, the p-value is displayed.

[*pval*, *f*, *df_num*, *df_den*] = **f_test_regression** (*y*, *x*, *rr*, *r*) [Function File]

Perform an F test for the null hypothesis rr * b = r in a classical normal regression model y = X * b + e.

Under the null, the test statistic *f* follows an F distribution with *df_num* and *df_den* degrees of freedom.

The p-value (1 minus the CDF of this distribution at *f*) is returned in *pval*.

If not given explicitly, $r = 0$.

If no output argument is given, the p-value is displayed.

[*pval*, *tsq*] = **hotelling_test** (*x*, *m*) [Function File]

For a sample *x* from a multivariate normal distribution with unknown mean and covariance matrix, test the null hypothesis that `mean (x) == m`.

Hotelling's T^2 is returned in *tsq*. Under the null, $(n-p)T^2/(p(n-1))$ has an F distribution with p and $n-p$ degrees of freedom, where n and p are the numbers of samples and variables, respectively.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

[*pval*, *tsq*] = **hotelling_test_2** (*x*, *y*) [Function File]

For two samples *x* from multivariate normal distributions with the same number of variables (columns), unknown means and unknown equal covariance matrices, test the null hypothesis `mean (x) == mean (y)`.

Hotelling's two-sample T^2 is returned in *tsq*. Under the null,

$$\frac{(n_x + n_y - p - 1)T^2}{p(n_x + n_y - 2)}$$

has an F distribution with p and $n_x + n_y - p - 1$ degrees of freedom, where n_x and n_y are the sample sizes and p is the number of variables.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

[*pval*, *ks*] = **kolmogorov_smirnov_test** (*x*, *dist*, *params*, *alt*) [Function File]

Perform a Kolmogorov-Smirnov test of the null hypothesis that the sample *x* comes from the (continuous) distribution *dist*.

if F and G are the CDFs corresponding to the sample and dist, respectively, then the null is that F == G.

The optional argument *params* contains a list of parameters of *dist*. For example, to test whether a sample *x* comes from a uniform distribution on [2,4], use

```
kolmogorov_smirnov_test (x, "unif", 2, 4)
```

dist can be any string for which a function *distcdf* that calculates the CDF of distribution *dist* exists.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative F != G. In this case, the test statistic *ks* follows a two-sided Kolmogorov-Smirnov distribution. If *alt* is `">"`, the one-sided alternative F `>` G is considered. Similarly for `"<"`, the one-sided alternative F `>` G is considered. In this case, the test statistic *ks* has a one-sided Kolmogorov-Smirnov distribution. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[pval, ks, d] = kolmogorov_smirnov_test_2 (x, y, alt)` [Function File]

Perform a 2-sample Kolmogorov-Smirnov test of the null hypothesis that the samples *x* and *y* come from the same (continuous) distribution.

If F and G are the CDFs corresponding to the *x* and *y* samples, respectively, then the null is that F == G.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative F != G. In this case, the test statistic *ks* follows a two-sided Kolmogorov-Smirnov distribution. If *alt* is `">"`, the one-sided alternative F `>` G is considered. Similarly for `"<"`, the one-sided alternative F `<` G is considered. In this case, the test statistic *ks* has a one-sided Kolmogorov-Smirnov distribution. The default is the two-sided case.

The p-value of the test is returned in *pval*.

The third returned value, *d*, is the test statistic, the maximum vertical distance between the two cumulative distribution functions.

If no output argument is given, the p-value is displayed.

`[pval, k, df] = kruskal_wallis_test (x1, ...)` [Function File]

Perform a Kruskal-Wallis one-factor analysis of variance.

Suppose a variable is observed for *k* `>` 1 different groups, and let *x1*, ..., *xk* be the corresponding data vectors.

Under the null hypothesis that the ranks in the pooled sample are not affected by the group memberships, the test statistic *k* is approximately chi-square with *df* = *k* - 1 degrees of freedom.

If the data contains ties (some value appears more than once) *k* is divided by

1 - *sum_ties* / (*n*^3 - *n*)

where *sum_ties* is the sum of *t*^2 - *t* over each group of ties where *t* is the number of ties in the group and *n* is the total number of values in the input data. For more info on this adjustment see William H. Kruskal and W. Allen Wallis, *Use of Ranks in One-Criterion Variance Analysis*, Journal of the American Statistical Association, Vol. 47, No. 260 (Dec 1952).

The p-value (1 minus the CDF of this distribution at *k*) is returned in *pval*.

If no output argument is given, the p-value is displayed.

`manova (x, g)` [Function File]

Perform a one-way multivariate analysis of variance (MANOVA).

The goal is to test whether the p-dimensional population means of data taken from *k* different groups are all equal. All data are assumed drawn independently from p-dimensional normal distributions with the same covariance matrix.

The data matrix is given by *x*. As usual, rows are observations and columns are variables. The vector *g* specifies the corresponding group labels (e.g., numbers from 1 to *k*).

The LR test statistic (Wilks' Lambda) and approximate p-values are computed and displayed.

See also: [anova], page 620.

`[pval, chisq, df] = mcnemar_test (x)` [Function File]

For a square contingency table *x* of data cross-classified on the row and column variables, McNemar's test can be used for testing the null hypothesis of symmetry of the classification probabilities.

Under the null, *chisq* is approximately distributed as chisquare with *df* degrees of freedom.

The p-value (1 minus the CDF of this distribution at *chisq*) is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = prop_test_2 (x1, n1, x2, n2, alt)` [Function File]

If *x1* and *n1* are the counts of successes and trials in one sample, and *x2* and *n2* those in a second one, test the null hypothesis that the success probabilities *p1* and *p2* are the same.

Under the null, the test statistic *z* approximately follows a standard normal distribution.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative *p1* != *p2*. If *alt* is `">"`, the one-sided alternative *p1* `>` *p2* is used. Similarly for `"<"`, the one-sided alternative *p1* `<` *p2* is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, chisq] = run_test (x)` [Function File]

Perform a chi-square test with 6 degrees of freedom based on the upward runs in the columns of *x*.

`run_test` can be used to decide whether *x* contains independent data.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value is displayed.

`[pval, b, n] = sign_test (x, y, alt)` [Function File]

For two matched-pair samples *x* and *y*, perform a sign test of the null hypothesis PROB (*x* `>` *y*) == PROB (*x* `<` *y*) == 1/2.

Under the null, the test statistic *b* roughly follows a binomial distribution with parameters `n = sum (x != y)` and $p = 1/2$.

With the optional argument `alt`, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null hypothesis is tested against the two-sided alternative PROB (x `<` y) != 1/2. If *alt* is `">"`, the one-sided alternative PROB (x `>` y) `>` 1/2 ("x is stochastically greater than y") is considered. Similarly for `"<"`, the one-sided alternative PROB (x `>` y) `<` 1/2 ("x is stochastically less than y") is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = t_test (x, m, alt)` [Function File]

For a sample *x* from a normal distribution with unknown mean and variance, perform a t-test of the null hypothesis `mean (x) == m`.

Under the null, the test statistic *t* follows a Student distribution with `df = length (x) - 1` degrees of freedom.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `mean (x) != m`. If *alt* is `">"`, the one-sided alternative `mean (x) > m` is considered. Similarly for `"<"`, the one-sided alternative `mean (x) < m` is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = t_test_2 (x, y, alt)` [Function File]

For two samples x and y from normal distributions with unknown means and unknown equal variances, perform a two-sample t-test of the null hypothesis of equal means.

Under the null, the test statistic *t* follows a Student distribution with *df* degrees of freedom.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `mean (x) != mean (y)`. If *alt* is `">"`, the one-sided alternative `mean (x) > mean (y)` is used. Similarly for `"<"`, the one-sided alternative `mean (x) < mean (y)` is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = t_test_regression (y, x, rr, r, alt)` [Function File]

Perform a t test for the null hypothesis `rr * b = r` in a classical normal regression model `y = x * b + e`.

Under the null, the test statistic *t* follows a *t* distribution with *df* degrees of freedom.

If *r* is omitted, a value of 0 is assumed.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `rr * b != r`. If *alt* is `">"`, the one-sided alternative `rr * b > r` is used. Similarly for `"<"`, the one-sided alternative `rr * b < r` is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = u_test (x, y, alt)` [Function File]

For two samples x and y, perform a Mann-Whitney U-test of the null hypothesis PROB (x > y) == 1/2 == PROB (x < y).

Under the null, the test statistic z approximately follows a standard normal distribution. Note that this test is equivalent to the Wilcoxon rank-sum test.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative PROB (x > y) != 1/2. If *alt* is `">"`, the one-sided alternative PROB (x > y) > 1/2 is considered. Similarly for `"<"`, the one-sided alternative PROB (x > y) < 1/2 is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, f, df_num, df_den] = var_test (x, y, alt)` [Function File]

For two samples x and y from normal distributions with unknown means and unknown variances, perform an F-test of the null hypothesis of equal variances.

Under the null, the test statistic f follows an F-distribution with *df_num* and *df_den* degrees of freedom.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `var (x) != var (y)`. If *alt* is `">"`, the one-sided alternative `var (x) > var (y)` is used. Similarly for `"<"`, the one-sided alternative `var (x) > var (y)` is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, t, df] = welch_test (x, y, alt)` [Function File]

For two samples x and y from normal distributions with unknown means and unknown and not necessarily equal variances, perform a Welch test of the null hypothesis of equal means.

Under the null, the test statistic t approximately follows a Student distribution with *df* degrees of freedom.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `mean (x) != m`. If *alt* is `">"`, the one-sided alternative mean(x) > m is considered. Similarly for `"<"`, the one-sided alternative mean(x) < m is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = wilcoxon_test (x, y, alt)` [Function File]

For two matched-pair sample vectors x and y, perform a Wilcoxon signed-rank test of the null hypothesis PROB (x > y) == 1/2.

Under the null, the test statistic z approximately follows a standard normal distribution when n > 25.

Caution: This function assumes a normal distribution for z and thus is invalid for $n \leq 25$.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative PROB (x > y) != 1/2. If alt is `">"`, the one-sided alternative PROB (x > y) > 1/2 is considered. Similarly for `"<"`, the one-sided alternative PROB (x > y) < 1/2 is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed.

`[pval, z] = z_test (x, m, v, alt)` [Function File]

Perform a Z-test of the null hypothesis `mean (x) == m` for a sample x from a normal distribution with unknown mean and known variance v.

Under the null, the test statistic z follows a standard normal distribution.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `mean (x) != m`. If *alt* is `">"`, the one-sided alternative `mean (x) > m` is considered. Similarly for `"<"`, the one-sided alternative `mean (x) < m` is considered. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed along with some information.

`[pval, z] = z_test_2 (x, y, v_x, v_y, alt)` [Function File]

For two samples x and y from normal distributions with unknown means and known variances v_x and v_y, perform a Z-test of the hypothesis of equal means.

Under the null, the test statistic z follows a standard normal distribution.

With the optional argument string *alt*, the alternative of interest can be selected. If *alt* is `"!="` or `"<>"`, the null is tested against the two-sided alternative `mean (x) != mean (y)`. If alt is `">"`, the one-sided alternative `mean (x) > mean (y)` is used. Similarly for `"<"`, the one-sided alternative `mean (x) < mean (y)` is used. The default is the two-sided case.

The p-value of the test is returned in *pval*.

If no output argument is given, the p-value of the test is displayed along with some information.

26.7 Random Number Generation

Octave can generate random numbers from a large number of distributions. The random number generators are based on the random number generators described in Section 16.3 [Special Utility Matrices], page 418.

The following table summarizes the available random number generators (in alphabetical order).

Distribution	Function
Beta Distribution	betarnd
Binomial Distribution	binornd
Cauchy Distribution	cauchy_rnd
Chi-Square Distribution	chi2rnd
Univariate Discrete Distribution	discrete_rnd
Empirical Distribution	empirical_rnd
Exponential Distribution	exprnd
F Distribution	frnd
Gamma Distribution	gamrnd
Geometric Distribution	geornd
Hypergeometric Distribution	hygernd
Laplace Distribution	laplace_rnd
Logistic Distribution	logistic_rnd
Log-Normal Distribution	lognrnd
Pascal Distribution	nbinrnd
Univariate Normal Distribution	normrnd
Poisson Distribution	poissrnd
Standard Normal Distribution	stdnormal_rnd
t (Student) Distribution	trnd
Univariate Discrete Distribution	unidrnd
Uniform Distribution	unifrnd
Weibull Distribution	wblrnd
Wiener Process	wienrnd

betarnd (*a*, *b*) [Function File]
betarnd (*a*, *b*, *r*) [Function File]
betarnd (*a*, *b*, *r*, *c*, . . .) [Function File]
betarnd (*a*, *b*, [*sz*]) [Function File]

Return a matrix of random samples from the Beta distribution with parameters *a* and *b*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *a* and *b*.

binornd (*n*, *p*) [Function File]
binornd (*n*, *p*, *r*) [Function File]
binornd (*n*, *p*, *r*, *c*, . . .) [Function File]
binornd (*n*, *p*, [*sz*]) [Function File]

Return a matrix of random samples from the binomial distribution with parameters *n* and *p*, where *n* is the number of trials and *p* is the probability of success.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *n* and *p*.

`cauchy_rnd` (*location*, *scale*) [Function File]
`cauchy_rnd` (*location*, *scale*, *r*) [Function File]
`cauchy_rnd` (*location*, *scale*, *r*, *c*, ...) [Function File]
`cauchy_rnd` (*location*, *scale*, [*sz*]) [Function File]

Return a matrix of random samples from the Cauchy distribution with parameters *location* and *scale*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *location* and *scale*.

`chi2rnd` (*n*) [Function File]
`chi2rnd` (*n*, *r*) [Function File]
`chi2rnd` (*n*, *r*, *c*, ...) [Function File]
`chi2rnd` (*n*, [*sz*]) [Function File]

Return a matrix of random samples from the chi-square distribution with *n* degrees of freedom.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the size of *n*.

`discrete_rnd` (*v*, *p*) [Function File]
`discrete_rnd` (*v*, *p*, *r*) [Function File]
`discrete_rnd` (*v*, *p*, *r*, *c*, ...) [Function File]
`discrete_rnd` (*v*, *p*, [*sz*]) [Function File]

Return a matrix of random samples from the univariate distribution which assumes the values in *v* with probabilities *p*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *v* and *p*.

`empirical_rnd` (*data*) [Function File]
`empirical_rnd` (*data*, *r*) [Function File]
`empirical_rnd` (*data*, *r*, *c*, ...) [Function File]
`empirical_rnd` (*data*, [*sz*]) [Function File]

Return a matrix of random samples from the empirical distribution obtained from the univariate sample *data*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is a random ordering of the sample *data*.

`exprnd` (*lambda*) [Function File]
`exprnd` (*lambda*, *r*) [Function File]
`exprnd` (*lambda*, *r*, *c*, ...) [Function File]
`exprnd` (*lambda*, [*sz*]) [Function File]

Return a matrix of random samples from the exponential distribution with mean *lambda*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the size of *lambda*.

`frnd` (*m*, *n*) [Function File]
`frnd` (*m*, *n*, *r*) [Function File]
`frnd` (*m*, *n*, *r*, *c*, ...) [Function File]
`frnd` (*m*, *n*, [*sz*]) [Function File]

Return a matrix of random samples from the F distribution with *m* and *n* degrees of freedom.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *m* and *n*.

`gamrnd` (*a*, *b*) [Function File]
`gamrnd` (*a*, *b*, *r*) [Function File]
`gamrnd` (*a*, *b*, *r*, *c*, ...) [Function File]
`gamrnd` (*a*, *b*, [*sz*]) [Function File]

Return a matrix of random samples from the Gamma distribution with shape parameter *a* and scale *b*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *a* and *b*.

`geornd` (*p*) [Function File]
`geornd` (*p*, *r*) [Function File]
`geornd` (*p*, *r*, *c*, ...) [Function File]
`geornd` (*p*, [*sz*]) [Function File]

Return a matrix of random samples from the geometric distribution with parameter *p*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the size of *p*.

The geometric distribution models the number of failures (*x*-1) of a Bernoulli trial with probability *p* before the first success (*x*).

`hygernd` (*t*, *m*, *n*) [Function File]
`hygernd` (*t*, *m*, *n*, *r*) [Function File]
`hygernd` (*t*, *m*, *n*, *r*, *c*, ...) [Function File]
`hygernd` (*t*, *m*, *n*, [*sz*]) [Function File]

Return a matrix of random samples from the hypergeometric distribution with parameters *t*, *m*, and *n*.

The parameters *t*, *m*, and *n* must be positive integers with *m* and *n* not greater than *t*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *t*, *m*, and *n*.

`laplace_rnd` (*r*) [Function File]
`laplace_rnd` (*r*, *c*, ...) [Function File]
`laplace_rnd` ([*sz*]) [Function File]

Return a matrix of random samples from the Laplace distribution.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

`logistic_rnd` (*r*) [Function File]
`logistic_rnd` (*r*, *c*, ...) [Function File]
`logistic_rnd` ([*sz*]) [Function File]

Return a matrix of random samples from the logistic distribution.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

`lognrnd` (*mu*, *sigma*) [Function File]
`lognrnd` (*mu*, *sigma*, *r*) [Function File]
`lognrnd` (*mu*, *sigma*, *r*, *c*, ...) [Function File]
`lognrnd` (*mu*, *sigma*, [*sz*]) [Function File]

Return a matrix of random samples from the lognormal distribution with parameters *mu* and *sigma*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *mu* and *sigma*.

`nbinrnd` (*n*, *p*) [Function File]
`nbinrnd` (*n*, *p*, *r*) [Function File]
`nbinrnd` (*n*, *p*, *r*, *c*, ...) [Function File]
`nbinrnd` (*n*, *p*, [*sz*]) [Function File]

Return a matrix of random samples from the negative binomial distribution with parameters *n* and *p*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *n* and *p*.

`normrnd` (*mu*, *sigma*) [Function File]
`normrnd` (*mu*, *sigma*, *r*) [Function File]
`normrnd` (*mu*, *sigma*, *r*, *c*, ...) [Function File]
`normrnd` (*mu*, *sigma*, [*sz*]) [Function File]

Return a matrix of random samples from the normal distribution with parameters mean *mu* and standard deviation *sigma*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *mu* and *sigma*.

`poissrnd` (*lambda*) [Function File]
`poissrnd` (*lambda*, *r*) [Function File]
`poissrnd` (*lambda*, *r*, *c*, ...) [Function File]
`poissrnd` (*lambda*, [*sz*]) [Function File]

Return a matrix of random samples from the Poisson distribution with parameter *lambda*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the size of *lambda*.

`stdnormal_rnd` (*r*) [Function File]
`stdnormal_rnd` (*r*, *c*, ...) [Function File]

`stdnormal_rnd ([sz])` [Function File]

Return a matrix of random samples from the standard normal distribution (mean = 0, standard deviation = 1).

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

`trnd (n)` [Function File]
`trnd (n, r)` [Function File]
`trnd (n, r, c, ...)` [Function File]
`trnd (n, [sz])` [Function File]

Return a matrix of random samples from the t (Student) distribution with *n* degrees of freedom.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the size of *n*.

`unidrnd (n)` [Function File]
`unidrnd (n, r)` [Function File]
`unidrnd (n, r, c, ...)` [Function File]
`unidrnd (n, [sz])` [Function File]

Return a matrix of random samples from the discrete uniform distribution which assumes the integer values 1–*n* with equal probability.

n may be a scalar or a multi-dimensional array.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the size of *n*.

`unifrnd (a, b)` [Function File]
`unifrnd (a, b, r)` [Function File]
`unifrnd (a, b, r, c, ...)` [Function File]
`unifrnd (a, b, [sz])` [Function File]

Return a matrix of random samples from the uniform distribution on [*a*, *b*].

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *a* and *b*.

`wblrnd (scale, shape)` [Function File]
`wblrnd (scale, shape, r)` [Function File]

`wblrnd` (*scale*, *shape*, *r*, *c*, ...) [Function File]
`wblrnd` (*scale*, *shape*, [*sz*]) [Function File]

Return a matrix of random samples from the Weibull distribution with parameters *scale* and *shape*.

When called with a single size argument, return a square matrix with the dimension specified. When called with more than one scalar argument the first two arguments are taken as the number of rows and columns and any further arguments specify additional matrix dimensions. The size may also be specified with a vector of dimensions *sz*.

If no size arguments are given then the result matrix is the common size of *scale* and *shape*.

`wienrnd` (*t*, *d*, *n*) [Function File]

Return a simulated realization of the d-dimensional Wiener Process on the interval $[0, t]$.

If d is omitted, $d = 1$ is used. The first column of the return matrix contains time, the remaining columns contain the Wiener process.

The optional parameter n defines the number of summands used for simulating the process over an interval of length 1. If n is omitted, $n = 1000$ is used.

27 Sets

Octave has a number of functions for managing sets of data. A set is defined as a collection of unique elements and is typically represented by a vector of numbers sorted in ascending order. Any vector or matrix can be converted to a set by removing duplicates through the use of the `unique` function. However, it isn't necessary to explicitly create a set as all of the functions which operate on sets will convert their input to a set before proceeding.

`unique (x)` [Function File]
`unique (x, "rows")` [Function File]
`[y, i, j] = unique (...)` [Function File]
`[y, i, j] = unique (..., "first")` [Function File]
`[y, i, j] = unique (..., "last")` [Function File]

Return the unique elements of *x* sorted in ascending order.

If the input *x* is a column vector then return a column vector; Otherwise, return a row vector. *x* may also be a cell array of strings.

If the optional argument `"rows"` is given then return the unique rows of *x* sorted in ascending order. The input must be a 2-D matrix to use this option.

If requested, return index vectors *i* and *j* such that `y = x(i)` and `x = y(j)`.

Additionally, if *i* is a requested output then one of `"first"` or `"last"` may be given as an input. If `"last"` is specified, return the highest possible indices in *i*, otherwise, if `"first"` is specified, return the lowest. The default is `"last"`.

See also: [union], page 636, [intersect], page 635, [setdiff], page 636, [setxor], page 636, [ismember], page 636.

27.1 Set Operations

Octave supports several basic set operations. Octave can compute the union, intersection, and difference of two sets. Octave also supports the *Exclusive Or* set operation.

The functions for set operations all work in the same way by accepting two input sets and returning a third set. As an example, assume that `a` and `b` contains two sets, then

```
union (a, b)
```

computes the union of the two sets.

Finally, determining whether elements belong to a set can be done with the `ismember` function. Because sets are ordered this operation is very efficient and is of order O(log2(n)) which is preferable to the `find` function which is of order O(n).

`c = intersect (a, b)` [Function File]
`c = intersect (a, b, "rows")` [Function File]
`[c, ia, ib] = intersect (...)` [Function File]

Return the unique elements common to both *a* and *b* sorted in ascending order.

If *a* and *b* are both row vectors then return a row vector; Otherwise, return a column vector. The inputs may also be cell arrays of strings.

If the optional input `"rows"` is given then return the common rows of *a* and *b*. The inputs must be 2-D matrices to use this option.

If requested, return index vectors *ia* and *ib* such that `c = a(ia)` and `c = b(ib)`.

See also: [unique], page 635, [union], page 636, [setdiff], page 636, [setxor], page 636, [ismember], page 636.

`c = union (a, b)` [Function File]
`c = union (a, b, "rows")` [Function File]
`[c, ia, ib] = union (...)` [Function File]

Return the unique elements that are in either *a* or *b* sorted in ascending order.

If *a* and *b* are both row vectors then return a row vector; Otherwise, return a column vector. The inputs may also be cell arrays of strings.

If the optional input `"rows"` is given then return rows that are in either *a* or *b*. The inputs must be 2-D matrices to use this option.

The optional outputs *ia* and *ib* are index vectors such that `a(ia)` and `b(ib)` are disjoint sets whose union is *c*.

See also: [unique], page 635, [intersect], page 635, [setdiff], page 636, [setxor], page 636, [ismember], page 636.

`c = setdiff (a, b)` [Function File]
`c = setdiff (a, b, "rows")` [Function File]
`[c, ia] = setdiff (...)` [Function File]

Return the unique elements in *a* that are not in *b* sorted in ascending order.

If *a* is a row vector return a column vector; Otherwise, return a column vector. The inputs may also be cell arrays of strings.

If the optional input `"rows"` is given then return the rows in *a* that are not in *b*. The inputs must be 2-D matrices to use this option.

If requested, return the index vector *ia* such that `c = a(ia)`.

See also: [unique], page 635, [union], page 636, [intersect], page 635, [setxor], page 636, [ismember], page 636.

`c = setxor (a, b)` [Function File]
`c = setxor (a, b, "rows")` [Function File]
`[c, ia, ib] = setxor (...)` [Function File]

Return the unique elements exclusive to sets *a* or *b* sorted in ascending order.

If *a* and *b* are both row vectors then return a row vector; Otherwise, return a column vector. The inputs may also be cell arrays of strings.

If the optional input `"rows"` is given then return the rows exclusive to sets *a* and *b*. The inputs must be 2-D matrices to use this option.

If requested, return index vectors *ia* and *ib* such that `a(ia)` and `b(ib)` are disjoint sets whose union is *c*.

See also: [unique], page 635, [union], page 636, [intersect], page 635, [setdiff], page 636, [ismember], page 636.

`tf = ismember (a, s)` [Function File]
`tf = ismember (a, s, "rows")` [Function File]
`[tf, s_idx] = ismember (...)` [Function File]

Return a logical matrix *tf* with the same shape as *a* which is true (1) if the element in *a* is found in *s* and false (0) if it is not.

If a second output argument is requested then the index into *s* of each matching element is also returned.

```
a = [3, 10, 1];
s = [0:9];
[tf, s_idx] = ismember (a, s)
     ⇒ tf = [1, 0, 1]
     ⇒ s_idx = [4, 0, 2]
```

The inputs *a* and *s* may also be cell arrays.

```
a = {"abc"};
s = {"abc", "def"};
[tf, s_idx] = ismember (a, s)
     ⇒ tf = [1, 0]
     ⇒ s_idx = [1, 0]
```

If the optional third argument `"rows"` is given then compare rows in *a* with rows in *s*. The inputs must be 2-D matrices with the same number of columns to use this option.

```
a = [1:3; 5:7; 4:6];
s = [0:2; 1:3; 2:4; 3:5; 4:6];
[tf, s_idx] = ismember (a, s, "rows")
     ⇒ tf = logical ([1; 0; 1])
     ⇒ s_idx = [2; 0; 5];
```

See also: [lookup], page 408, [unique], page 635, [union], page 636, [intersect], page 635, [setdiff], page 636, [setxor], page 636.

`powerset (a)` [Function File]
`powerset (a, "`*rows*`")` [Function File]

Compute the powerset (all subsets) of the set *a*.

The set *a* must be a numerical matrix or a cell array of strings. The output will always be a cell array of either vectors or strings.

With the optional argument `"rows"`, each row of the set *a* is considered one element of the set. The input must be a 2-D numeric matrix to use this argument.

See also: [unique], page 635, [union], page 636, [intersect], page 635, [setdiff], page 636, [setxor], page 636, [ismember], page 636.

28 Polynomial Manipulations

In Octave, a polynomial is represented by its coefficients (arranged in descending order). For example, a vector c of length $N+1$ corresponds to the following polynomial of order N

$$p(x) = c_1 x^N + \ldots + c_N x + c_{N+1}.$$

28.1 Evaluating Polynomials

The value of a polynomial represented by the vector c can be evaluated at the point x very easily, as the following example shows:

```
N = length (c) - 1;
val = dot (x.^(N:-1:0), c);
```

While the above example shows how easy it is to compute the value of a polynomial, it isn't the most stable algorithm. With larger polynomials you should use more elegant algorithms, such as Horner's Method, which is exactly what the Octave function `polyval` does.

In the case where x is a square matrix, the polynomial given by c is still well-defined. As when x is a scalar the obvious implementation is easily expressed in Octave, but also in this case more elegant algorithms perform better. The `polyvalm` function provides such an algorithm.

y = `polyval` (*p*, *x*) [Function File]
y = `polyval` (*p*, *x*, [], *mu*) [Function File]
[*y*, *dy*] = `polyval` (*p*, *x*, *s*) [Function File]
[*y*, *dy*] = `polyval` (*p*, *x*, *s*, *mu*) [Function File]

Evaluate the polynomial *p* at the specified values of *x*.

If *x* is a vector or matrix, the polynomial is evaluated for each of the elements of *x*.

When *mu* is present, evaluate the polynomial for (*x*-*mu*(1))/*mu*(2).

In addition to evaluating the polynomial, the second output represents the prediction interval, *y* `+/-` *dy*, which contains at least 50% of the future predictions. To calculate the prediction interval, the structured variable *s*, originating from `polyfit`, must be supplied.

See also: [polyvalm], page 639, [polyaffine], page 645, [polyfit], page 645, [roots], page 640, [poly], page 654.

`polyvalm` (*c*, *x*) [Function File]

Evaluate a polynomial in the matrix sense.

`polyvalm (c, x)` will evaluate the polynomial in the matrix sense, i.e., matrix multiplication is used instead of element by element multiplication as used in `polyval`.

The argument *x* must be a square matrix.

See also: [polyval], page 639, [roots], page 640, [poly], page 654.

28.2 Finding Roots

Octave can find the roots of a given polynomial. This is done by computing the companion matrix of the polynomial (see the `compan` function for a definition), and then finding its eigenvalues.

`roots (c)` [Function File]

Compute the roots of the polynomial *c*.

For a vector *c* with *N* components, return the roots of the polynomial

$$c_1 x^{N-1} + \cdots + c_{N-1} x + c_N.$$

As an example, the following code finds the roots of the quadratic polynomial

$$p(x) = x^2 - 5.$$

```
c = [1, 0, -5];
roots (c)
⇒  2.2361
⇒ -2.2361
```

Note that the true result is $\pm\sqrt{5}$ which is roughly ± 2.2361.

See also: [poly], page 654, [compan], page 640, [fzero], page 507.

`z = polyeig (C0, C1, ..., Cl)` [Function File]
`[v, z] = polyeig (C0, C1, ..., Cl)` [Function File]

Solve the polynomial eigenvalue problem of degree *l*.

Given an *n*n* matrix polynomial

`C(s) = C0 + C1 s + ... + Cl s^l`

`polyeig` solves the eigenvalue problem

`(C0 + C1 + ... + Cl)v = 0.`

Note that the eigenvalues *z* are the zeros of the matrix polynomial. *z* is a row vector with *n*l* elements. *v* is a matrix (*n* x *n*l*) with columns that correspond to the eigenvectors.

See also: [eig], page 467, [eigs], page 545, [compan], page 640.

`compan (c)` [Function File]

Compute the companion matrix corresponding to polynomial coefficient vector *c*.

The companion matrix is

$$A = \begin{bmatrix} -c_2/c_1 & -c_3/c_1 & \cdots & -c_N/c_1 & -c_{N+1}/c_1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

The eigenvalues of the companion matrix are equal to the roots of the polynomial.

See also: [roots], page 640, [poly], page 654, [eig], page 467.

`[multp, idxp] = mpoles (p)` [Function File]
`[multp, idxp] = mpoles (p, tol)` [Function File]
`[multp, idxp] = mpoles (p, tol, reorder)` [Function File]

Identify unique poles in *p* and their associated multiplicity.

The output is ordered from largest pole to smallest pole.

If the relative difference of two poles is less than *tol* then they are considered to be multiples. The default value for *tol* is 0.001.

If the optional parameter *reorder* is zero, poles are not sorted.

The output *multp* is a vector specifying the multiplicity of the poles. `multp(n)` refers to the multiplicity of the Nth pole `p(idxp(n))`.

For example:

```
p = [2 3 1 1 2];
[m, n] = mpoles (p)
   ⇒ m = [1; 1; 2; 1; 2]
   ⇒ n = [2; 5; 1; 4; 3]
   ⇒ p(n) = [3, 2, 2, 1, 1]
```

See also: [residue], page 643, [poly], page 654, [roots], page 640, [conv], page 641, [deconv], page 642.

28.3 Products of Polynomials

`conv (a, b)` [Function File]
`conv (a, b, shape)` [Function File]

Convolve two vectors *a* and *b*.

The output convolution is a vector with length equal to `length (a) + length (b) - 1`. When *a* and *b* are the coefficient vectors of two polynomials, the convolution represents the coefficient vector of the product polynomial.

The optional *shape* argument may be

shape = `"full"`
: Return the full convolution. (default)

shape = `"same"`
: Return the central part of the convolution with the same size as *a*.

See also: [deconv], page 642, [conv2], page 642, [convn], page 641, [fftconv], page 684.

`C = convn (A, B)` [Built-in Function]
`C = convn (A, B, shape)` [Built-in Function]

Return the n-D convolution of *A* and *B*.

The size of the result is determined by the optional *shape* argument which takes the following values

shape = `"full"`
: Return the full convolution. (default)

shape = "same"
: Return central part of the convolution with the same size as *A*. The central part of the convolution begins at the indices `floor ([size(B)/2] + 1)`.

shape = "valid"
: Return only the parts which do not include zero-padded edges. The size of the result is `max (size (A) - size (B) + 1, 0)`.

See also: [conv2], page 642, [conv], page 641.

`deconv (y, a)` [Function File]

Deconvolve two vectors.

`[b, r] = deconv (y, a)` solves for *b* and *r* such that `y = conv (a, b) + r`.

If *y* and *a* are polynomial coefficient vectors, *b* will contain the coefficients of the polynomial quotient and *r* will be a remainder polynomial of lowest order.

See also: [conv], page 641, [residue], page 643.

`conv2 (A, B)` [Built-in Function]
`conv2 (v1, v2, m)` [Built-in Function]
`conv2 (..., shape)` [Built-in Function]

Return the 2-D convolution of *A* and *B*.

The size of the result is determined by the optional *shape* argument which takes the following values

shape = "full"
: Return the full convolution. (default)

shape = "same"
: Return the central part of the convolution with the same size as *A*. The central part of the convolution begins at the indices `floor ([size(B)/2] + 1)`.

shape = "valid"
: Return only the parts which do not include zero-padded edges. The size of the result is `max (size (A) - size (B) + 1, 0)`.

When the third argument is a matrix, return the convolution of the matrix *m* by the vector *v1* in the column direction and by the vector *v2* in the row direction.

See also: [conv], page 641, [convn], page 641.

`q = polygcd (b, a)` [Function File]
`q = polygcd (b, a, tol)` [Function File]

Find the greatest common divisor of two polynomials.

This is equivalent to the polynomial found by multiplying together all the common roots. Together with deconv, you can reduce a ratio of two polynomials.

The tolerance *tol* defaults to `sqrt (eps)`.

Caution: This is a numerically unstable algorithm and should not be used on large polynomials.

Example code:

```
polygcd (poly (1:8), poly (3:12)) - poly (3:8)
⇒ [ 0, 0, 0, 0, 0, 0, 0 ]
deconv (poly (1:8), polygcd (poly (1:8), poly (3:12))) - poly (1:2)
⇒ [ 0, 0, 0 ]
```

See also: [poly], page 654, [roots], page 640, [conv], page 641, [deconv], page 642, [residue], page 643.

`[r, p, k, e] = residue (b, a)` [Function File]
`[b, a] = residue (r, p, k)` [Function File]
`[b, a] = residue (r, p, k, e)` [Function File]

The first calling form computes the partial fraction expansion for the quotient of the polynomials, *b* and *a*.

The quotient is defined as

$$\frac{B(s)}{A(s)} = \sum_{m=1}^{M} \frac{r_m}{(s-p_m)_m^e} + \sum_{i=1}^{N} k_i s^{N-i}.$$

where M is the number of poles (the length of the *r*, *p*, and *e*), the *k* vector is a polynomial of order $N-1$ representing the direct contribution, and the *e* vector specifies the multiplicity of the m-th residue's pole.

For example,

```
b = [1, 1, 1];
a = [1, -5, 8, -4];
[r, p, k, e] = residue (b, a)
   ⇒ r = [-2; 7; 3]
   ⇒ p = [2; 2; 1]
   ⇒ k = [](0x0)
   ⇒ e = [1; 2; 1]
```

which represents the following partial fraction expansion

$$\frac{s^2+s+1}{s^3-5s^2+8s-4} = \frac{-2}{s-2} + \frac{7}{(s-2)^2} + \frac{3}{s-1}$$

The second calling form performs the inverse operation and computes the reconstituted quotient of polynomials, *b*(s)/*a*(s), from the partial fraction expansion; represented by the residues, poles, and a direct polynomial specified by *r*, *p* and *k*, and the pole multiplicity *e*.

If the multiplicity, *e*, is not explicitly specified the multiplicity is determined by the function `mpoles`.

For example:

```
r = [-2; 7; 3];
p = [2; 2; 1];
k = [1, 0];
[b, a] = residue (r, p, k)
   ⇒ b = [1, -5, 9, -3, 1]
   ⇒ a = [1, -5, 8, -4]

where mpoles is used to determine e = [1; 2; 1]
```

Alternatively the multiplicity may be defined explicitly, for example,

```
r = [7; 3; -2];
p = [2; 1; 2];
k = [1, 0];
e = [2; 1; 1];
[b, a] = residue (r, p, k, e)
   ⇒ b = [1, -5, 9, -3, 1]
   ⇒ a = [1, -5, 8, -4]
```

which represents the following partial fraction expansion

$$\frac{-2}{s-2} + \frac{7}{(s-2)^2} + \frac{3}{s-1} + s = \frac{s^4 - 5s^3 + 9s^2 - 3s + 1}{s^3 - 5s^2 + 8s - 4}$$

See also: [mpoles], page 641, [poly], page 654, [roots], page 640, [conv], page 641, [deconv], page 642.

28.4 Derivatives / Integrals / Transforms

Octave comes with functions for computing the derivative and the integral of a polynomial. The functions `polyder` and `polyint` both return new polynomials describing the result. As an example we'll compute the definite integral of $p(x) = x^2 + 1$ from 0 to 3.

```
c = [1, 0, 1];
integral = polyint (c);
area = polyval (integral, 3) - polyval (integral, 0)
⇒ 12
```

`polyder (`*p*`)` [Function File]
`[`*k*`] = polyder (`*a*`, `*b*`)` [Function File]
`[`*q*`, `*d*`] = polyder (`*b*`, `*a*`)` [Function File]

Return the coefficients of the derivative of the polynomial whose coefficients are given by the vector *p*.

If a pair of polynomials is given, return the derivative of the product $a * b$.

If two inputs and two outputs are given, return the derivative of the polynomial quotient b/a. The quotient numerator is in *q* and the denominator in *d*.

See also: [polyint], page 644, [polyval], page 639, [polyreduce], page 654.

`polyint (`*p*`)` [Function File]
`polyint (`*p*`, `*k*`)` [Function File]

Return the coefficients of the integral of the polynomial whose coefficients are represented by the vector *p*.

The variable k is the constant of integration, which by default is set to zero.

See also: [polyder], page 644, [polyval], page 639.

`polyaffine (f, mu)` [Function File]

Return the coefficients of the polynomial vector f after an affine transformation.

If f is the vector representing the polynomial f(x), then `g = polyaffine (f, mu)` is the vector representing:

```
g(x) = f( (x - mu(1)) / mu(2) )
```

See also: [polyval], page 639, [polyfit], page 645.

28.5 Polynomial Interpolation

Octave comes with good support for various kinds of interpolation, most of which are described in Chapter 29 [Interpolation], page 655. One simple alternative to the functions described in the aforementioned chapter, is to fit a single polynomial, or a piecewise polynomial (spline) to some given data points. To avoid a highly fluctuating polynomial, one most often wants to fit a low-order polynomial to data. This usually means that it is necessary to fit the polynomial in a least-squares sense, which just is what the `polyfit` function does.

`p = polyfit (x, y, n)` [Function File]
`[p, s] = polyfit (x, y, n)` [Function File]
`[p, s, mu] = polyfit (x, y, n)` [Function File]

Return the coefficients of a polynomial $p(x)$ of degree n that minimizes the least-squares-error of the fit to the points `[x, y]`.

If n is a logical vector, it is used as a mask to selectively force the corresponding polynomial coefficients to be used or ignored.

The polynomial coefficients are returned in a row vector.

The optional output s is a structure containing the following fields:

‘`R`’ Triangular factor R from the QR decomposition.

‘`X`’ The Vandermonde matrix used to compute the polynomial coefficients.

‘`C`’ The unscaled covariance matrix, formally equal to the inverse of $x'*x$, but computed in a way minimizing roundoff error propagation.

‘`df`’ The degrees of freedom.

‘`normr`’ The norm of the residuals.

‘`yf`’ The values of the polynomial for each value of x.

The second output may be used by `polyval` to calculate the statistical error limits of the predicted values. In particular, the standard deviation of p coefficients is given by

`sqrt (diag (s.C)/s.df)*s.normr`.

When the third output, mu, is present the coefficients, p, are associated with a polynomial in

`xhat = (x - mu(1)) / mu(2)`
where $mu(1) = \text{mean}\ (x)$, and $mu(2) = \text{std}\ (x)$.

This linear transformation of *x* improves the numerical stability of the fit.

See also: [polyval], page 639, [polyaffine], page 645, [roots], page 640, [vander], page 434, [zscore], page 606.

In situations where a single polynomial isn't good enough, a solution is to use several polynomials pieced together. The function `splinefit` fits a peicewise polynomial (spline) to a set of data.

pp = splinefit (*x*, *y*, *breaks*) [Function File]
pp = splinefit (*x*, *y*, *p*) [Function File]
pp = splinefit (. . . , "*periodic*", *periodic*) [Function File]
pp = splinefit (. . . , "*robust*", *robust*) [Function File]
pp = splinefit (. . . , "*beta*", *beta*) [Function File]
pp = splinefit (. . . , "*order*", *order*) [Function File]
pp = splinefit (. . . , "*constraints*", *constraints*) [Function File]

Fit a piecewise cubic spline with breaks (knots) *breaks* to the noisy data, *x* and *y*.

x is a vector, and *y* is a vector or N-D array. If *y* is an N-D array, then *x*(j) is matched to *y*(:,. . .,:,j).

p is a positive integer defining the number of intervals along *x*, and *p*+1 is the number of breaks. The number of points in each interval differ by no more than 1.

The optional property *periodic* is a logical value which specifies whether a periodic boundary condition is applied to the spline. The length of the period is `max (breaks) - min (breaks)`. The default value is `false`.

The optional property *robust* is a logical value which specifies if robust fitting is to be applied to reduce the influence of outlying data points. Three iterations of weighted least squares are performed. Weights are computed from previous residuals. The sensitivity of outlier identification is controlled by the property *beta*. The value of *beta* is restricted to the range, $0 < beta < 1$. The default value is $beta = 1/2$. Values close to 0 give all data equal weighting. Increasing values of *beta* reduce the influence of outlying data. Values close to unity may cause instability or rank deficiency.

The fitted spline is returned as a piecewise polynomial, *pp*, and may be evaluated using `ppval`.

The splines are constructed of polynomials with degree *order*. The default is a cubic, *order*=3. A spline with P pieces has P+*order* degrees of freedom. With periodic boundary conditions the degrees of freedom are reduced to P.

The optional property, *constaints*, is a structure specifying linear constraints on the fit. The structure has three fields, `"xc"`, `"yc"`, and `"cc"`.

`"xc"` Vector of the x-locations of the constraints.

`"yc"` Constraining values at the locations *xc*. The default is an array of zeros.

`"cc"` Coefficients (matrix). The default is an array of ones. The number of rows is limited to the order of the piecewise polynomials, *order*.

Constraints are linear combinations of derivatives of order 0 to *order*-1 according to

$$cc(1,j) \cdot y(xc(j)) + cc(2,j) \cdot y\prime(xc(j)) + ... = yc(:,\ldots,:,j).$$

See also: [interp1], page 655, [unmkpp], page 653, [ppval], page 653, [spline], page 658, [pchip], page 690, [ppder], page 653, [ppint], page 653, [ppjumps], page 653.

The number of *breaks* (or knots) used to construct the piecewise polynomial is a significant factor in suppressing the noise present in the input data, *x* and *y*. This is demonstrated by the example below.

```
x = 2 * pi * rand (1, 200);
y = sin (x) + sin (2 * x) + 0.2 * randn (size (x));
## Uniform breaks
breaks = linspace (0, 2 * pi, 41); % 41 breaks, 40 pieces
pp1 = splinefit (x, y, breaks);
## Breaks interpolated from data
pp2 = splinefit (x, y, 10);  % 11 breaks, 10 pieces
## Plot
xx = linspace (0, 2 * pi, 400);
y1 = ppval (pp1, xx);
y2 = ppval (pp2, xx);
plot (x, y, ".", xx, [y1; y2])
axis tight
ylim auto
legend ({"data", "41 breaks, 40 pieces", "11 breaks, 10 pieces"})
```

The result of which can be seen in Figure 28.1.

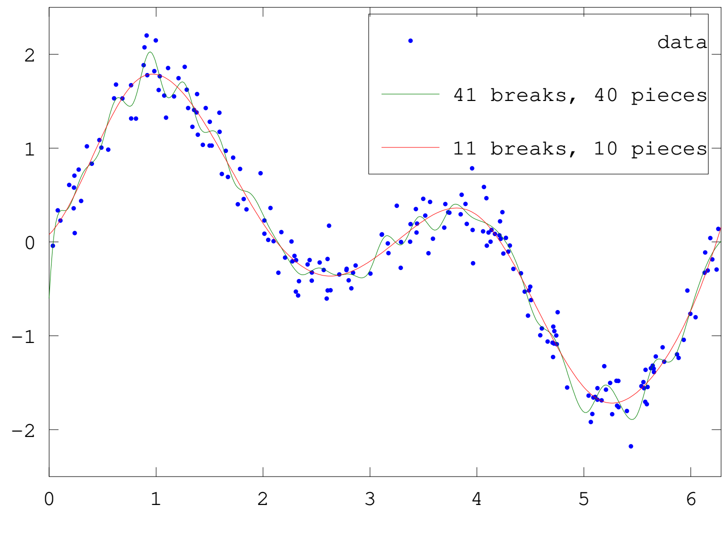

Figure 28.1: Comparison of a fitting a piecewise polynomial with 41 breaks to one with 11 breaks. The fit with the large number of breaks exhibits a fast ripple that is not present in the underlying function.

The piecewise polynomial fit, provided by `splinefit`, has continuous derivatives up to the *order*-1. For example, a cubic fit has continuous first and second derivatives. This is demonstrated by the code

```
## Data (200 points)
x = 2 * pi * rand (1, 200);
y = sin (x) + sin (2 * x) + 0.1 * randn (size (x));
## Piecewise constant
pp1 = splinefit (x, y, 8, "order", 0);
## Piecewise linear
pp2 = splinefit (x, y, 8, "order", 1);
## Piecewise quadratic
pp3 = splinefit (x, y, 8, "order", 2);
## Piecewise cubic
pp4 = splinefit (x, y, 8, "order", 3);
## Piecewise quartic
pp5 = splinefit (x, y, 8, "order", 4);
## Plot
xx = linspace (0, 2 * pi, 400);
y1 = ppval (pp1, xx);
y2 = ppval (pp2, xx);
y3 = ppval (pp3, xx);
y4 = ppval (pp4, xx);
y5 = ppval (pp5, xx);
plot (x, y, ".", xx, [y1; y2; y3; y4; y5])
axis tight
ylim auto
legend ({"data", "order 0", "order 1", "order 2", "order 3", "order 4"})
```

The result of which can be seen in Figure 28.2.

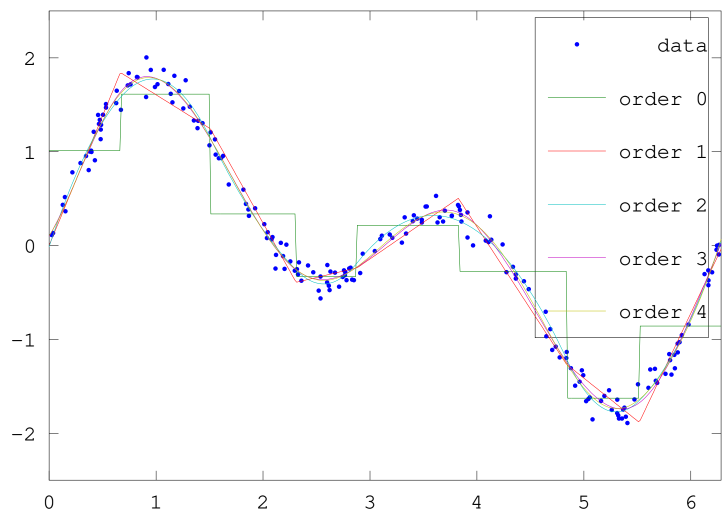

Figure 28.2: Comparison of a piecewise constant, linear, quadratic, cubic, and quartic polynomials with 8 breaks to noisy data. The higher order solutions more accurately represent the underlying function, but come with the expense of computational complexity.

When the underlying function to provide a fit to is periodic, `splinefit` is able to apply the boundary conditions needed to manifest a periodic fit. This is demonstrated by the code below.

```
## Data (100 points)
x = 2 * pi * [0, (rand (1, 98)), 1];
y = sin (x) - cos (2 * x) + 0.2 * randn (size (x));
## No constraints
pp1 = splinefit (x, y, 10, "order", 5);
## Periodic boundaries
pp2 = splinefit (x, y, 10, "order", 5, "periodic", true);
## Plot
xx = linspace (0, 2 * pi, 400);
y1 = ppval (pp1, xx);
y2 = ppval (pp2, xx);
plot (x, y, ".", xx, [y1; y2])
axis tight
ylim auto
legend ({"data", "no constraints", "periodic"})
```

The result of which can be seen in Figure 28.3.

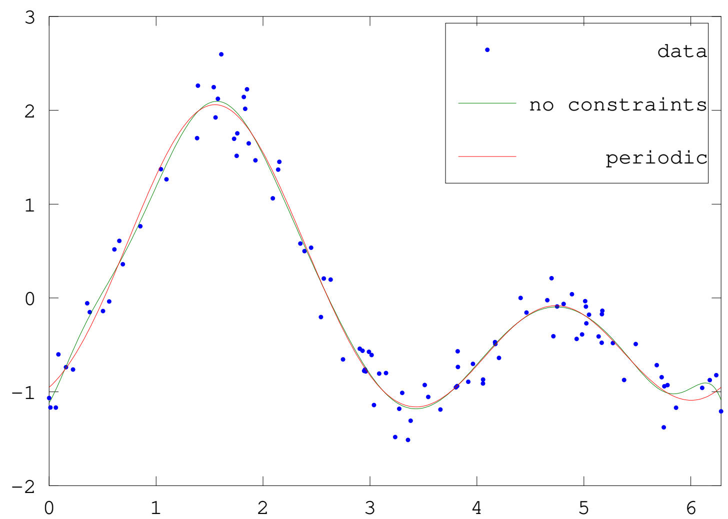

Figure 28.3: Comparison of piecewise polynomial fits to a noisy periodic function with, and without, periodic boundary conditions.

More complex constraints may be added as well. For example, the code below illustrates a periodic fit with values that have been clamped at the endpoints, and a second periodic fit which is hinged at the endpoints.

```
## Data (200 points)
x = 2 * pi * rand (1, 200);
y = sin (2 * x) + 0.1 * randn (size (x));
## Breaks
breaks = linspace (0, 2 * pi, 10);
## Clamped endpoints, y = y' = 0
xc = [0, 0, 2*pi, 2*pi];
cc = [(eye (2)), (eye (2))];
con = struct ("xc", xc, "cc", cc);
pp1 = splinefit (x, y, breaks, "constraints", con);
## Hinged periodic endpoints, y = 0
con = struct ("xc", 0);
pp2 = splinefit (x, y, breaks, "constraints", con, "periodic", true);
## Plot
xx = linspace (0, 2 * pi, 400);
y1 = ppval (pp1, xx);
y2 = ppval (pp2, xx);
plot (x, y, ".", xx, [y1; y2])
axis tight
ylim auto
legend ({"data", "clamped", "hinged periodic"})
```

The result of which can be seen in Figure 28.4.

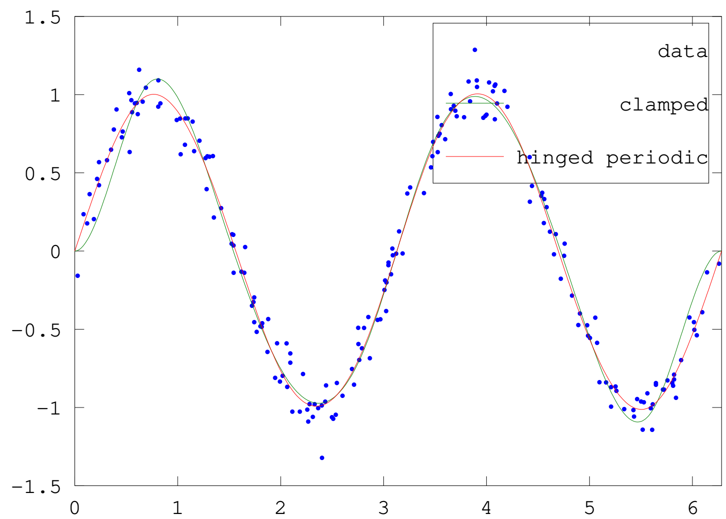

Figure 28.4: Comparison of two periodic piecewise cubic fits to a noisy periodic signal. One fit has its endpoints clamped and the second has its endpoints hinged.

The `splinefit` function also provides the convenience of a *robust* fitting, where the effect of outlying data is reduced. In the example below, three different fits are provided. Two with differing levels of outlier suppression and a third illustrating the non-robust solution.

```
## Data
x = linspace (0, 2*pi, 200);
y = sin (x) + sin (2 * x) + 0.05 * randn (size (x));
## Add outliers
x = [x, linspace(0,2*pi,60)];
y = [y, -ones(1,60)];
## Fit splines with hinged conditions
con = struct ("xc", [0, 2*pi]);
## Robust fitting, beta = 0.25
pp1 = splinefit (x, y, 8, "constraints", con, "beta", 0.25);
## Robust fitting, beta = 0.75
pp2 = splinefit (x, y, 8, "constraints", con, "beta", 0.75);
## No robust fitting
pp3 = splinefit (x, y, 8, "constraints", con);
## Plot
xx = linspace (0, 2*pi, 400);
y1 = ppval (pp1, xx);
y2 = ppval (pp2, xx);
y3 = ppval (pp3, xx);
plot (x, y, ".", xx, [y1; y2; y3])
legend ({"data with outliers","robust, beta = 0.25", ...
         "robust, beta = 0.75", "no robust fitting"})
axis tight
```

```
ylim auto
```

The result of which can be seen in Figure 28.5.

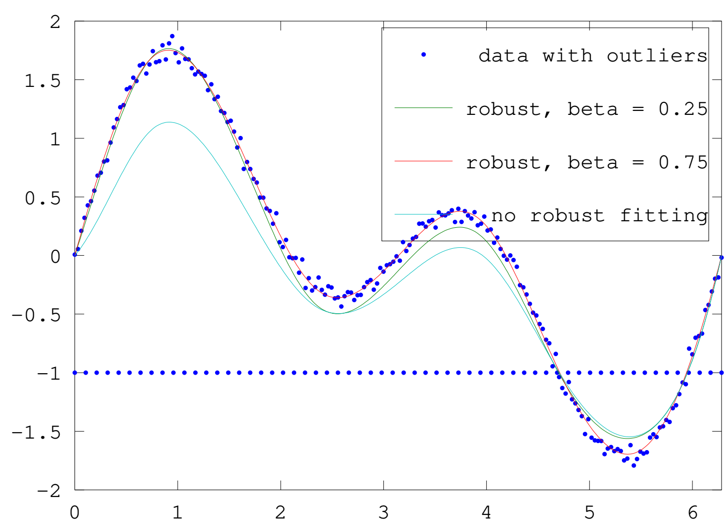

Figure 28.5: Comparison of two different levels of robust fitting (*beta* = 0.25 and 0.75) to noisy data combined with outlying data. A conventional fit, without robust fitting (*beta* = 0) is also included.

The function, `ppval`, evaluates the piecewise polynomials, created by `mkpp` or other means, and `unmkpp` returns detailed information about the piecewise polynomial.

The following example shows how to combine two linear functions and a quadratic into one function. Each of these functions is expressed on adjoined intervals.

```
x = [-2, -1, 1, 2];
p = [ 0,  1, 0;
      1, -2, 1;
      0, -1, 1 ];
pp = mkpp (x, p);
xi = linspace (-2, 2, 50);
yi = ppval (pp, xi);
plot (xi, yi);
```

pp = mkpp (*breaks*, *coefs*) [Function File]
pp = mkpp (*breaks*, *coefs*, *d*) [Function File]

Construct a piecewise polynomial (pp) structure from sample points *breaks* and coefficients *coefs*.

breaks must be a vector of strictly increasing values. The number of intervals is given by `ni = length (breaks) - 1`.

When *m* is the polynomial order *coefs* must be of size: *ni* x *m* + 1.

The i-th row of *coefs*, `coefs (i,:)`, contains the coefficients for the polynomial over the *i*-th interval, ordered from highest (*m*) to lowest (*0*).

coefs may also be a multi-dimensional array, specifying a vector-valued or array-valued polynomial. In that case the polynomial order is defined by the length of the last dimension of *coefs*. The size of first dimension(s) are given by the scalar or vector *d*. If *d* is not given it is set to `1`. In any case *coefs* is reshaped to a 2-D matrix of size `[ni*prod(d m)]`

See also: [unmkpp], page 653, [ppval], page 653, [spline], page 658, [pchip], page 690, [ppder], page 653, [ppint], page 653, [ppjumps], page 653.

`[x, p, n, k, d] = unmkpp (pp)` [Function File]

Extract the components of a piecewise polynomial structure *pp*.

The components are:

x Sample points.

p Polynomial coefficients for points in sample interval. `p (i, :)` contains the coefficients for the polynomial over interval *i* ordered from highest to lowest. If `d > 1`, `p (r, i, :)` contains the coefficients for the r-th polynomial defined on interval *i*.

n Number of polynomial pieces.

k Order of the polynomial plus 1.

d Number of polynomials defined for each interval.

See also: [mkpp], page 652, [ppval], page 653, [spline], page 658, [pchip], page 690.

`yi = ppval (pp, xi)` [Function File]

Evaluate the piecewise polynomial structure *pp* at the points *xi*.

If *pp* describes a scalar polynomial function, the result is an array of the same shape as *xi*. Otherwise, the size of the result is `[pp.dim, length(xi)]` if *xi* is a vector, or `[pp.dim, size(xi)]` if it is a multi-dimensional array.

See also: [mkpp], page 652, [unmkpp], page 653, [spline], page 658, [pchip], page 690.

`ppd = ppder (pp)` [Function File]
`ppd = ppder (pp, m)` [Function File]

Compute the piecewise *m*-th derivative of a piecewise polynomial struct *pp*.

If *m* is omitted the first derivative is calculated.

See also: [mkpp], page 652, [ppval], page 653, [ppint], page 653.

`ppi = ppint (pp)` [Function File]
`ppi = ppint (pp, c)` [Function File]

Compute the integral of the piecewise polynomial struct *pp*.

c, if given, is the constant of integration.

See also: [mkpp], page 652, [ppval], page 653, [ppder], page 653.

`jumps = ppjumps (pp)` [Function File]

Evaluate the boundary jumps of a piecewise polynomial.

If there are *n* intervals, and the dimensionality of *pp* is *d*, the resulting array has dimensions `[d, n-1]`.

See also: [mkpp], page 652.

28.6 Miscellaneous Functions

`poly (A)` [Function File]
`poly (x)` [Function File]

If A is a square N-by-N matrix, `poly (A)` is the row vector of the coefficients of `det (z * eye (N) - A)`, the characteristic polynomial of A.

For example, the following code finds the eigenvalues of A which are the roots of `poly (A)`.

```
roots (poly (eye (3)))
    ⇒ 1.00001 + 0.00001i
       1.00001 - 0.00001i
       0.99999 + 0.00000i
```

In fact, all three eigenvalues are exactly 1 which emphasizes that for numerical performance the `eig` function should be used to compute eigenvalues.

If x is a vector, `poly (x)` is a vector of the coefficients of the polynomial whose roots are the elements of x. That is, if c is a polynomial, then the elements of `d = roots (poly (c))` are contained in c. The vectors c and d are not identical, however, due to sorting and numerical errors.

See also: [roots], page 640, [eig], page 467.

`polyout (c)` [Function File]
`polyout (c, x)` [Function File]
`str = polyout (...)` [Function File]

Display a formatted version of the polynomial c.

The formatted polynomial

$$c(x) = c_1 x^n + \ldots + c_n x + c_{n+1}$$

is returned as a string or written to the screen if `nargout` is zero.

The second argument x specifies the variable name to use for each term and defaults to the string `"s"`.

See also: [polyreduce], page 654.

`polyreduce (c)` [Function File]

Reduce a polynomial coefficient vector to a minimum number of terms by stripping off any leading zeros.

See also: [polyout], page 654.

29 Interpolation

29.1 One-dimensional Interpolation

Octave supports several methods for one-dimensional interpolation, most of which are described in this section. Section 28.5 [Polynomial Interpolation], page 645, and Section 30.4 [Interpolation on Scattered Data], page 677, describe additional methods.

yi = `interp1` (*x*, *y*, *xi*) [Function File]
yi = `interp1` (*y*, *xi*) [Function File]
yi = `interp1` (..., *method*) [Function File]
yi = `interp1` (..., *extrap*) [Function File]
yi = `interp1` (..., "*left*") [Function File]
yi = `interp1` (..., "*right*") [Function File]
pp = `interp1` (..., "*pp*") [Function File]

One-dimensional interpolation.

Interpolate input data to determine the value of *yi* at the points *xi*. If not specified, *x* is taken to be the indices of *y* (`1:length (`*y*`)`). If *y* is a matrix or an N-dimensional array, the interpolation is performed on each column of *y*.

The interpolation *method* is one of:

`"nearest"`
: Return the nearest neighbor.

`"previous"`
: Return the previous neighbor.

`"next"`
: Return the next neighbor.

`"linear"` (default)
: Linear interpolation from nearest neighbors.

`"pchip"`
: Piecewise cubic Hermite interpolating polynomial—shape-preserving interpolation with smooth first derivative.

`"cubic"`
: Cubic interpolation (same as `"pchip"`).

`"spline"`
: Cubic spline interpolation—smooth first and second derivatives throughout the curve.

Adding '*' to the start of any method above forces `interp1` to assume that *x* is uniformly spaced, and only `x(1)` and `x(2)` are referenced. This is usually faster, and is never slower. The default method is `"linear"`.

If *extrap* is the string `"extrap"`, then extrapolate values beyond the endpoints using the current *method*. If *extrap* is a number, then replace values beyond the endpoints with that number. When unspecified, *extrap* defaults to `NA`.

If the string argument `"pp"` is specified, then *xi* should not be supplied and `interp1` returns a piecewise polynomial object. This object can later be used with `ppval` to evaluate the interpolation. There is an equivalence, such that `ppval (interp1 (`*x*`, `*y*`, `*method*`, "pp"), `*xi*`) == interp1 (`*x*`, `*y*`, `*xi*`, `*method*`, "extrap")`.

Duplicate points in *x* specify a discontinuous interpolant. There may be at most 2 consecutive points with the same value. If *x* is increasing, the default discontinuous interpolant is right-continuous. If *x* is decreasing, the default discontinuous interpolant is left-continuous. The continuity condition of the interpolant may be specified by using the options `"left"` or `"right"` to select a left-continuous or right-continuous interpolant, respectively. Discontinuous interpolation is only allowed for `"nearest"` and `"linear"` methods; in all other cases, the *x*-values must be unique.

An example of the use of `interp1` is

```
xf = [0:0.05:10];
yf = sin (2*pi*xf/5);
xp = [0:10];
yp = sin (2*pi*xp/5);
lin = interp1 (xp, yp, xf);
near = interp1 (xp, yp, xf, "nearest");
pch = interp1 (xp, yp, xf, "pchip");
spl = interp1 (xp, yp, xf, "spline");
plot (xf,yf,"r", xf,near,"g", xf,lin,"b", xf,pch,"c", xf,spl,"m",
      xp,yp,"r*");
legend ("original", "nearest", "linear", "pchip", "spline");
```

See also: [pchip], page 690, [spline], page 658, [interpft], page 657, [interp2], page 659, [interp3], page 660, [interpn], page 661.

There are some important differences between the various interpolation methods. The `"spline"` method enforces that both the first and second derivatives of the interpolated values have a continuous derivative, whereas the other methods do not. This means that the results of the `"spline"` method are generally smoother. If the function to be interpolated is in fact smooth, then `"spline"` will give excellent results. However, if the function to be evaluated is in some manner discontinuous, then `"pchip"` interpolation might give better results.

This can be demonstrated by the code

```
t = -2:2;
dt = 1;
ti =-2:0.025:2;
dti = 0.025;
y = sign (t);
ys = interp1 (t,y,ti,"spline");
yp = interp1 (t,y,ti,"pchip");
ddys = diff (diff (ys)./dti) ./ dti;
ddyp = diff (diff (yp)./dti) ./ dti;
figure (1);
plot (ti,ys,"r-", ti,yp,"g-");
legend ("spline", "pchip", 4);
figure (2);
plot (ti,ddys,"r+", ti,ddyp,"g*");
legend ("spline", "pchip");
```

The result of which can be seen in Figure 29.1 and Figure 29.2.

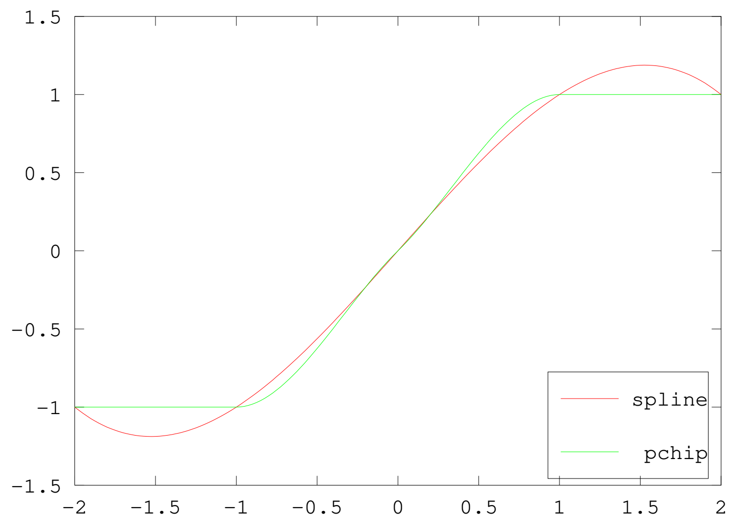

Figure 29.1: Comparison of "`pchip`" and "`spline`" interpolation methods for a step function

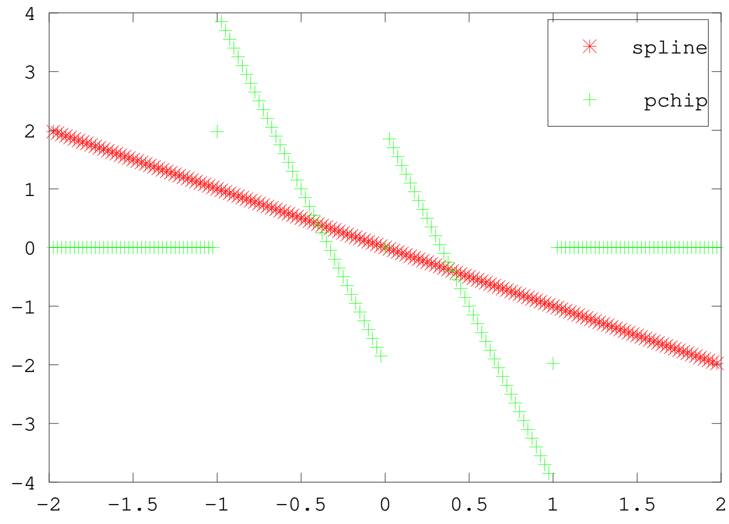

Figure 29.2: Comparison of the second derivative of the "`pchip`" and "`spline`" interpolation methods for a step function

Fourier interpolation, is a resampling technique where a signal is converted to the frequency domain, padded with zeros and then reconverted to the time domain.

`interpft` (*x*, *n*) [Function File]
`interpft` (*x*, *n*, *dim*) [Function File]

Fourier interpolation.

If *x* is a vector then *x* is resampled with *n* points. The data in *x* is assumed to be equispaced. If *x* is a matrix or an N-dimensional array, the interpolation is performed on each column of *x*.

If *dim* is specified, then interpolate along the dimension *dim*.

`interpft` assumes that the interpolated function is periodic, and so assumptions are made about the endpoints of the interpolation.

See also: [interp1], page 655.

There are two significant limitations on Fourier interpolation. First, the function signal is assumed to be periodic, and so non-periodic signals will be poorly represented at the edges. Second, both the signal and its interpolation are required to be sampled at equispaced points. An example of the use of `interpft` is

```
t = 0 : 0.3 : pi; dt = t(2)-t(1);
n = length (t); k = 100;
ti = t(1) + [0 : k-1]*dt*n/k;
y = sin (4*t + 0.3) .* cos (3*t - 0.1);
yp = sin (4*ti + 0.3) .* cos (3*ti - 0.1);
plot (ti, yp, "g", ti, interp1 (t, y, ti, "spline"), "b", ...
      ti, interpft (y, k), "c", t, y, "r+");
legend ("sin(4t+0.3)cos(3t-0.1)", "spline", "interpft", "data");
```

which demonstrates the poor behavior of Fourier interpolation for non-periodic functions, as can be seen in Figure 29.3.

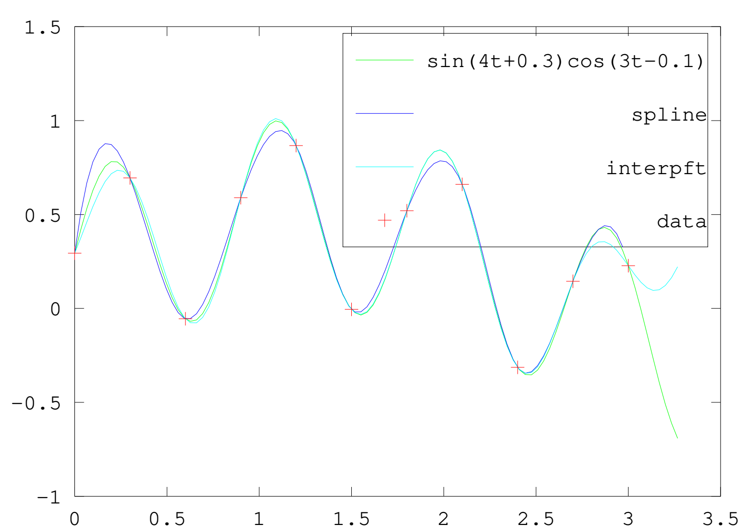

Figure 29.3: Comparison of `interp1` and `interpft` for non-periodic data

In addition, the support functions `spline` and `lookup` that underlie the `interp1` function can be called directly.

pp = `spline` (*x*, *y*) [Function File]
yi = `spline` (*x*, *y*, *xi*) [Function File]

Return the cubic spline interpolant of points *x* and *y*.

When called with two arguments, return the piecewise polynomial *pp* that may be used with `ppval` to evaluate the polynomial at specific points.

When called with a third input argument, `spline` evaluates the spline at the points *xi*. The third calling form `spline (x, y, xi)` is equivalent to `ppval (spline (x, y), xi)`.

The variable *x* must be a vector of length *n*.

y can be either a vector or array. If *y* is a vector it must have a length of either *n* or `n + 2`. If the length of *y* is *n*, then the `"not-a-knot"` end condition is used. If the length of *y* is `n + 2`, then the first and last values of the vector *y* are the values of the first derivative of the cubic spline at the endpoints.

If *y* is an array, then the size of *y* must have the form

$$[s_1, s_2, \cdots, s_k, n]$$

or

$$[s_1, s_2, \cdots, s_k, n + 2].$$

The array is reshaped internally to a matrix where the leading dimension is given by

$$s_1 s_2 \cdots s_k$$

and each row of this matrix is then treated separately. Note that this is exactly the opposite of `interp1` but is done for MATLAB compatibility.

See also: [pchip], page 690, [ppval], page 653, [mkpp], page 652, [unmkpp], page 653.

29.2 Multi-dimensional Interpolation

There are three multi-dimensional interpolation functions in Octave, with similar capabilities. Methods using Delaunay tessellation are described in Section 30.4 [Interpolation on Scattered Data], page 677.

zi = `interp2` (*x*, *y*, *z*, *xi*, *yi*) [Function File]
zi = `interp2` (*z*, *xi*, *yi*) [Function File]
zi = `interp2` (*z*, *n*) [Function File]
zi = `interp2` (*z*) [Function File]
zi = `interp2` (. . ., *method*) [Function File]
zi = `interp2` (. . ., *method*, *extrap*) [Function File]

Two-dimensional interpolation.

Interpolate reference data *x*, *y*, *z* to determine *zi* at the coordinates *xi*, *yi*. The reference data *x*, *y* can be matrices, as returned by `meshgrid`, in which case the sizes of *x*, *y*, and *z* must be equal. If *x*, *y* are vectors describing a grid then `length (x) == columns (z)` and `length (y) == rows (z)`. In either case the input data must be strictly monotonic.

If called without *x*, *y*, and just a single reference data matrix *z*, the 2-D region `x = 1:columns (z), y = 1:rows (z)` is assumed. This saves memory if the grid is regular and the distance between points is not important.

If called with a single reference data matrix *z* and a refinement value *n*, then perform interpolation over a grid where each original interval has been recursively subdivided *n* times. This results in `2^n-1` additional points for every interval in the original grid. If *n* is omitted a value of 1 is used. As an example, the interval [0,1] with `n==2` results in a refined interval with points at [0, 1/4, 1/2, 3/4, 1].

The interpolation *method* is one of:

`"nearest"`
: Return the nearest neighbor.

`"linear"` (default)
: Linear interpolation from nearest neighbors.

`"pchip"`
: Piecewise cubic Hermite interpolating polynomial—shape-preserving interpolation with smooth first derivative.

`"cubic"`
: Cubic interpolation (same as `"pchip"`).

`"spline"`
: Cubic spline interpolation—smooth first and second derivatives throughout the curve.

extrap is a scalar number. It replaces values beyond the endpoints with *extrap*. Note that if *extrapval* is used, *method* must be specified as well. If *extrap* is omitted and the *method* is `"spline"`, then the extrapolated values of the `"spline"` are used. Otherwise the default *extrap* value for any other *method* is `"NA"`.

See also: [interp1], page 655, [interp3], page 660, [interpn], page 661, [meshgrid], page 317.

`vi = interp3 (x, y, z, v, xi, yi, zi)` [Function File]
`vi = interp3 (v, xi, yi, zi)` [Function File]
`vi = interp3 (v, n)` [Function File]
`vi = interp3 (v)` [Function File]
`vi = interp3 (..., method)` [Function File]
`vi = interp3 (..., method, extrapval)` [Function File]

Three-dimensional interpolation.

Interpolate reference data *x*, *y*, *z*, *v* to determine *vi* at the coordinates *xi*, *yi*, *zi*. The reference data *x*, *y*, *z* can be matrices, as returned by `meshgrid`, in which case the sizes of *x*, *y*, *z*, and *v* must be equal. If *x*, *y*, *z* are vectors describing a cubic grid then `length (x) == columns (v)`, `length (y) == rows (v)`, and `length (z) == size (v, 3)`. In either case the input data must be strictly monotonic.

If called without *x*, *y*, *z*, and just a single reference data matrix *v*, the 3-D region `x = 1:columns (v), y = 1:rows (v), z = 1:size (v, 3)` is assumed. This saves memory if the grid is regular and the distance between points is not important.

If called with a single reference data matrix *v* and a refinement value *n*, then perform interpolation over a 3-D grid where each original interval has been recursively subdivided *n* times. This results in `2^n-1` additional points for every interval in the original grid. If *n* is omitted a value of 1 is used. As an example, the interval [0,1] with `n==2` results in a refined interval with points at [0, 1/4, 1/2, 3/4, 1].

The interpolation *method* is one of:

`"nearest"`
: Return the nearest neighbor.

`"linear"` (default)
: Linear interpolation from nearest neighbors.

`"cubic"`
: Piecewise cubic Hermite interpolating polynomial—shape-preserving interpolation with smooth first derivative (not implemented yet).

`"spline"`
: Cubic spline interpolation—smooth first and second derivatives throughout the curve.

extrapval is a scalar number. It replaces values beyond the endpoints with *extrapval*. Note that if *extrapval* is used, *method* must be specified as well. If *extrapval* is omitted and the *method* is `"spline"`, then the extrapolated values of the `"spline"` are used. Otherwise the default *extrapval* value for any other *method* is `"NA"`.

See also: [interp1], page 655, [interp2], page 659, [interpn], page 661, [meshgrid], page 317.

vi = **interpn** (*x1*, *x2*, ..., *v*, *y1*, *y2*, ...) [Function File]
vi = **interpn** (*v*, *y1*, *y2*, ...) [Function File]
vi = **interpn** (*v*, *m*) [Function File]
vi = **interpn** (*v*) [Function File]
vi = **interpn** (..., *method*) [Function File]
vi = **interpn** (..., *method*, *extrapval*) [Function File]

Perform *n*-dimensional interpolation, where *n* is at least two.

Each element of the *n*-dimensional array *v* represents a value at a location given by the parameters *x1*, *x2*, ..., *xn*. The parameters *x1*, *x2*, ..., *xn* are either *n*-dimensional arrays of the same size as the array *v* in the `"ndgrid"` format or vectors. The parameters *y1*, etc. respect a similar format to *x1*, etc., and they represent the points at which the array *vi* is interpolated.

If *x1*, ..., *xn* are omitted, they are assumed to be `x1 = 1 : size (v, 1)`, etc. If *m* is specified, then the interpolation adds a point half way between each of the interpolation points. This process is performed *m* times. If only *v* is specified, then *m* is assumed to be `1`.

The interpolation *method* is one of:

`"nearest"`
: Return the nearest neighbor.

`"linear"` (default)
: Linear interpolation from nearest neighbors.

`"pchip"`
: Piecewise cubic Hermite interpolating polynomial—shape-preserving interpolation with smooth first derivative (not implemented yet).

`"cubic"`
: Cubic interpolation (same as `"pchip"` [not implemented yet]).

`"spline"` Cubic spline interpolation—smooth first and second derivatives throughout the curve.

The default method is `"linear"`.

extrapval is a scalar number. It replaces values beyond the endpoints with *extrapval*. Note that if *extrapval* is used, *method* must be specified as well. If *extrapval* is omitted and the *method* is `"spline"`, then the extrapolated values of the `"spline"` are used. Otherwise the default *extrapval* value for any other *method* is `"NA"`.

See also: [interp1], page 655, [interp2], page 659, [interp3], page 660, [spline], page 658, [ndgrid], page 318.

A significant difference between `interpn` and the other two multi-dimensional interpolation functions is the fashion in which the dimensions are treated. For `interp2` and `interp3`, the y-axis is considered to be the columns of the matrix, whereas the x-axis corresponds to the rows of the array. As Octave indexes arrays in column major order, the first dimension of any array is the columns, and so `interpn` effectively reverses the 'x' and 'y' dimensions. Consider the example,

```
x = y = z = -1:1;
f = @(x,y,z) x.^2 - y - z.^2;
[xx, yy, zz] = meshgrid (x, y, z);
v = f (xx,yy,zz);
xi = yi = zi = -1:0.1:1;
[xxi, yyi, zzi] = meshgrid (xi, yi, zi);
vi = interp3 (x, y, z, v, xxi, yyi, zzi, "spline");
[xxi, yyi, zzi] = ndgrid (xi, yi, zi);
vi2 = interpn (x, y, z, v, xxi, yyi, zzi, "spline");
mesh (zi, yi, squeeze (vi2(1,:,:)));
```

where `vi` and `vi2` are identical. The reversal of the dimensions is treated in the `meshgrid` and `ndgrid` functions respectively. The result of this code can be seen in Figure 29.4.

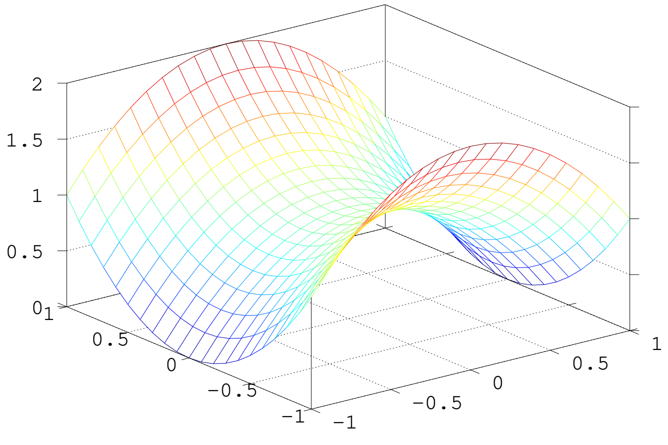

Figure 29.4: Demonstration of the use of **interpn**

30 Geometry

Much of the geometry code in Octave is based on the Qhull library[1]. Some of the documentation for Qhull, particularly for the options that can be passed to `delaunay`, `voronoi` and `convhull`, etc., is relevant to Octave users.

30.1 Delaunay Triangulation

The Delaunay triangulation is constructed from a set of circum-circles. These circum-circles are chosen so that there are at least three of the points in the set to triangulation on the circumference of the circum-circle. None of the points in the set of points falls within any of the circum-circles.

In general there are only three points on the circumference of any circum-circle. However, in some cases, and in particular for the case of a regular grid, 4 or more points can be on a single circum-circle. In this case the Delaunay triangulation is not unique.

tri = delaunay (*x*, *y*) [Function File]
tetr = delaunay (*x*, *y*, *z*) [Function File]
tri = delaunay (*x*) [Function File]
tri = delaunay (. . ., *options*) [Function File]

Compute the Delaunay triangulation for a 2-D or 3-D set of points.

For 2-D sets, the return value *tri* is a set of triangles which satisfies the Delaunay circum-circle criterion, i.e., only a single data point from [*x*, *y*] is within the circum-circle of the defining triangle. The set of triangles *tri* is a matrix of size [n, 3]. Each row defines a triangle and the three columns are the three vertices of the triangle. The value of `tri(i,j)` is an index into *x* and *y* for the location of the j-th vertex of the i-th triangle.

For 3-D sets, the return value *tetr* is a set of tetrahedrons which satisfies the Delaunay circum-circle criterion, i.e., only a single data point from [*x*, *y*, *z*] is within the circum-circle of the defining tetrahedron. The set of tetrahedrons is a matrix of size [n, 4]. Each row defines a tetrahedron and the four columns are the four vertices of the tetrahedron. The value of `tetr(i,j)` is an index into *x*, *y*, *z* for the location of the j-th vertex of the i-th tetrahedron.

The input *x* may also be a matrix with two or three columns where the first column contains x-data, the second y-data, and the optional third column contains z-data.

The optional last argument, which must be a string or cell array of strings, contains options passed to the underlying qhull command. See the documentation for the Qhull library for details `http://www.qhull.org/html/qh-quick.htm#options`. The default options are `{"Qt", "Qbb", "Qc", "Qz"}`.

If *options* is not present or `[]` then the default arguments are used. Otherwise, *options* replaces the default argument list. To append user options to the defaults it is necessary to repeat the default arguments in *options*. Use a null string to pass no arguments.

[1] Barber, C.B., Dobkin, D.P., and Huhdanpaa, H.T., *The Quickhull Algorithm for Convex Hulls*, ACM Trans. on Mathematical Software, 22(4):469–483, Dec 1996, `http://www.qhull.org`

```
x = rand (1, 10);
y = rand (1, 10);
tri = delaunay (x, y);
triplot (tri, x, y);
hold on;
plot (x, y, "r*");
axis ([0,1,0,1]);
```

See also: [delaunayn], page 666, [convhull], page 675, [voronoi], page 672, [triplot], page 667, [trimesh], page 667, [tetramesh], page 668, [trisurf], page 668.

For 3-D inputs `delaunay` returns a set of tetrahedra that satisfy the Delaunay circum-circle criteria. Similarly, `delaunayn` returns the N-dimensional simplex satisfying the Delaunay circum-circle criteria. The N-dimensional extension of a triangulation is called a tessellation.

T = `delaunayn` (*pts*) [Function File]
T = `delaunayn` (*pts*, *options*) [Function File]

Compute the Delaunay triangulation for an N-dimensional set of points.

The Delaunay triangulation is a tessellation of the convex hull of a set of points such that no N-sphere defined by the N-triangles contains any other points from the set.

The input matrix *pts* of size [n, dim] contains n points in a space of dimension dim. The return matrix *T* has size [m, dim+1]. Each row of *T* contains a set of indices back into the original set of points *pts* which describes a simplex of dimension dim. For example, a 2-D simplex is a triangle and 3-D simplex is a tetrahedron.

An optional second argument, which must be a string or cell array of strings, contains options passed to the underlying qhull command. See the documentation for the Qhull library for details `http://www.qhull.org/html/qh-quick.htm#options`. The default options depend on the dimension of the input:

- 2-D and 3-D: *options* = `{"Qt", "Qbb", "Qc", "Qz"}`
- 4-D and higher: *options* = `{"Qt", "Qbb", "Qc", "Qx"}`

If *options* is not present or `[]` then the default arguments are used. Otherwise, *options* replaces the default argument list. To append user options to the defaults it is necessary to repeat the default arguments in *options*. Use a null string to pass no arguments.

See also: [delaunay], page 665, [convhulln], page 676, [voronoin], page 672, [trimesh], page 667, [tetramesh], page 668.

An example of a Delaunay triangulation of a set of points is

```
rand ("state", 2);
x = rand (10, 1);
y = rand (10, 1);
T = delaunay (x, y);
X = [ x(T(:,1)); x(T(:,2)); x(T(:,3)); x(T(:,1)) ];
Y = [ y(T(:,1)); y(T(:,2)); y(T(:,3)); y(T(:,1)) ];
axis ([0, 1, 0, 1]);
plot (X, Y, "b", x, y, "r*");
```

The result of which can be seen in Figure 30.1.

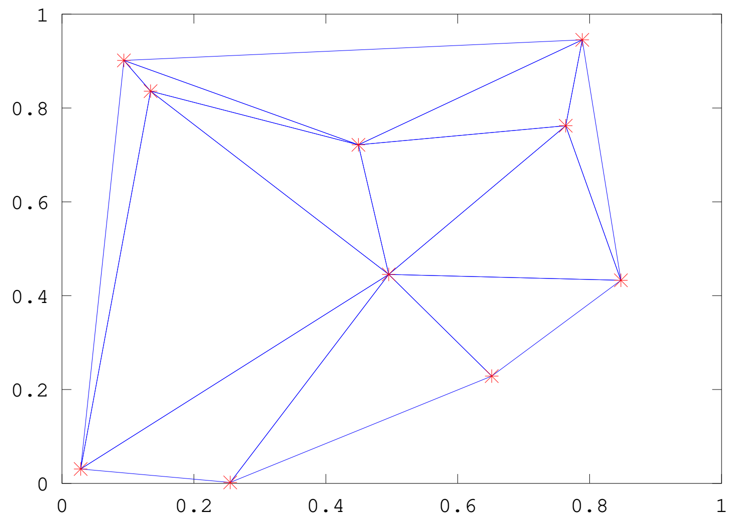

Figure 30.1: Delaunay triangulation of a random set of points

30.1.1 Plotting the Triangulation

Octave has the functions `triplot`, `trimesh`, and `trisurf` to plot the Delaunay triangulation of a 2-dimensional set of points. `tetramesh` will plot the triangulation of a 3-dimensional set of points.

`triplot (`*`tri`*`, `*`x`*`, `*`y`*`)` [Function File]
`triplot (`*`tri`*`, `*`x`*`, `*`y`*`, `*`linespec`*`)` [Function File]
`h` `= triplot (...)` [Function File]

Plot a 2-D triangular mesh.

tri is typically the output of a Delaunay triangulation over the grid of *x*, *y*. Every row of *tri* represents one triangle and contains three indices into [*x*, *y*] which are the vertices of the triangles in the x-y plane.

The linestyle to use for the plot can be defined with the argument *linespec* of the same format as the `plot` command.

The optional return value *h* is a graphics handle to the created patch object.

See also: [plot], page 272, [trimesh], page 667, [trisurf], page 668, [delaunay], page 665.

`trimesh (`*`tri`*`, `*`x`*`, `*`y`*`, `*`z`*`, `*`c`*`)` [Function File]
`trimesh (`*`tri`*`, `*`x`*`, `*`y`*`, `*`z`*`)` [Function File]
`trimesh (`*`tri`*`, `*`x`*`, `*`y`*`)` [Function File]
`trimesh (..., `*`prop`*`, `*`val`*`, ...)` [Function File]
`h` `= trimesh (...)` [Function File]

Plot a 3-D triangular wireframe mesh.

In contrast to `mesh`, which plots a mesh using rectangles, `trimesh` plots the mesh using triangles.

tri is typically the output of a Delaunay triangulation over the grid of *x*, *y*. Every row of *tri* represents one triangle and contains three indices into [*x*, *y*] which are the vertices of the triangles in the x-y plane. *z* determines the height above the plane of each vertex. If no *z* input is given then the triangles are plotted as a 2-D figure.

The color of the trimesh is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally, the color of the mesh can be specified independently of *z* by supplying a color matrix, *c*. If *z* has N elements, then *c* should be an Nx1 vector for colormap data or an Nx3 matrix for RGB data.

Any property/value pairs are passed directly to the underlying patch object.

The optional return value *h* is a graphics handle to the created patch object.

See also: [mesh], page 307, [tetramesh], page 668, [triplot], page 667, [trisurf], page 668, [delaunay], page 665, [patch], page 357, [hidden], page 309.

`trisurf (`*tri*`,` *x*`,` *y*`,` *z*`,` *c*`)` [Function File]
`trisurf (`*tri*`,` *x*`,` *y*`,` *z*`)` [Function File]
`trisurf (...,` *prop*`,` *val*`, ...)` [Function File]
h `= trisurf (...)` [Function File]

Plot a 3-D triangular surface.

In contrast to `surf`, which plots a surface mesh using rectangles, `trisurf` plots the mesh using triangles.

tri is typically the output of a Delaunay triangulation over the grid of *x*, *y*. Every row of *tri* represents one triangle and contains three indices into [*x*, *y*] which are the vertices of the triangles in the x-y plane. *z* determines the height above the plane of each vertex.

The color of the trimesh is computed by linearly scaling the *z* values to fit the range of the current colormap. Use `caxis` and/or change the colormap to control the appearance.

Optionally, the color of the mesh can be specified independently of *z* by supplying a color matrix, *c*. If *z* has N elements, then *c* should be an Nx1 vector for colormap data or an Nx3 matrix for RGB data.

Any property/value pairs are passed directly to the underlying patch object.

The optional return value *h* is a graphics handle to the created patch object.

See also: [surf], page 309, [triplot], page 667, [trimesh], page 667, [delaunay], page 665, [patch], page 357, [shading], page 320.

`tetramesh (`*T*`,` *X*`)` [Function File]
`tetramesh (`*T*`,` *X*`,` *C*`)` [Function File]
`tetramesh (...,` *property*`,` *val*`, ...)` [Function File]
h `= tetramesh (...)` [Function File]

Display the tetrahedrons defined in the m-by-4 matrix *T* as 3-D patches.

T is typically the output of a Delaunay triangulation of a 3-D set of points. Every row of *T* contains four indices into the n-by-3 matrix *X* of the vertices of a tetrahedron. Every row in *X* represents one point in 3-D space.

The vector *C* specifies the color of each tetrahedron as an index into the current colormap. The default value is 1:m where m is the number of tetrahedrons; the indices are scaled to map to the full range of the colormap. If there are more tetrahedrons than colors in the colormap then the values in *C* are cyclically repeated.

Calling `tetramesh (..., "property", "value", ...)` passes all property/value pairs directly to the patch function as additional arguments.

The optional return value *h* is a vector of patch handles where each handle represents one tetrahedron in the order given by *T*. A typical use case for *h* is to turn the respective patch `"visible"` property `"on"` or `"off"`.

Type `demo tetramesh` to see examples on using `tetramesh`.

See also: [trimesh], page 667, [delaunay], page 665, [delaunayn], page 666, [patch], page 357.

The difference between `triplot`, and `trimesh` or `triplot`, is that the former only plots the 2-dimensional triangulation itself, whereas the second two plot the value of a function `f (x, y)`. An example of the use of the `triplot` function is

```
rand ("state", 2)
x = rand (20, 1);
y = rand (20, 1);
tri = delaunay (x, y);
triplot (tri, x, y);
```

which plots the Delaunay triangulation of a set of random points in 2-dimensions. The output of the above can be seen in Figure 30.2.

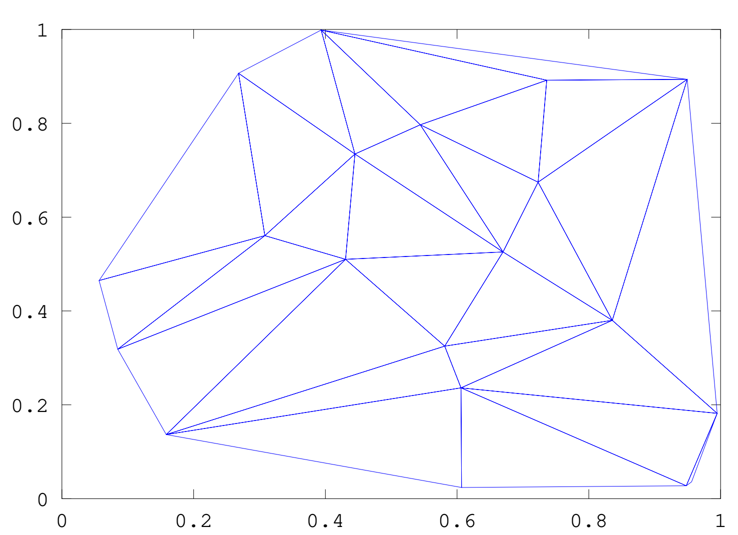

Figure 30.2: Delaunay triangulation of a random set of points

30.1.2 Identifying Points in Triangulation

It is often necessary to identify whether a particular point in the N-dimensional space is within the Delaunay tessellation of a set of points in this N-dimensional space, and if so which N-simplex contains the point and which point in the tessellation is closest to the desired point. The functions `tsearch` and `dsearch` perform this function in a triangulation, and `tsearchn` and `dsearchn` in an N-dimensional tessellation.

To identify whether a particular point represented by a vector *p* falls within one of the simplices of an N-simplex, we can write the Cartesian coordinates of the point in a parametric form with respect to the N-simplex. This parametric form is called the Barycentric Coordinates of the point. If the points defining the N-simplex are given by `N + 1` vectors *t*(*i*,:), then the Barycentric coordinates defining the point *p* are given by

```
p = sum (beta(1:N+1) * t(1:N+1),:)
```

where there are `N + 1` values `beta(i)` that together as a vector represent the Barycentric coordinates of the point *p*. To ensure a unique solution for the values of `beta(i)` an additional criteria of

```
sum (beta(1:N+1)) == 1
```

is imposed, and we can therefore write the above as

```
p - t(end, :) = beta(1:end-1) * (t(1:end-1, :)
      - ones (N, 1) * t(end, :)
```

Solving for *beta* we can then write

```
beta(1:end-1) = (p - t(end, :)) / (t(1:end-1, :)
      - ones (N, 1) * t(end, :))
beta(end) = sum (beta(1:end-1))
```

which gives the formula for the conversion of the Cartesian coordinates of the point *p* to the Barycentric coordinates *beta*. An important property of the Barycentric coordinates is that for all points in the N-simplex

```
0 <= beta(i) <= 1
```

Therefore, the test in `tsearch` and `tsearchn` essentially only needs to express each point in terms of the Barycentric coordinates of each of the simplices of the N-simplex and test the values of *beta*. This is exactly the implementation used in `tsearchn`. `tsearch` is optimized for 2-dimensions and the Barycentric coordinates are not explicitly formed.

`idx = tsearch (x, y, t, xi, yi)` [Built-in Function]

Search for the enclosing Delaunay convex hull.

For `t = delaunay (x, y)`, finds the index in *t* containing the points `(xi, yi)`. For points outside the convex hull, *idx* is NaN.

See also: [delaunay], page 665, [delaunayn], page 666.

`idx = tsearchn (x, t, xi)` [Function File]
`[idx, p] = tsearchn (x, t, xi)` [Function File]

Search for the enclosing Delaunay convex hull.

For `t = delaunayn (x)`, finds the index in *t* containing the points *xi*. For points outside the convex hull, *idx* is NaN.

If requested `tsearchn` also returns the Barycentric coordinates *p* of the enclosing triangles.

See also: [delaunay], page 665, [delaunayn], page 666.

An example of the use of `tsearch` can be seen with the simple triangulation

```
x = [-1; -1; 1; 1];
y = [-1; 1; -1; 1];
tri = [1, 2, 3; 2, 3, 1];
```

consisting of two triangles defined by *tri*. We can then identify which triangle a point falls in like

```
tsearch (x, y, tri, -0.5, -0.5)
⇒ 1
tsearch (x, y, tri, 0.5, 0.5)
⇒ 2
```

and we can confirm that a point doesn't lie within one of the triangles like

```
tsearch (x, y, tri, 2, 2)
⇒ NaN
```

The `dsearch` and `dsearchn` find the closest point in a tessellation to the desired point. The desired point does not necessarily have to be in the tessellation, and even if it the returned point of the tessellation does not have to be one of the vertexes of the N-simplex within which the desired point is found.

idx = `dsearch` (*x*, *y*, *tri*, *xi*, *yi*) [Function File]
idx = `dsearch` (*x*, *y*, *tri*, *xi*, *yi*, *s*) [Function File]

Return the index *idx* of the closest point in `x, y` to the elements `[xi(:), yi(:)]`.

The variable *s* is accepted for compatibility but is ignored.

See also: [dsearchn], page 671, [tsearch], page 670.

idx = `dsearchn` (*x*, *tri*, *xi*) [Function File]
idx = `dsearchn` (*x*, *tri*, *xi*, *outval*) [Function File]
idx = `dsearchn` (*x*, *xi*) [Function File]
[*idx*, *d*] = `dsearchn` (...) [Function File]

Return the index *idx* of the closest point in *x* to the elements *xi*.

If *outval* is supplied, then the values of *xi* that are not contained within one of the simplices *tri* are set to *outval*. Generally, *tri* is returned from `delaunayn (x)`.

See also: [dsearch], page 671, [tsearch], page 670.

An example of the use of `dsearch`, using the above values of *x*, *y* and *tri* is

```
dsearch (x, y, tri, -2, -2)
⇒ 1
```

If you wish the points that are outside the tessellation to be flagged, then `dsearchn` can be used as

```
dsearchn ([x, y], tri, [-2, -2], NaN)
⇒ NaN
dsearchn ([x, y], tri, [-0.5, -0.5], NaN)
⇒ 1
```

where the point outside the tessellation are then flagged with `NaN`.

30.2 Voronoi Diagrams

A Voronoi diagram or Voronoi tessellation of a set of points *s* in an N-dimensional space, is the tessellation of the N-dimensional space such that all points in `v(p)`, a partitions of the tessellation where *p* is a member of *s*, are closer to *p* than any other point in *s*. The Voronoi diagram is related to the Delaunay triangulation of a set of points, in that the vertexes of the Voronoi tessellation are the centers of the circum-circles of the simplices of the Delaunay tessellation.

`voronoi (`*x*`,` *y*`)` [Function File]
`voronoi (`*x*`,` *y*`,` *options*`)` [Function File]
`voronoi (`. . .`, "`*linespec*`")` [Function File]
`voronoi (`*hax*`, . . .)` [Function File]
h `= voronoi (. . .)` [Function File]
`[`*vx*`,` *vy*`] = voronoi (. . .)` [Function File]

Plot the Voronoi diagram of points `(`*x*`,` *y*`)`.

The Voronoi facets with points at infinity are not drawn.

The *options* argument, which must be a string or cell array of strings, contains options passed to the underlying qhull command. See the documentation for the Qhull library for details `http://www.qhull.org/html/qh-quick.htm#options`.

If `"linespec"` is given it is used to set the color and line style of the plot.

If an axis graphics handle *hax* is supplied then the Voronoi diagram is drawn on the specified axis rather than in a new figure.

If a single output argument is requested then the Voronoi diagram will be plotted and a graphics handle *h* to the plot is returned.

[*vx*, *vy*] = voronoi (. . .) returns the Voronoi vertices instead of plotting the diagram.

```
x = rand (10, 1);
y = rand (size (x));
h = convhull (x, y);
[vx, vy] = voronoi (x, y);
plot (vx, vy, "-b", x, y, "o", x(h), y(h), "-g");
legend ("", "points", "hull");
```

See also: [voronoin], page 672, [delaunay], page 665, [convhull], page 675.

`[`*C*`,` *F*`] = voronoin (`*pts*`)` [Function File]
`[`*C*`,` *F*`] = voronoin (`*pts*`,` *options*`)` [Function File]

Compute N-dimensional Voronoi facets.

The input matrix *pts* of size [n, dim] contains n points in a space of dimension dim.

C contains the points of the Voronoi facets. The list *F* contains, for each facet, the indices of the Voronoi points.

An optional second argument, which must be a string or cell array of strings, contains options passed to the underlying qhull command. See the documentation for the Qhull library for details `http://www.qhull.org/html/qh-quick.htm#options`.

The default options depend on the dimension of the input:

- 2-D and 3-D: *options* = `{"Qbb"}`
- 4-D and higher: *options* = `{"Qbb", "Qx"}`

If *options* is not present or `[]` then the default arguments are used. Otherwise, *options* replaces the default argument list. To append user options to the defaults it is necessary to repeat the default arguments in *options*. Use a null string to pass no arguments.

See also: [voronoi], page 672, [convhulln], page 676, [delaunayn], page 666.

An example of the use of `voronoi` is

```
rand ("state",9);
x = rand (10,1);
y = rand (10,1);
tri = delaunay (x, y);
[vx, vy] = voronoi (x, y, tri);
triplot (tri, x, y, "b");
hold on;
plot (vx, vy, "r");
```

The result of which can be seen in Figure 30.3. Note that the circum-circle of one of the triangles has been added to this figure, to make the relationship between the Delaunay tessellation and the Voronoi diagram clearer.

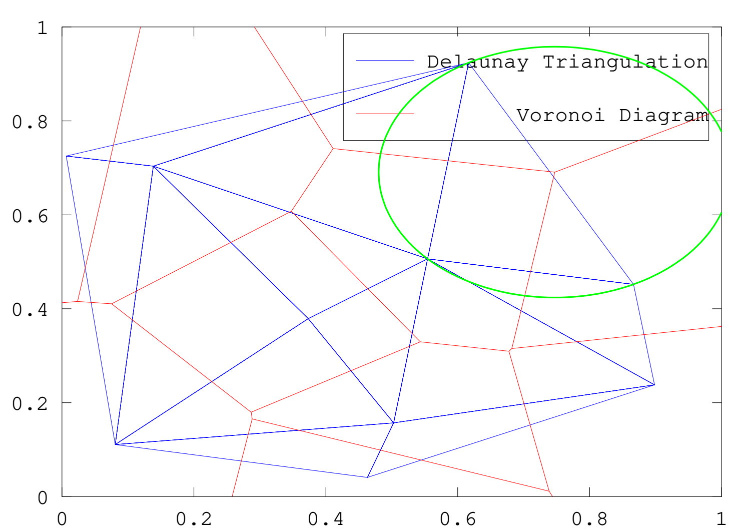

Figure 30.3: Delaunay triangulation and Voronoi diagram of a random set of points

Additional information about the size of the facets of a Voronoi diagram, and which points of a set of points is in a polygon can be had with the `polyarea` and `inpolygon` functions respectively.

`polyarea (x, y)` [Function File]
`polyarea (x, y, dim)` [Function File]

Determine area of a polygon by triangle method.

The variables *x* and *y* define the vertex pairs, and must therefore have the same shape. They can be either vectors or arrays. If they are arrays then the columns of *x* and *y* are treated separately and an area returned for each.

If the optional *dim* argument is given, then `polyarea` works along this dimension of the arrays *x* and *y*.

An example of the use of `polyarea` might be

```
rand ("state", 2);
x = rand (10, 1);
y = rand (10, 1);
[c, f] = voronoin ([x, y]);
af = zeros (size (f));
for i = 1 : length (f)
  af(i) = polyarea (c (f {i, :}, 1), c (f {i, :}, 2));
endfor
```

Facets of the Voronoi diagram with a vertex at infinity have infinity area. A simplified version of `polyarea` for rectangles is available with `rectint`

`area = rectint (a, b)` [Function File]

Compute area or volume of intersection of rectangles or N-D boxes.

Compute the area of intersection of rectangles in *a* and rectangles in *b*. N-dimensional boxes are supported in which case the volume, or hypervolume is computed according to the number of dimensions.

2-dimensional rectangles are defined as `[xpos ypos width height]` where xpos and ypos are the position of the bottom left corner. Higher dimensions are supported where the coordinates for the minimum value of each dimension follow the length of the box in that dimension, e.g., `[xpos ypos zpos kpos ... width height depth k_length ...]`.

Each row of *a* and *b* define a rectangle, and if both define multiple rectangles, then the output, *area*, is a matrix where the i-th row corresponds to the i-th row of a and the j-th column corresponds to the j-th row of b.

See also: [polyarea], page 673.

`in = inpolygon (x, y, xv, yv)` [Function File]
`[in, on] = inpolygon (x, y, xv, yv)` [Function File]

For a polygon defined by vertex points `(xv, yv)`, return true if the points `(x, y)` are inside (or on the boundary) of the polygon; Otherwise, return false.

The input variables *x* and *y*, must have the same dimension.

The optional output *on* returns true if the points are exactly on the polygon edge, and false otherwise.

See also: [delaunay], page 665.

An example of the use of `inpolygon` might be

```
randn ("state", 2);
x = randn (100, 1);
y = randn (100, 1);
vx = cos (pi * [-1 : 0.1: 1]);
vy = sin (pi * [-1 : 0.1 : 1]);
in = inpolygon (x, y, vx, vy);
plot (vx, vy, x(in), y(in), "r+", x(!in), y(!in), "bo");
axis ([-2, 2, -2, 2]);
```

The result of which can be seen in Figure 30.4.

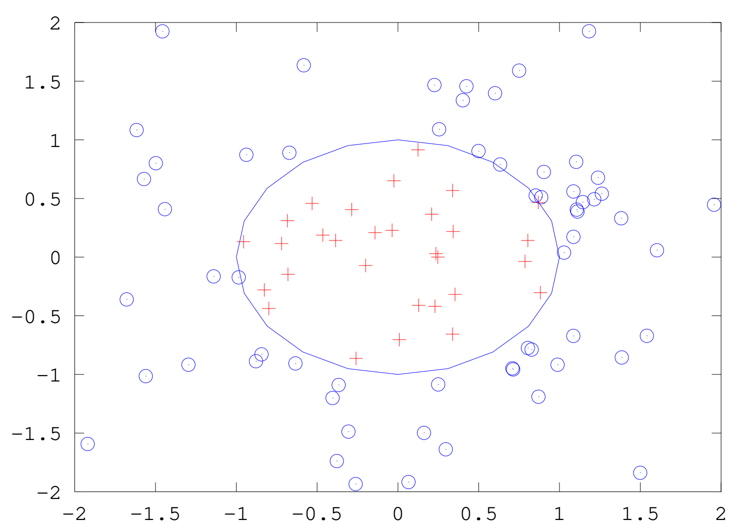

Figure 30.4: Demonstration of the `inpolygon` function to determine the points inside a polygon

30.3 Convex Hull

The convex hull of a set of points is the minimum convex envelope containing all of the points. Octave has the functions `convhull` and `convhulln` to calculate the convex hull of 2-dimensional and N-dimensional sets of points.

H = `convhull` (*x*, *y*) [Function File]
H = `convhull` (*x*, *y*, *options*) [Function File]

Compute the convex hull of the set of points defined by the arrays *x* and *y*. The hull *H* is an index vector into the set of points and specifies which points form the enclosing hull.

An optional third argument, which must be a string or cell array of strings, contains options passed to the underlying qhull command. See the documentation for the Qhull library for details `http://www.qhull.org/html/qh-quick.htm#options`. The default option is `{"Qt"}`.

If *options* is not present or `[]` then the default arguments are used. Otherwise, *options* replaces the default argument list. To append user options to the defaults it is necessary to repeat the default arguments in *options*. Use a null string to pass no arguments.

See also: [convhulln], page 676, [delaunay], page 665, [voronoi], page 672.

`h = convhulln (pts)` [Loadable Function]
`h = convhulln (pts, options)` [Loadable Function]
`[h, v] = convhulln (...)` [Loadable Function]

Compute the convex hull of the set of points *pts*.

pts is a matrix of size [n, dim] containing n points in a space of dimension dim.

The hull *h* is an index vector into the set of points and specifies which points form the enclosing hull.

An optional second argument, which must be a string or cell array of strings, contains options passed to the underlying qhull command. See the documentation for the Qhull library for details `http://www.qhull.org/html/qh-quick.htm#options`. The default options depend on the dimension of the input:

- 2D, 3D, 4D: *options* = `{"Qt"}`
- 5D and higher: *options* = `{"Qt", "Qx"}`

If *options* is not present or `[]` then the default arguments are used. Otherwise, *options* replaces the default argument list. To append user options to the defaults it is necessary to repeat the default arguments in *options*. Use a null string to pass no arguments.

If the second output *v* is requested the volume of the enclosing convex hull is calculated.

See also: [convhull], page 675, [delaunayn], page 666, [voronoin], page 672.

An example of the use of `convhull` is

```
x = -3:0.05:3;
y = abs (sin (x));
k = convhull (x, y);
plot (x(k), y(k), "r-", x, y, "b+");
axis ([-3.05, 3.05, -0.05, 1.05]);
```

The output of the above can be seen in Figure 30.5.

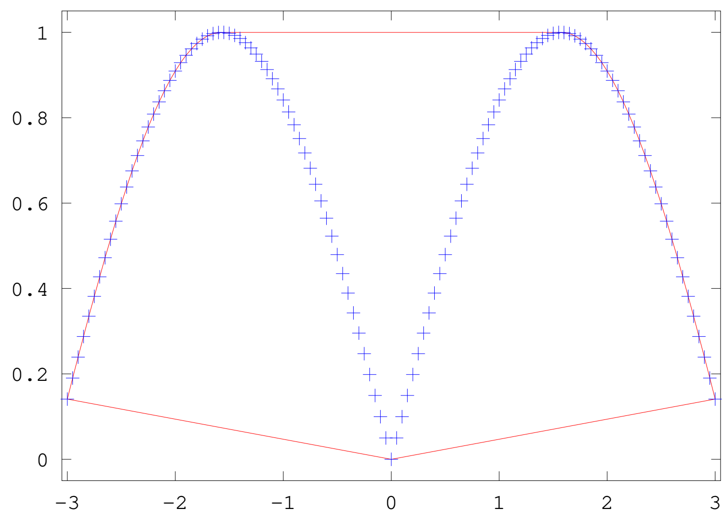

Figure 30.5: The convex hull of a simple set of points

30.4 Interpolation on Scattered Data

An important use of the Delaunay tessellation is that it can be used to interpolate from scattered data to an arbitrary set of points. To do this the N-simplex of the known set of points is calculated with `delaunay` or `delaunayn`. Then the simplices in to which the desired points are found are identified. Finally the vertices of the simplices are used to interpolate to the desired points. The functions that perform this interpolation are `griddata`, `griddata3` and `griddatan`.

zi = `griddata` (*x*, *y*, *z*, *xi*, *yi*) [Function File]
zi = `griddata` (*x*, *y*, *z*, *xi*, *yi*, *method*) [Function File]
[*xi*, *yi*, *zi*] = `griddata` (...) [Function File]

Generate a regular mesh from irregular data using interpolation.

The function is defined by `z = f (x, y)`. Inputs `x, y, z` are vectors of the same length or `x, y` are vectors and `z` is matrix.

The interpolation points are all `(xi, yi)`. If *xi*, *yi* are vectors then they are made into a 2-D mesh.

The interpolation method can be `"nearest"`, `"cubic"` or `"linear"`. If method is omitted it defaults to `"linear"`.

See also: [griddata3], page 677, [griddatan], page 678, [delaunay], page 665.

vi = `griddata3` (*x*, *y*, *z*, *v*, *xi*, *yi*, *zi*) [Function File]
vi = `griddata3` (*x*, *y*, *z*, *v*, *xi*, *yi*, *zi*, *method*) [Function File]
vi = `griddata3` (*x*, *y*, *z*, *v*, *xi*, *yi*, *zi*, *method*, *options*) [Function File]

Generate a regular mesh from irregular data using interpolation.

The function is defined by `v = f (x, y, z)`. The interpolation points are specified by *xi*, *yi*, *zi*.

The interpolation method can be `"nearest"` or `"linear"`. If method is omitted it defaults to `"linear"`.

The optional argument *options* is passed directly to Qhull when computing the Delaunay triangulation used for interpolation. See `delaunayn` for information on the defaults and how to pass different values.

See also: [griddata], page 677, [griddatan], page 678, [delaunayn], page 666.

yi = **griddatan** (*x*, *y*, *xi*) [Function File]
yi = **griddatan** (*x*, *y*, *xi*, *method*) [Function File]
yi = **griddatan** (*x*, *y*, *xi*, *method*, *options*) [Function File]

Generate a regular mesh from irregular data using interpolation.

The function is defined by `y = f (x)`. The interpolation points are all *xi*.

The interpolation method can be `"nearest"` or `"linear"`. If method is omitted it defaults to `"linear"`.

The optional argument *options* is passed directly to Qhull when computing the Delaunay triangulation used for interpolation. See `delaunayn` for information on the defaults and how to pass different values.

See also: [griddata], page 677, [griddata3], page 677, [delaunayn], page 666.

An example of the use of the `griddata` function is

```
rand ("state", 1);
x = 2*rand (1000,1) - 1;
y = 2*rand (size (x)) - 1;
z = sin (2*(x.^2+y.^2));
[xx,yy] = meshgrid (linspace (-1,1,32));
griddata (x,y,z,xx,yy);
```

that interpolates from a random scattering of points, to a uniform grid. The output of the above can be seen in Figure 30.6.

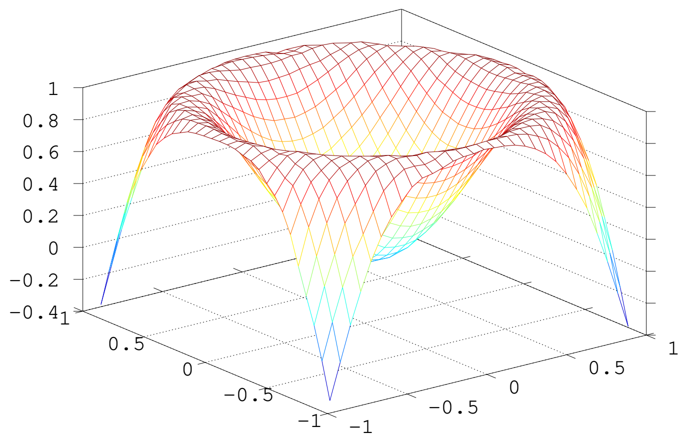

Figure 30.6: Interpolation from a scattered data to a regular grid

31 Signal Processing

This chapter describes the signal processing and fast Fourier transform functions available in Octave. Fast Fourier transforms are computed with the FFTW or FFTPACK libraries depending on how Octave is built.

`fft (x)` [Built-in Function]
`fft (x, n)` [Built-in Function]
`fft (x, n, dim)` [Built-in Function]

Compute the discrete Fourier transform of A using a Fast Fourier Transform (FFT) algorithm.

The FFT is calculated along the first non-singleton dimension of the array. Thus if x is a matrix, `fft (x)` computes the FFT for each column of x.

If called with two arguments, n is expected to be an integer specifying the number of elements of x to use, or an empty matrix to specify that its value should be ignored. If n is larger than the dimension along which the FFT is calculated, then x is resized and padded with zeros. Otherwise, if n is smaller than the dimension along which the FFT is calculated, then x is truncated.

If called with three arguments, *dim* is an integer specifying the dimension of the matrix along which the FFT is performed

See also: [ifft], page 681, [fft2], page 681, [fftn], page 682, [fftw], page 682.

`ifft (x)` [Built-in Function]
`ifft (x, n)` [Built-in Function]
`ifft (x, n, dim)` [Built-in Function]

Compute the inverse discrete Fourier transform of A using a Fast Fourier Transform (FFT) algorithm.

The inverse FFT is calculated along the first non-singleton dimension of the array. Thus if x is a matrix, `fft (x)` computes the inverse FFT for each column of x.

If called with two arguments, n is expected to be an integer specifying the number of elements of x to use, or an empty matrix to specify that its value should be ignored. If n is larger than the dimension along which the inverse FFT is calculated, then x is resized and padded with zeros. Otherwise, if n is smaller than the dimension along which the inverse FFT is calculated, then x is truncated.

If called with three arguments, *dim* is an integer specifying the dimension of the matrix along which the inverse FFT is performed

See also: [fft], page 681, [ifft2], page 682, [ifftn], page 682, [fftw], page 682.

`fft2 (A)` [Built-in Function]
`fft2 (A, m, n)` [Built-in Function]

Compute the two-dimensional discrete Fourier transform of A using a Fast Fourier Transform (FFT) algorithm.

The optional arguments m and n may be used specify the number of rows and columns of A to use. If either of these is larger than the size of A, A is resized and padded with zeros.

If *A* is a multi-dimensional matrix, each two-dimensional sub-matrix of *A* is treated separately.

See also: [ifft2], page 682, [fft], page 681, [fftn], page 682, [fftw], page 682.

`ifft2 (A)` [Built-in Function]
`ifft2 (A, m, n)` [Built-in Function]

Compute the inverse two-dimensional discrete Fourier transform of *A* using a Fast Fourier Transform (FFT) algorithm.

The optional arguments *m* and *n* may be used specify the number of rows and columns of *A* to use. If either of these is larger than the size of *A*, *A* is resized and padded with zeros.

If *A* is a multi-dimensional matrix, each two-dimensional sub-matrix of *A* is treated separately

See also: [fft2], page 681, [ifft], page 681, [ifftn], page 682, [fftw], page 682.

`fftn (A)` [Built-in Function]
`fftn (A, size)` [Built-in Function]

Compute the N-dimensional discrete Fourier transform of *A* using a Fast Fourier Transform (FFT) algorithm.

The optional vector argument *size* may be used specify the dimensions of the array to be used. If an element of *size* is smaller than the corresponding dimension of *A*, then the dimension of *A* is truncated prior to performing the FFT. Otherwise, if an element of *size* is larger than the corresponding dimension then *A* is resized and padded with zeros.

See also: [ifftn], page 682, [fft], page 681, [fft2], page 681, [fftw], page 682.

`ifftn (A)` [Built-in Function]
`ifftn (A, size)` [Built-in Function]

Compute the inverse N-dimensional discrete Fourier transform of *A* using a Fast Fourier Transform (FFT) algorithm.

The optional vector argument *size* may be used specify the dimensions of the array to be used. If an element of *size* is smaller than the corresponding dimension of *A*, then the dimension of *A* is truncated prior to performing the inverse FFT. Otherwise, if an element of *size* is larger than the corresponding dimension then *A* is resized and padded with zeros.

See also: [fftn], page 682, [ifft], page 681, [ifft2], page 682, [fftw], page 682.

Octave uses the FFTW libraries to perform FFT computations. When Octave starts up and initializes the FFTW libraries, they read a system wide file (on a Unix system, it is typically `/etc/fftw/wisdom`) that contains information useful to speed up FFT computations. This information is called the *wisdom*. The system-wide file allows wisdom to be shared between all applications using the FFTW libraries.

Use the `fftw` function to generate and save wisdom. Using the utilities provided together with the FFTW libraries (`fftw-wisdom` on Unix systems), you can even add wisdom generated by Octave to the system-wide wisdom file.

method = fftw ("*planner*") [Loadable Function]
fftw ("*planner*", *method*) [Loadable Function]
wisdom = fftw ("*dwisdom*") [Loadable Function]
fftw ("*dwisdom*", *wisdom*) [Loadable Function]
fftw ("*threads*", *nthreads*) [Loadable Function]
nthreads = fftw ("*threads*") [Loadable Function]

Manage FFTW wisdom data.

Wisdom data can be used to significantly accelerate the calculation of the FFTs, but implies an initial cost in its calculation. When the FFTW libraries are initialized, they read a system wide wisdom file (typically in `/etc/fftw/wisdom`), allowing wisdom to be shared between applications other than Octave. Alternatively, the `fftw` function can be used to import wisdom. For example,

```
wisdom = fftw ("dwisdom")
```

will save the existing wisdom used by Octave to the string *wisdom*. This string can then be saved to a file and restored using the `save` and `load` commands respectively. This existing wisdom can be re-imported as follows

```
fftw ("dwisdom", wisdom)
```

If *wisdom* is an empty string, then the wisdom used is cleared.

During the calculation of Fourier transforms further wisdom is generated. The fashion in which this wisdom is generated is also controlled by the `fftw` function. There are five different manners in which the wisdom can be treated:

`"estimate"`
: Specifies that no run-time measurement of the optimal means of calculating a particular is performed, and a simple heuristic is used to pick a (probably sub-optimal) plan. The advantage of this method is that there is little or no overhead in the generation of the plan, which is appropriate for a Fourier transform that will be calculated once.

`"measure"`
: In this case a range of algorithms to perform the transform is considered and the best is selected based on their execution time.

`"patient"`
: Similar to `"measure"`, but a wider range of algorithms is considered.

`"exhaustive"`
: Like `"measure"`, but all possible algorithms that may be used to treat the transform are considered.

`"hybrid"`
: As run-time measurement of the algorithm can be expensive, this is a compromise where `"measure"` is used for transforms up to the size of 8192 and beyond that the `"estimate"` method is used.

The default method is `"estimate"`. The current method can be queried with

```
method = fftw ("planner")
```

or set by using

```
fftw ("planner", method)
```

Note that calculated wisdom will be lost when restarting Octave. However, the wisdom data can be reloaded if it is saved to a file as described above. Saved wisdom files should not be used on different platforms since they will not be efficient and the point of calculating the wisdom is lost.

The number of threads used for computing the plans and executing the transforms can be set with

```
fftw ("threads", NTHREADS)
```

Note that octave must be compiled with multi-threaded FFTW support for this feature. The number of processors available to the current process is used per default.

See also: [fft], page 681, [ifft], page 681, [fft2], page 681, [ifft2], page 682, [fftn], page 682, [ifftn], page 682.

`fftconv` (*x*, *y*) [Function File]
`fftconv` (*x*, *y*, *n*) [Function File]

Convolve two vectors using the FFT for computation.

`c = fftconv (x, y)` returns a vector of length equal to `length (x) + length (y) - 1`. If *x* and *y* are the coefficient vectors of two polynomials, the returned value is the coefficient vector of the product polynomial.

The computation uses the FFT by calling the function `fftfilt`. If the optional argument *n* is specified, an N-point FFT is used.

See also: [deconv], page 642, [conv], page 641, [conv2], page 642.

`fftfilt` (*b*, *x*) [Function File]
`fftfilt` (*b*, *x*, *n*) [Function File]

Filter *x* with the FIR filter *b* using the FFT.

If *x* is a matrix, filter each column of the matrix.

Given the optional third argument, *n*, `fftfilt` uses the overlap-add method to filter *x* with *b* using an N-point FFT. The FFT size must be an even power of 2 and must be greater than or equal to the length of *b*. If the specified *n* does not meet these criteria, it is automatically adjusted to the nearest value that does.

See also: [filter], page 684, [filter2], page 685.

y = `filter` (*b*, *a*, *x*) [Built-in Function]
[*y*, *sf*] = `filter` (*b*, *a*, *x*, *si*) [Built-in Function]
[*y*, *sf*] = `filter` (*b*, *a*, *x*, [], *dim*) [Built-in Function]
[*y*, *sf*] = `filter` (*b*, *a*, *x*, *si*, *dim*) [Built-in Function]

Apply a 1-D digital filter to the data *x*.

`filter` returns the solution to the following linear, time-invariant difference equation:

$$\sum_{k=0}^{N} a_{k+1} y_{n-k} = \sum_{k=0}^{M} b_{k+1} x_{n-k}, \qquad 1 \le n \le P$$

where $a \in \Re^{N-1}$, $b \in \Re^{M-1}$, and $x \in \Re^{P}$. The result is calculated over the first non-singleton dimension of *x* or over *dim* if supplied.

An equivalent form of the equation is:

$$y_n = -\sum_{k=1}^{N} c_{k+1} y_{n-k} + \sum_{k=0}^{M} d_{k+1} x_{n-k}, \qquad 1 \le n \le P$$

where $c = a/a_1$ and $d = b/a_1$.

If the fourth argument *si* is provided, it is taken as the initial state of the system and the final state is returned as *sf*. The state vector is a column vector whose length is equal to the length of the longest coefficient vector minus one. If *si* is not supplied, the initial state vector is set to all zeros.

In terms of the Z Transform, y is the result of passing the discrete-time signal x through a system characterized by the following rational system function:

$$H(z) = \frac{\displaystyle\sum_{k=0}^{M} d_{k+1} z^{-k}}{1 + \displaystyle\sum_{k+1}^{N} c_{k+1} z^{-k}}$$

See also: [filter2], page 685, [fftfilt], page 684, [freqz], page 685.

`y = filter2 (b, x)` [Function File]
`y = filter2 (b, x, shape)` [Function File]

Apply the 2-D FIR filter b to x.

If the argument *shape* is specified, return an array of the desired shape. Possible values are:

`"full"` pad x with zeros on all sides before filtering.

`"same"` unpadded x (default)

`"valid"` trim x after filtering so edge effects are no included.

Note this is just a variation on convolution, with the parameters reversed and b rotated 180 degrees.

See also: [conv2], page 642.

`[h, w] = freqz (b, a, n, "whole")` [Function File]
`[h, w] = freqz (b)` [Function File]
`[h, w] = freqz (b, a)` [Function File]
`[h, w] = freqz (b, a, n)` [Function File]
`h = freqz (b, a, w)` [Function File]
`[h, w] = freqz (..., Fs)` [Function File]
`freqz (...)` [Function File]

Return the complex frequency response h of the rational IIR filter whose numerator and denominator coefficients are b and a, respectively.

The response is evaluated at n angular frequencies between 0 and 2π.

The output value w is a vector of the frequencies.

If *a* is omitted, the denominator is assumed to be 1 (this corresponds to a simple FIR filter).

If *n* is omitted, a value of 512 is assumed. For fastest computation, *n* should factor into a small number of small primes.

If the fourth argument, `"whole"`, is omitted the response is evaluated at frequencies between 0 and π.

`freqz (b, a, w)`

Evaluate the response at the specific frequencies in the vector *w*. The values for *w* are measured in radians.

`[...] = freqz (..., Fs)`

Return frequencies in Hz instead of radians assuming a sampling rate *Fs*. If you are evaluating the response at specific frequencies *w*, those frequencies should be requested in Hz rather than radians.

`freqz (...)`

Plot the magnitude and phase response of *h* rather than returning them.

See also: [freqz_plot], page 686.

`freqz_plot (w, h)` [Function File]
`freqz_plot (w, h, freq_norm)` [Function File]

Plot the magnitude and phase response of *h*.

If the optional *freq_norm* argument is true, the frequency vector *w* is in units of normalized radians. If *freq_norm* is false, or not given, then *w* is measured in Hertz.

See also: [freqz], page 685.

`sinc (x)` [Function File]

Compute the sinc function.

Return $\sin(\pi x)/(\pi x)$.

`b = unwrap (x)` [Function File]
`b = unwrap (x, tol)` [Function File]
`b = unwrap (x, tol, dim)` [Function File]

Unwrap radian phases by adding multiples of 2*pi as appropriate to remove jumps greater than *tol*.

tol defaults to pi.

Unwrap will work along the dimension *dim*. If *dim* is unspecified it defaults to the first non-singleton dimension.

`[a, b] = arch_fit (y, x, p, iter, gamma, a0, b0)` [Function File]

Fit an ARCH regression model to the time series *y* using the scoring algorithm in Engle's original ARCH paper.

The model is

```
y(t) = b(1) * x(t,1) + ... + b(k) * x(t,k) + e(t),
h(t) = a(1) + a(2) * e(t-1)^2 + ... + a(p+1) * e(t-p)^2
```

in which $e(t)$ is $N(0, h(t))$, given a time-series vector y up to time $t-1$ and a matrix of (ordinary) regressors x up to t. The order of the regression of the residual variance is specified by p.

If invoked as `arch_fit (y, k, p)` with a positive integer k, fit an ARCH(k, p) process, i.e., do the above with the t-th row of x given by

```
[1, y(t-1), ..., y(t-k)]
```

Optionally, one can specify the number of iterations *iter*, the updating factor *gamma*, and initial values $a0$ and $b0$ for the scoring algorithm.

`arch_rnd (a, b, t)` [Function File]

Simulate an ARCH sequence of length t with AR coefficients b and CH coefficients a.

The result $y(t)$ follows the model

```
y(t) = b(1) + b(2) * y(t-1) + ... + b(lb) * y(t-lb+1) + e(t),
```

where $e(t)$, given y up to time $t-1$, is $N(0, h(t))$, with

```
h(t) = a(1) + a(2) * e(t-1)^2 + ... + a(la) * e(t-la+1)^2
```

`[pval, lm] = arch_test (y, x, p)` [Function File]

For a linear regression model

```
y = x * b + e
```

perform a Lagrange Multiplier (LM) test of the null hypothesis of no conditional heteroscedascity against the alternative of CH(p).

I.e., the model is

```
y(t) = b(1) * x(t,1) + ... + b(k) * x(t,k) + e(t),
```

given y up to $t-1$ and x up to t, e(t) is $N(0, h(t))$ with

```
h(t) = v + a(1) * e(t-1)^2 + ... + a(p) * e(t-p)^2,
```

and the null is $a(1) == \ldots == a(p) == 0$.

If the second argument is a scalar integer, k, perform the same test in a linear autoregression model of order k, i.e., with

```
[1, y(t-1), ..., y(t-k)]
```

as the t-th row of x.

Under the null, LM approximately has a chisquare distribution with p degrees of freedom and *pval* is the p-value (1 minus the CDF of this distribution at LM) of the test.

If no output argument is given, the p-value is displayed.

`arma_rnd (a, b, v, t, n)` [Function File]

Return a simulation of the ARMA model.

The ARMA model is defined by

```
x(n) = a(1) * x(n-1) + ... + a(k) * x(n-k)
     + e(n) + b(1) * e(n-1) + ... + b(l) * e(n-l)
```

in which k is the length of vector a, l is the length of vector b and e is Gaussian white noise with variance v. The function returns a vector of length t.

The optional parameter *n* gives the number of dummy *x*(*i*) used for initialization, i.e., a sequence of length *t*+*n* is generated and *x*(*n*+1:*t*+*n*) is returned. If *n* is omitted, *n* = 100 is used.

`autoreg_matrix (y, k)` [Function File]

Given a time series (vector) *y*, return a matrix with ones in the first column and the first *k* lagged values of *y* in the other columns.

In other words, for *t* > *k*, `[1, y(t-1), ..., y(t-k)]` is the t-th row of the result.

The resulting matrix may be used as a regressor matrix in autoregressions.

`bartlett (m)` [Function File]

Return the filter coefficients of a Bartlett (triangular) window of length *m*.

For a definition of the Bartlett window see, e.g., A.V. Oppenheim & R. W. Schafer, *Discrete-Time Signal Processing*.

`blackman (m)` [Function File]
`blackman (m, "periodic")` [Function File]
`blackman (m, "symmetric")` [Function File]

Return the filter coefficients of a Blackman window of length *m*.

If the optional argument `"periodic"` is given, the periodic form of the window is returned. This is equivalent to the window of length *m*+1 with the last coefficient removed. The optional argument `"symmetric"` is equivalent to not specifying a second argument.

For a definition of the Blackman window, see, e.g., A.V. Oppenheim & R. W. Schafer, *Discrete-Time Signal Processing*.

`detrend (x, p)` [Function File]

If *x* is a vector, `detrend (x, p)` removes the best fit of a polynomial of order *p* from the data *x*.

If *x* is a matrix, `detrend (x, p)` does the same for each column in *x*.

The second argument *p* is optional. If it is not specified, a value of 1 is assumed. This corresponds to removing a linear trend.

The order of the polynomial can also be given as a string, in which case *p* must be either `"constant"` (corresponds to `p=0`) or `"linear"` (corresponds to `p=1`).

See also: [polyfit], page 645.

`[d, dd] = diffpara (x, a, b)` [Function File]

Return the estimator *d* for the differencing parameter of an integrated time series.

The frequencies from $[2 * pi * a/t, 2 * pi * b/T]$ are used for the estimation. If *b* is omitted, the interval $[2 * pi/T, 2 * pi * a/T]$ is used. If both *b* and *a* are omitted then $a = 0.5 * sqrt(T)$ and $b = 1.5 * sqrt(T)$ is used, where *T* is the sample size. If *x* is a matrix, the differencing parameter of each column is estimated.

The estimators for all frequencies in the intervals described above is returned in *dd*.

The value of *d* is simply the mean of *dd*.

Reference: P.J. Brockwell & R.A. Davis. *Time Series: Theory and Methods*. Springer 1987.

`durbinlevinson` (*c*, *oldphi*, *oldv*) [Function File]

Perform one step of the Durbin-Levinson algorithm.

The vector *c* specifies the autocovariances `[gamma_0, ..., gamma_t]` from lag 0 to *t*, *oldphi* specifies the coefficients based on *c*(*t*-1) and *oldv* specifies the corresponding error.

If *oldphi* and *oldv* are omitted, all steps from 1 to *t* of the algorithm are performed.

`fftshift` (*x*) [Function File]
`fftshift` (*x*, *dim*) [Function File]

Perform a shift of the vector *x*, for use with the `fft` and `ifft` functions, in order the move the frequency 0 to the center of the vector or matrix.

If *x* is a vector of *N* elements corresponding to *N* time samples spaced by *dt*, then `fftshift (fft (x))` corresponds to frequencies

```
f = [ -(ceil((N-1)/2):-1:1)*df 0 (1:floor((N-1)/2))*df ]
```

where *df* = 1 / *dt*.

If *x* is a matrix, the same holds for rows and columns. If *x* is an array, then the same holds along each dimension.

The optional *dim* argument can be used to limit the dimension along which the permutation occurs.

See also: [ifftshift], page 689.

`ifftshift` (*x*) [Function File]
`ifftshift` (*x*, *dim*) [Function File]

Undo the action of the `fftshift` function.

For even length *x*, `fftshift` is its own inverse, but odd lengths differ slightly.

See also: [fftshift], page 689.

`fractdiff` (*x*, *d*) [Function File]

Compute the fractional differences $(1-L)^d x$ where L denotes the lag-operator and d is greater than -1.

`hamming` (*m*) [Function File]
`hamming` (*m*, "*periodic*") [Function File]
`hamming` (*m*, "*symmetric*") [Function File]

Return the filter coefficients of a Hamming window of length *m*.

If the optional argument `"periodic"` is given, the periodic form of the window is returned. This is equivalent to the window of length *m*+1 with the last coefficient removed. The optional argument `"symmetric"` is equivalent to not specifying a second argument.

For a definition of the Hamming window see, e.g., A.V. Oppenheim & R. W. Schafer, *Discrete-Time Signal Processing*.

`hanning` (*m*) [Function File]
`hanning` (*m*, "*periodic*") [Function File]
`hanning` (*m*, "*symmetric*") [Function File]

Return the filter coefficients of a Hanning window of length *m*.

If the optional argument `"periodic"` is given, the periodic form of the window is returned. This is equivalent to the window of length *m*+1 with the last coefficient removed. The optional argument `"symmetric"` is equivalent to not specifying a second argument.

For a definition of the Hanning window see, e.g., A.V. Oppenheim & R. W. Schafer, *Discrete-Time Signal Processing*.

`hurst` (*x*) [Function File]

Estimate the Hurst parameter of sample *x* via the rescaled range statistic.

If *x* is a matrix, the parameter is estimated for every column.

pp = `pchip` (*x*, *y*) [Function File]
yi = `pchip` (*x*, *y*, *xi*) [Function File]

Return the Piecewise Cubic Hermite Interpolating Polynomial (pchip) of points *x* and *y*.

If called with two arguments, return the piecewise polynomial *pp* that may be used with `ppval` to evaluate the polynomial at specific points.

When called with a third input argument, `pchip` evaluates the pchip polynomial at the points *xi*. The third calling form is equivalent to `ppval (pchip (x, y), xi)`.

The variable *x* must be a strictly monotonic vector (either increasing or decreasing) of length *n*.

y can be either a vector or array. If *y* is a vector then it must be the same length *n* as *x*. If *y* is an array then the size of *y* must have the form

$$[s_1, s_2, \cdots, s_k, n]$$

The array is reshaped internally to a matrix where the leading dimension is given by

$$s_1 s_2 \cdots s_k$$

and each row of this matrix is then treated separately. Note that this is exactly opposite to `interp1` but is done for MATLAB compatibility.

See also: [spline], page 658, [ppval], page 653, [mkpp], page 652, [unmkpp], page 653.

[*Pxx*, *w*] = `periodogram` (*x*) [Function File]
[*Pxx*, *w*] = `periodogram` (*x*, *win*) [Function File]
[*Pxx*, *w*] = `periodogram` (*x*, *win*, *nfft*) [Function File]
[*Pxx*, *f*] = `periodogram` (*x*, *win*, *nfft*, *Fs*) [Function File]
[*Pxx*, *f*] = `periodogram` (..., "*range*") [Function File]
`periodogram` (...) [Function File]

Return the periodogram (Power Spectral Density) of *x*.

The possible inputs are:

x

data vector. If *x* is real-valued a one-sided spectrum is estimated. If *x* is complex-valued, or "*range*" specifies `"twosided"`, the full spectrum is estimated.

win window weight data. If window is empty or unspecified a default rectangular window is used. Otherwise, the window is applied to the signal (`x .* win`) before computing the periodogram. The window data must be a vector of the same length as *x*.

nfft number of frequency bins. The default is 256 or the next higher power of 2 greater than the length of *x* (`max (256, 2.^nextpow2 (length (x)))`). If *nfft* is greater than the length of the input then *x* will be zero-padded to the length of *nfft*.

Fs sampling rate. The default is 1.

range range of spectrum. `"onesided"` computes spectrum from [0..nfft/2+1]. `"twosided"` computes spectrum from [0..nfft-1].

The optional second output *w* are the normalized angular frequencies. For a one-sided calculation *w* is in the range [0, pi] if *nfft* is even and [0, pi) if *nfft* is odd. Similarly, for a two-sided calculation *w* is in the range [0, 2*pi] or [0, 2*pi) depending on *nfft*.

If a sampling frequency is specified, *Fs*, then the output frequencies *f* will be in the range [0, *Fs*/2] or [0, *Fs*/2) for one-sided calculations. For two-sided calculations the range will be [0, *Fs*).

When called with no outputs the periodogram is immediately plotted in the current figure window.

See also: [fft], page 681.

`sinetone (freq, rate, sec, ampl)` [Function File]

Return a sinetone of frequency *freq* with a length of *sec* seconds at sampling rate *rate* and with amplitude *ampl*.

The arguments *freq* and *ampl* may be vectors of common size.

The defaults are *rate* = 8000, *sec* = 1, and *ampl* = 64.

See also: [sinewave], page 691.

`sinewave (m, n, d)` [Function File]

Return an *m*-element vector with *i*-th element given by `sin (2 * pi * (i+d-1) / n)`.

The default value for *d* is 0 and the default value for *n* is *m*.

See also: [sinetone], page 691.

`spectral_adf (c)` [Function File]
`spectral_adf (c, win)` [Function File]
`spectral_adf (c, win, b)` [Function File]

Return the spectral density estimator given a vector of autocovariances *c*, window name *win*, and bandwidth, *b*.

The window name, e.g., `"triangle"` or `"rectangle"` is used to search for a function called `win_lw`.

If *win* is omitted, the triangle window is used.

If *b* is omitted, `1 / sqrt (length (x))` is used.

See also: [spectral_xdf], page 692.

`spectral_xdf (x)` [Function File]
`spectral_xdf (x, win)` [Function File]
`spectral_xdf (x, win, b)` [Function File]

Return the spectral density estimator given a data vector *x*, window name *win*, and bandwidth, *b*.

The window name, e.g., `"triangle"` or `"rectangle"` is used to search for a function called `win_sw`.

If *win* is omitted, the triangle window is used.

If *b* is omitted, `1 / sqrt (length (x))` is used.

See also: [spectral_adf], page 691.

`spencer (x)` [Function File]

Return Spencer's 15 point moving average of each column of *x*.

`y = stft (x)` [Function File]
`y = stft (x, win_size)` [Function File]
`y = stft (x, win_size, inc)` [Function File]
`y = stft (x, win_size, inc, num_coef)` [Function File]
`y = stft (x, win_size, inc, num_coef, win_type)` [Function File]
`[y, c] = stft (...)` [Function File]

Compute the short-time Fourier transform of the vector *x* with *num_coef* coefficients by applying a window of *win_size* data points and an increment of *inc* points.

Before computing the Fourier transform, one of the following windows is applied:

`"hanning"`
: win_type = 1

`"hamming"`
: win_type = 2

`"rectangle"`
: win_type = 3

The window names can be passed as strings or by the *win_type* number.

The following defaults are used for unspecified arguments: *win_size* = 80, *inc* = 24, *num_coef* = 64, and *win_type* = 1.

`y = stft (x, ...)` returns the absolute values of the Fourier coefficients according to the *num_coef* positive frequencies.

`[y, c] = stft (x, ...)` returns the entire STFT-matrix *y* and a 3-element vector *c* containing the window size, increment, and window type, which is needed by the `synthesis` function.

See also: [synthesis], page 692.

`x = synthesis (y, c)` [Function File]

Compute a signal from its short-time Fourier transform *y* and a 3-element vector *c* specifying window size, increment, and window type.

The values *y* and *c* can be derived by

```
[y, c] = stft (x , ...)
```

See also: [stft], page 692.

`[a, v] = yulewalker (c)` [Function File]

Fit an AR (p)-model with Yule-Walker estimates given a vector *c* of autocovariances `[gamma_0, ..., gamma_p]`.

Returns the AR coefficients, *a*, and the variance of white noise, *v*.

32 Image Processing

Since an image is basically a matrix, Octave is a very powerful environment for processing and analyzing images. To illustrate how easy it is to do image processing in Octave, the following example will load an image, smooth it by a 5-by-5 averaging filter, and compute the gradient of the smoothed image.

```
I = imread ("myimage.jpg");
S = conv2 (I, ones (5, 5) / 25, "same");
[Dx, Dy] = gradient (S);
```

In this example `S` contains the smoothed image, and `Dx` and `Dy` contains the partial spatial derivatives of the image.

32.1 Loading and Saving Images

The first step in most image processing tasks is to load an image into Octave which is done with the `imread` function. The `imwrite` function is the corresponding function for writing images to the disk.

In summary, most image processing code will follow the structure of this code

```
I = imread ("my_input_image.img");
J = process_my_image (I);
imwrite (J, "my_output_image.img");
```

[*img*, *map*, *alpha*] = imread (*filename*) [Function File]
[...] = imread (*url*) [Function File]
[...] = imread (..., *ext*) [Function File]
[...] = imread (..., *idx*) [Function File]
[...] = imread (..., *param1*, *val1*, ...) [Function File]

Read images from various file formats.

Read an image as a matrix from the file *filename*. If there is no file *filename*, and *ext* was specified, it will look for a file with the extension *ext*. Finally, it will attempt to download and read an image from *url*.

The size and class of the output depends on the format of the image. A color image is returned as an MxNx3 matrix. Gray-level and black-and-white images are of size MxN. Multipage images will have an additional 4th dimension.

The bit depth of the image determines the class of the output: `"uint8"`, `"uint16"` or `"single"` for gray and color, and `"logical"` for black and white. Note that indexed images always return the indexes for a colormap, independent if *map* is a requested output. To obtain the actual RGB image, use `ind2rgb`. When more than one indexed image is being read, *map* is obtained from the first. In some rare cases this may be incorrect and `imfinfo` can be used to obtain the colormap of each image.

See the Octave manual for more information in representing images.

Some file formats, such as TIFF and GIF, are able to store multiple images in a single file. *idx* can be a scalar or vector specifying the index of the images to read. By default, Octave will only read the first page.

Depending on the file format, it is possible to configure the reading of images with *param*, *val* pairs. The following options are supported:

'`"Frames" or "Index"`'
: This is an alternative method to specify *idx*. When specifying it in this way, its value can also be the string `"all"`.

'`"Info"`'
: This option exists for MATLAB compatibility and has no effect. For maximum performance while reading multiple images from a single file, use the Index option.

'`"PixelRegion"`'
: Controls the image region that is read. Takes as value a cell array with two arrays of 3 elements `{rows cols}`. The elements in the array are the start, increment and end pixel to be read. If the increment value is omitted, defaults to 1. For example, the following are all equivalent:

```
imread (filename, "PixelRegion", {[200 600] [300 700]});
imread (filename, "PixelRegion", {[200 1 600] [300 1 700]});
imread (filename)(200:600, 300:700);
```

See also: [imwrite], page 696, [imfinfo], page 697, [imformats], page 700.

`imwrite` (*img*, *filename*) [Function File]
`imwrite` (*img*, *filename*, *ext*) [Function File]
`imwrite` (*img*, *map*, *filename*) [Function File]
`imwrite` (..., *param1*, *val1*, ...) [Function File]

Write images in various file formats.

The image *img* can be a binary, grayscale, RGB, or multi-dimensional image. The size and class of *img* should be the same as what should be expected when reading it with `imread`: the 3rd and 4th dimensions reserved for color space, and multiple pages respectively. If it's an indexed image, the colormap *map* must also be specified.

If *ext* is not supplied, the file extension of *filename* is used to determine the format. The actual supported formats are dependent on options made during the build of Octave. Use `imformats` to check the support of the different image formats.

Depending on the file format, it is possible to configure the writing of images with *param*, *val* pairs. The following options are supported:

'`Alpha`'
: Alpha (transparency) channel for the image. This must be a matrix with same class, and number of rows and columns of *img*. In case of a multipage image, the size of the 4th dimension must also match and the third dimension must be a singleton. By default, image will be completely opaque.

'`DelayTime`'
: For formats that accept animations (such as GIF), controls for how long a frame is displayed until it moves to the next one. The value must be scalar (which will applied to all frames in *img*), or a vector of length equal to the number of frames in *im*. The value is in seconds, must be between 0 and 655.35, and defaults to 0.5.

'`DisposalMethod`'
: For formats that accept animations (such as GIF), controls what happens to a frame before drawing the next one. Its value can be one of the following strings: "doNotSpecify" (default); "leaveInPlace"; "restoreBG"; and "restorePrevious", or a cell array of those string with length equal to the number of frames in *img*.

'`LoopCount`'
: For formats that accept animations (such as GIF), controls how many times the sequence is repeated. A value of Inf means an infinite loop (default), a value of 0 or 1 that the sequence is played only once (loops zero times), while a value of 2 or above loops that number of times (looping twice means it plays the complete sequence 3 times). This option is ignored when there is only a single image at the end of writing the file.

'`Quality`' Set the quality of the compression. The value should be an integer between 0 and 100, with larger values indicating higher visual quality and lower compression. Defaults to 75.

'`WriteMode`'
: Some file formats, such as TIFF and GIF, are able to store multiple images in a single file. This option specifies if *img* should be appended to the file (if it exists) or if a new file should be created for it (possibly overwriting an existing file). The value should be the string `"Overwrite"` (default), or `"Append"`.

: Despite this option, the most efficient method of writing a multipage image is to pass a 4 dimensional *img* to `imwrite`, the same matrix that could be expected when using `imread` with the option `"Index"` set to `"all"`.

See also: [imread], page 695, [imfinfo], page 697, [imformats], page 700.

val = IMAGE_PATH () [Built-in Function]
old_val = IMAGE_PATH (*new_val*) [Built-in Function]
IMAGE_PATH (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies a colon separated list of directories in which to search for image files.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [EXEC_PATH], page 779, [OCTAVE_HOME], page 789.

It is possible to get information about an image file on disk, without actually reading it into Octave. This is done using the `imfinfo` function which provides read access to many of the parameters stored in the header of the image file.

info = imfinfo (*filename*) [Function File]
info = imfinfo (*url*) [Function File]
info = imfinfo (..., *ext*) [Function File]

Read image information from a file.

`imfinfo` returns a structure containing information about the image stored in the file *filename*. If there is no file *filename*, and *ext* was specified, it will look for a file named *filename* and extension *ext*, i.e., a file named *filename.ext*.

The output structure *info* contains the following fields:

'`Filename`'
: The full name of the image file.

'`FileModDate`'
: Date of last modification to the file.

'`FileSize`'
: Number of bytes of the image on disk

'`Format`'
: Image format (e.g., `"jpeg"`).

'`Height`'
: Image height in pixels.

'`Width`'
: Image Width in pixels.

'`BitDepth`'
: Number of bits per channel per pixel.

'`ColorType`'
: Image type. Value is `"grayscale"`, `"indexed"`, `"truecolor"`, `"CMYK"`, or `"undefined"`.

'`XResolution`'
: X resolution of the image.

'`YResolution`'
: Y resolution of the image.

'`ResolutionUnit`'
: Units of image resolution. Value is `"Inch"`, `"Centimeter"`, or `"undefined"`.

'`DelayTime`'
: Time in 1/100ths of a second (0 to 65535) which must expire before displaying the next image in an animated sequence.

'`LoopCount`'
: Number of iterations to loop an animation.

'`ByteOrder`'
: Endian option for formats that support it. Value is `"little-endian"`, `"big-endian"`, or `"undefined"`.

'`Gamma`'
: Gamma level of the image. The same color image displayed on two different workstations may look different due to differences in the display monitor.

'`Quality`'
: JPEG/MIFF/PNG compression level. Value is an integer in the range [0 100].

'DisposalMethod'
: Only valid for GIF images, control how successive frames are rendered (how the preceding frame is disposed of) when creating a GIF animation. Values can be `"doNotSpecify"`, `"leaveInPlace"`, `"restoreBG"`, or `"restorePrevious"`. For non-GIF files, value is an empty string.

'Chromaticities'
: Value is a 1x8 Matrix with the x,y chromaticity values for white, red, green, and blue points, in that order.

'Comment'
: Image comment.

'Compression'
: Compression type. Value can be `"none"`, `"bzip"`, `"fax3"`, `"fax4"`, `"jpeg"`, `"lzw"`, `"rle"`, `"deflate"`, `"lzma"`, `"jpeg2000"`, `"jbig2"`, `"jbig2"`, or `"undefined"`.

'Colormap'
: Colormap for each image.

'Orientation'
: The orientation of the image with respect to the rows and columns. Value is an integer between 1 and 8 as defined in the TIFF 6 specifications, and for MATLAB compatibility.

'Software'
: Name and version of the software or firmware of the camera or image input device used to generate the image.

'Make'
: The manufacturer of the recording equipment. This is the manufacture of the DSC, scanner, video digitizer or other equipment that generated the image.

'Model'
: The model name or model number of the recording equipment as mentioned on the field `"Make"`.

'DateTime'
: The date and time of image creation as defined by the Exif standard, i.e., it is the date and time the file was changed.

'ImageDescription'
: The title of the image as defined by the Exif standard.

'Artist'
: Name of the camera owner, photographer or image creator.

'Copyright'
: Copyright notice of the person or organization claiming rights to the image.

'DigitalCamera'
: A struct with information retrieved from the Exif tag.

'GPSInfo'
: A struct with geotagging information retrieved from the Exif tag.

See also: [imread], page 695, [imwrite], page 696, [imshow], page 701, [imformats], page 700.

By default, Octave's image IO functions (`imread`, `imwrite`, and `imfinfo`) use the `GraphicsMagick` library for their operations. This means a vast number of image formats is supported but considering the large amount of image formats in science and its commonly closed nature, it is impossible to have a library capable of reading them all. Because of this, the function `imformats` keeps a configurable list of available formats, their extensions, and what functions should the image IO functions use. This allows one to expand Octave's image IO capabilities by creating functions aimed at acting on specific file formats.

While it would be possible to call the extra functions directly, properly configuring Octave with `imformats` allows one to keep a consistent code that is abstracted from file formats.

It is important to note that a file format is not actually defined by its file extension and that `GraphicsMagick` is capable to read and write more file formats than the ones listed by `imformats`. What this means is that even with an incorrect or missing extension the image may still be read correctly, and that even unlisted formats are not necessarily unsupported.

`imformats ()` [Function File]
formats = `imformats (`*ext*`)` [Function File]
formats = `imformats (`*format*`)` [Function File]
formats = `imformats ("`*add*`",` *format*`)` [Function File]
formats = `imformats ("`*remove*`",` *ext*`)` [Function File]
formats = `imformats ("`*update*`",` *ext*, *format*`)` [Function File]
formats = `imformats ("`*factory*`")` [Function File]

Manage supported image formats.

formats is a structure with information about each supported file format, or from a specific format *ext*, the value displayed on the field `ext`. It contains the following fields:

ext — The name of the file format. This may match the file extension but Octave will automatically detect the file format.

description — A long description of the file format.

isa — A function handle to confirm if a file is of the specified format.

write — A function handle to write if a file is of the specified format.

read — A function handle to open files the specified format.

info — A function handle to obtain image information of the specified format.

alpha — Logical value if format supports alpha channel (transparency or matte).

multipage — Logical value if format supports multipage (multiple images per file).

It is possible to change the way Octave manages file formats with the options `"add"`, `"remove"`, and `"update"`, and supplying a structure *format* with the required fields. The option `"factory"` resets the configuration to the default.

This can be used by Octave packages to extend the image reading capabilities Octave, through use of the PKG_ADD and PKG_DEL commands.

See also: [imfinfo], page 697, [imread], page 695, [imwrite], page 696.

32.2 Displaying Images

A natural part of image processing is visualization of an image. The most basic function for this is the `imshow` function that shows the image given in the first input argument.

`imshow (`*im*`)` [Function File]
`imshow (`*im*`, `*limits*`)` [Function File]
`imshow (`*im*`, `*map*`)` [Function File]
`imshow (`*rgb*`, ...)` [Function File]
`imshow (`*filename*`)` [Function File]
`imshow (..., `*string_param1*`, `*value1*`, ...)` [Function File]
h `= imshow (...)` [Function File]

Display the image *im*, where *im* can be a 2-dimensional (grayscale image) or a 3-dimensional (RGB image) matrix.

If *limits* is a 2-element vector `[low, high]`, the image is shown using a display range between *low* and *high*. If an empty matrix is passed for *limits*, the display range is computed as the range between the minimal and the maximal value in the image.

If *map* is a valid color map, the image will be shown as an indexed image using the supplied color map.

If a file name is given instead of an image, the file will be read and shown.

If given, the parameter *string_param1* has value *value1*. *string_param1* can be any of the following:

`"displayrange"`
: *value1* is the display range as described above.

`"colormap"`
: *value1* is the colormap to use when displaying an indexed image.

`"xdata"`
: If *value1* is a two element vector, it must contain horizontal axis limits in the form [xmin xmax]; Otherwise *value1* must be a vector and only the first and last elements will be used for xmin and xmax respectively.

`"ydata"`
: If *value1* is a two element vector, it must contain vertical axis limits in the form [ymin ymax]; Otherwise *value1* must be a vector and only the first and last elements will be used for ymin and ymax respectively.

The optional return value *h* is a graphics handle to the image.

See also: [image], page 701, [imagesc], page 702, [colormap], page 704, [gray2ind], page 703, [rgb2ind], page 704.

`image (`*img*`)` [Function File]
`image (`*x*`, `*y*`, `*img*`)` [Function File]
`image (..., "`*prop*`", `*val*`, ...)` [Function File]
`image ("`*prop1*`", `*val1*`, ...)` [Function File]
h `= image (...)` [Function File]

Display a matrix as an indexed color image.

The elements of *img* are indices into the current colormap.

x and *y* are optional 2-element vectors, `[min, max]`, which specify the range for the axis labels. If a range is specified as `[max, min]` then the image will be reversed along that axis. For convenience, *x* and *y* may be specified as N-element vectors matching the length of the data in *img*. However, only the first and last elements will be used to determine the axis limits. **Warning:** *x* and *y* are ignored when using gnuplot 4.0 or earlier.

Multiple property/value pairs may be specified for the image object, but they must appear in pairs.

The optional return value *h* is a graphics handle to the image.

Implementation Note: The origin (0, 0) for images is located in the upper left. For ordinary plots, the origin is located in the lower left. Octave handles this inversion by plotting the data normally, and then reversing the direction of the y-axis by setting the `ydir` property to `"reverse"`. This has implications whenever an image and an ordinary plot need to be overlaid. The recommended solution is to display the image and then plot the reversed ydata using, for example, `flipud (ydata)`.

Calling Forms: The `image` function can be called in two forms: High-Level and Low-Level. When invoked with normal options, the High-Level form is used which first calls `newplot` to prepare the graphic figure and axes. When the only inputs to `image` are property/value pairs the Low-Level form is used which creates a new instance of an image object and inserts it in the current axes.

See also: [imshow], page 701, [imagesc], page 702, [colormap], page 704.

`imagesc (`*`img`*`)` [Function File]
`imagesc (`*`x`*`, `*`y`*`, `*`img`*`)` [Function File]
`imagesc (..., `*`climits`*`)` [Function File]
`imagesc (..., "`*`prop`*`", `*`val`*`, ...)` [Function File]
`imagesc ("`*`prop1`*`", `*`val1`*`, ...)` [Function File]
`imagesc (`*`hax`*`, ...)` [Function File]
`h`` = imagesc (...)` [Function File]

Display a scaled version of the matrix *img* as a color image.

The colormap is scaled so that the entries of the matrix occupy the entire colormap. If *`climits`*` = [`*`lo`*`, `*`hi`*`]` is given, then that range is set to the `"clim"` of the current axes.

The axis values corresponding to the matrix elements are specified in *x* and *y*, either as pairs giving the minimum and maximum values for the respective axes, or as values for each row and column of the matrix *img*.

The optional return value *h* is a graphics handle to the image.

Calling Forms: The `imagesc` function can be called in two forms: High-Level and Low-Level. When invoked with normal options, the High-Level form is used which first calls `newplot` to prepare the graphic figure and axes. When the only inputs to `image` are property/value pairs the Low-Level form is used which creates a new instance of an image object and inserts it in the current axes.

See also: [image], page 701, [imshow], page 701, [caxis], page 300.

32.3 Representing Images

In general Octave supports four different kinds of images, grayscale images, RGB images, binary images, and indexed images. A grayscale image is represented with an M-by-N matrix in which each element corresponds to the intensity of a pixel. An RGB image is represented with an M-by-N-by-3 array where each 3-vector corresponds to the red, green, and blue intensities of each pixel.

The actual meaning of the value of a pixel in a grayscale or RGB image depends on the class of the matrix. If the matrix is of class `double` pixel intensities are between 0 and 1, if it is of class `uint8` intensities are between 0 and 255, and if it is of class `uint16` intensities are between 0 and 65535.

A binary image is an M-by-N matrix of class `logical`. A pixel in a binary image is black if it is `false` and white if it is `true`.

An indexed image consists of an M-by-N matrix of integers and a C-by-3 color map. Each integer corresponds to an index in the color map, and each row in the color map corresponds to an RGB color. The color map must be of class `double` with values between 0 and 1.

`iscolormap (`*cmap*`)` [Function File]

Return true if *cmap* is a colormap.

A colormap is a real matrix with *n* rows and 3 columns. Each row represents a single color. The columns contain red, green, and blue intensities respectively. All entries must be between 0 and 1 inclusive.

See also: [colormap], page 704, [rgbplot], page 705.

img `= gray2ind (`*I*`)` [Function File]
img `= gray2ind (`*I*, *n*`)` [Function File]
img `= gray2ind (`*BW*`)` [Function File]
img `= gray2ind (`*BW*, *n*`)` [Function File]
`[`*img*`,` *map*`] = gray2ind (...)` [Function File]

Convert a grayscale or binary intensity image to an indexed image.

The indexed image will consist of *n* different intensity values. If not given *n* defaults to 64 for grayscale images or 2 for binary black and white images.

The output *img* is of class uint8 if *n* is less than or equal to 256; Otherwise the return class is uint16.

See also: [ind2gray], page 703, [rgb2ind], page 704.

I `= ind2gray (`*x*, *map*`)` [Function File]

Convert a color indexed image to a grayscale intensity image.

The image *x* must be an indexed image which will be converted using the colormap *cmap*. If *cmap* does not contain enough colors for the image, pixels in *x* outside the range are mapped to the last color in the map before conversion to grayscale.

The output *I* is of the same class as the input *x* and may be one of `uint8`, `uint16`, `single`, or `double`.

Implementation Note: There are several ways of converting colors to grayscale intensities. This functions uses the luminance value obtained from `rgb2ntsc` which is `I =`

0.299*R + 0.587*G + 0.114*B. Other possibilities include the value component from rgb2hsv or using a single color channel from ind2rgb.

See also: [gray2ind], page 703, [ind2rgb], page 704.

`[x, map] = rgb2ind (rgb)` [Function File]
`[x, map] = rgb2ind (R, G, B)` [Function File]

Convert an image in red-green-blue (RGB) color space to an indexed image.

The input image *rgb* can be specified as a single matrix of size MxNx3, or as three separate variables, *R*, *G*, and *B*, its three color channels, red, green, and blue.

It outputs an indexed image *x* and a colormap *map* to interpret an image exactly the same as the input. No dithering or other form of color quantization is performed. The output class of the indexed image *x* can be uint8, uint16 or double, whichever is required to specify the number of unique colors in the image (which will be equal to the number of rows in *map*) in order

Multi-dimensional indexed images (of size MxNx3xK) are also supported, both via a single input (*rgb*) or its three color channels as separate variables.

See also: [ind2rgb], page 704, [rgb2hsv], page 711, [rgb2ntsc], page 712.

`rgb = ind2rgb (x, map)` [Function File]
`[R, G, B] = ind2rgb (x, map)` [Function File]

Convert an indexed image to red, green, and blue color components.

The image *x* must be an indexed image which will be converted using the colormap *map*. If *map* does not contain enough colors for the image, pixels in *x* outside the range are mapped to the last color in the map.

The output may be a single RGB image (MxNx3 matrix where M and N are the original image *x* dimensions, one for each of the red, green and blue channels). Alternatively, the individual red, green, and blue color matrices of size MxN may be returned.

Multi-dimensional indexed images (of size MxNx1xK) are also supported.

See also: [rgb2ind], page 704, [ind2gray], page 703, [hsv2rgb], page 711, [ntsc2rgb], page 712.

`cmap = colormap ()` [Function File]
`cmap = colormap (map)` [Function File]
`cmap = colormap ("default")` [Function File]
`cmap = colormap ("map_name")` [Function File]
`cmap = colormap (hax, ...)` [Function File]
`colormap map_name` [Command]
`cmaps = colormap ("list")` [Function File]
`colormap ("register", "name")` [Function File]
`colormap ("unregister", "name")` [Function File]

Query or set the current colormap.

With no input arguments, `colormap` returns the current color map.

`colormap (map)` sets the current colormap to *map*. The colormap should be an *n* row by 3 column matrix. The columns contain red, green, and blue intensities respectively. All entries must be between 0 and 1 inclusive. The new colormap is returned.

`colormap ("default")` restores the default colormap (the `jet` map with 64 entries). The default colormap is returned.

The map may also be specified by a string, `"map_name"`, where *map_name* is the name of a function that returns a colormap.

If the first argument *hax* is an axes handle, then the colormap for the parent figure of *hax* is queried or set.

For convenience, it is also possible to use this function with the command form, `colormap map_name`.

`colormap ("list")` returns a cell array with all of the available colormaps. The options `"register"` and `"unregister"` add or remove the colormap *name* from this list.

See also: [jet], page 707.

`rgbplot (cmap)` [Function File]
`rgbplot (cmap, style)` [Function File]
`h = rgbplot (...)` [Function File]

Plot the components of a colormap.

Two different *style*s are available for displaying the *cmap*:

profile (default)
: Plot the RGB line profile of the colormap for each of the channels (red, green and blue) with the plot lines colored appropriately. Each line represents the intensity of each RGB components across the colormap.

composite
: Draw the colormap across the X-axis so that the actual index colors are visible rather than the individual color components.

The optional return value *h* is a graphics handle to the created plot.

Run `demo rgbplot` to see an example of `rgbplot` and each style option.

See also: [colormap], page 704.

`map = autumn ()` [Function File]
`map = autumn (n)` [Function File]

Create color colormap. This colormap ranges from red through orange to yellow.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

`map = bone ()` [Function File]
`map = bone (n)` [Function File]

Create color colormap. This colormap varies from black to white with gray-blue shades.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = colorcube () [Function File]
map = colorcube (*n*) [Function File]

Create color colormap. This colormap is composed of as many equally spaced colors (not grays) in the RGB color space as possible.

If there are not a perfect number *n* of regularly spaced colors then the remaining entries in the colormap are gradients of pure red, green, blue, and gray.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = cool () [Function File]
map = cool (*n*) [Function File]

Create color colormap. The colormap varies from cyan to magenta.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = copper () [Function File]
map = copper (*n*) [Function File]

Create color colormap. This colormap varies from black to a light copper tone.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = cubehelix () [Function File]
map = cubehelix (*n*) [Function File]

Create cubehelix colormap.

This colormap varies from black to white going though blue, green, and red tones while maintaining a monotonically increasing perception of intensity. This is achieved by transversing a color cube from black to white through a helix, hence the name cubehelix, while taking into account the perceived brightness of each channel according to the NTSC specifications from 1953.

```
rgbplot (cubehelix (256))
```

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

Reference: Green, D. A., 2011, "*A colour scheme for the display of astronomical intensity images*", Bulletin of the Astronomical Society of India, 39, 289.

See also: [colormap], page 704.

map = flag () [Function File]
map = flag (*n*) [Function File]

Create color colormap. This colormap cycles through red, white, blue, and black with each index change.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = gray () [Function File]
map = gray (*n*) [Function File]

Create gray colormap. This colormap varies from black to white with shades of gray.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = hot () [Function File]
map = hot (*n*) [Function File]

Create color colormap. This colormap ranges from black through dark red, red, orange, yellow, to white.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

hsv (*n*) [Function File]

Create color colormap. This colormap begins with red, changes through yellow, green, cyan, blue, and magenta, before returning to red.

It is useful for displaying periodic functions. The map is obtained by linearly varying the hue through all possible values while keeping constant maximum saturation and value. The equivalent code is `hsv2rgb ([(0:N-1)'/N, ones(N,2)])`.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = jet () [Function File]
map = jet (*n*) [Function File]

Create color colormap. This colormap ranges from dark blue through blue, cyan, green, yellow, red, to dark red.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = lines () [Function File]
map = lines (*n*) [Function File]

Create color colormap. This colormap is composed of the list of colors in the current axes `"ColorOrder"` property. The default is blue, green, red, cyan, pink, yellow, and gray.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = ocean () [Function File]
map = ocean (*n*) [Function File]

Create color colormap. This colormap varies from black to white with shades of blue.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = `pink ()` [Function File]
map = `pink (`*n*`)` [Function File]

Create color colormap. This colormap varies from black to white with shades of gray-pink.

This colormap gives a sepia tone when used on grayscale images.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = `prism ()` [Function File]
map = `prism (`*n*`)` [Function File]

Create color colormap. This colormap cycles through red, orange, yellow, green, blue and violet with each index change.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = `rainbow ()` [Function File]
map = `rainbow (`*n*`)` [Function File]

Create color colormap. This colormap ranges from red through orange, yellow, green, blue, to violet.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = `spring ()` [Function File]
map = `spring (`*n*`)` [Function File]

Create color colormap. This colormap varies from magenta to yellow.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = `summer ()` [Function File]
map = `summer (`*n*`)` [Function File]

Create color colormap. This colormap varies from green to yellow.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = white () [Function File]
map = white (*n*) [Function File]

Create color colormap. This colormap is completely white.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

map = winter () [Function File]
map = winter (*n*) [Function File]

Create color colormap. This colormap varies from blue to green.

The argument *n* must be a scalar. If unspecified, the length of the current colormap, or 64, is used.

See also: [colormap], page 704.

cmap = contrast (*x*) [Function File]
cmap = contrast (*x*, *n*) [Function File]

Return a gray colormap that maximizes the contrast in an image.

The returned colormap will have *n* rows. If *n* is not defined then the size of the current colormap is used.

See also: [colormap], page 704, [brighten], page 709.

The following three functions modify the existing colormap rather than replace it.

map_out = brighten (*beta*) [Function File]
map_out = brighten (*map*, *beta*) [Function File]
map_out = brighten (*h*, *beta*) [Function File]
brighten (...) [Function File]

Brighten or darken a colormap.

The argument *beta* must be a scalar between -1 and 1, where a negative value darkens and a positive value brightens the colormap.

If the *map* argument is omitted, the function is applied to the current colormap.

The first argument can also be a valid graphics handle *h*, in which case `brighten` is applied to the colormap associated with this handle.

If no output is specified then the result is written to the current colormap.

See also: [colormap], page 704, [contrast], page 709.

spinmap () [Function File]
spinmap (*t*) [Function File]
spinmap (*t*, *inc*) [Function File]
spinmap ("*inf*") [Function File]

Cycle the colormap for *t* seconds with a color increment of *inc*.

Both parameters are optional. The default cycle time is 5 seconds and the default increment is 2. If the option `"inf"` is given then cycle continuously until *Control-C* is pressed.

When rotating, the original color 1 becomes color 2, color 2 becomes color 3, etc. A positive or negative increment is allowed and a higher value of *inc* will cause faster cycling through the colormap.

See also: [colormap], page 704.

`whitebg ()` [Function File]
`whitebg (`*color*`)` [Function File]
`whitebg ("`*none*`")` [Function File]
`whitebg (`*hfig*`, ...)` [Function File]

Invert the colors in the current color scheme.

The root properties are also inverted such that all subsequent plot use the new color scheme.

If the optional argument *color* is present then the background color is set to *color* rather than inverted. *color* may be a string representing one of the eight known colors or an RGB triplet. The special string argument `"none"` restores the plot to the default colors.

If the first argument *hfig* is a figure handle, then operate on this figure rather than the current figure returned by `gcf`. The root properties will not be changed.

See also: [reset], page 386, [get], page 361, [set], page 361.

The following functions can be used to manipulate colormaps.

`[`*Y*`, `*newmap*`] = cmunique (`*X*`, `*map*`)` [Function File]
`[`*Y*`, `*newmap*`] = cmunique (`*RGB*`)` [Function File]
`[`*Y*`, `*newmap*`] = cmunique (`*I*`)` [Function File]

Convert an input image X to an ouput indexed image Y which uses the smallest colormap possible *newmap*.

When the input is an indexed image (X with colormap *map*) the output is a colormap *newmap* from which any repeated rows have been eliminated. The output image, Y, is the original input image with the indices adjusted to match the new, possibly smaller, colormap.

When the input is an RGB image (an MxNx3 array), the output colormap will contain one entry for every unique color in the original image. In the worst case the new map could have as many rows as the number of pixels in the original image.

When the input is a grayscale image I, the output colormap will contain one entry for every unique intensity value in the original image. In the worst case the new map could have as many rows as the number of pixels in the original image.

Implementation Details:

newmap is always an Mx3 matrix, even if the input image is an intensity grayscale image I (all three RGB planes are assigned the same value).

The output image is of class uint8 if the size of the new colormap is less than or equal to 256. Otherwise, the output image is of class double.

See also: [rgb2ind], page 704, [gray2ind], page 703.

[*Y*, *newmap*] = cmpermute (*X*, *map*) [Function File]
[*Y*, *newmap*] = cmpermute (*X*, *map*, *index*) [Function File]

Reorder colors in a colormap.

When called with only two arguments, `cmpermute` randomly rearranges the colormap *map* and returns a new colormap *newmap*. It also returns the indexed image Y which is the equivalent of the original input image X when displayed using *newmap*.

When called with an optional third argument the order of colors in the new colormap is defined by *index*.

Caution: `index` should not have repeated elements or the function will fail.

32.4 Plotting on top of Images

If gnuplot is being used to display images it is possible to plot on top of images. Since an image is a matrix it is indexed by row and column values. The plotting system is, however, based on the traditional (x, y) system. To minimize the difference between the two systems Octave places the origin of the coordinate system in the point corresponding to the pixel at $(1, 1)$. So, to plot points given by row and column values on top of an image, one should simply call `plot` with the column values as the first argument and the row values as the second. As an example the following code generates an image with random intensities between 0 and 1, and shows the image with red circles over pixels with an intensity above 0.99.

```
I = rand (100, 100);
[row, col] = find (I > 0.99);
hold ("on");
imshow (I);
plot (col, row, "ro");
hold ("off");
```

32.5 Color Conversion

Octave supports conversion from the RGB color system to NTSC and HSV and vice versa.

hsv_map = rgb2hsv (*rgb*) [Function File]
hsv_map = rgb2hsv (*rgb*) [Function File]

Transform a colormap or image from red-green-blue (RGB) space to hue-saturation-value (HSV) space.

A color in the RGB space consists of red, green, and blue intensities.

A color in HSV space is represented by hue, saturation, and value (brightness) levels. Value gives the amount of light in the color. Hue describes the dominant wavelength. Saturation is the amount of hue mixed into the color.

See also: [hsv2rgb], page 711, [rgb2ind], page 704, [rgb2ntsc], page 712.

rgb_map = hsv2rgb (*hsv_map*) [Function File]
rgb_img = hsv2rgb (*hsv_img*) [Function File]

Transform a colormap or image from hue-saturation-value (HSV) space to red-green-blue (RGB) space.

A color in HSV space is represented by hue, saturation and value (brightness) levels. Value gives the amount of light in the color. Hue describes the dominant wavelength. Saturation is the amount of hue mixed into the color.

A color in the RGB space consists of red, green, and blue intensities.

See also: [rgb2hsv], page 711, [ind2rgb], page 704, [ntsc2rgb], page 712.

yiq_map = rgb2ntsc (*rgb_map*) [Function File]
yiq_img = rgb2ntsc (*rgb_img*) [Function File]

Transform a colormap or image from red-green-blue (RGB) color space to luminance-chrominance (NTSC) space. The input may be of class uint8, uint16, single, or double. The output is of class double.

Implementation Note: The reference matrix for the transformation is

```
/Y\     0.299  0.587  0.114  /R\
|I|  =  0.596 -0.274 -0.322  |G|
\Q/     0.211 -0.523  0.312  \B/
```

as documented in `http://en.wikipedia.org/wiki/YIQ` and truncated to 3 significant figures. Note: The FCC version of NTSC uses only 2 significant digits and is slightly different.

See also: [ntsc2rgb], page 712, [rgb2hsv], page 711, [rgb2ind], page 704.

rgb_map = ntsc2rgb (*yiq_map*) [Function File]
rgb_img = ntsc2rgb (*yiq_img*) [Function File]

Transform a colormap or image from luminance-chrominance (NTSC) space to red-green-blue (RGB) color space.

Implementation Note: The conversion matrix is chosen to be the inverse of the matrix used for rgb2ntsc such that

```
x == ntsc2rgb (rgb2ntsc (x))
```

MATLAB uses a slightly different matrix where rounding means the equality above does not hold.

See also: [rgb2ntsc], page 712, [hsv2rgb], page 711, [ind2rgb], page 704.

33 Audio Processing

33.1 Audio File Utilities

The following functions allow you to read, write and retrieve information about audio files. Various formats are supported including wav, flac and ogg vorbis.

`info = audioinfo (filename)` [Loadable Function]

Return information about an audio file specified by *filename*.

`[y, fs] = audioread (filename)` [Loadable Function]
`[y, fs] = audioread (filename, samples)` [Loadable Function]
`[y, fs] = audioread (filename, datatype)` [Loadable Function]
`[y, fs] = audioread (filename, samples, datatype)` [Loadable Function]

Read the audio file *filename* and return the audio data *y* and sampling rate *fs*.

The audio data is stored as matrix with rows corresponding to audio frames and columns corresponding to channels.

The optional two-element vector argument *samples* specifies starting and ending frames.

The optional argument *datatype* specifies the datatype to return. If it is `"native"`, then the type of data depends on how the data is stored in the audio file.

`audiowrite (filename, y, fs)` [Loadable Function]
`audiowrite (filename, y, fs, name, value, ...)` [Loadable Function]

Write audio data from the matrix *y* to *filename* at sampling rate *fs* with the file format determined by the file extension.

Additional name/value argument pairs may be used to specify the following options:

‘`BitsPerSample`’
: Number of bits per sample, valid values are 8, 16, 24 and 32. Default is 16.

‘`BitRate`’
: Valid argument name, but ignored. Left for compatibility with MATLAB.

‘`Quality`’
: Quality setting for the Ogg Vorbis compressor. Values can range between 0 and 100 with 100 being the highest quality setting. Default is 75.

‘`Title`’
: Title for the audio file.

‘`Artist`’
: Artist name.

‘`Comment`’
: Comment.

33.2 Audio Device Information

`devinfo = audiodevinfo ()` [Loadable Function]
`devs = audiodevinfo (io)` [Loadable Function]
`name = audiodevinfo (io, id)` [Loadable Function]
`id = audiodevinfo (io, name)` [Loadable Function]
`id = audiodevinfo (io, rate, bits, chans)` [Loadable Function]

supports = audiodevinfo (*io*, *id*, *rate*, *bits*, *chans*) [Loadable Function]

Return a structure describing the available audio input and output devices.

The *devinfo* structure has two fields "input" and "output". The value of each field is a structure array with fields "Name", "DriverVersion" and "ID" describing an audio device.

If the optional argument *io* is 1, return information about input devices only. If it is 0, return information about output devices only.

If the optional argument *id* is provided, return information about the corresponding device.

If the optional argument *name* is provided, return the id of the named device.

Given a sampling rate, bits per sample, and number of channels for an input or output device, return the ID of the first device that supports playback or recording using the specified parameters.

If also given a device ID, return true if the device supports playback or recording using those parameters.

33.3 Audio Player

The following methods are used to create and use audioplayer objects. These objects can be used to play back audio data stored in Octave matrices and arrays. The audioplayer object supports playback from various devices available to the system, blocking and non-blocking playback, convenient pausing and resuming and much more.

player = audioplayer (*y*, *fs*) [Function File]
player = audioplayer (*y*, *fs*, *nbits*) [Function File]
player = audioplayer (*y*, *fs*, *nbits*, *id*) [Function File]
player = audioplayer (*recorder*) [Function File]
player = audioplayer (*recorder*, *id*) [Function File]

Create an audioplayer object that will play back data *y* at sample rate *fs*.

The optional arguments *nbits*, and *id* specify the bit depth and player device id, respectively. Device IDs may be found using the audiodevinfo function. Given an audioplayer object, use the data from the object to initialize the player.

The signal *y* can be a vector or a two-dimensional array.

The following example will create an audioplayer object that will play back one second of white noise at 44100 sample rate using 8 bits per sample.

```
y = randn (2, 44100) - 0.5;
player = audioplayer (y, 44100, 8);
play (player);
```

33.3.1 Playback

The following methods are used to control player playback.

play (*player*) [Function File]
play (*player*, *start*) [Function File]
play (*player*, *limits*) [Function File]

Play audio stored in the audioplayer object *player* without blocking.

Given optional argument start, begin playing at *start* seconds in the recording. Given a two-element vector *limits*, begin and end playing at the number of seconds specified by the elements of the vector.

`playblocking` (*player*) [Function File]
`playblocking` (*player*, *start*) [Function File]
`playblocking` (*player*, *limits*) [Function File]

Play audio stored in the audioplayer object *player* with blocking.

Given optional argument start, begin playing at *start* seconds in the recording. Given a two-element vector *limits*, begin and end playing at the number of seconds specified by the elements of the vector.

`pause` (*player*) [Function File]

Pause the audioplayer *player*.

`resume` (*player*) [Function File]

Resume playback for the paused audioplayer object *player*.

`stop` (*player*) [Function File]

Stop the playback for the audioplayer *player* and reset the relevant variables to their starting values.

`isplaying` (*player*) [Function File]

Return true if the audioplayer object *player* is currently playing back audio and false otherwise.

33.3.2 Properties

The remaining couple of methods are used to get and set various properties of the audioplayer object.

value = `get` (*player*, *name*) [Function File]
values = `get` (*player*) [Function File]

Return the *value* of the property identified by *name*.

If *name* is a cell array return the values of the properties identified by the elements of the cell array. Given only the player object, return a scalar structure with values of all properties of *player*. The field names of the structure correspond to property names.

`set` (*player*, *name*, *value*) [Function File]
`set` (*player*, *properties*) [Function File]
properties = `set` (*player*) [Function File]

Set the value of property specified by *name* to a given *value*.

If *name* and *value* are cell arrays, set each property to the corresponding value. Given a structure of *properties* with fields corresponding to property names, set the value of those properties to the field values. Given only the audioplayer object, return a structure of settable properties.

33.4 Audio Recorder

The following methods are used to create and use audiorecorder objects. These objects can be used to record audio data from various devices available to the system. You can use convenient methods to retrieve that data or audioplayer objects created from that data. Methods for blocking and non-blocking recording, pausing and resuming recording and much more is available.

`recorder = audiorecorder ()` [Function File]
`recorder = audiorecorder (fs, nbits, channels)` [Function File]
`recorder = audiorecorder (fs, nbits, channels, id)` [Function File]

Create an audiorecorder object recording 8 bit mono audio at 8000 Hz sample rate.

The optional arguments *fs*, *nbits*, *channels*, and *id* specify the sample rate, bit depth, number of channels and recording device id, respectively. Device IDs may be found using the audiodevinfo function.

33.4.1 Recording

The following methods control the recording process.

`record (recorder)` [Function File]
`record (recorder, length)` [Function File]

Record audio without blocking using the audiorecorder object *recorder* until stopped or paused by the *stop* or *pause* method.

Given the optional argument *length*, record for *length* seconds.

`recordblocking (recorder, length)` [Function File]

Record audio with blocking (synchronous I/O).

The length of the recording in seconds (*length*) must be specified.

`pause (recorder)` [Function File]

Pause recording with audiorecorder object *recorder*.

`resume (recorder)` [Function File]

Resume recording with the paused audiorecorder object *recorder*.

`stop (recorder)` [Function File]

Stop the audiorecorder object *recorder* and clean up any audio streams.

`isrecording (recorder)` [Function File]

Return true if the audiorecorder object *recorder* is currently recording audio and false otherwise.

33.4.2 Data Retrieval

The following methods allow you to retrieve recorded audio data in various ways.

`data = getaudiodata (recorder)` [Function File]
`data = getaudiodata (recorder, datatype)` [Function File]

Return recorder audio data as a matrix with values between -1.0 and 1.0 and with as many columns as there are channels in the recorder.

Given the optional argument *datatype*, convert the recorded data to the specified type, which may be one of `"double"`, `"single"`, `"int16"`, `"int8"` or `"uint8"`.

player = getplayer (*recorder*) [Function File]

Return an audioplayer object with data recorded by the audiorecorder object *recorder*.

player = play (*recorder*) [Function File]
player = play (*recorder*, *start*) [Function File]
player = play (*recorder*, [*start*, *end*]) [Function File]

Play the audio recorded in *recorder* and return a corresponding audioplayer object.

If the optional argument *start* is provided, begin playing *start* seconds in to the recording.

If the optional argument *end* is provided, stop playing at *end* seconds in the recording.

33.4.3 Properties

The remaining two methods allow you to read or alter the properties of audiorecorder objects.

value = get (*recorder*, *name*) [Function File]
values = get (*recorder*) [Function File]

Return the *value* of the property identified by *name*.

If *name* is a cell array, return the values of the properties corresponding to the elements of the cell array. Given only the recorder object, return a scalar structure with values of all properties of *recorder*. The field names of the structure correspond to property names.

set (*recorder*, *name*, *value*) [Function File]
set (*recorder*, *properties*) [Function File]
properties = set (*recorder*) [Function File]

Set the value of property specified by *name* to a given *value*.

If *name* and *value* are cell arrays of the same size, set each property to a corresponding value. Given a structure with fields corresponding to property names, set the value of those properties to the corresponding field values. Given only the recorder object, return a structure of settable properties.

33.5 Audio Data Processing

Octave provides a few functions for dealing with audio data. An audio 'sample' is a single output value from an A/D converter, i.e., a small integer number (usually 8 or 16 bits), and audio data is just a series of such samples. It can be characterized by three parameters: the sampling rate (measured in samples per second or Hz, e.g., 8000 or 44100), the number of bits per sample (e.g., 8 or 16), and the number of channels (1 for mono, 2 for stereo, etc.).

There are many different formats for representing such data. Currently, only the two most popular, *linear encoding* and *mu-law encoding*, are supported by Octave. There is an excellent FAQ on audio formats by Guido van Rossum `guido@cwi.nl` which can be found at any FAQ ftp site, in particular in the directory `/pub/usenet/news.answers/audio-fmts` of the archive site `rtfm.mit.edu`.

Octave simply treats audio data as vectors of samples (non-mono data are not supported yet). It is assumed that audio files using linear encoding have one of the extensions `lin` or `raw`, and that files holding data in mu-law encoding end in `au`, `mu`, or `snd`.

`lin2mu (x, n)` [Function File]

Convert audio data from linear to mu-law.

Mu-law values use 8-bit unsigned integers. Linear values use *n*-bit signed integers or floating point values in the range -1 ≤ *x* ≤ 1 if *n* is 0.

If *n* is not specified it defaults to 0, 8, or 16 depending on the range of values in *x*.

See also: [mu2lin], page 718.

`mu2lin (x, n)` [Function File]

Convert audio data from mu-law to linear.

Mu-law values are 8-bit unsigned integers. Linear values use *n*-bit signed integers or floating point values in the range -1≤y≤1 if *n* is 0.

If *n* is not specified it defaults to 0.

See also: [lin2mu], page 717.

`record (sec)` [Function File]
`record (sec, fs)` [Function File]

Record *sec* seconds of audio from the system's default audio input at a sampling rate of 8000 samples per second.

If the optional argument *fs* is given, it specifies the sampling rate for recording.

For more control over audio recording, use the `audiorecorder` class.

See also: [sound], page 718, [soundsc], page 718.

`sound (y)` [Function File]
`sound (y, fs)` [Function File]
`sound (y, fs, nbits)` [Function File]

Play audio data *y* at sample rate *fs* to the default audio device.

The audio signal *y* can be a vector or a two-column array, representing mono or stereo audio, respectively.

If *fs* is not given, a default sample rate of 8000 samples per second is used.

The optional argument *nbits* specifies the bit depth to play to the audio device and defaults to 8 bits.

For more control over audio playback, use the `audioplayer` class.

See also: [soundsc], page 718, [record], page 718.

`soundsc (y)` [Function File]
`soundsc (y, fs)` [Function File]
`soundsc (y, fs, nbits)` [Function File]
`soundsc (..., [ymin, ymax])` [Function File]

Scale the audio data *y* and play it at sample rate *fs* to the default audio device.

The audio signal *y* can be a vector or a two-column array, representing mono or stereo audio, respectively.

If *fs* is not given, a default sample rate of 8000 samples per second is used.

The optional argument *nbits* specifies the bit depth to play to the audio device and defaults to 8 bits.

By default, *y* is automatically normalized to the range [-1, 1]. If the range [*ymin*, *ymax*] is given, then elements of *y* that fall within the range $ymin \le y \le ymax$ are scaled to the range [-1, 1] instead.

For more control over audio playback, use the `audioplayer` class.

See also: [sound], page 718, [record], page 718.

`y = wavread (filename)` [Function File]
`[y, fs, nbits] = wavread (filename)` [Function File]
`[...] = wavread (filename, n)` [Function File]
`[...] = wavread (filename, [n1 n2])` [Function File]
`[...] = wavread (..., datatype)` [Function File]
`sz = wavread (filename, "size")` [Function File]
`[n_samp, n_chan] = wavread (filename, "size")` [Function File]

Read the audio signal *y* from the RIFF/WAVE sound file *filename*.

If the file contains multichannel data, then *y* is a matrix with the channels represented as columns.

If *n* is specified, only the first *n* samples of the file are returned. If [*n1 n2*] is specified, only the range of samples from *n1* to *n2* is returned. A value of `Inf` can be used to represent the total number of samples in the file.

If the option `"size"` is given, then the size of the audio signal is returned instead of the data. The size is returned in a row vector of the form [*samples channels*]. If there are two output arguments, the number of samples is assigned to the first and the number of channels is assigned to the second.

The optional return value *fs* is the sample rate of the audio file in Hz. The optional return value *nbits* is the number of bits per sample as encoded in the file.

See also: [audioread], page 713, [audiowrite], page 713, [wavwrite], page 719.

`wavwrite (y, filename)` [Function File]
`wavwrite (y, fs, filename)` [Function File]
`wavwrite (y, fs, nbits, filename)` [Function File]

Write the audio signal *y* to the RIFF/WAVE sound file *filename*.

If *y* is a matrix, the columns represent multiple audio channels.

The optional argument *fs* specifies the sample rate of the audio signal in Hz.

The optional argument *nbits* specifies the number of bits per sample to write to *filename*.

The default sample rate is 8000 Hz and the default bit depth is 16 bits per sample.

See also: [audiowrite], page 713, [audioread], page 713, [wavread], page 719.

34 Object Oriented Programming

Octave includes the capability to include user classes, including the features of operator and function overloading. Equally a user class can be used to encapsulate certain properties of the class so that they cannot be altered accidentally and can be set up to address the issue of class precedence in mixed class operations.

This chapter discussions the means of constructing a user class with the example of a polynomial class, how to query and set the properties of this class, together with the means to overload operators and functions.

34.1 Creating a Class

We use in the following text a polynomial class to demonstrate the use of object oriented programming within Octave. This class was chosen as it is simple, and so doesn't distract unnecessarily from the discussion of the programming features of Octave. However, even still a small understand of the polynomial class itself is necessary to fully grasp the techniques described.

The polynomial class is used to represent polynomials of the form

$$a_0 + a_1 x + a_2 x^2 + \ldots a_n x^n$$

where a_0, a_1, etc. are elements of $\Re$. Thus the polynomial can be represented by a vector

```
a = [a0, a1, a2, ..., an];
```

We therefore now have sufficient information about the requirements of the class constructor for our polynomial class to write it. All object oriented classes in Octave, must be contained with a directory taking the name of the class, prepended with the @ symbol. For example, with our polynomial class, we would place the methods defining the class in the @polynomial directory.

The constructor of the class, must have the name of the class itself and so in our example the constructor with have the name `@polynomial/polynomial.m`. Also ideally when the constructor is called with no arguments to should return a value object. So for example our polynomial might look like

```
## -*- texinfo -*-
## @deftypefn  {Function File} {} polynomial ()
## @deftypefnx {Function File} {} polynomial (@var{a})
## Create a polynomial object representing the polynomial
##
## @example
## a0 + a1 * x + a2 * x^2 + @dots{} + an * x^n
## @end example
##
## @noindent
## from a vector of coefficients [a0 a1 a2 @dots{} an].
## @end deftypefn

function p = polynomial (a)
```

```
  if (nargin == 0)
    p.poly = [0];
    p = class (p, "polynomial");
  elseif (nargin == 1)
    if (strcmp (class (a), "polynomial"))
      p = a;
    elseif (isvector (a) && isreal (a))
      p.poly = a(:).';
      p = class (p, "polynomial");
    else
      error ("polynomial: expecting real vector");
    endif
  else
    print_usage ();
  endif
endfunction
```

Note that the return value of the constructor must be the output of the `class` function called with the first argument being a structure and the second argument being the class name. An example of the call to this constructor function is then

```
p = polynomial ([1, 0, 1]);
```

Note that methods of a class can be documented. The help for the constructor itself can be obtained with the constructor name, that is for the polynomial constructor `help polynomial` will return the help string. Also the help can be obtained by restricting the search for the help to a particular class, for example `help @polynomial/polynomial`. This second method is the only means of getting help for the overloaded methods and functions of the class.

The same is true for other Octave functions that take a function name as an argument. For example `type @polynomial/display` will print the code of the display method of the polynomial class to the screen, and `dbstop @polynomial/display` will set a breakpoint at the first executable line of the display method of the polynomial class.

To check where a variable is a user class, the `isobject` and `isa` functions can be used. For example:

```
p = polynomial ([1, 0, 1]);
isobject (p)
  ⇒ 1
isa (p, "polynomial")
  ⇒ 1
```

`isobject` (*x*) [Built-in Function]

Return true if *x* is a class object.

See also: [class], page 39, [typeinfo], page 39, [isa], page 39, [ismethod], page 723, [isprop], page 356.

The available methods of a class can be displayed with the `methods` function.

`methods (obj)` [Function File]
`methods ("classname")` [Function File]
`mtds = methods (...)` [Function File]

Return a cell array containing the names of the methods for the object *obj* or the named class *classname*.

obj may be an Octave class object or a Java object.

See also: [fieldnames], page 107.

To inquire whether a particular method is available to a user class, the `ismethod` function can be used.

`ismethod (obj, method)` [Built-in Function]

Return true if *obj* is a class object and the string *method* is a method of this class.

See also: [isprop], page 356, [isobject], page 722.

For example:

```
p = polynomial ([1, 0, 1]);
ismethod (p, "roots")
  ⇒ 1
```

34.2 Manipulating Classes

There are a number of basic classes methods that can be defined to allow the contents of the classes to be queried and set. The most basic of these is the `display` method. The `display` method is used by Octave when displaying a class on the screen, due to an expression that is not terminated with a semicolon. If this method is not defined, then Octave will printed nothing when displaying the contents of a class.

`display (a)` [Function File]

Display the contents of an object.

If *a* is an object of the class `"myclass"`, then `display` is called in a case like

```
myclass (...)
```

where Octave is required to display the contents of a variable of the type `"myclass"`.

See also: [class], page 39, [subsref], page 726, [subsasgn], page 728.

An example of a display method for the polynomial class might be

```
function display (p)
  a = p.poly;
  first = true;
  fprintf ("%s =", inputname (1));
  for i = 1 : length (a);
    if (a(i) != 0)
      if (first)
        first = false;
      elseif (a(i) > 0)
        fprintf (" +");
```

```
        endif
        if (a(i) < 0)
          fprintf (" -");
        endif
        if (i == 1)
          fprintf (" %g", abs (a(i)));
        elseif (abs(a(i)) != 1)
          fprintf (" %g *", abs (a(i)));
        endif
        if (i > 1)
          fprintf (" X");
        endif
        if (i > 2)
          fprintf (" ^ %d", i - 1);
        endif
      endif
    endfor
    if (first)
      fprintf (" 0");
    endif
    fprintf ("\n");
  endfunction
```

Note that in the display method, it makes sense to start the method with the line `fprintf ("%s =", inputname (1))` to be consistent with the rest of Octave and print the variable name to be displayed when displaying the class.

To be consistent with the Octave graphic handle classes, a class should also define the `get` and `set` methods. The `get` method should accept one or two arguments, and given one argument of the appropriate class it should return a structure with all of the properties of the class. For example:

```
  function s = get (p, f)
    if (nargin == 1)
      s.poly = p.poly;
    elseif (nargin == 2)
      if (ischar (f))
        switch (f)
          case "poly"
            s = p.poly;
          otherwise
            error ("get: invalid property %s", f);
        endswitch
      else
        error ("get: expecting the property to be a string");
      endif
    else
      print_usage ();
```

```
  endif
endfunction
```

Similarly, the `set` method should taken as its first argument an object to modify, and then take property/value pairs to be modified.

```
function s = set (p, varargin)
  s = p;
  if (length (varargin) < 2 || rem (length (varargin), 2) != 0)
    error ("set: expecting property/value pairs");
  endif
  while (length (varargin) > 1)
    prop = varargin{1};
    val = varargin{2};
    varargin(1:2) = [];
    if (ischar (prop) && strcmp (prop, "poly"))
      if (isvector (val) && isreal (val))
        s.poly = val(:).';
      else
        error ("set: expecting the value to be a real vector");
      endif
    else
      error ("set: invalid property of polynomial class");
    endif
  endwhile
endfunction
```

Note that as Octave does not implement pass by reference, than the modified object is the return value of the `set` method and it must be called like

```
p = set (p, "a", [1, 0, 0, 0, 1]);
```

Also the `set` method makes use of the `subsasgn` method of the class, and this method must be defined. The `subsasgn` method is discussed in the next section.

Finally, user classes can be considered as a special type of a structure, and so they can be saved to a file in the same manner as a structure. For example:

```
p = polynomial ([1, 0, 1]);
save userclass.mat p
clear p
load userclass.mat
```

All of the file formats supported by `save` and `load` are supported. In certain circumstances, a user class might either contain a field that it makes no sense to save or a field that needs to be initialized before it is saved. This can be done with the `saveobj` method of the class

b = `saveobj` (*a*) [Function File]

Method of a class to manipulate an object prior to saving it to a file.

The function `saveobj` is called when the object *a* is saved using the `save` function. An example of the use of `saveobj` might be to remove fields of the object that don't make sense to be saved or it might be used to ensure that certain fields of the object are initialized before the object is saved. For example:

```
function b = saveobj (a)
  b = a;
  if (isempty (b.field))
     b.field = initfield (b);
  endif
endfunction
```

See also: [loadobj], page 726, [class], page 39.

`saveobj` is called just prior to saving the class to a file. Likely, the `loadobj` method is called just after a class is loaded from a file, and can be used to ensure that any removed fields are reinserted into the user object.

b = `loadobj` (*a*) [Function File]

Method of a class to manipulate an object after loading it from a file.

The function `loadobj` is called when the object *a* is loaded using the `load` function. An example of the use of `saveobj` might be to add fields to an object that don't make sense to be saved. For example:

```
function b = loadobj (a)
  b = a;
  b.addmissingfield = addfield (b);
endfunction
```

See also: [saveobj], page 725, [class], page 39.

34.3 Indexing Objects

34.3.1 Defining Indexing And Indexed Assignment

Objects can be indexed with parentheses, either like `a (`*idx*`)` or like `a {`*idx*`}`, or even like `a (`*idx*`).`*field*. However, it is up to the user to decide what this indexing actually means. In the case of our polynomial class *p* `(`*n*`)` might mean either the coefficient of the *n*-th power of the polynomial, or it might be the evaluation of the polynomial at *n*. The meaning of this subscripted referencing is determined by the `subsref` method.

`subsref` (*val*, *idx*) [Built-in Function]

Perform the subscripted element selection operation according to the subscript specified by *idx*.

The subscript *idx* is expected to be a structure array with fields '`type`' and '`subs`'. Valid values for '`type`' are '`"()"`', '`"{}"`', and '`"."`'. The '`subs`' field may be either '`":"`' or a cell array of index values.

The following example shows how to extract the first two columns of a matrix

```
val = magic (3)
    ⇒ val = [ 8   1   6
              3   5   7
              4   9   2 ]
idx.type = "()";
idx.subs = {":", 1:2};
subsref (val, idx)
     ⇒ [ 8   1
         3   5
         4   9 ]
```

Note that this is the same as writing `val(:,1:2)`.

If *idx* is an empty structure array with fields '`type`' and '`subs`', return *val*.

See also: [subsasgn], page 728, [substruct], page 111.

For example we might decide that indexing with `"()"` evaluates the polynomial and indexing with `"{}"` returns the n-th coefficient (of n-th power). In this case the `subsref` method of our polynomial class might look like

```
function b = subsref (a, s)
  if (isempty (s))
    error ("polynomial: missing index");
  endif
  switch (s(1).type)
    case "()"
      ind = s(1).subs;
      if (numel (ind) != 1)
        error ("polynomial: need exactly one index");
      else
        b = polyval (fliplr (a.poly), ind{1});
      endif
    case "{}"
      ind = s(1).subs;
      if (numel (ind) != 1)
        error ("polynomial: need exactly one index");
      else
        if (isnumeric (ind{1}))
          b = a.poly(ind{1}+1);
        else
          b = a.poly(ind{1});
        endif
      endif
    case "."
      fld = s.subs;
      if (strcmp (fld, "poly"))
        b = a.poly;
      else
        error ("@polynomial/subsref: invalid property \"%s\"", fld);
```

```
      endif
    otherwise
      error ("invalid subscript type");
  endswitch
  if (numel (s) > 1)
    b = subsref (b, s(2:end));
  endif
endfunction
```

The equivalent functionality for subscripted assignments uses the `subsasgn` method.

`subsasgn (`*`val`*`, `*`idx`*`, `*`rhs`*`)` [Built-in Function]

Perform the subscripted assignment operation according to the subscript specified by *idx*.

The subscript *idx* is expected to be a structure array with fields '`type`' and '`subs`'. Valid values for '`type`' are '`"()"`', '`"{}"`', and '`"."`'. The '`subs`' field may be either '`":"`' or a cell array of index values.

The following example shows how to set the two first columns of a 3-by-3 matrix to zero.

```
val = magic (3);
idx.type = "()";
idx.subs = {":", 1:2};
subsasgn (val, idx, 0)
     ⇒  [ 0   0   6
          0   0   7
          0   0   2 ]
```

Note that this is the same as writing `val(:,1:2) = 0`.

If *idx* is an empty structure array with fields '`type`' and '`subs`', return *rhs*.

See also: [subsref], page 726, [substruct], page 111.

`val` `= optimize_subsasgn_calls ()` [Built-in Function]
`old_val` `= optimize_subsasgn_calls (`*`new_val`*`)` [Built-in Function]
`optimize_subsasgn_calls (`*`new_val`*`, "`*local*`")` [Built-in Function]

Query or set the internal flag for subsasgn method call optimizations.

If true, Octave will attempt to eliminate the redundant copying when calling the subsasgn method of a user-defined class.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

Note that the `subsref` and `subsasgn` methods always receive the whole index chain, while they usually handle only the first element. It is the responsibility of these methods to handle the rest of the chain (if needed), usually by forwarding it again to `subsref` or `subsasgn`.

If you wish to use the end keyword in subscripted expressions of an object, then the user needs to define the end method for the class. For example, the end method for our polynomial class might look like

```
function r = end (obj, index_pos, num_indices)

  if (num_indices != 1)
    error ("polynomial object may only have one index")
  endif

  r = length (obj.poly) - 1;

endfunction
```

which is a fairly generic end method that has a behavior similar to the end keyword for Octave Array classes. It can then be used as follows:

```
p = polynomial ([1,2,3,4]);
p(end-1)
  ⇒ 3
```

Objects can also be used as the index in a subscripted expression themselves and this is controlled with the subsindex function.

idx = subsindex (*a*) [Function File]

Convert an object to an index vector.

When *a* is a class object defined with a class constructor, then subsindex is the overloading method that allows the conversion of this class object to a valid indexing vector. It is important to note that subsindex must return a zero-based real integer vector of the class "double". For example, if the class constructor

```
function b = myclass (a)
  b = class (struct ("a", a), "myclass");
endfunction
```

then the subsindex function

```
function idx = subsindex (a)
  idx = double (a.a) - 1.0;
endfunction
```

can then be used as follows

```
a = myclass (1:4);
b = 1:10;
b(a)
⇒ 1  2  3  4
```

See also: [class], page 39, [subsref], page 726, [subsasgn], page 728.

Finally, objects can equally be used like ranges, using the colon method

r = colon (*base*, *limit*) [Built-in Function]
r = colon (*base*, *increment*, *limit*) [Built-in Function]

Return the result of the colon expression corresponding to *base*, *limit*, and optionally, *increment*.

This function is equivalent to the operator syntax `base : limit` or `base : increment : limit`.

34.3.2 Indexed Assignment Optimization

Octave's ubiquitous lazily-copied pass-by-value semantics implies a problem for performance of user-defined subsasgn methods. Imagine a call to subsasgn:

```
ss = substruct ("()",{1});
x = subsasgn (x, ss, 1);
```

and the corresponding method looking like this:

```
function x = subsasgn (x, ss, val)
  ...
  x.myfield (ss.subs{1}) = val;
endfunction
```

The problem is that on entry to the subsasgn method, `x` is still referenced from the caller's scope, which means that the method will first need to unshare (copy) `x` and `x.myfield` before performing the assignment. Upon completing the call, unless an error occurs, the result is immediately assigned to `x` in the caller's scope, so that the previous value of `x.myfield` is forgotten. Hence, the Octave language implies a copy of N elements (N being the size of `x.myfield`), where modifying just a single element would actually suffice, i.e., degrades a constant-time operation to linear-time one. This may be a real problem for user classes that intrinsically store large arrays.

To partially solve the problem, Octave uses a special optimization for user-defined subsasgn methods coded as m-files. When the method gets called as a result of the built-in assignment syntax (not direct subsasgn call as shown above), i.e.

```
x(1) = 1;
```

AND if the subsasgn method is declared with identical input and output argument, like in the example above, then Octave will ignore the copy of `x` inside the caller's scope; therefore, any changes made to `x` during the method execution will directly affect the caller's copy as well. This allows, for instance, defining a polynomial class where modifying a single element takes constant time.

It is important to understand the implications that this optimization brings. Since no extra copy of `x` in the caller's scope will exist, it is *solely* the callee's responsibility to not leave `x` in an invalid state if an error occurs throughout the execution. Also, if the method partially changes `x` and then errors out, the changes *will* affect `x` in the caller's scope. Deleting or completely replacing `x` inside subsasgn will not do anything, however, only indexed assignments matter.

Since this optimization may change the way code works (especially if badly written), a built-in variable `optimize_subsasgn_calls` is provided to control it. It is on by default. Another option to avoid the effect is to declare subsasgn methods with different output and input arguments, like this:

```
  function y = subsasgn (x, ss, val)
    ...
  endfunction
```

34.4 Overloading Objects

34.4.1 Function Overloading

Any Octave function can be overloaded, and allows an object specific version of this function to be called as needed. A pertinent example for our polynomial class might be to overload the `polyval` function like

```
function [y, dy] = polyval (p, varargin)
  if (nargout == 2)
    [y, dy] = polyval (fliplr (p.poly), varargin{:});
  else
    y = polyval (fliplr (p.poly), varargin{:});
  endif
endfunction
```

This function just hands off the work to the normal Octave `polyval` function. Another interesting example for an overloaded function for our polynomial class is the `plot` function.

```
function h = plot (p, varargin)
  n = 128;
  rmax = max (abs (roots (p.poly)));
  x = [0 : (n - 1)] / (n - 1) * 2.2 * rmax - 1.1 * rmax;
  if (nargout > 0)
    h = plot (x, p(x), varargin{:});
  else
    plot (x, p(x), varargin{:});
  endif
endfunction
```

which allows polynomials to be plotted in the domain near the region of the roots of the polynomial.

Functions that are of particular interest to be overloaded are the class conversion functions such as `double`. Overloading these functions allows the `cast` function to work with the user class and can aid in the use of methods of other classes with the user class. An example `double` function for our polynomial class might look like.

```
function b = double (a)
  b = a.poly;
endfunction
```

34.4.2 Operator Overloading

The following table shows, for each built-in numerical operation, the corresponding function name to use when providing an overloaded method for a user class.

Operation	Method	Description
$a + b$	plus (a, b)	Binary addition operator
$a - b$	minus (a, b)	Binary subtraction operator
$+a$	uplus (a)	Unary addition operator
$-a$	uminus (a)	Unary subtraction operator
$a. * b$	times (a, b)	Element-wise multiplication operator
$a * b$	mtimes (a, b)	Matrix multiplication operator
$a./b$	rdivide (a, b)	Element-wise right division operator
a/b	mrdivide (a, b)	Matrix right division operator
$a.\backslash b$	ldivide (a, b)	Element-wise left division operator
$a\backslash b$	mldivide (a, b)	Matrix left division operator
$a.\hat{b}$	power (a, b)	Element-wise power operator
$a\hat{b}$	mpower (a, b)	Matrix power operator
$a < b$	lt (a, b)	Less than operator
$a <= b$	le (a, b)	Less than or equal to operator
$a > b$	gt (a, b)	Greater than operator
$a >= b$	ge (a, b)	Greater than or equal to operator
$a == b$	eq (a, b)	Equal to operator
$a! = b$	ne (a, b)	Not equal to operator
$a\&b$	and (a, b)	Logical and operator
$a\|b$	or (a, b)	Logical or operator
$!b$	not (a)	Logical not operator
a'	ctranspose (a)	Complex conjugate transpose operator
$a.'$	transpose (a)	Transpose operator
$a : b$	colon (a, b)	Two element range operator
$a : b : c$	colon (a, b, c)	Three element range operator
$[a, b]$	horzcat (a, b)	Horizontal concatenation operator
$[a; b]$	vertcat (a, b)	Vertical concatenation operator
$a(s_1, \ldots, s_n)$	subsref (a, s)	Subscripted reference
$a(s_1, \ldots, s_n) = b$	subsasgn (a, s, b)	Subscripted assignment
$b(a)$	subsindex (a)	Convert to zero-based index
display	display (a)	Commandline display function

Table 34.1: Available overloaded operators and their corresponding class method

An example `mtimes` method for our polynomial class might look like

```
function y = mtimes (a, b)
  y = polynomial (conv (double (a), double (b)));
endfunction
```

34.4.3 Precedence of Objects

Many functions and operators take two or more arguments and so the case can easily arise that these functions are called with objects of different classes. It is therefore necessary to determine the precedence of which method of which class to call when there are mixed objects given to a function or operator. To do this the `superiorto` and `inferiorto` functions can be used

`superiorto (`*`class_name`*`, ...)` [Built-in Function]

When called from a class constructor, mark the object currently constructed as having a higher precedence than *class_name*.

More that one such class can be specified in a single call. This function may only be called from a class constructor.

See also: [inferiorto], page 733.

`inferiorto (`*`class_name`*`, ...)` [Built-in Function]

When called from a class constructor, mark the object currently constructed as having a lower precedence than *class_name*.

More that one such class can be specified in a single call. This function may only be called from a class constructor.

See also: [superiorto], page 733.

For example with our polynomial class consider the case

```
2 * polynomial ([1, 0, 1]);
```

That mixes an object of the class `"double"` with an object of the class `"polynomial"`. In this case we like to ensure that the return type of the above is of the type `"polynomial"` and so we use the `superiorto` function in the class constructor. In particular our polynomial class constructor would be modified to be

```
## -*- texinfo -*-
## @deftypefn  {Function File} {} polynomial ()
## @deftypefnx {Function File} {} polynomial (@var{a})
## Create a polynomial object representing the polynomial
##
## @example
## a0 + a1 * x + a2 * x^2 + @dots{} + an * x^n
## @end example
##
## @noindent
## from a vector of coefficients [a0 a1 a2 @dots{} an].
## @end deftypefn

function p = polynomial (a)
  if (nargin == 0)
    p.poly = [0];
    p = class (p, "polynomial");
  elseif (nargin == 1)
    if (strcmp (class (a), "polynomial"))
      p = a;
    elseif (isvector (a) && isreal (a))
      p.poly = a(:).';
```

```
      p = class (p, "polynomial");
    else
      error ("polynomial: expecting real vector");
    endif
  else
    print_usage ();
  endif
  superiorto ("double");
endfunction
```

Note that user classes always have higher precedence than built-in Octave types. So in fact marking our polynomial class higher than the `"double"` class is in fact not necessary.

When faced with two objects that have the same precedence, Octave will use the method of the object that appears first on the list of arguments.

34.5 Inheritance and Aggregation

Using classes to build new classes is supported by octave through the use of both inheritance and aggregation.

Class inheritance is provided by octave using the `class` function in the class constructor. As in the case of the polynomial class, the octave programmer will create a struct that contains the data fields required by the class, and then call the class function to indicate that an object is to be created from the struct. Creating a child of an existing object is done by creating an object of the parent class and providing that object as the third argument of the class function.

This is easily demonstrated by example. Suppose the programmer needs an FIR filter, i.e., a filter with a numerator polynomial but a unity denominator polynomial. In traditional octave programming, this would be performed as follows.

```
octave:1> x = [some data vector];
octave:2> n = [some coefficient vector];
octave:3> y = filter (n, 1, x);
```

The equivalent class could be implemented in a class directory @FIRfilter that is on the octave path. The constructor is a file FIRfilter.m in the class directory.

```
## -*- texinfo -*-
## @deftypefn  {Function File} {} FIRfilter ()
## @deftypefnx {Function File} {} FIRfilter (@var{p})
## Create a FIR filter with polynomial @var{p} as coefficient vector.
## @end deftypefn

function f = FIRfilter (p)

  f.polynomial = [];
  if (nargin == 0)
    p = @polynomial ([1]);
  elseif (nargin == 1)
    if (!isa (p, "polynomial"))
```

```
      error ("FIRfilter: expecting polynomial as input argument");
    endif
  else
    print_usage ();
  endif
  f = class (f, "FIRfilter", p);
endfunction
```

As before, the leading comments provide command-line documentation for the class constructor. This constructor is very similar to the polynomial class constructor, except that we pass a polynomial object as the third argument to the class function, telling octave that the FIRfilter class will be derived from the polynomial class. Our FIR filter does not have any data fields, but we must provide a struct to the `class` function. The `class` function will add an element named polynomial to the object struct, so we simply add a dummy element named polynomial as the first line of the constructor. This dummy element will be overwritten by the class function.

Note further that all our examples provide for the case in which no arguments are supplied. This is important since octave will call the constructor with no arguments when loading objects from save files to determine the inheritance structure.

A class may be a child of more than one class (see the documentation for the `class` function), and inheritance may be nested. There is no limitation to the number of parents or the level of nesting other than memory or other physical issues.

As before, we need a `display` method. A simple example might be

```
function display (f)

  display (f.polynomial);

endfunction
```

Note that we have used the polynomial field of the struct to display the filter coefficients.

Once we have the class constructor and display method, we may create an object by calling the class constructor. We may also check the class type and examine the underlying structure.

```
octave:1> f = FIRfilter (polynomial ([1 1 1]/3))
f.polynomial = 0.333333 + 0.333333 * X + 0.333333 * X ^ 2
octave:2> class (f)
ans = FIRfilter
octave:3> isa (f,"FIRfilter")
ans =  1
octave:4> isa (f,"polynomial")
ans =  1
octave:5> struct (f)
ans =
{
polynomial = 0.333333 + 0.333333 * X + 0.333333 * X ^ 2
}
```

We only need to define a method to actually process data with our filter and our class is usable. It is also useful to provide a means of changing the data stored in the class. Since the fields in the underlying struct are private by default, we could provide a mechanism to access the fields. The `subsref` method may be used for both.

```
function out = subsref (f, x)
  switch (x.type)
    case "()"
      n = f.polynomial;
      out = filter (n.poly, 1, x.subs{1});
    case "."
      fld = x.subs;
      if (strcmp (fld, "polynomial"))
        out = f.polynomial;
      else
        error ("@FIRfilter/subsref: invalid property \"%s\"", fld);
      endif
    otherwise
      error ("@FIRfilter/subsref: invalid subscript type for FIR filter");
  endswitch
endfunction
```

The `"()"` case allows us to filter data using the polynomial provided to the constructor.

```
octave:2> f = FIRfilter (polynomial ([1 1 1]/3));
octave:3> x = ones (5,1);
octave:4> y = f(x)
y =

   0.33333
   0.66667
   1.00000
   1.00000
   1.00000
```

The `"."` case allows us to view the contents of the polynomial field.

```
octave:1> f = FIRfilter (polynomial ([1 1 1]/3));
octave:2> f.polynomial
ans = 0.333333 + 0.333333 * X + 0.333333 * X ^ 2
```

In order to change the contents of the object, we need to define a `subsasgn` method. For example, we may make the polynomial field publicly writable.

```
function out = subsasgn (f, index, val)
  switch (index.type)
    case "."
      fld = index.subs;
      if (strcmp (fld, "polynomial"))
        out = f;
        out.polynomial = val;
      else
        error ("@FIRfilter/subsref: invalid property \"%s\"", fld);
      endif
    otherwise
      error ("FIRfilter/subsagn: Invalid index type")
  endswitch
endfunction
```

So that

```
octave:6> f = FIRfilter ();
octave:7> f.polynomial = polynomial ([1 2 3]);
f.polynomial = 1 + 2 * X + 3 * X ^ 2
```

Defining the FIRfilter class as a child of the polynomial class implies that and FIRfilter object may be used any place that a polynomial may be used. This is not a normal use of a filter, so that aggregation may be a more sensible design approach. In this case, the polynomial is simply a field in the class structure. A class constructor for this case might be

```
## -*- texinfo -*-
## @deftypefn  {Function File} {} FIRfilter ()
## @deftypefnx {Function File} {} FIRfilter (@var{p})
## Create a FIR filter with polynomial @var{p} as coefficient vector.
## @end deftypefn

function f = FIRfilter (p)

  if (nargin == 0)
    f.polynomial = @polynomial ([1]);
  elseif (nargin == 1)
    if (isa (p, "polynomial"))
      f.polynomial = p;
    else
      error ("FIRfilter: expecting polynomial as input argument");
    endif
```

```
    else
      print_usage ();
    endif
    f = class (f, "FIRfilter");
  endfunction
```

For our example, the remaining class methods remain unchanged.

35 GUI Development

Octave is principally a batch or command-line language. However, it does offer some features for constructing graphical interfaces that interact with users.

The GUI elements available are I/O dialogs, a progress bar, and UI elements for plot windows. For example, rather than hardcoding a filename for output results a script can open a dialog box and allow the user to choose a file. Similarly, if a calculation is expected to take a long time a script can display a progress bar. The various UI elements can be used to fully customize the plot window with menubars, context menus,

Several utility functions make it possible to store private data for use with a GUI which will not pollute the user's variable space.

Finally, a program written in Octave might want to have long term storage of preferences or state variables. This can be done with user-defined preferences.

35.1 I/O Dialogs

Simple dialog menus are available for choosing directories or files. They return a string variable which can then be used with any command requiring a file name.

dirname = `uigetdir` () [Function File]
dirname = `uigetdir` (*init_path*) [Function File]
dirname = `uigetdir` (*init_path*, *dialog_name*) [Function File]

Open a GUI dialog for selecting a directory.

If *init_path* is not given the current working directory is used.

dialog_name may be used to customize the dialog title.

See also: [uigetfile], page 739, [uiputfile], page 740.

[*fname*, *fpath*, *fltidx*] = `uigetfile` () [Function File]
[...] = `uigetfile` (*flt*) [Function File]
[...] = `uigetfile` (*flt*, *dialog_name*) [Function File]
[...] = `uigetfile` (*flt*, *dialog_name*, *default_file*) [Function File]
[...] = `uigetfile` (..., "*Position*", [*px py*]) [Function File]
[...] = `uigetfile` (..., "*MultiSelect*", *mode*) [Function File]

Open a GUI dialog for selecting a file and return the filename *fname*, the path to this file *fpath*, and the filter index *fltidx*.

flt contains a (list of) file filter string(s) in one of the following formats:

`"/path/to/filename.ext"`
: If a filename is given then the file extension is extracted and used as filter. In addition, the path is selected as current path and the filename is selected as default file. Example: `uigetfile ("myfun.m")`

A single file extension `"*.ext"`
: Example: `uigetfile ("*.ext")`

A 2-column cell array
: containing a file extension in the first column and a brief description in the second column. Example: `uigetfile ({"*.ext", "My Description";"*.xyz", "XYZ-Format"})`

The filter string can also contain a semicolon separated list of filter extensions. Example: `uigetfile ({"*.gif;*.png;*.jpg", "Supported Picture Formats"})`

dialog_name can be used to customize the dialog title.

If *default_file* is given then it will be selected in the GUI dialog. If, in addition, a path is given it is also used as current path.

The screen position of the GUI dialog can be set using the `"Position"` key and a 2-element vector containing the pixel coordinates. Two or more files can be selected when setting the `"MultiSelect"` key to `"on"`. In that case *fname* is a cell array containing the files.

See also: [uiputfile], page 740, [uigetdir], page 739.

`[fname, fpath, fltidx] = uiputfile ()` [Function File]
`[fname, fpath, fltidx] = uiputfile (flt)` [Function File]
`[fname, fpath, fltidx] = uiputfile (flt, dialog_name)` [Function File]
`[fname, fpath, fltidx] = uiputfile (flt, dialog_name, default_file)` [Function File]

Open a GUI dialog for selecting a file.

flt contains a (list of) file filter string(s) in one of the following formats:

`"/path/to/filename.ext"`
: If a filename is given the file extension is extracted and used as filter. In addition the path is selected as current path and the filename is selected as default file. Example: `uiputfile ("myfun.m")`

`"*.ext"`
: A single file extension. Example: `uiputfile ("*.ext")`

`{"*.ext", "My Description"}`
: A 2-column cell array containing the file extension in the 1st column and a brief description in the 2nd column. Example: `uiputfile ({"*.ext","My Description";"*.xyz", "XYZ-Format"})`

The filter string can also contain a semicolon separated list of filter extensions. Example: `uiputfile ({"*.gif;*.png;*.jpg", "Supported Picture Formats"})`

dialog_name can be used to customize the dialog title. If *default_file* is given it is preselected in the GUI dialog. If, in addition, a path is given it is also used as current path.

See also: [uigetfile], page 739, [uigetdir], page 739.

35.2 Progress Bar

`h = waitbar (frac)` [Function File]
`h = waitbar (frac, msg)` [Function File]
`h = waitbar (..., "FigureProperty", "Value", ...)` [Function File]
`waitbar (frac)` [Function File]
`waitbar (frac, hwbar)` [Function File]
`waitbar (frac, hwbar, msg)` [Function File]

Return a handle *h* to a new waitbar object.

The waitbar is filled to fraction *frac* which must be in the range [0, 1].

The optional message *msg* is centered and displayed above the waitbar.

The appearance of the waitbar figure window can be configured by passing property/value pairs to the function.

When called with a single input the current waitbar, if it exists, is updated to the new value *frac*. If there are multiple outstanding waitbars they can be updated individually by passing the handle *hwbar* of the specific waitbar to modify.

35.3 UI Elements

The ui* series of functions work best with the `qt` graphics toolkit, although some functionality is available with the `fltk` toolkit. There is no support for the `gnuplot` toolkit.

hui = `uicontextmenu` (*property*, *value*, ...) [Function File]
hui = `uicontextmenu` (*h*, *property*, *value*, ...) [Function File]

Create a uicontextmenu object and return a handle to it.

If *h* is omitted then a uicontextmenu for the current figure is created. If no figure is available, a new figure is created first.

If *h* is given then a uicontextmenu relative to *h* is created.

Any provided property value pairs will override the default values of the created uicontextmenu object.

Examples:

```
% create figure and uicontextmenu
f = figure;
c = uicontextmenu (f);

% create menus in the context menu
m1 = uimenu ("parent",c,"label","Menu item 1","callback","disp('menu item 1')");
m2 = uimenu ("parent",c,"label","Menu item 2","callback","disp('menu item 2')");

% set the context menu for the figure
set (f, "uicontextmenu", c);
```

See also: [figure], page 336, [uimenu], page 352.

hui = `uicontrol` (*property*, *value*, ...) [Function File]
hui = `uicontrol` (*parent*, *property*, *value*, ...) [Function File]
`uicontrol` (*h*) [Function File]

Create a uicontrol object and return a handle to it.

uicontrols are used to create simple interactive controls such as push buttons, checkboxes, edit and list controls.

If *parent* is omitted then a uicontrol for the current figure is created. If no figure is available, a new figure is created first.

If *parent* is given then a uicontrol relative to *parent* is created.

Any provided property value pairs will override the default values of the created uicontrol object.

Control of the type of uicontrol created is through the use of the *style* property. If no style property is provided, a push button will be created.

Valid styles for uicontrol are:

`"checkbox"`
: Create a checkbox control that allows user on/off selection.

`"edit"`
: Create a edit control that allows user input of single or multiple lines of text.

`"listbox"`
: Create a listbox control that displays a lit of items and allows user slelection of single or multiple items.

`"popupmenu"`
: Create a popupmenu control that displays a list of options that can be selected when the user clicks on the control.

`"pushbutton"`
: Create a push button control that allows user to press to cause an action.

`"radiobutton"`
: Create a radio button control intended to be used for mutually exclusive input in a group of of radiobutton controls.

`"slider"`
: Create a slider control that allows user selection from a range of values by sliding knob on the control.

`"text"`
: Create a static text control to display single or multiple lines of text.

`"togglebutton"`
: Create a toggle button control that appears like a push button but allows the user to select between two states.

Examples:

```
% create figure and panel on it
f = figure;
% create a button (default style)
b1 = uicontrol (f, "string", "A Button", "position",[10 10 150 40]);
% create an edit control
e1 = uicontrol (f, "style", "edit", "string", "editable text", "position",[10 60 300 40]);
% create a checkbox
c1 = uicontrol (f, "style", "checkbox", "string", "a checkbox", "position",[10 120 150 40]);
```

See also: [figure], page 336, [uipanel], page 742.

hui = `uipanel` (*property*, *value*, ...) [Function File]
hui = `uipanel` (*parent*, "*property*, *value*, ...) [Function File]

Create a uipanel object and return a handle to it.

uipanels are used as containers to group other uicontrol objects.

If *parent* is omitted then a uipanel for the current figure is created. If no figure is available, a new figure is created first.

If *parent* is given then a uipanel relative to *parent* is created.

Any provided property value pairs will override the default values of the created uipanel object.

Examples:

```
% create figure and panel on it
f = figure;
p = uipanel ("title", "Panel Title", "position", [.25 .25 .5 .5]);

% add two buttons to the panel
b1 = uicontrol ("parent", p, "string", "A Button", "position",[18 10 150 36]);
b2 = uicontrol ("parent", p, "string", "Another Button", "position",[18 60 150 36]);
```

See also: [figure], page 336, [uicontrol], page 741.

hui = **uipushtool** (*property*, *value*, . . .) [Function File]
hui = **uipushtool** (*parent*, *property*, *value*, . . .) [Function File]

Create a uipushtool object and return a handle to it.

uipushtools are buttons that appear on a figure toolbar. The button is created with a border that is shown when the user hovers over the button. An image can be set using the cdata property.

If *parent* is omitted then a uipushtool for the current figure is created. If no figure is available, a new figure is created first. If a figure is available, but does not contain a uitoolbar, a uitoolbar will be created.

If *parent* is given then a uipushtools is created on the *parent* uitoolbar.

Any provided property value pairs will override the default values of the created uipushtool object.

Examples:

```
% create figure without a default toolbar
f = figure ("toolbar", "none");
% create empty toolbar
t = uitoolbar (f);
% create a 19x19x3 black square
img=zeros(19,19,3);
% add pushtool button to toolbar
b = uipushtool (t, "cdata", img);
```

See also: [figure], page 336, [uitoolbar], page 744, [uitoggletool], page 743.

hui = **uitoggletool** (*property*, *value*, . . .) [Function File]
hui = **uitoggletool** (*parent*, *property*, *value*, . . .) [Function File]

Create a uitoggletool object and return a handle to it.

uitoggletool are togglebuttons that appear on a figure toolbar. The button is created with a border that is shown when the user hovers over the button. An image can be set using the cdata property.

If *parent* is omitted then a uitoggletool for the current figure is created. If no figure is available, a new figure is created first. If a figure is available, but does not contain a uitoolbar, a uitoolbar will be created.

If *parent* is given then a uitoggletool is created on the *parent* uitoolbar.

Any provided property value pairs will override the default values of the created uitoggletool object.

Examples:

```
% create figure without a default toolbar
f = figure ("toolbar", "none");
% create empty toolbar
t = uitoolbar (f);
% create a 19x19x3 black square
img=zeros(19,19,3);
% add uitoggletool button to toolbar
b = uitoggletool (t, "cdata", img);
```

See also: [figure], page 336, [uitoolbar], page 744, [uipushtool], page 743.

hui = uitoolbar (*property*, *value*, ...) [Function File]
hui = uitoolbar (*parent*, *property*, *value*, ...) [Function File]

Create a uitoolbar object and return a handle to it. A uitoolbar displays uitoggletool and uipushtool buttons.

If *parent* is omitted then a uitoolbar for the current figure is created. If no figure is available, a new figure is created first.

If *parent* is given then a uitoolbar relative to *parent* is created.

Any provided property value pairs will override the default values of the created uitoolbar object.

Examples:

```
% create figure without a default toolbar
f = figure ("toolbar", "none");
% create empty toolbar
t = uitoolbar (f);
```

See also: [figure], page 336, [uitoggletool], page 743, [uipushtool], page 743.

35.4 GUI Utility Functions

These functions do not implement a GUI element but are useful when developing programs that do. The functions `uiwait`, `uiresume`, and `waitfor` are only available with the `qt` or `fltk` toolkits.

used = desktop ("*-inuse*") [Function File]

Return true if the desktop (GUI) is currently in use.

See also: [isguirunning], page 745.

data = guidata (*h*) [Function File]
guidata (*h*, *data*) [Function File]

Query or set user-custom GUI data.

The GUI data is stored in the figure handle *h*. If *h* is not a figure handle then it's parent figure will be used for storage.

data must be a single object which means it is usually preferable for it to be a data container such as a cell array or struct so that additional data items can be added easily.

See also: [getappdata], page 389, [setappdata], page 389, [get], page 361, [set], page 361, [getpref], page 746, [setpref], page 747.

`hdata = guihandles (h)` [Function File]
`hdata = guihandles` [Function File]

Return a structure of object handles for the figure associated with handle *h*.

If no handle is specified the current figure returned by `gcf` is used.

The fieldname for each entry of *hdata* is taken from the `"tag"` property of the graphic object. If the tag is empty then the handle is not returned. If there are multiple graphic objects with the same tag then the entry in *hdata* will be a vector of handles. `guihandles` includes all possible handles, including those for which `"HandleVisibility"` is `"off"`.

See also: [guidata], page 744, [findobj], page 384, [findall], page 385, [allchild], page 363.

`have_window_system ()` [Built-in Function]

Return true if a window system is available (X11, Windows, or Apple OS X) and false otherwise.

See also: [isguirunning], page 745.

`isguirunning ()` [Built-in Function]

Return true if Octave is running in GUI mode and false otherwise.

See also: [have_window_system], page 745.

`uiwait` [Function File]
`uiwait (h)` [Function File]
`uiwait (h, timeout)` [Function File]

Suspend program execution until the figure with handle *h* is deleted or `uiresume` is called.

When no figure handle is specified this function uses the current figure. If the figure handle is invalid or there is no current figure, this functions returns immediately.

When specified, *timeout* defines the number of seconds to wait for the figure deletion or the `uiresume` call. The timeout value must be at least 1. If a smaller value is specified, a warning is issued and a timeout value of 1 is used instead. If a non-integer value is specified, it is truncated towards 0. If *timeout* is not specified, the program execution is suspended indefinitely.

See also: [uiresume], page 745, [waitfor], page 746.

`uiresume (h)` [Function File]

Resume program execution suspended with `uiwait`.

The handle *h* must be the same as the on specified in `uiwait`. If the handle is invalid or there is no `uiwait` call pending for the figure with handle *h*, this function does nothing.

See also: [uiwait], page 745.

`waitfor (h)` [Built-in Function]
`waitfor (h, prop)` [Built-in Function]
`waitfor (h, prop, value)` [Built-in Function]
`waitfor (..., "timeout", timeout)` [Built-in Function]

Suspend the execution of the current program until a condition is satisfied on the graphics handle *h*.

While the program is suspended graphics events are still processed normally, allowing callbacks to modify the state of graphics objects. This function is reentrant and can be called from a callback, while another `waitfor` call is pending at the top-level.

In the first form, program execution is suspended until the graphics object *h* is destroyed. If the graphics handle is invalid, the function returns immediately.

In the second form, execution is suspended until the graphics object is destroyed or the property named *prop* is modified. If the graphics handle is invalid or the property does not exist, the function returns immediately.

In the third form, execution is suspended until the graphics object is destroyed or the property named *prop* is set to *value*. The function `isequal` is used to compare property values. If the graphics handle is invalid, the property does not exist or the property is already set to *value*, the function returns immediately.

An optional timeout can be specified using the property `timeout`. This timeout value is the number of seconds to wait for the condition to be true. *timeout* must be at least 1. If a smaller value is specified, a warning is issued and a value of 1 is used instead. If the timeout value is not an integer, it is truncated towards 0.

To define a condition on a property named `timeout`, use the string `\timeout` instead.

In all cases, typing CTRL-C stops program execution immediately.

See also: [waitforbuttonpress], page 352, [isequal], page 146.

35.5 User-Defined Preferences

`getpref (group, pref)` [Function File]
`getpref (group, pref, default)` [Function File]
`getpref (group)` [Function File]

Return the preference value corresponding to the named preference *pref* in the preference group *group*.

The named preference group must be a character string.

If *pref* does not exist in *group* and *default* is specified, return *default*.

The preference *pref* may be a character string or a cell array of character strings.

The corresponding default value *default* may be any value, or, if *pref* is a cell array of strings, *default* must be a cell array of values with the same size as *pref*.

If neither *pref* nor *default* are specified, return a structure of preferences for the preference group *group*.

If no arguments are specified, return a structure containing all groups of preferences and their values.

See also: [addpref], page 747, [setpref], page 747, [ispref], page 747, [rmpref], page 747.

`setpref (group, pref, val)` [Function File]

Set a preference *pref* to the given *val* in the named preference group *group*.

The named preference group must be a character string.

The preference *pref* may be a character string or a cell array of character strings.

The corresponding value *val* may be any value, or, if *pref* is a cell array of strings, *val* must be a cell array of values with the same size as *pref*.

If the named preference or group does not exist, it is added.

See also: [addpref], page 747, [getpref], page 746, [ispref], page 747, [rmpref], page 747.

`addpref (group, pref, val)` [Function File]

Add a preference *pref* and associated value *val* to the named preference group *group*.

The named preference group must be a character string.

The preference *pref* may be a character string or a cell array of character strings.

The corresponding value *val* may be any value, or, if *pref* is a cell array of strings, *val* must be a cell array of values with the same size as *pref*.

See also: [setpref], page 747, [getpref], page 746, [ispref], page 747, [rmpref], page 747.

`rmpref (group, pref)` [Function File]
`rmpref (group)` [Function File]

Remove the named preference *pref* from the preference group *group*.

The named preference group must be a character string.

The preference *pref* may be a character string or cell array of strings.

If *pref* is not specified, remove the preference group *group*.

It is an error to remove a nonexistent preference or group.

See also: [addpref], page 747, [ispref], page 747, [setpref], page 747, [getpref], page 746.

`ispref (group, pref)` [Function File]
`ispref (group)` [Function File]

Return true if the named preference *pref* exists in the preference group *group*.

The named preference group must be a character string.

The preference *pref* may be a character string or a cell array of character strings.

If *pref* is not specified, return true if the preference group *group* exists.

See also: [getpref], page 746, [addpref], page 747, [setpref], page 747, [rmpref], page 747.

`prefdir` [Command]
dir = `prefdir` [Command]

Return the directory that contains the preferences for Octave.

Examples:

Display the preferences directory

```
prefdir
```

Change to the preferences folder

```
cd (prefdir)
```

See also: [getpref], page 746, [setpref], page 747, [addpref], page 747, [rmpref], page 747, [ispref], page 747.

`preferences` [Command]

Display the GUI preferences dialog window for Octave.

36 System Utilities

This chapter describes the functions that are available to allow you to get information about what is happening outside of Octave, while it is still running, and use this information in your program. For example, you can get information about environment variables, the current time, and even start other programs from the Octave prompt.

36.1 Timing Utilities

Octave's core set of functions for manipulating time values are patterned after the corresponding functions from the standard C library. Several of these functions use a data structure for time that includes the following elements:

`usec` Microseconds after the second (0-999999).

`sec` Seconds after the minute (0-60). This number can be 60 to account for leap seconds.

`min` Minutes after the hour (0-59).

`hour` Hours since midnight (0-23).

`mday` Day of the month (1-31).

`mon` Months since January (0-11).

`year` Years since 1900.

`wday` Days since Sunday (0-6).

`yday` Days since January 1 (0-365).

`isdst` Daylight Savings Time flag.

`zone` Time zone.

In the descriptions of the following functions, this structure is referred to as a *tm_struct*.

`*seconds* = time ()` [Built-in Function]

Return the current time as the number of seconds since the epoch.

The epoch is referenced to 00:00:00 CUT (Coordinated Universal Time) 1 Jan 1970. For example, on Monday February 17, 1997 at 07:15:06 CUT, the value returned by `time` was 856163706.

See also: [strftime], page 751, [strptime], page 753, [localtime], page 750, [gmtime], page 750, [mktime], page 751, [now], page 749, [date], page 754, [clock], page 753, [datenum], page 756, [datestr], page 757, [datevec], page 758, [calendar], page 759, [weekday], page 759.

`t = now ()` [Function File]

Return the current local date/time as a serial day number (see `datenum`).

The integral part, `floor (now)` corresponds to the number of days between today and Jan 1, 0000.

The fractional part, `rem (now, 1)` corresponds to the current time.

See also: [clock], page 753, [date], page 754, [datenum], page 756.

`ctime` (*t*) [Function File]

Convert a value returned from `time` (or any other non-negative integer), to the local time and return a string of the same form as `asctime`.

The function `ctime (time)` is equivalent to `asctime (localtime (time))`. For example:

```
ctime (time ())
   ⇒ "Mon Feb 17 01:15:06 1997\n"
```

See also: [asctime], page 751, [time], page 749, [localtime], page 750.

tm_struct = `gmtime` (*t*) [Built-in Function]

Given a value returned from `time`, or any non-negative integer, return a time structure corresponding to CUT (Coordinated Universal Time).

For example:

```
gmtime (time ())
     ⇒ {
           usec = 0
           sec = 6
           min = 15
           hour = 7
           mday = 17
           mon = 1
           year = 97
           wday = 1
           yday = 47
           isdst = 0
           zone = CST
        }
```

See also: [strftime], page 751, [strptime], page 753, [localtime], page 750, [mktime], page 751, [time], page 749, [now], page 749, [date], page 754, [clock], page 753, [datenum], page 756, [datestr], page 757, [datevec], page 758, [calendar], page 759, [weekday], page 759.

tm_struct = `localtime` (*t*) [Built-in Function]

Given a value returned from `time`, or any non-negative integer, return a time structure corresponding to the local time zone.

```
localtime (time ())
     ⇒ {
          usec = 0
          sec = 6
          min = 15
          hour = 1
          mday = 17
          mon = 1
          year = 97
          wday = 1
          yday = 47
          isdst = 0
          zone = CST
        }
```

See also: [strftime], page 751, [strptime], page 753, [gmtime], page 750, [mktime], page 751, [time], page 749, [now], page 749, [date], page 754, [clock], page 753, [datenum], page 756, [datestr], page 757, [datevec], page 758, [calendar], page 759, [weekday], page 759.

seconds = `mktime` (*tm_struct*) [Built-in Function]

Convert a time structure corresponding to the local time to the number of seconds since the epoch.

For example:

```
mktime (localtime (time ()))
     ⇒ 856163706
```

See also: [strftime], page 751, [strptime], page 753, [localtime], page 750, [gmtime], page 750, [time], page 749, [now], page 749, [date], page 754, [clock], page 753, [datenum], page 756, [datestr], page 757, [datevec], page 758, [calendar], page 759, [weekday], page 759.

`asctime` (*tm_struct*) [Function File]

Convert a time structure to a string using the following format: `"ddd mmm mm HH:MM:SS yyyy\n"`.

For example:

```
asctime (localtime (time ()))
     ⇒ "Mon Feb 17 01:15:06 1997\n"
```

This is equivalent to `ctime (time ())`.

See also: [ctime], page 750, [localtime], page 750, [time], page 749.

`strftime` (*fmt*, *tm_struct*) [Built-in Function]

Format the time structure *tm_struct* in a flexible way using the format string *fmt* that contains '%' substitutions similar to those in `printf`.

Except where noted, substituted fields have a fixed size; numeric fields are padded if necessary. Padding is with zeros by default; for fields that display a single number, padding can be changed or inhibited by following the '%' with one of the modifiers described below. Unknown field specifiers are copied as normal characters. All other characters are copied to the output without change. For example:

```
strftime ("%r (%Z) %A %e %B %Y", localtime (time ()))
     ⇒ "01:15:06 AM (CST) Monday 17 February 1997"
```

Octave's `strftime` function supports a superset of the ANSI C field specifiers.

Literal character fields:

`%%` % character.

`%n` Newline character.

`%t` Tab character.

Numeric modifiers (a nonstandard extension):

`- (dash)` Do not pad the field.

`_ (underscore)`
Pad the field with spaces.

Time fields:

`%H` Hour (00-23).

`%I` Hour (01-12).

`%k` Hour (0-23).

`%l` Hour (1-12).

`%M` Minute (00-59).

`%p` Locale's AM or PM.

`%r` Time, 12-hour (hh:mm:ss [AP]M).

`%R` Time, 24-hour (hh:mm).

`%s` Time in seconds since 00:00:00, Jan 1, 1970 (a nonstandard extension).

`%S` Second (00-61).

`%T` Time, 24-hour (hh:mm:ss).

`%X` Locale's time representation (%H:%M:%S).

`%Z` Time zone (EDT), or nothing if no time zone is determinable.

Date fields:

`%a` Locale's abbreviated weekday name (Sun-Sat).

`%A` Locale's full weekday name, variable length (Sunday-Saturday).

`%b` Locale's abbreviated month name (Jan-Dec).

`%B` Locale's full month name, variable length (January-December).

`%c` Locale's date and time (Sat Nov 04 12:02:33 EST 1989).

`%C` Century (00-99).

`%d` Day of month (01-31).

`%e` Day of month (1-31).

`%D` Date (mm/dd/yy).

`%h` Same as %b.

`%j` Day of year (001-366).

`%m` Month (01-12).

`%U` Week number of year with Sunday as first day of week (00-53).

`%w` Day of week (0-6).

`%W` Week number of year with Monday as first day of week (00-53).

`%x` Locale's date representation (mm/dd/yy).

`%y` Last two digits of year (00-99).

`%Y` Year (1970-).

See also: [strptime], page 753, [localtime], page 750, [gmtime], page 750, [mktime], page 751, [time], page 749, [now], page 749, [date], page 754, [clock], page 753, [datenum], page 756, [datestr], page 757, [datevec], page 758, [calendar], page 759, [weekday], page 759.

`[`*`tm_struct`*`, `*`nchars`*`] = strptime (`*`str`*`, `*`fmt`*`)` [Built-in Function]

Convert the string *str* to the time structure *tm_struct* under the control of the format string *fmt*.

If *fmt* fails to match, *nchars* is 0; otherwise, it is set to the position of last matched character plus 1. Always check for this unless you're absolutely sure the date string will be parsed correctly.

See also: [strftime], page 751, [localtime], page 750, [gmtime], page 750, [mktime], page 751, [time], page 749, [now], page 749, [date], page 754, [clock], page 753, [datenum], page 756, [datestr], page 757, [datevec], page 758, [calendar], page 759, [weekday], page 759.

Most of the remaining functions described in this section are not patterned after the standard C library. Some are available for compatibility with MATLAB and others are provided because they are useful.

`clock ()` [Function File]

Return the current local date and time as a date vector.

The date vector contains the following fields: current year, month (1-12), day (1-31), hour (0-23), minute (0-59), and second (0-61). The seconds field has a fractional part after the decimal point for extended accuracy.

For example:

```
fix (clock ())
     ⇒ [ 1993, 8, 20, 4, 56, 1 ]
```

`clock` is more accurate on systems that have the `gettimeofday` function.

See also: [now], page 749, [date], page 754, [datevec], page 758.

`date ()` [Function File]

Return the current date as a character string in the form DD-MMM-YYYY.

For example:

```
date ()
  ⇒ "20-Aug-1993"
```

See also: [now], page 749, [clock], page 753, [datestr], page 757, [localtime], page 750.

`etime (`*t2*`, `*t1*`)` [Function File]

Return the difference in seconds between two time values returned from `clock` (*t2* − *t1*).

For example:

```
t0 = clock ();
# many computations later...
elapsed_time = etime (clock (), t0);
```

will set the variable `elapsed_time` to the number of seconds since the variable `t0` was set.

See also: [tic], page 754, [toc], page 754, [clock], page 753, [cputime], page 754, [addtodate], page 759.

`[`*total*`, `*user*`, `*system*`] = cputime ();` [Built-in Function]

Return the CPU time used by your Octave session.

The first output is the total time spent executing your process and is equal to the sum of second and third outputs, which are the number of CPU seconds spent executing in user mode and the number of CPU seconds spent executing in system mode, respectively.

If your system does not have a way to report CPU time usage, `cputime` returns 0 for each of its output values.

Note that because Octave used some CPU time to start, it is reasonable to check to see if `cputime` works by checking to see if the total CPU time used is nonzero.

See also: [tic], page 754, [toc], page 754.

`is_leap_year ()` [Function File]
`is_leap_year (`*year*`)` [Function File]

Return true if *year* is a leap year and false otherwise.

If no year is specified, `is_leap_year` uses the current year.

For example:

```
is_leap_year (2000)
   ⇒ 1
```

See also: [weekday], page 759, [eomday], page 759, [calendar], page 759.

`tic ()` [Built-in Function]
id `= tic ()` [Built-in Function]
`toc ()` [Built-in Function]
`toc (`*id*`)` [Built-in Function]

`val = toc (...)` [Built-in Function]

Set or check a wall-clock timer.

Calling `tic` without an output argument sets the internal timer state. Subsequent calls to `toc` return the number of seconds since the timer was set. For example,

```
tic ();
# many computations later...
elapsed_time = toc ();
```

will set the variable `elapsed_time` to the number of seconds since the most recent call to the function `tic`.

If called with one output argument, `tic` returns a scalar of type `uint64` that may be later passed to `toc`.

```
id = tic; sleep (5); toc (id)
      ⇒ 5.0010
```

Calling `tic` and `toc` this way allows nested timing calls.

If you are more interested in the CPU time that your process used, you should use the `cputime` function instead. The `tic` and `toc` functions report the actual wall clock time that elapsed between the calls. This may include time spent processing other jobs or doing nothing at all.

See also: [toc], page 754, [cputime], page 754.

`pause ()` [Built-in Function]
`pause (n)` [Built-in Function]

Suspend the execution of the program for *n* seconds.

n is a positive real value and may be a fraction of a second.

If invoked without an input arguments then the program is suspended until a character is typed.

The following example prints a message and then waits 5 seconds before clearing the screen.

```
fprintf (stderr, "wait please...\n");
pause (5);
clc;
```

See also: [kbhit], page 237, [sleep], page 755.

`sleep (seconds)` [Built-in Function]

Suspend the execution of the program for the given number of seconds.

See also: [usleep], page 755, [pause], page 755.

`usleep (microseconds)` [Built-in Function]

Suspend the execution of the program for the given number of microseconds.

On systems where it is not possible to sleep for periods of time less than one second, `usleep` will pause the execution for `round (microseconds / 1e6)` seconds.

See also: [sleep], page 755, [pause], page 755.

days = datenum (*datevec*) [Function File]
days = datenum (*year*, *month*, *day*) [Function File]
days = datenum (*year*, *month*, *day*, *hour*) [Function File]
days = datenum (*year*, *month*, *day*, *hour*, *minute*) [Function File]
days = datenum (*year*, *month*, *day*, *hour*, *minute*, *second*) [Function File]
days = datenum ("*datestr*") [Function File]
days = datenum ("*datestr*", *f*) [Function File]
days = datenum ("*datestr*", *p*) [Function File]
[*days*, *secs*] = datenum (...) [Function File]

Return the date/time input as a serial day number, with Jan 1, 0000 defined as day 1.

The integer part, `floor (days)` counts the number of complete days in the date input.

The fractional part, `rem (days, 1)` corresponds to the time on the given day.

The input may be a date vector (see `datevec`), datestr (see `datestr`), or directly specified as input.

When processing input datestrings, *f* is the format string used to interpret date strings (see `datestr`). If no format *f* is specified, then a relatively slow search is performed through various formats. It is always preferable to specify the format string *f* if it is known. Formats which do not specify a particular time component will have the value set to zero. Formats which do not specify a date will default to January 1st of the current year.

p is the year at the start of the century to which two-digit years will be referenced. If not specified, it defaults to the current year minus 50.

The optional output *secs* holds the time on the specified day with greater precision than *days*.

Notes:

- Years can be negative and/or fractional.
- Months below 1 are considered to be January.
- Days of the month start at 1.
- Days beyond the end of the month go into subsequent months.
- Days before the beginning of the month go to the previous month.
- Days can be fractional.

Caution: this function does not attempt to handle Julian calendars so dates before October 15, 1582 are wrong by as much as eleven days. Also, be aware that only Roman Catholic countries adopted the calendar in 1582. It took until 1924 for it to be adopted everywhere. See the Wikipedia entry on the Gregorian calendar for more details.

Warning: leap seconds are ignored. A table of leap seconds is available on the Wikipedia entry for leap seconds.

See also: [datestr], page 757, [datevec], page 758, [now], page 749, [clock], page 753, [date], page 754.

str = datestr (*date*) [Function File]
str = datestr (*date*, *f*) [Function File]
str = datestr (*date*, *f*, *p*) [Function File]

Format the given date/time according to the format `f` and return the result in *str*.

date is a serial date number (see `datenum`) or a date vector (see `datevec`). The value of *date* may also be a string or cell array of strings.

f can be an integer which corresponds to one of the codes in the table below, or a date format string.

p is the year at the start of the century in which two-digit years are to be interpreted in. If not specified, it defaults to the current year minus 50.

For example, the date 730736.65149 (2000-09-07 15:38:09.0934) would be formatted as follows:

Code	**Format**	**Example**
0	dd-mmm-yyyy HH:MM:SS	07-Sep-2000 15:38:09
1	dd-mmm-yyyy	07-Sep-2000
2	mm/dd/yy	09/07/00
3	mmm	Sep
4	m	S
5	mm	09
6	mm/dd	09/07
7	dd	07
8	ddd	Thu
9	d	T
10	yyyy	2000
11	yy	00
12	mmmyy	Sep00
13	HH:MM:SS	15:38:09
14	HH:MM:SS PM	03:38:09 PM
15	HH:MM	15:38
16	HH:MM PM	03:38 PM
17	QQ-YY	Q3-00
18	QQ	Q3
19	dd/mm	07/09
20	dd/mm/yy	07/09/00
21	mmm.dd,yyyy HH:MM:SS	Sep.07,2000 15:38:08
22	mmm.dd,yyyy	Sep.07,2000
23	mm/dd/yyyy	09/07/2000
24	dd/mm/yyyy	07/09/2000
25	yy/mm/dd	00/09/07
26	yyyy/mm/dd	2000/09/07
27	QQ-YYYY	Q3-2000
28	mmmyyyy	Sep2000
29	yyyy-mm-dd	2000-09-07
30	yyyymmddTHHMMSS	20000907T153808
31	yyyy-mm-dd HH:MM:SS	2000-09-07 15:38:08

If *f* is a format string, the following symbols are recognized:

Symbol	**Meaning**	**Example**
yyyy	Full year	2005
yy	Two-digit year	05
mmmm	Full month name	December
mmm	Abbreviated month name	Dec
mm	Numeric month number (padded with zeros)	01, 08, 12
m	First letter of month name (capitalized)	D
dddd	Full weekday name	Sunday
ddd	Abbreviated weekday name	Sun
dd	Numeric day of month (padded with zeros)	11
d	First letter of weekday name (capitalized)	S
HH	Hour of day, padded with zeros if PM is set	09:00
	and not padded with zeros otherwise	9:00 AM
MM	Minute of hour (padded with zeros)	10:05
SS	Second of minute (padded with zeros)	10:05:03
FFF	Milliseconds of second (padded with zeros)	10:05:03.012
AM	Use 12-hour time format	11:30 AM
PM	Use 12-hour time format	11:30 PM

If *f* is not specified or is `-1`, then use 0, 1 or 16, depending on whether the date portion or the time portion of *date* is empty.

If *p* is nor specified, it defaults to the current year minus 50.

If a matrix or cell array of dates is given, a column vector of date strings is returned.

See also: [datenum], page 756, [datevec], page 758, [date], page 754, [now], page 749, [clock], page 753.

v = `datevec` (*date*) [Function File]
v = `datevec` (*date*, *f*) [Function File]
v = `datevec` (*date*, *p*) [Function File]
v = `datevec` (*date*, *f*, *p*) [Function File]
[*y*, *m*, *d*, *h*, *mi*, *s*] = `datevec (...)` [Function File]

Convert a serial date number (see `datenum`) or date string (see `datestr`) into a date vector.

A date vector is a row vector with six members, representing the year, month, day, hour, minute, and seconds respectively.

f is the format string used to interpret date strings (see `datestr`). If *date* is a string, but no format is specified, then a relatively slow search is performed through various formats. It is always preferable to specify the format string *f* if it is known. Formats which do not specify a particular time component will have the value set to zero. Formats which do not specify a date will default to January 1st of the current year.

p is the year at the start of the century to which two-digit years will be referenced. If not specified, it defaults to the current year minus 50.

See also: [datenum], page 756, [datestr], page 757, [clock], page 753, [now], page 749, [date], page 754.

`d = addtodate (d, q, f)` [Function File]

Add *q* amount of time (with units *f*) to the serial datenum, *d*.

f must be one of `"year"`, `"month"`, `"day"`, `"hour"`, `"minute"`, `"second"`, or `"millisecond"`.

See also: [datenum], page 756, [datevec], page 758, [etime], page 754.

`c = calendar ()` [Function File]
`c = calendar (d)` [Function File]
`c = calendar (y, m)` [Function File]
`calendar (...)` [Function File]

Return the current monthly calendar in a 6x7 matrix.

If *d* is specified, return the calendar for the month containing the date *d*, which must be a serial date number or a date string.

If *y* and *m* are specified, return the calendar for year *y* and month *m*.

If no output arguments are specified, print the calendar on the screen instead of returning a matrix.

See also: [datenum], page 756, [datestr], page 757.

`[n, s] = weekday (d)` [Function File]
`[n, s] = weekday (d, format)` [Function File]

Return the day of the week as a number in *n* and as a string in *s*.

The days of the week are numbered 1–7 with the first day being Sunday.

d is a serial date number or a date string.

If the string *format* is not present or is equal to `"short"` then *s* will contain the abbreviated name of the weekday. If *format* is `"long"` then *s* will contain the full name.

Table of return values based on *format*:

n	`"short"`	`"long"`
1	Sun	Sunday
2	Mon	Monday
3	Tue	Tuesday
4	Wed	Wednesday
5	Thu	Thursday
6	Fri	Friday
7	Sat	Saturday

See also: [eomday], page 759, [is_leap_year], page 754, [calendar], page 759, [datenum], page 756, [datevec], page 758.

`e = eomday (y, m)` [Function File]

Return the last day of the month *m* for the year *y*.

See also: [weekday], page 759, [datenum], page 756, [datevec], page 758, [is_leap_year], page 754, [calendar], page 759.

`datetick ()` [Function File]
`datetick (`*form*`)` [Function File]
`datetick (`*axis*, *form*`)` [Function File]
`datetick (..., "`*keeplimits*`")` [Function File]
`datetick (..., "`*keepticks*`")` [Function File]
`datetick (`*hax*, ...`)` [Function File]

Add date formatted tick labels to an axis.

The axis to apply the ticks to is determined by *axis* which can take the values `"x"`, `"y"`, or `"z"`. The default value is `"x"`.

The formatting of the labels is determined by the variable *form*, which can either be a string or positive integer that `datestr` accepts.

See also: [datenum], page 756, [datestr], page 757.

36.2 Filesystem Utilities

Octave includes many utility functions for copying, moving, renaming, and deleting files; for creating, reading, and deleting directories; for retrieving status information on files; and for manipulating file and path names.

`movefile (`*f1*`)` [Function File]
`movefile (`*f1*, *f2*`)` [Function File]
`movefile (`*f1*, *f2*, *'f'*`)` [Function File]
`[`*status*, *msg*, *msgid*`] = movefile (...)` [Function File]

Move the source files or directories *f1* to the destination *f2*.

The name *f1* may contain globbing patterns. If *f1* expands to multiple file names, *f2* must be a directory. If no destination *f2* is specified then the destination is the present working directory. If *f2* is a file name then *f1* is renamed to *f2*.

When the force flag `'f'` is given any existing files will be overwritten without prompting.

If successful, *status* is 1, and *msg*, *msgid* are empty character strings (`""`). Otherwise, *status* is 0, *msg* contains a system-dependent error message, and *msgid* contains a unique message identifier. Note that the status code is exactly opposite that of the `system` command.

See also: [rename], page 760, [copyfile], page 760, [unlink], page 761, [delete], page 341, [glob], page 766.

`rename` *old new* [Built-in Function]
`[`*err*, *msg*`] = rename (`*old*, *new*`)` [Built-in Function]

Change the name of file *old* to *new*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [movefile], page 760, [copyfile], page 760, [ls], page 785, [dir], page 785.

`[`*status*, *msg*, *msgid*`] = copyfile (`*f1*, *f2*`)` [Function File]
`[`*status*, *msg*, *msgid*`] = copyfile (`*f1*, *f2*, *'f'*`)` [Function File]

Copy the source files or directories *f1* to the destination *f2*.

The name *f1* may contain globbing patterns. If *f1* expands to multiple file names, *f2* must be a directory.

When the force flag `'f'` is given any existing files will be overwritten without prompting.

If successful, *status* is 1, and *msg*, *msgid* are empty character strings (`""`). Otherwise, *status* is 0, *msg* contains a system-dependent error message, and *msgid* contains a unique message identifier. Note that the status code is exactly opposite that of the `system` command.

See also: [movefile], page 760, [rename], page 760, [unlink], page 761, [delete], page 341, [glob], page 766.

`[err, msg] = unlink (file)` [Built-in Function]

Delete the file named *file*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [delete], page 341, [rmdir], page 762.

`link old new` [Built-in Function]
`[err, msg] = link (old, new)` [Built-in Function]

Create a new link (also known as a hard link) to an existing file.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [symlink], page 761, [unlink], page 761, [readlink], page 761, [lstat], page 763.

`symlink old new` [Built-in Function]
`[err, msg] = symlink (old, new)` [Built-in Function]

Create a symbolic link *new* which contains the string *old*.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [link], page 761, [unlink], page 761, [readlink], page 761, [lstat], page 763.

`readlink symlink` [Built-in Function]
`[result, err, msg] = readlink (symlink)` [Built-in Function]

Read the value of the symbolic link *symlink*.

If successful, *result* contains the contents of the symbolic link *symlink*, *err* is 0, and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [lstat], page 763, [symlink], page 761, [link], page 761, [unlink], page 761, [delete], page 341.

`mkdir dir` [Built-in Function]
`mkdir (parent, dir)` [Built-in Function]
`[status, msg, msgid] = mkdir (...)` [Built-in Function]

Create a directory named *dir* in the directory *parent*.

If no *parent* directory is specified the present working directory is used.

If successful, *status* is 1, and *msg*, *msgid* are empty character strings (""). Otherwise, *status* is 0, *msg* contains a system-dependent error message, and *msgid* contains a unique message identifier.

When creating a directory permissions will be set to `0777 - umask`.

See also: [rmdir], page 762, [pwd], page 786, [cd], page 784, [umask], page 762.

`rmdir dir` [Built-in Function]
`rmdir (dir, "s")` [Built-in Function]
`[status, msg, msgid] = rmdir (...)` [Built-in Function]

Remove the directory named *dir*.

If the optional second parameter is supplied with value `"s"`, recursively remove all subdirectories as well.

If successful, *status* is 1, and *msg*, *msgid* are empty character strings (""). Otherwise, *status* is 0, *msg* contains a system-dependent error message, and *msgid* contains a unique message identifier.

See also: [mkdir], page 761, [confirm_recursive_rmdir], page 762, [pwd], page 786.

`val = confirm_recursive_rmdir ()` [Built-in Function]
`old_val = confirm_recursive_rmdir (new_val)` [Built-in Function]
`confirm_recursive_rmdir (new_val, "local")` [Built-in Function]

Query or set the internal variable that controls whether Octave will ask for confirmation before recursively removing a directory tree.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [rmdir], page 762.

`err = mkfifo (name, mode)` [Built-in Function]
`[err, msg] = mkfifo (name, mode)` [Built-in Function]

Create a FIFO special file named *name* with file mode *mode*.

mode is interpreted as a decimal number (*not* octal) and is subject to umask processing. The final calculated mode is `mode - umask`.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [pipe], page 779, [umask], page 762.

`umask (mask)` [Built-in Function]

Set the permission mask for file creation.

The parameter *mask* is an integer, interpreted as an octal number.

If successful, returns the previous value of the mask (as an integer to be interpreted as an octal number); otherwise an error message is printed.

The permission mask is a UNIX concept used when creating new objects on a file system such as files, directories, or named FIFOs. The object to be created has base permissions in an octal number *mode* which are modified according to the octal value of *mask*. The final permissions for the new object are `mode - mask`.

See also: [fopen], page 250, [mkdir], page 761, [mkfifo], page 762.

`[info, err, msg] = stat (file)` [Built-in Function]
`[info, err, msg] = stat (fid)` [Built-in Function]
`[info, err, msg] = lstat (file)` [Built-in Function]
`[info, err, msg] = lstat (fid)` [Built-in Function]

Return a structure *info* containing the following information about *file* or file identifier *fid*.

`dev` ID of device containing a directory entry for this file.

`ino` File number of the file.

`mode` File mode, as an integer. Use the functions `S_ISREG`, `S_ISDIR`, `S_ISCHR`, `S_ISBLK`, `S_ISFIFO`, `S_ISLNK`, or `S_ISSOCK` to extract information from this value.

`modestr` File mode, as a string of ten letters or dashes as would be returned by `ls -l`.

`nlink` Number of links.

`uid` User ID of file's owner.

`gid` Group ID of file's group.

`rdev` ID of device for block or character special files.

`size` Size in bytes.

`atime` Time of last access in the same form as time values returned from `time`. See Section 36.1 [Timing Utilities], page 749.

`mtime` Time of last modification in the same form as time values returned from `time`. See Section 36.1 [Timing Utilities], page 749.

`ctime` Time of last file status change in the same form as time values returned from `time`. See Section 36.1 [Timing Utilities], page 749.

`blksize` Size of blocks in the file.

`blocks` Number of blocks allocated for file.

If the call is successful *err* is 0 and *msg* is an empty string. If the file does not exist, or some other error occurs, *info* is an empty matrix, *err* is -1, and *msg* contains the corresponding system error message.

If *file* is a symbolic link, `stat` will return information about the actual file that is referenced by the link. Use `lstat` if you want information about the symbolic link itself.

For example:

```
[info, err, msg] = stat ("/vmlinuz")
  ⇒ info =
```

```
     {
       atime = 855399756
       rdev = 0
       ctime = 847219094
       uid = 0
       size = 389218
       blksize = 4096
       mtime = 847219094
       gid = 6
       nlink = 1
       blocks = 768
       mode = -rw-r--r--
       modestr = -rw-r--r--
       ino = 9316
       dev = 2049
     }
  ⇒ err = 0
  ⇒ msg =
```

See also: [lstat], page 763, [ls], page 785, [dir], page 785.

`S_ISBLK` (*mode*) [Built-in Function]

Return true if *mode* corresponds to a block device.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`S_ISCHR` (*mode*) [Built-in Function]

Return true if *mode* corresponds to a character device.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`S_ISDIR` (*mode*) [Built-in Function]

Return true if *mode* corresponds to a directory.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`S_ISFIFO` (*mode*) [Built-in Function]

Return true if *mode* corresponds to a fifo.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`S_ISLNK` (*mode*) [Built-in Function]

Return true if *mode* corresponds to a symbolic link.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`S_ISREG (mode)` [Built-in Function]

Return true if *mode* corresponds to a regular file.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`S_ISSOCK (mode)` [Built-in Function]

Return true if *mode* corresponds to a socket.

The value of *mode* is assumed to be returned from a call to `stat`.

See also: [stat], page 763, [lstat], page 763.

`[status, result, msgid] = fileattrib (file)` [Function File]

Return information about *file*.

If successful, *status* is 1, with *result* containing a structure with the following fields:

`Name` Full name of *file*.

`archive` True if *file* is an archive (Windows).

`system` True if *file* is a system file (Windows).

`hidden` True if *file* is a hidden file (Windows).

`directory`
True if *file* is a directory.

`UserRead`
`GroupRead`
`OtherRead`
True if the user (group; other users) has read permission for *file*.

`UserWrite`
`GroupWrite`
`OtherWrite`
True if the user (group; other users) has write permission for *file*.

`UserExecute`
`GroupExecute`
`OtherExecute`
True if the user (group; other users) has execute permission for *file*.

If an attribute does not apply (i.e., archive on a Unix system) then the field is set to NaN.

With no input arguments, return information about the current directory.

If *file* contains globbing characters, return information about all the matching files.

See also: [glob], page 766.

`isdir (f)` [Function File]

Return true if *f* is a directory.

See also: [exist], page 130, [stat], page 763, [is_absolute_filename], page 768, [is_rooted_relative_filename], page 769.

`files = readdir (dir)` [Built-in Function]
`[files, err, msg] = readdir (dir)` [Built-in Function]

Return the names of files in the directory *dir* as a cell array of strings.

If an error occurs, return an empty cell array in *files*. If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [ls], page 785, [dir], page 785, [glob], page 766, [what], page 132.

`glob (pattern)` [Built-in Function]

Given an array of pattern strings (as a char array or a cell array) in *pattern*, return a cell array of file names that match any of them, or an empty cell array if no patterns match.

The pattern strings are interpreted as filename globbing patterns (as they are used by Unix shells).

Within a pattern

`*` matches any string, including the null string,

`?` matches any single character, and

`[...]` matches any of the enclosed characters.

Tilde expansion is performed on each of the patterns before looking for matching file names. For example:

```
ls
   ⇒
      file1  file2  file3  myfile1 myfile1b
glob ("*file1")
   ⇒
      {
        [1,1] = file1
        [2,1] = myfile1
      }
glob ("myfile?")
   ⇒
      {
        [1,1] = myfile1
      }
glob ("file[12]")
   ⇒
      {
        [1,1] = file1
        [2,1] = file2
      }
```

See also: [ls], page 785, [dir], page 785, [readdir], page 766, [what], page 132.

`file_in_path (path, file)` [Built-in Function]
`file_in_path (path, file, "all")` [Built-in Function]

Return the absolute name of *file* if it can be found in *path*.

The value of *path* should be a colon-separated list of directories in the format described for `path`. If no file is found, return an empty character string. For example:

```
file_in_path (EXEC_PATH, "sh")
     ⇒ "/bin/sh"
```

If the second argument is a cell array of strings, search each directory of the path for element of the cell array and return the first that matches.

If the third optional argument `"all"` is supplied, return a cell array containing the list of all files that have the same name in the path. If no files are found, return an empty cell array.

See also: [file_in_loadpath], page 191, [dir_in_loadpath], page 192, [path], page 190.

`filesep ()` [Built-in Function]
`filesep ("`*all*`")` [Built-in Function]

Return the system-dependent character used to separate directory names.

If `"all"` is given, the function returns all valid file separators in the form of a string. The list of file separators is system-dependent. It is '`/`' (forward slash) under UNIX or Mac OS X, '`/`' and '`\`' (forward and backward slashes) under Windows.

See also: [pathsep], page 191.

val `= filemarker ()` [Built-in Function]
old_val `= filemarker (`*new_val*`)` [Built-in Function]
`filemarker (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the character used to separate the filename from the subfunction names contained within the file.

By default this is the character '`>`'. This can be used in a generic manner to interact with subfunctions. For example,

```
help (["myfunc", filemarker, "mysubfunc"])
```

returns the help string associated with the subfunction `mysubfunc` located in the file `myfunc.m`.

`filemarker` is also useful during debugging for placing breakpoints within subfunctions or nested functions. For example,

```
dbstop (["myfunc", filemarker, "mysubfunc"])
```

will set a breakpoint at the first line of the subfunction `mysubfunc`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

`[`*dir*`, `*name*`, `*ext*`] = fileparts (`*filename*`)` [Function File]

Return the directory, name, and extension components of *filename*.

The input *filename* is a string which is parsed. There is no attempt to check whether the filename or directory specified actually exists.

See also: [fullfile], page 768, [filesep], page 767.

`filename = fullfile (dir1, dir2, ..., file)` [Function File]
`filenames = fullfile (..., files)` [Function File]

Build complete filename from separate parts.

Joins any number of path components intelligently. The return value is the concatenation of each component with exactly one file separator between each non empty part and at most one leading and/or trailing file separator.

If the last component part is a cell array, returns a cell array of filepaths, one for each element in the last component, e.g.:

```
fullfile ("/home/username", "data", {"f1.csv", "f2.csv", "f3.csv"})
⇒  /home/username/data/f1.csv
    /home/username/data/f2.csv
    /home/username/data/f3.csv
```

On Windows systems, while forward slash file separators do work, they are replaced by backslashes; in addition drive letters are stripped of leading file separators to obtain a valid file path.

See also: [fileparts], page 767, [filesep], page 767.

`tilde_expand (string)` [Built-in Function]

Perform tilde expansion on *string*.

If *string* begins with a tilde character, ('~'), all of the characters preceding the first slash (or all characters, if there is no slash) are treated as a possible user name, and the tilde and the following characters up to the slash are replaced by the home directory of the named user. If the tilde is followed immediately by a slash, the tilde is replaced by the home directory of the user running Octave.

For example:

```
tilde_expand ("~joeuser/bin")
     ⇒ "/home/joeuser/bin"
tilde_expand ("~/bin")
     ⇒ "/home/jwe/bin"
```

`[cname, status, msg] = canonicalize_file_name (fname)` [Built-in Function]

Return the canonical name of file *fname*.

If the file does not exist the empty string ("") is returned.

See also: [make_absolute_filename], page 768, [is_absolute_filename], page 768, [is_rooted_relative_filename], page 769.

`make_absolute_filename (file)` [Built-in Function]

Return the full name of *file* beginning from the root of the file system.

No check is done for the existence of *file*.

See also: [canonicalize_file_name], page 768, [is_absolute_filename], page 768, [is_rooted_relative_filename], page 769, [isdir], page 765.

`is_absolute_filename (file)` [Built-in Function]

Return true if *file* is an absolute filename.

See also: [is_rooted_relative_filename], page 769, [make_absolute_filename], page 768, [isdir], page 765.

`is_rooted_relative_filename (`*`file`*`)` [Built-in Function]

Return true if *file* is a rooted-relative filename.

See also: [is_absolute_filename], page 768, [make_absolute_filename], page 768, [isdir], page 765.

`current_state` `= recycle ()` [Function File]
`old_state` `= recycle (`*`new_state`*`)` [Function File]

Query or set the preference for recycling deleted files.

When recycling is enabled, commands which would permanently erase files instead move them to a temporary location (such as the directory labeled Trash).

Programming Note: This function is provided for MATLAB compatibility, but recycling is not implemented in Octave. To help avoid accidental data loss an error will be raised if an attempt is made to enable file recycling.

See also: [delete], page 341, [rmdir], page 762.

36.3 File Archiving Utilities

`filelist` `= bunzip2 (`*`bzfile`*`)` [Function File]
`filelist` `= bunzip2 (`*`bzfile`*`,` *`dir`*`)` [Function File]

Unpack the bzip2 archive *bzfile*.

If *dir* is specified the files are unpacked in this directory rather than the one where *bzfile* is located.

The optional output *filelist* is a list of the uncompressed files.

See also: [bzip2], page 771, [unpack], page 771, [gunzip], page 770, [unzip], page 771, [untar], page 770.

`filelist` `= gzip (`*`files`*`)` [Function File]
`filelist` `= gzip (`*`files`*`,` *`dir`*`)` [Function File]

Compress the list of files and directories specified in *files*.

files is a character array or cell array of strings. Shell wildcards in the filename such as '`*`' or '`?`' are accepted and expanded. Each file is compressed separately and a new file with a `".gz"` extension is created. The original files are not modified, but existing compressed files will be silently overwritten. If a directory is specified then `gzip` recursively compresses all files in the directory.

If *dir* is defined the compressed files are placed in this directory, rather than the original directory where the uncompressed file resides. If *dir* does not exist it is created.

The optional output *filelist* is a list of the compressed files.

See also: [gunzip], page 770, [unpack], page 771, [bzip2], page 771, [zip], page 770, [tar], page 770.

filelist = gunzip (*gzfile*) [Function File]
filelist = gunzip (*gzfile*, *dir*) [Function File]

Unpack the gzip archive *gzfile*.

If *gzfile* is a directory, all gzfiles in the directory will be recursively unpacked.

If *dir* is specified the files are unpacked in this directory rather than the one where *gzfile* is located.

The optional output *filelist* is a list of the uncompressed files.

See also: [gzip], page 769, [unpack], page 771, [bunzip2], page 769, [unzip], page 771, [untar], page 770.

filelist = tar (*tarfile*, *files*) [Function File]
filelist = tar (*tarfile*, *files*, *rootdir*) [Function File]

Pack the list of files and directories specified in *files* into the TAR archive *tarfile*.

files is a character array or cell array of strings. Shell wildcards in the filename such as '*' or '?' are accepted and expanded. Directories are recursively traversed and all files are added to the archive.

If *rootdir* is defined then any files without absolute pathnames are located relative to *rootdir* rather than the current directory.

The optional output *filelist* is a list of the files that were included in the archive.

See also: [untar], page 770, [unpack], page 771, [bzip2], page 771, [gzip], page 769, [zip], page 770.

untar (*tarfile*) [Function File]
untar (*tarfile*, *dir*) [Function File]

Unpack the TAR archive *tarfile*.

If *dir* is specified the files are unpacked in this directory rather than the one where *tarfile* is located.

The optional output *filelist* is a list of the uncompressed files.

See also: [tar], page 770, [unpack], page 771, [bunzip2], page 769, [gunzip], page 770, [unzip], page 771.

filelist = zip (*zipfile*, *files*) [Function File]
filelist = zip (*zipfile*, *files*, *rootdir*) [Function File]

Compress the list of files and directories specified in *files* into the ZIP archive *zipfile*.

files is a character array or cell array of strings. Shell wildcards in the filename such as '*' or '?' are accepted and expanded. Directories are recursively traversed and all files are compressed and added to the archive.

If *rootdir* is defined then any files without absolute pathnames are located relative to *rootdir* rather than the current directory.

The optional output *filelist* is a list of the files that were included in the archive.

See also: [unzip], page 771, [unpack], page 771, [bzip2], page 771, [gzip], page 769, [tar], page 770.

filelist = unzip (*zipfile*) [Function File]
filelist = unzip (*zipfile*, *dir*) [Function File]

Unpack the ZIP archive *zipfile*.

If *dir* is specified the files are unpacked in this directory rather than the one where *zipfile* is located.

The optional output *filelist* is a list of the uncompressed files.

See also: [zip], page 770, [unpack], page 771, [bunzip2], page 769, [gunzip], page 770, [untar], page 770.

files = unpack (*file*) [Function File]
files = unpack (*file*, *dir*) [Function File]
files = unpack (*file*, *dir*, *filetype*) [Function File]

Unpack the archive *file* based on its extension to the directory *dir*.

If *file* is a list of strings, then each file is unpacked individually. Shell wildcards in the filename such as '`*`' or '`?`' are accepted and expanded.

If *dir* is not specified or is empty (`[]`), it defaults to the current directory. If a directory is in the file list, then *filetype* must also be specified.

The specific archive filetype is inferred from the extension of the file. The *filetype* may also be specified directly using a string which corresponds to a known extension.

Valid filetype extensions:

`bz`
`bz2` bzip archive

`gz` gzip archive

`tar` tar archive

`tarbz`
`tarbz2`
`tbz`
`tbz2` tar + bzip archive

`targz`
`tgz` tar + gzip archive

`z` compress archive

`zip` zip archive

The optional return value is a list of *files* unpacked.

See also: [bunzip2], page 769, [gunzip], page 770, [unzip], page 771, [untar], page 770, [bzip2], page 771, [gzip], page 769, [zip], page 770, [tar], page 770.

filelist = bzip2 (*files*) [Function File]
filelist = bzip2 (*files*, *dir*) [Function File]

Compress the list of files specified in *files*.

files is a character array or cell array of strings. Shell wildcards in the filename such as '`*`' or '`?`' are accepted and expanded. Each file is compressed separately and a new file with a `".bz2"` extension is created. The original files are not modified, but existing compressed files will be silently overwritten.

If *dir* is defined the compressed files are placed in this directory, rather than the original directory where the uncompressed file resides. If *dir* does not exist it is created.

The optional output *filelist* is a list of the compressed files.

See also: [bunzip2], page 769, [unpack], page 771, [gzip], page 769, [zip], page 770, [tar], page 770.

36.4 Networking Utilities

`gethostname ()` [Built-in Function]

Return the hostname of the system where Octave is running.

36.4.1 FTP Objects

Octave supports the FTP protocol through an object-oriented interface. Use the function `ftp` to create an FTP object which represents the connection. All FTP functions take an FTP object as the first argument.

`f = ftp (host)` [Function File]
`f = ftp (host, username, password)` [Function File]

Connect to the FTP server *host* with *username* and *password*.

If *username* and *password* are not specified, user `"anonymous"` with no password is used. The returned FTP object *f* represents the established FTP connection.

The list of actions for an FTP object are shown below. All functions require an FTP object as the first argument.

Method	**Description**
ascii	Set transfer type to ascii
binary	Set transfer type to binary
cd	Change remote working directory
close	Close FTP connection
delete	Delete remote file
dir	List remote directory contents
mget	Download remote files
mkdir	Create remote directory
mput	Upload local files
rename	Rename remote file or directory
rmdir	Remove remote directory

`close (f)` [Function File]

Close the FTP connection represented by the FTP object *f*.

f is an FTP object returned by the `ftp` function.

`mget (f, file)` [Function File]
`mget (f, dir)` [Function File]

`mget (f, remote_name, target)` [Function File]

Download a remote file *file* or directory *dir* to the local directory on the FTP connection *f*.

f is an FTP object returned by the `ftp` function.

The arguments *file* and *dir* can include wildcards and any files or directories on the remote server that match will be downloaded.

If a third argument *target* is given, then a single file or directory will be downloaded to the local directory and the local name will be changed to *target*.

`mput (f, file)` [Function File]

Upload the local file *file* into the current remote directory on the FTP connection *f*.

f is an FTP object returned by the ftp function.

The argument *file* is passed through the `glob` function and any files that match the wildcards in *file* will be uploaded.

`cd (f)` [Function File]
`cd (f, path)` [Function File]

Get or set the remote directory on the FTP connection *f*.

f is an FTP object returned by the `ftp` function.

If *path* is not specified, return the remote current working directory. Otherwise, set the remote directory to *path* and return the new remote working directory.

If the directory does not exist, an error message is printed and the working directory is not changed.

`lst = dir (f)` [Function File]

List the current directory in verbose form for the FTP connection *f*.

f is an FTP object returned by the `ftp` function.

`ascii (f)` [Function File]

Set the FTP connection *f* to use ASCII mode for transfers.

ASCII mode is only appropriate for text files as it will convert the remote host's newline representation to the local host's newline representation.

f is an FTP object returned by the `ftp` function.

`binary (f)` [Function File]

Set the FTP connection *f* to use binary mode for transfers.

In binary mode there is no conversion of newlines from the remote representation to the local representation.

f is an FTP object returned by the `ftp` function.

`delete (f, file)` [Function File]

Delete the remote file *file* over the FTP connection *f*.

f is an FTP object returned by the `ftp` function.

`rename (f, oldname, newname)` [Function File]

Rename or move the remote file or directory *oldname* to *newname*, over the FTP connection *f*.

f is an FTP object returned by the ftp function.

`mkdir (f, path)` [Function File]

Create the remote directory *path*, over the FTP connection *f*.

f is an FTP object returned by the `ftp` function.

`rmdir (f, path)` [Function File]

Remove the remote directory *path*, over the FTP connection *f*.

f is an FTP object returned by the `ftp` function.

36.4.2 URL Manipulation

`s = urlread (url)` [Loadable Function]
`[s, success] = urlread (url)` [Loadable Function]
`[s, success, message] = urlread (url)` [Loadable Function]
`[...] = urlread (url, method, param)` [Loadable Function]

Download a remote file specified by its *url* and return its content in string *s*.

For example:

```
s = urlread ("ftp://ftp.octave.org/pub/README");
```

The variable *success* is 1 if the download was successful, otherwise it is 0 in which case *message* contains an error message.

If no output argument is specified and an error occurs, then the error is signaled through Octave's error handling mechanism.

This function uses libcurl. Curl supports, among others, the HTTP, FTP and FILE protocols. Username and password may be specified in the URL. For example:

```
s = urlread ("http://user:password@example.com/file.txt");
```

GET and POST requests can be specified by *method* and *param*. The parameter *method* is either '`get`' or '`post`' and *param* is a cell array of parameter and value pairs. For example:

```
s = urlread ("http://www.google.com/search", "get",
             {"query", "octave"});
```

See also: [urlwrite], page 774.

`urlwrite (url, localfile)` [Loadable Function]
`f = urlwrite (url, localfile)` [Loadable Function]
`[f, success] = urlwrite (url, localfile)` [Loadable Function]
`[f, success, message] = urlwrite (url, localfile)` [Loadable Function]

Download a remote file specified by its *url* and save it as *localfile*.

For example:

```
urlwrite ("ftp://ftp.octave.org/pub/README",
          "README.txt");
```

The full path of the downloaded file is returned in *f*.

The variable *success* is 1 if the download was successful, otherwise it is 0 in which case *message* contains an error message.

If no output argument is specified and an error occurs, then the error is signaled through Octave's error handling mechanism.

This function uses libcurl. Curl supports, among others, the HTTP, FTP and FILE protocols. Username and password may be specified in the URL, for example:

```
urlwrite ("http://username:password@example.com/file.txt",
          "file.txt");
```

GET and POST requests can be specified by *method* and *param*. The parameter *method* is either '`get`' or '`post`' and *param* is a cell array of parameter and value pairs. For example:

```
urlwrite ("http://www.google.com/search", "search.html",
          "get", {"query", "octave"});
```

See also: [urlread], page 774.

36.4.3 Base64 and Binary Data Transmission

Some transmission channels can not accept binary data. It is customary to encode binary data in Base64 for transmission and to decode the data upon reception.

s = `base64_encode` (*x*) [Built-in Function]

Encode a double matrix or array *x* into the base64 format string *s*.

See also: [base64_decode], page 775.

x = `base64_decode` (*s*) [Built-in Function]
x = `base64_decode` (*s*, *dims*) [Built-in Function]

Decode the double matrix or array *x* from the base64 encoded string *s*.

The optional input parameter *dims* should be a vector containing the dimensions of the decoded array.

See also: [base64_encode], page 775.

36.5 Controlling Subprocesses

Octave includes some high-level commands like `system` and `popen` for starting subprocesses. If you want to run another program to perform some task and then look at its output, you will probably want to use these functions.

Octave also provides several very low-level Unix-like functions which can also be used for starting subprocesses, but you should probably only use them if you can't find any way to do what you need with the higher-level functions.

`system ("`*string*`")` [Built-in Function]
`system ("`*string*`",` *return_output*`)` [Built-in Function]
`system ("`*string*`",` *return_output*`,` *type*`)` [Built-in Function]
`[`*status*`,` *output*`] = system (...)` [Built-in Function]

Execute a shell command specified by *string*.

If the optional argument *type* is `"async"`, the process is started in the background and the process ID of the child process is returned immediately. Otherwise, the child process is started and Octave waits until it exits. If the *type* argument is omitted, it defaults to the value `"sync"`.

If *system* is called with one or more output arguments, or if the optional argument *return_output* is true and the subprocess is started synchronously, then the output from the command is returned as a variable. Otherwise, if the subprocess is executed synchronously, its output is sent to the standard output. To send the output of a command executed with `system` through the pager, use a command like

```
[output, text] = system ("cmd");
disp (text);
```

or

```
printf ("%s\n", nthargout (2, "system", "cmd"));
```

The `system` function can return two values. The first is the exit status of the command and the second is any output from the command that was written to the standard output stream. For example,

```
[status, output] = system ("echo foo; exit 2");
```

will set the variable `output` to the string '`foo`', and the variable `status` to the integer '2'.

For commands run asynchronously, *status* is the process id of the command shell that is started to run the command.

See also: [unix], page 776, [dos], page 776.

`unix ("command")` [Function File]
`status = unix ("command")` [Function File]
`[status, text] = unix ("command")` [Function File]
`[...] = unix ("command", "-echo")` [Function File]

Execute a system command if running under a Unix-like operating system, otherwise do nothing.

Octave waits for the external command to finish before returning the exit status of the program in *status* and any output in *text*.

When called with no output argument, or the `"-echo"` argument is given, then *text* is also sent to standard output.

See also: [dos], page 776, [system], page 775, [isunix], page 789, [ismac], page 789, [ispc], page 789.

`dos ("command")` [Function File]
`status = dos ("command")` [Function File]
`[status, text] = dos ("command")` [Function File]
`[...] = dos ("command", "-echo")` [Function File]

Execute a system command if running under a Windows-like operating system, otherwise do nothing.

Octave waits for the external command to finish before returning the exit status of the program in *status* and any output in *text*.

When called with no output argument, or the `"-echo"` argument is given, then *text* is also sent to standard output.

See also: [unix], page 776, [system], page 775, [isunix], page 789, [ismac], page 789, [ispc], page 789.

output = open *file* [Function File]
output = open (*file*) [Function File]

Open the file *file* in Octave or in an external application based on the file type as determined by the file name extension.

Recognized file types are

`.m` Open file in the editor.

`.mat` Load the file in the base workspace.

`.exe` Execute the program (on Windows systems only).

Other file types are opened in the appropriate external application.

output = perl (*scriptfile*) [Function File]
output = perl (*scriptfile*, *argument1*, *argument2*, ...) [Function File]
[*output*, *status*] = perl (...) [Function File]

Invoke Perl script *scriptfile*, possibly with a list of command line arguments.

Return output in *output* and optional status in *status*. If *scriptfile* is not an absolute file name it is searched for in the current directory and then in the Octave loadpath.

See also: [system], page 775, [python], page 777.

output = python (*scriptfile*) [Function File]
output = python (*scriptfile*, *argument1*, *argument2*, ...) [Function File]
[*output*, *status*] = python (...) [Function File]

Invoke Python script *scriptfile*, possibly with a list of command line arguments.

Return output in *output* and optional status in *status*. If *scriptfile* is not an absolute file name it is searched for in the current directory and then in the Octave loadpath.

See also: [system], page 775, [perl], page 777.

fid = popen (*command*, *mode*) [Built-in Function]

Start a process and create a pipe.

The name of the command to run is given by *command*. The argument *mode* may be

`"r"` The pipe will be connected to the standard output of the process, and open for reading.

`"w"` The pipe will be connected to the standard input of the process, and open for writing.

The file identifier corresponding to the input or output stream of the process is returned in *fid*.

For example:

```
fid = popen ("ls -ltr / | tail -3", "r");
while (ischar (s = fgets (fid)))
  fputs (stdout, s);
endwhile

   ⊣ drwxr-xr-x  33 root  root  3072 Feb 15 13:28 etc
   ⊣ drwxr-xr-x   3 root  root  1024 Feb 15 13:28 lib
   ⊣ drwxrwxrwt  15 root  root  2048 Feb 17 14:53 tmp
```

See also: [popen2], page 778.

`pclose (`*`fid`*`)` [Built-in Function]

Close a file identifier that was opened by `popen`.

The function `fclose` may also be used for the same purpose.

See also: [fclose], page 251, [popen], page 777.

`[`*`in`*`, `*`out`*`, `*`pid`*`] = popen2 (`*`command`*`, `*`args`*`)` [Built-in Function]

Start a subprocess with two-way communication.

The name of the process is given by *command*, and *args* is an array of strings containing options for the command.

The file identifiers for the input and output streams of the subprocess are returned in *in* and *out*. If execution of the command is successful, *pid* contains the process ID of the subprocess. Otherwise, *pid* is −1.

For example:

```
[in, out, pid] = popen2 ("sort", "-r");
fputs (in, "these\nare\nsome\nstrings\n");
fclose (in);
EAGAIN = errno ("EAGAIN");
done = false;
do
  s = fgets (out);
  if (ischar (s))
    fputs (stdout, s);
  elseif (errno () == EAGAIN)
    sleep (0.1);
    fclear (out);
  else
    done = true;
  endif
until (done)
fclose (out);
waitpid (pid);

   ⊣ these
   ⊣ strings
   ⊣ some
   ⊣ are
```

Note that `popen2`, unlike `popen`, will not "reap" the child process. If you don't use `waitpid` to check the child's exit status, it will linger until Octave exits.

See also: [popen], page 777, [waitpid], page 780.

val = EXEC_PATH () [Built-in Function]
old_val = EXEC_PATH (*new_val*) [Built-in Function]
EXEC_PATH (*new_val*, "*local*") [Built-in Function]

Query or set the internal variable that specifies a colon separated list of directories to append to the shell PATH when executing external programs.

The initial value of is taken from the environment variable `OCTAVE_EXEC_PATH`, but that value can be overridden by the command line argument `--exec-path PATH`.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [IMAGE_PATH], page 697, [OCTAVE_HOME], page 789.

In most cases, the following functions simply decode their arguments and make the corresponding Unix system calls. For a complete example of how they can be used, look at the definition of the function `popen2`.

[*pid*, *msg*] = fork () [Built-in Function]

Create a copy of the current process.

Fork can return one of the following values:

> 0
: You are in the parent process. The value returned from `fork` is the process id of the child process. You should probably arrange to wait for any child processes to exit.

0
: You are in the child process. You can call `exec` to start another process. If that fails, you should probably call `exit`.

< 0
: The call to `fork` failed for some reason. You must take evasive action. A system dependent error message will be waiting in *msg*.

[*err*, *msg*] = exec (*file*, *args*) [Built-in Function]

Replace current process with a new process.

Calling `exec` without first calling `fork` will terminate your current Octave process and replace it with the program named by *file*. For example,

```
exec ("ls", "-l")
```

will run `ls` and return you to your shell prompt.

If successful, `exec` does not return. If `exec` does return, *err* will be nonzero, and *msg* will contain a system-dependent error message.

[*read_fd*, *write_fd*, *err*, *msg*] = pipe () [Built-in Function]

Create a pipe and return the reading and writing ends of the pipe into *read_fd* and *write_fd* respectively.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [mkfifo], page 762.

`[fid, msg] = dup2 (old, new)` [Built-in Function]

Duplicate a file descriptor.

If successful, *fid* is greater than zero and contains the new file ID. Otherwise, *fid* is negative and *msg* contains a system-dependent error message.

See also: [fopen], page 250, [fclose], page 251, [fcntl], page 782.

`[pid, status, msg] = waitpid (pid, options)` [Built-in Function]

Wait for process *pid* to terminate.

The *pid* argument can be:

-1 Wait for any child process.

0 Wait for any child process whose process group ID is equal to that of the Octave interpreter process.

> 0 Wait for termination of the child process with ID *pid*.

The *options* argument can be a bitwise OR of zero or more of the following constants:

`0` Wait until signal is received or a child process exits (this is the default if the *options* argument is missing).

`WNOHANG` Do not hang if status is not immediately available.

`WUNTRACED`
Report the status of any child processes that are stopped, and whose status has not yet been reported since they stopped.

`WCONTINUE`
Return if a stopped child has been resumed by delivery of `SIGCONT`. This value may not be meaningful on all systems.

If the returned value of *pid* is greater than 0, it is the process ID of the child process that exited. If an error occurs, *pid* will be less than zero and *msg* will contain a system-dependent error message. The value of *status* contains additional system-dependent information about the subprocess that exited.

See also: [WCONTINUE], page 780, [WCOREDUMP], page 780, [WEXITSTATUS], page 781, [WIFCONTINUED], page 781, [WIFSIGNALED], page 781, [WIFSTOPPED], page 781, [WNOHANG], page 781, [WSTOPSIG], page 782, [WTERMSIG], page 782, [WUNTRACED], page 782.

`WCONTINUE ()` [Built-in Function]

Return the numerical value of the option argument that may be passed to `waitpid` to indicate that it should also return if a stopped child has been resumed by delivery of a `SIGCONT` signal.

See also: [waitpid], page 780, [WNOHANG], page 781, [WUNTRACED], page 782.

`WCOREDUMP (status)` [Built-in Function]

Given *status* from a call to `waitpid`, return true if the child produced a core dump.

This function should only be employed if `WIFSIGNALED` returned true. The macro used to implement this function is not specified in POSIX.1-2001 and is not available on some Unix implementations (e.g., AIX, SunOS).

See also: [waitpid], page 780, [WIFEXITED], page 781, [WEXITSTATUS], page 781, [WIFSIGNALED], page 781, [WTERMSIG], page 782, [WIFSTOPPED], page 781, [WSTOPSIG], page 782, [WIFCONTINUED], page 781.

`WEXITSTATUS (`*status*`)` [Built-in Function]

Given *status* from a call to `waitpid`, return the exit status of the child.

This function should only be employed if `WIFEXITED` returned true.

See also: [waitpid], page 780, [WIFEXITED], page 781, [WIFSIGNALED], page 781, [WTERMSIG], page 782, [WCOREDUMP], page 780, [WIFSTOPPED], page 781, [WSTOPSIG], page 782, [WIFCONTINUED], page 781.

`WIFCONTINUED (`*status*`)` [Built-in Function]

Given *status* from a call to `waitpid`, return true if the child process was resumed by delivery of `SIGCONT`.

See also: [waitpid], page 780, [WIFEXITED], page 781, [WEXITSTATUS], page 781, [WIFSIGNALED], page 781, [WTERMSIG], page 782, [WCOREDUMP], page 780, [WIFSTOPPED], page 781, [WSTOPSIG], page 782.

`WIFSIGNALED (`*status*`)` [Built-in Function]

Given *status* from a call to `waitpid`, return true if the child process was terminated by a signal.

See also: [waitpid], page 780, [WIFEXITED], page 781, [WEXITSTATUS], page 781, [WTERMSIG], page 782, [WCOREDUMP], page 780, [WIFSTOPPED], page 781, [WSTOPSIG], page 782, [WIFCONTINUED], page 781.

`WIFSTOPPED (`*status*`)` [Built-in Function]

Given *status* from a call to `waitpid`, return true if the child process was stopped by delivery of a signal.

This is only possible if the call was done using `WUNTRACED` or when the child is being traced (see ptrace(2)).

See also: [waitpid], page 780, [WIFEXITED], page 781, [WEXITSTATUS], page 781, [WIFSIGNALED], page 781, [WTERMSIG], page 782, [WCOREDUMP], page 780, [WSTOPSIG], page 782, [WIFCONTINUED], page 781.

`WIFEXITED (`*status*`)` [Built-in Function]

Given *status* from a call to `waitpid`, return true if the child terminated normally.

See also: [waitpid], page 780, [WEXITSTATUS], page 781, [WIFSIGNALED], page 781, [WTERMSIG], page 782, [WCOREDUMP], page 780, [WIFSTOPPED], page 781, [WSTOPSIG], page 782, [WIFCONTINUED], page 781.

`WNOHANG ()` [Built-in Function]

Return the numerical value of the option argument that may be passed to `waitpid` to indicate that it should return its status immediately instead of waiting for a process to exit.

See also: [waitpid], page 780, [WUNTRACED], page 782, [WCONTINUE], page 780.

`WSTOPSIG (status)` [Built-in Function]

Given *status* from a call to `waitpid`, return the number of the signal which caused the child to stop.

This function should only be employed if `WIFSTOPPED` returned true.

See also: [waitpid], page 780, [WIFEXITED], page 781, [WEXITSTATUS], page 781, [WIFSIGNALED], page 781, [WTERMSIG], page 782, [WCOREDUMP], page 780, [WIFSTOPPED], page 781, [WIFCONTINUED], page 781.

`WTERMSIG (status)` [Built-in Function]

Given *status* from a call to `waitpid`, return the number of the signal that caused the child process to terminate.

This function should only be employed if `WIFSIGNALED` returned true.

See also: [waitpid], page 780, [WIFEXITED], page 781, [WEXITSTATUS], page 781, [WIFSIGNALED], page 781, [WCOREDUMP], page 780, [WIFSTOPPED], page 781, [WSTOPSIG], page 782, [WIFCONTINUED], page 781.

`WUNTRACED ()` [Built-in Function]

Return the numerical value of the option argument that may be passed to `waitpid` to indicate that it should also return if the child process has stopped but is not traced via the `ptrace` system call

See also: [waitpid], page 780, [WNOHANG], page 781, [WCONTINUE], page 780.

`[err, msg] = fcntl (fid, request, arg)` [Built-in Function]

Change the properties of the open file *fid*.

The following values may be passed as *request*:

`F_DUPFD` Return a duplicate file descriptor.

`F_GETFD` Return the file descriptor flags for *fid*.

`F_SETFD` Set the file descriptor flags for *fid*.

`F_GETFL` Return the file status flags for *fid*. The following codes may be returned (some of the flags may be undefined on some systems).

- `O_RDONLY` Open for reading only.
- `O_WRONLY` Open for writing only.
- `O_RDWR` Open for reading and writing.
- `O_APPEND` Append on each write.
- `O_CREAT` Create the file if it does not exist.
- `O_NONBLOCK` Non-blocking mode.
- `O_SYNC` Wait for writes to complete.
- `O_ASYNC` Asynchronous I/O.

`F_SETFL` Set the file status flags for *fid* to the value specified by *arg*. The only flags that can be changed are `O_APPEND` and `O_NONBLOCK`.

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

See also: [fopen], page 250, [dup2], page 780.

`[err, msg] = kill (pid, sig)` [Built-in Function]

Send signal *sig* to process *pid*.

If *pid* is positive, then signal *sig* is sent to *pid*.

If *pid* is 0, then signal *sig* is sent to every process in the process group of the current process.

If *pid* is -1, then signal *sig* is sent to every process except process 1.

If *pid* is less than -1, then signal *sig* is sent to every process in the process group -*pid*.

If *sig* is 0, then no signal is sent, but error checking is still performed.

Return 0 if successful, otherwise return -1.

`SIG ()` [Built-in Function]

Return a structure containing Unix signal names and their defined values.

36.6 Process, Group, and User IDs

`pgid = getpgrp ()` [Built-in Function]

Return the process group id of the current process.

`pid = getpid ()` [Built-in Function]

Return the process id of the current process.

See also: [getppid], page 783.

`pid = getppid ()` [Built-in Function]

Return the process id of the parent process.

See also: [getpid], page 783.

`euid = geteuid ()` [Built-in Function]

Return the effective user id of the current process.

See also: [getuid], page 783.

`uid = getuid ()` [Built-in Function]

Return the real user id of the current process.

See also: [geteuid], page 783.

`egid = getegid ()` [Built-in Function]

Return the effective group id of the current process.

See also: [getgid], page 783.

`gid = getgid ()` [Built-in Function]

Return the real group id of the current process.

See also: [getegid], page 783.

36.7 Environment Variables

`getenv (var)` [Built-in Function]

Return the value of the environment variable *var*.

For example,

```
getenv ("PATH")
```

returns a string containing the value of your path.

See also: [setenv], page 784, [unsetenv], page 784.

`setenv (var, value)` [Built-in Function]
`setenv (var)` [Built-in Function]
`putenv (...)` [Built-in Function]

Set the value of the environment variable *var* to *value*.

If no *value* is specified then the variable will be assigned the null string.

See also: [unsetenv], page 784, [getenv], page 784.

`status = unsetenv (var)` [Built-in Function]

Delete the environment variable *var*.

Return 0 if the variable was deleted, or did not exist, and -1 if an error occurred.

See also: [setenv], page 784, [getenv], page 784.

`homedir = get_home_directory ()` [Built-in Function]

Return the current home directory.

On most systems, this is equivalent to `getenv ("HOME")`. On Windows systems, if the environment variable `HOME` is not set then it is equivalent to `fullfile (getenv ("HOMEDRIVE"), getenv ("HOMEPATH"))`

See also: [getenv], page 784.

36.8 Current Working Directory

`cd dir` [Command]
`cd` [Command]
`old_dir = cd (dir)` [Built-in Function]
`chdir ...` [Command]

Change the current working directory to *dir*.

If *dir* is omitted, the current directory is changed to the user's home directory (`"~"`).

For example,

```
cd ~/octave
```

changes the current working directory to `~/octave`. If the directory does not exist, an error message is printed and the working directory is not changed.

`chdir` is an alias for `cd` and can be used in all of the same calling formats.

Compatibility Note: When called with no arguments, MATLAB prints the present working directory rather than changing to the user's home directory.

See also: [pwd], page 786, [mkdir], page 761, [rmdir], page 762, [dir], page 785, [ls], page 785.

`ls` [Command]
`ls filenames` [Command]
`ls options` [Command]
`ls options filenames` [Command]
`list = ls (...)` [Function File]

List directory contents.

The `ls` command is implemented by calling the native operating system's directory listing command—available *options* will vary from system to system.

Filenames are subject to shell expansion if they contain any wildcard characters '`*`', '`?`', '`[]`'. To find a literal example of a wildcard character the wildcard must be escaped using the backslash operator '`\`'.

If the optional output *list* is requested then `ls` returns a character array with one row for each file/directory name.

Example usage on a UNIX-like system:

```
ls -l
     ⊣ total 12
     ⊣ -rw-r--r--   1 jwe  users  4488 Aug 19 04:02 foo.m
     ⊣ -rw-r--r--   1 jwe  users  1315 Aug 17 23:14 bar.m
```

See also: [dir], page 785, [readdir], page 766, [glob], page 766, [what], page 132, [stat], page 763, [filesep], page 767, [ls_command], page 785.

`val = ls_command ()` [Function File]
`old_val = ls_command (new_val)` [Function File]

Query or set the shell command used by Octave's `ls` command.

See also: [ls], page 785.

`dir` [Function File]
`dir (directory)` [Function File]
`[list] = dir (directory)` [Function File]

Display file listing for directory *directory*.

If *directory* is not specified then list the present working directory.

If a return value is requested, return a structure array with the fields

name	File or directory name.
date	Timestamp of file modification (string value).
bytes	File size in bytes.
isdir	True if name is a directory.
datenum	Timestamp of file modification as serial date number (double).
statinfo	Information structure returned from `stat`.

If *directory* is a filename, rather than a directory, then return information about the named file. *directory* may also be a list rather than a single directory or file.

directory is subject to shell expansion if it contains any wildcard characters '`*`', '`?`', '`[]`'. To find a literal example of a wildcard character the wildcard must be escaped using the backslash operator '`\`'.

Note that for symbolic links, `dir` returns information about the file that the symbolic link points to rather than the link itself. However, if the link points to a nonexistent file, `dir` returns information about the link.

See also: [ls], page 785, [readdir], page 766, [glob], page 766, [what], page 132, [stat], page 763, [lstat], page 763.

`pwd ()` [Built-in Function]
`dir` `= pwd ()` [Built-in Function]

Return the current working directory.

See also: [cd], page 784, [dir], page 785, [ls], page 785, [mkdir], page 761, [rmdir], page 762.

36.9 Password Database Functions

Octave's password database functions return information in a structure with the following fields.

`name` The user name.

`passwd` The encrypted password, if available.

`uid` The numeric user id.

`gid` The numeric group id.

`gecos` The GECOS field.

`dir` The home directory.

`shell` The initial shell.

In the descriptions of the following functions, this data structure is referred to as a *pw_struct*.

`pw_struct` `= getpwent ()` [Built-in Function]

Return a structure containing an entry from the password database, opening it if necessary.

Once the end of the data has been reached, `getpwent` returns 0.

See also: [setpwent], page 787, [endpwent], page 787.

`pw_struct` `= getpwuid (`*`uid`*`).` [Built-in Function]

Return a structure containing the first entry from the password database with the user ID *uid*.

If the user ID does not exist in the database, `getpwuid` returns 0.

See also: [getpwnam], page 786.

`pw_struct` `= getpwnam (`*`name`*`)` [Built-in Function]

Return a structure containing the first entry from the password database with the user name *name*.

If the user name does not exist in the database, `getpwname` returns 0.

See also: [getpwuid], page 786.

`setpwent ()` [Built-in Function]

Return the internal pointer to the beginning of the password database.

See also: [getpwent], page 786, [endpwent], page 787.

`endpwent ()` [Built-in Function]

Close the password database.

See also: [getpwent], page 786, [setpwent], page 787.

36.10 Group Database Functions

Octave's group database functions return information in a structure with the following fields.

`name` The user name.

`passwd` The encrypted password, if available.

`gid` The numeric group id.

`mem` The members of the group.

In the descriptions of the following functions, this data structure is referred to as a *grp_struct*.

grp_struct = `getgrent ()` [Built-in Function]

Return an entry from the group database, opening it if necessary.

Once the end of data has been reached, `getgrent` returns 0.

See also: [setgrent], page 787, [endgrent], page 787.

grp_struct = `getgrgid (`*gid*`).` [Built-in Function]

Return the first entry from the group database with the group ID *gid*.

If the group ID does not exist in the database, `getgrgid` returns 0.

See also: [getgrnam], page 787.

grp_struct = `getgrnam (`*name*`)` [Built-in Function]

Return the first entry from the group database with the group name *name*.

If the group name does not exist in the database, `getgrnam` returns 0.

See also: [getgrgid], page 787.

`setgrent ()` [Built-in Function]

Return the internal pointer to the beginning of the group database.

See also: [getgrent], page 787, [endgrent], page 787.

`endgrent ()` [Built-in Function]

Close the group database.

See also: [getgrent], page 787, [setgrent], page 787.

36.11 System Information

`computer ()` [Function File]
c `= computer ()` [Function File]
`[`*c*`,` *maxsize*`] = computer ()` [Function File]
`[`*c*`,` *maxsize*`,` *endian*`] = computer ()` [Function File]
arch `= computer ("`*arch*`")` [Function File]

Print or return a string of the form *cpu-vendor-os* that identifies the type of computer that Octave is running on.

If invoked with an output argument, the value is returned instead of printed. For example:

```
computer ()
   ⊣ i586-pc-linux-gnu

mycomp = computer ()
   ⇒ mycomp = "i586-pc-linux-gnu"
```

If two output arguments are requested, also return the maximum number of elements for an array. This will depend on whether Octave has been compiled with 32-bit or 64-bit index vectors.

If three output arguments are requested, also return the byte order of the current system as a character (`"B"` for big-endian or `"L"` for little-endian).

If the argument `"arch"` is specified, return a string indicating the architecture of the computer on which Octave is running.

See also: [isunix], page 789, [ismac], page 789, [ispc], page 789.

`[`*uts*`,` *err*`,` *msg*`] = uname ()` [Built-in Function]

Return system information in the structure.

For example:

```
uname ()
   ⇒ {
         sysname = x86_64
         nodename = segfault
         release = 2.6.15-1-amd64-k8-smp
         version = Linux
         machine = #2 SMP Thu Feb 23 04:57:49 UTC 2006
      }
```

If successful, *err* is 0 and *msg* is an empty string. Otherwise, *err* is nonzero and *msg* contains a system-dependent error message.

`nproc ()` [Built-in Function]
`nproc (`*query*`)` [Built-in Function]

Return the current number of available processors.

If called with the optional argument *query*, modify how processors are counted as follows:

`all` total number of processors.

`current` processors available to the current process.

`overridable`
same as `current`, but overridable through the `OMP_NUM_THREADS` environment variable.

ispc () [Function File]

Return true if Octave is running on a Windows system and false otherwise.

See also: [isunix], page 789, [ismac], page 789.

isunix () [Function File]

Return true if Octave is running on a Unix-like system and false otherwise.

See also: [ismac], page 789, [ispc], page 789.

ismac () [Function File]

Return true if Octave is running on a Mac OS X system and false otherwise.

See also: [isunix], page 789, [ispc], page 789.

isieee () [Built-in Function]

Return true if your computer *claims* to conform to the IEEE standard for floating point calculations.

No actual tests are performed.

isdeployed () [Function File]

Return true if the current program has been compiled and is running separately from the Octave interpreter and false if it is running in the Octave interpreter.

Currently, this function always returns false in Octave.

isstudent () [Built-in Function]

Return true if running in the student edition of MATLAB.

`isstudent` always returns false in Octave.

See also: [false], page 61.

OCTAVE_HOME () [Built-in Function]

Return the name of the top-level Octave installation directory.

See also: [EXEC_PATH], page 779, [IMAGE_PATH], page 697.

matlabroot () [Function File]

Return the name of the top-level Octave installation directory.

This is an alias for the function `OCTAVE_HOME` provided for compatibility.

See also: [OCTAVE_HOME], page 789.

OCTAVE_VERSION () [Built-in Function]

Return the version number of Octave as a string.

See also: [ver], page 790, [version], page 790.

`version ()` [Function File]

Return the version number of Octave as a string.

This is an alias for the function `OCTAVE_VERSION` provided for compatibility.

See also: [OCTAVE_VERSION], page 789, [ver], page 790.

`ver` [Function File]
`ver` *Octave* [Function File]
`ver` *package* [Function File]
`v = ver (...)` [Function File]

Display a header containing the current Octave version number, license string, and operating system. The header is followed by a list of installed packages, versions, and installation directories.

Use the package name *package* or Octave to limit the listing to a desired component.

When called with an output argument, return a vector of structures describing Octave and each installed package. The structure includes the following fields.

`Name` Package name.

`Version` Version of the package.

`Revision` Revision of the package.

`Date` Date of the version/revision.

See also: [version], page 790, [octave_config_info], page 791, [usejava], page 797, [pkg], page 807.

`compare_versions (v1, v2, operator)` [Function File]

Compare two version strings using the given *operator*.

This function assumes that versions *v1* and *v2* are arbitrarily long strings made of numeric and period characters possibly followed by an arbitrary string (e.g., `"1.2.3"`, `"0.3"`, `"0.1.2+"`, or `"1.2.3.4-test1"`).

The version is first split into numeric and character portions and then the parts are padded to be the same length (i.e., `"1.1"` would be padded to be `"1.1.0"` when being compared with `"1.1.1"`, and separately, the character parts of the strings are padded with nulls).

The operator can be any logical operator from the set

- `"=="` equal
- `"<"` less than
- `"<="` less than or equal to
- `">"` greater than
- `">="` greater than or equal to
- `"!="` not equal
- `"~="` not equal

Note that version `"1.1-test2"` will compare as greater than `"1.1-test10"`. Also, since the numeric part is compared first, `"a"` compares less than `"1a"` because the second string starts with a numeric part even though `double ("a")` is greater than `double ("1")`.

`license` [Command]
`license` *inuse* [Command]
`license` *inuse* ***feature*** [Command]
`license ("`*inuse*`")` [Function File]
retval `= license ("`*inuse*`")` [Function File]
retval `= license ("`*test*`",` ***feature***`)` [Function File]
retval `= license ("`*checkout*`",` ***feature***`)` [Function File]
`[`***retval***`,` ***errmsg***`] = license ("`*checkout*`",` ***feature***`)` [Function File]

Get license information for Octave and Octave packages.

GNU Octave is free software distributed under the GNU General Public License (GPL), and a license manager makes no sense. This function is provided only for MATLAB compatibility.

When called with no extra input arguments, it returns the Octave license, otherwise the first input defines the operation mode and must be one of the following strings: `inuse`, `test`, and `checkout`. The optional *feature* argument can either be `"octave"` (core), or an Octave package.

`"inuse"` Returns a list of loaded features, i.e., octave and the list of loaded packages. If an output is requested, it returns a struct array with the fields `"feature"`, and `"user"`.

`"test"` Return true if the specified *feature* is installed, false otherwise.

An optional third argument `"enable"` or `"disable"` is accepted but ignored.

`"checkout"`
Return true if the specified *feature* is installed, false otherwise. An optional second output will have an error message if a package is not installed.

See also: [pkg], page 807, [ver], page 790, [version], page 790.

`octave_config_info ()` [Built-in Function]
`octave_config_info (`***option***`)` [Built-in Function]

Return a structure containing configuration and installation information for Octave.

If *option* is a string, return the configuration information for the specified option.

See also: [computer], page 788.

`getrusage ()` [Built-in Function]

Return a structure containing a number of statistics about the current Octave process.

Not all fields are available on all systems. If it is not possible to get CPU time statistics, the CPU time slots are set to zero. Other missing data are replaced by NaN. The list of possible fields is:

`idrss` Unshared data size.

`inblock` Number of block input operations.

`isrss` Unshared stack size.

`ixrss` Shared memory size.

`majflt` Number of major page faults.

`maxrss` Maximum data size.

`minflt` Number of minor page faults.

`msgrcv` Number of messages received.

`msgsnd` Number of messages sent.

`nivcsw` Number of involuntary context switches.

`nsignals` Number of signals received.

`nswap` Number of swaps.

`nvcsw` Number of voluntary context switches.

`oublock` Number of block output operations.

`stime` A structure containing the system CPU time used. The structure has the elements `sec` (seconds) `usec` (microseconds).

`utime` A structure containing the user CPU time used. The structure has the elements `sec` (seconds) `usec` (microseconds).

36.12 Hashing Functions

It is often necessary to find if two strings or files are identical. This might be done by comparing them character by character and looking for differences. However, this can be slow, and so comparing a hash of the string or file can be a rapid way of finding if the files differ.

Another use of the hashing function is to check for file integrity. The user can check the hash of the file against a known value and find if the file they have is the same as the one that the original hash was produced with.

Octave supplies the `md5sum` function to perform MD5 hashes on strings and files. An example of the use of `md5sum` function might be

```
if exist (file, "file")
  hash = md5sum (file);
else
  # Treat the variable "file" as a string
  hash = md5sum (file, true);
endif
```

`md5sum` (*file*) [Built-in Function]
`md5sum` (*str*, *opt*) [Built-in Function]

Calculate the MD5 sum of the file *file*.

If the second parameter *opt* exists and is true, then calculate the MD5 sum of the string *str*.

37 Java Interface

The Java Interface is designed for calling Java functions from within Octave. If you want to do the reverse, and call Octave from within Java, try a library like `javaOctave` (`http://kenai.com/projects/javaOctave`) or `joPas` (`http://jopas.sourceforge.net`).

37.1 Java Interface Functions

The following functions are the core of the Java Interface. They provide a way to create a Java object, get and set its data fields, and call Java methods which return results to Octave.

jobj = javaObject (*classname*) [Built-in Function]
jobj = javaObject (*classname*, *arg1*, ...) [Built-in Function]

Create a Java object of class *classsname*, by calling the class constructor with the arguments *arg1*, ...

The first example below creates an uninitialized object, while the second example supplies an initial argument to the constructor.

```
x = javaObject ("java.lang.StringBuffer")
x = javaObject ("java.lang.StringBuffer", "Initial string")
```

See also: [javaMethod], page 795, [javaArray], page 793.

jary = javaArray (*classname*, *sz*) [Function File]
jary = javaArray (*classname*, *m*, *n*, ...) [Function File]

Create a Java array of size *sz* with elements of class *classname*.

classname may be a Java object representing a class or a string containing the fully qualified class name. The size of the object may also be specified with individual integer arguments *m*, *n*, etc.

The generated array is uninitialized. All elements are set to null if *classname* is a reference type, or to a default value (usually 0) if *classname* is a primitive type.

Sample code:

```
jary = javaArray ("java.lang.String", 2, 2);
jary(1,1) = "Hello";
```

See also: [javaObject], page 793.

There are many different variable types in Octave but only ones created through `javaObject` can use Java functions. Before using Java with an unknown object the type can be checked with `isjava`.

isjava (*x*) [Built-in Function]

Return true if *x* is a Java object.

See also: [class], page 39, [typeinfo], page 39, [isa], page 39, [javaObject], page 793.

Once an object has been created it is natural to find out what fields the object has and to read (get) and write (set) them.

In Octave the `fieldnames` function for structures has been overloaded to return the fields of a Java object. For example:

```
dobj = javaObject ("java.lang.Double", pi);
fieldnames (dobj)
⇒
{
  [1,1] = public static final double java.lang.Double.POSITIVE_INFINITY
  [1,2] = public static final double java.lang.Double.NEGATIVE_INFINITY
  [1,3] = public static final double java.lang.Double.NaN
  [1,4] = public static final double java.lang.Double.MAX_VALUE
  [1,5] = public static final double java.lang.Double.MIN_NORMAL
  [1,6] = public static final double java.lang.Double.MIN_VALUE
  [1,7] = public static final int java.lang.Double.MAX_EXPONENT
  [1,8] = public static final int java.lang.Double.MIN_EXPONENT
  [1,9] = public static final int java.lang.Double.SIZE
  [1,10] = public static final java.lang.Class java.lang.Double.TYPE
}
```

The analogy of objects with structures is carried over into reading and writing object fields. To read a field the object is indexed with the '`.`' operator from structures. This is the preferred method for reading fields, but Octave also provides a function interface to read fields with `java_get`. An example of both styles is shown below.

```
dobj = javaObject ("java.lang.Double", pi);
dobj.MAX_VALUE
⇒  1.7977e+308
java_get ("java.lang.Float", "MAX_VALUE")
⇒  3.4028e+38
```

val = `java_get` (*obj*, *name*) [Function File]

Get the value of the field *name* of the Java object *obj*.

For static fields, *obj* can be a string representing the fully qualified name of the corresponding class.

When *obj* is a regular Java object, structure-like indexing can be used as a shortcut syntax. For instance, the following two statements are equivalent

```
java_get (x, "field1")
x.field1
```

See also: [java_set], page 794, [javaMethod], page 795, [javaObject], page 793.

obj = `java_set` (*obj*, *name*, *val*) [Function File]

Set the value of the field *name* of the Java object *obj* to *val*.

For static fields, *obj* can be a string representing the fully qualified named of the corresponding Java class.

When *obj* is a regular Java object, structure-like indexing can be used as a shortcut syntax. For instance, the following two statements are equivalent

```
java_set (x, "field1", val)
x.field1 = val
```

See also: [java_get], page 794, [javaMethod], page 795, [javaObject], page 793.

To see what functions can be called with an object use `methods`. For example, using the previously created *dobj*:

```
methods (dobj)
⇒
Methods for class java.lang.Double:
boolean equals(java.lang.Object)
java.lang.String toString(double)
java.lang.String toString()
...
```

To call a method of an object the same structure indexing operator '`.`' is used. Octave also provides a functional interface to calling the methods of an object through `javaMethod`. An example showing both styles is shown below.

```
dobj = javaObject ("java.lang.Double", pi);
dobj.equals (3)
⇒  0
javaMethod ("equals", dobj, pi)
⇒  1
```

ret = javaMethod (*methodname*, *obj*) [Built-in Function]
ret = javaMethod (*methodname*, *obj*, *arg1*, ...) [Built-in Function]

Invoke the method *methodname* on the Java object *obj* with the arguments *arg1*, ...

For static methods, *obj* can be a string representing the fully qualified name of the corresponding class.

When *obj* is a regular Java object, structure-like indexing can be used as a shortcut syntax. For instance, the two following statements are equivalent

```
ret = javaMethod ("method1", x, 1.0, "a string")
ret = x.method1 (1.0, "a string")
```

`javaMethod` returns the result of the method invocation.

See also: [methods], page 722, [javaObject], page 793.

The following three functions are used to display and modify the class path used by the Java Virtual Machine. This is entirely separate from Octave's PATH variable and is used by the JVM to find the correct code to execute.

javaclasspath () [Function File]
dpath = javaclasspath () [Function File]
[*dpath*, *spath*] = javaclasspath () [Function File]
clspath = javaclasspath (*what*) [Function File]

Return the class path of the Java virtual machine in the form of a cell array of strings.

If called with no inputs:

- If no output is requested, the dynamic and static classpaths are printed to the standard output.
- If one output value *dpath* is requested, the result is the dynamic classpath.

- If two output valuesdpath and *spath* are requested, the first variable will contain the dynamic classpath and the second will contain the static classpath.

If called with a single input parameter *what*:

`"-dynamic"`
: Return the dynamic classpath.

`"-static"`
: Return the static classpath.

`"-all"`
: Return both the static and dynamic classpath in a single cellstr.

See also: [javaaddpath], page 796, [javarmpath], page 796.

`javaaddpath (`*`clspath`*`)` [Function File]
`javaaddpath (`*`clspath1`*`, ...)` [Function File]

Add *clspath* to the dynamic class path of the Java virtual machine.

clspath may either be a directory where `.class` files are found, or a `.jar` file containing Java classes. Multiple paths may be added at once by specifying additional arguments.

See also: [javarmpath], page 796, [javaclasspath], page 795.

`javarmpath (`*`clspath`*`)` [Function File]
`javarmpath (`*`clspath1`*`, ...)` [Function File]

Remove *clspath* from the dynamic class path of the Java virtual machine.

clspath may either be a directory where `.class` files are found, or a `.jar` file containing Java classes. Multiple paths may be removed at once by specifying additional arguments.

See also: [javaaddpath], page 796, [javaclasspath], page 795.

The following functions provide information and control over the interface between Octave and the Java Virtual Machine.

`javachk (`*`feature`*`)` [Function File]
`javachk (`*`feature`*`, `*`component`*`)` [Function File]
`msg`` = javachk (...)` [Function File]

Check for the presence of the Java *feature* in the current session and print or return an error message if it is not.

Possible features are:

`"awt"`
: Abstract Window Toolkit for GUIs.

`"desktop"`
: Interactive desktop is running.

`"jvm"`
: Java Virtual Machine.

`"swing"`
: Swing components for lightweight GUIs.

If *feature* is supported and

- no output argument is requested:

  Return an empty string

- an output argument is requested:

  Return a struct with fields `"feature"` and `"identifier"` both empty

If *feature* is not supported and

- no output argument is requested:

  Emit an error message

- an output argument is requested:

  Return a struct with field `"feature"` set to *feature* and field `"identifier"` set to *component*

The optional input *component* will be used in place of *feature* in any error messages for greater specificity.

`javachk` determines if specific Java features are available in an Octave session. This function is provided for scripts which may alter their behavior based on the availability of Java. The feature `"desktop"` is never available as Octave has no Java-based desktop. Other features may be available if Octave was compiled with the Java Interface and Java is installed.

See also: [usejava], page 797, [error], page 205.

`usejava (`*feature*`)` [Function File]

Return true if the Java element *feature* is available.

Possible features are:

`"awt"` Abstract Window Toolkit for GUIs.

`"desktop"`
Interactive desktop is running.

`"jvm"` Java Virtual Machine.

`"swing"` Swing components for lightweight GUIs.

`usejava` determines if specific Java features are available in an Octave session. This function is provided for scripts which may alter their behavior based on the availability of Java. The feature `"desktop"` always returns `false` as Octave has no Java-based desktop. Other features may be available if Octave was compiled with the Java Interface and Java is installed.

See also: [javachk], page 796.

`javamem ()` [Function File]

jmem `= javamem ()` [Function File]

Show the current memory usage of the Java virtual machine (JVM) and run the garbage collector.

When no return argument is given the info is printed to the screen. Otherwise, the output cell array *jmem* contains Maximum, Total, and Free memory (in bytes).

All Java-based routines are run in the JVM's shared memory pool, a dedicated and separate part of memory claimed by the JVM from your computer's total memory (which comprises physical RAM and virtual memory / swap space on hard disk).

The maximum allowable memory usage can be configured using the file `java.opts`. The directory where this file resides is determined by the environment variable `OCTAVE_JAVA_DIR`. If unset, the directory where `javaaddpath.m` resides is used instead (typically `OCTAVE_HOME/share/octave/OCTAVE_VERSION/m/java/`).

`java.opts` is a plain text file with one option per line. The default initial memory size and default maximum memory size (which are both system dependent) can be overridden like so:

-Xms64m

-Xmx512m

(in megabytes in this example). You can adapt these values to your own requirements if your system has limited available physical memory or if you get Java memory errors.

`"Total memory"` is what the operating system has currently assigned to the JVM and depends on actual and active memory usage. `"Free memory"` is self-explanatory. During operation of Java-based Octave functions the amount of Total and Free memory will vary, due to Java's own cleaning up and your operating system's memory management.

val = `java_matrix_autoconversion ()` [Built-in Function]
old_val = `java_matrix_autoconversion (`*new_val*`)` [Built-in Function]
`java_matrix_autoconversion (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that controls whether Java arrays are automatically converted to Octave matrices.

The default value is false.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [java_unsigned_autoconversion], page 798, [debug_java], page 798.

val = `java_unsigned_autoconversion ()` [Built-in Function]
old_val = `java_unsigned_autoconversion (`*new_val*`)` [Built-in Function]
`java_unsigned_autoconversion (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that controls how integer classes are converted when `java_matrix_autoconversion` is enabled.

When enabled, Java arrays of class Byte or Integer are converted to matrices of class uint8 or uint32 respectively. The default value is true.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [java_matrix_autoconversion], page 798, [debug_java], page 798.

val = `debug_java ()` [Built-in Function]
old_val = `debug_java (`*new_val*`)` [Built-in Function]
`debug_java (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that determines whether extra debugging information regarding the initialization of the JVM and any Java exceptions is printed.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [java_matrix_autoconversion], page 798, [java_unsigned_autoconversion], page 798.

37.2 Dialog Box Functions

The following functions all use the Java Interface to provide some form of dialog box.

h = `msgbox` (*msg*) [Function File]
h = `msgbox` (*msg*, *title*) [Function File]
h = `msgbox` (*msg*, *title*, *icon*) [Function File]
h = `msgbox` (. . ., *createmode*) [Function File]

Display *msg* using a message dialog box.

The message may have multiple lines separated by newline characters (`"\n"`), or it may be a cellstr array with one element for each line.

The optional input *title* (character string) can be used to decorate the dialog caption.

The optional argument *icon* selects a dialog icon. It can be one of `"none"` (default), `"error"`, `"help"`, or `"warn"`.

The return value is always 1.

Compatibility Note: The optional argument *createmode* is accepted for MATLAB compatibility, but is not implemented.

See also: [errordlg], page 799, [helpdlg], page 799, [inputdlg], page 800, [listdlg], page 800, [questdlg], page 801, [warndlg], page 802.

h = `errordlg` (*msg*) [Function File]
h = `errordlg` (*msg*, *title*) [Function File]
h = `errordlg` (*msg*, *title*, *createmode*) [Function File]

Display *msg* using an error dialog box.

The message may have multiple lines separated by newline characters (`"\n"`), or it may be a cellstr array with one element for each line.

The optional input *title* (character string) can be used to set the dialog caption. The default title is `"Error Dialog"`.

The return value is always 1.

Compatibility Note: The optional argument *createmode* is accepted for MATLAB compatibility, but is not implemented.

See also: [helpdlg], page 799, [inputdlg], page 800, [listdlg], page 800, [msgbox], page 799, [questdlg], page 801, [warndlg], page 802.

h = `helpdlg` (*msg*) [Function File]
h = `helpdlg` (*msg*, *title*) [Function File]

Display *msg* in a help dialog box.

The message may have multiple lines separated by newline characters (`"\n"`), or it may be a cellstr array with one element for each line.

The optional input *title* (character string) can be used to set the dialog caption. The default title is `"Help Dialog"`.

The return value is always 1.

See also: [errordlg], page 799, [inputdlg], page 800, [listdlg], page 800, [msgbox], page 799, [questdlg], page 801, [warndlg], page 802.

cstr = `inputdlg` (*prompt*) [Function File]
cstr = `inputdlg` (*prompt*, *title*) [Function File]
cstr = `inputdlg` (*prompt*, *title*, *rowscols*) [Function File]
cstr = `inputdlg` (*prompt*, *title*, *rowscols*, *defaults*) [Function File]

Return user input from a multi-textfield dialog box in a cell array of strings, or an empty cell array if the dialog is closed by the Cancel button.

Inputs:

prompt A cell array with strings labeling each text field. This input is required.

title String to use for the caption of the dialog. The default is `"Input Dialog"`.

rowscols Specifies the size of the text fields and can take three forms:

1. a scalar value which defines the number of rows used for each text field.
2. a vector which defines the individual number of rows used for each text field.
3. a matrix which defines the individual number of rows and columns used for each text field. In the matrix each row describes a single text field. The first column specifies the number of input rows to use and the second column specifies the text field width.

defaults A list of default values to place in each text fields. It must be a cell array of strings with the same size as *prompt*.

See also: [errordlg], page 799, [helpdlg], page 799, [listdlg], page 800, [msgbox], page 799, [questdlg], page 801, [warndlg], page 802.

[*sel*, *ok*] = `listdlg` (*key*, *value*, ...) [Function File]

Return user inputs from a list dialog box in a vector of selection indices *sel* and a flag *ok* indicating how the user closed the dialog box.

The value of *ok* is 1 if the user closed the box with the OK button, otherwise it is 0 and *sel* is empty.

The indices in *sel* are 1-based.

The arguments are specified in form of *key*, *value* pairs. The `"ListString"` argument pair must be specified.

Valid *key* and *value* pairs are:

`"ListString"`
a cell array of strings comprising the content of the list.

`"SelectionMode"`
can be either `"Single"` or `"Multiple"` (default).

`"ListSize"`
: a vector with two elements *width* and *height* defining the size of the list field in pixels. Default is [160 300].

`"InitialValue"`
: a vector containing 1-based indices of preselected elements. Default is 1 (first item).

`"Name"`
: a string to be used as the dialog caption. Default is `""`.

`"PromptString"`
: a cell array of strings to be displayed above the list field. Default is {}.

`"OKString"`
: a string used to label the OK button. Default is `"OK"`.

`"CancelString"`
: a string used to label the Cancel button. Default is `"Cancel"`.

Example:

```
[sel, ok] = listdlg ("ListString", {"An item", "another", "yet another"},
                     "SelectionMode", "Multiple");
if (ok == 1)
  for i = 1:numel (sel)
    disp (sel(i));
  endfor
endif
```

See also: [menu], page 237, [errordlg], page 799, [helpdlg], page 799, [inputdlg], page 800, [msgbox], page 799, [questdlg], page 801, [warndlg], page 802.

btn = **questdlg** (*msg*) [Function File]
btn = **questdlg** (*msg*, *title*) [Function File]
btn = **questdlg** (*msg*, *title*, *default*) [Function File]
btn = **questdlg** (*msg*, *title*, *btn1*, *btn2*, *default*) [Function File]
btn = **questdlg** (*msg*, *title*, *btn1*, *btn2*, *btn3*, *default*) [Function File]

Display *msg* using a question dialog box and return the caption of the activated button.

The dialog may contain two or three buttons which will all close the dialog.

The message may have multiple lines separated by newline characters (`"\n"`), or it may be a cellstr array with one element for each line.

The optional *title* (character string) can be used to decorate the dialog caption.

The string *default* identifies the default button, which is activated by pressing the `ENTER` key. It must match one of the strings given in *btn1*, *btn2*, or *btn3*.

If only *msg* and *title* are specified, three buttons with the default captions `"Yes"`, `"No"`, and `"Cancel"` are used.

If only two button captions, *btn1* and *btn2*, are specified the dialog will have only these two buttons.

See also: [errordlg], page 799, [helpdlg], page 799, [inputdlg], page 800, [listdlg], page 800, [warndlg], page 802.

h = `warndlg` (*msg*) [Function File]
h = `warndlg` (*msg*, *title*) [Function File]
h = `warndlg` (*msg*, *title*, *createmode*) [Function File]

Display *msg* using a warning dialog box.

The message may have multiple lines separated by newline characters ("\n"), or it may be a cellstr array with one element for each line.

The optional input *title* (character string) can be used to set the dialog caption. The default title is "Warning Dialog".

The return value is always 1.

Compatibility Note: The optional argument *createmode* is accepted for MATLAB compatibility, but is not implemented.

See also: [helpdlg], page 799, [inputdlg], page 800, [listdlg], page 800, [questdlg], page 801.

37.3 FAQ - Frequently asked Questions

37.3.1 How to distinguish between Octave and Matlab?

Octave and MATLAB are very similar, but handle Java slightly different. Therefore it may be necessary to detect the environment and use the appropriate functions. The following function can be used to detect the environment. Due to the persistent variable it can be called repeatedly without a heavy performance hit.

Example:

```
%%
%% Return: true if the environment is Octave.
%%
function retval = isOctave
  persistent cacheval;  % speeds up repeated calls

  if isempty (cacheval)
    cacheval = (exist ("OCTAVE_VERSION", "builtin") > 0);
  end

  retval = cacheval;
end
```

37.3.2 How to make Java classes available to Octave?

Java finds classes by searching a *classpath*. This is a list of Java archive files and/or directories containing class files. In Octave the *classpath* is composed of two parts:

- the *static classpath* is initialized once at startup of the JVM, and
- the *dynamic classpath* which can be modified at runtime.

Octave searches the *static classpath* first, then the *dynamic classpath*. Classes appearing in the *static* as well as in the *dynamic classpath* will therefore be found in the *static classpath* and loaded from this location. Classes which will be used frequently or must be available to all users should be added to the *static classpath*. The *static classpath* is populated once from the contents of a plain text file named `javaclasspath.txt` (or `classpath.txt` historically) when the Java Virtual Machine starts. This file contains one line for each individual classpath to be added to the *static classpath*. These lines can identify single class files, directories containing class files, or Java archives with complete class file hierarchies. Comment lines starting with a '#' or a '%' character are ignored.

The search rules for the file `javaclasspath.txt` (or `classpath.txt`) are:

- First, Octave tries to locate it in the current directory (where Octave was started from). If such a file is found, it is read and defines the initial *static classpath*. Thus, it is possible to define a static classpath on a 'per Octave invocation' basis.
- Next, Octave searches in the user's home directory. If a file `javaclasspath.txt` exists here, its contents are appended to the static classpath (if any). Thus, it is possible to build an initial static classpath on a 'per user' basis.
- Finally, Octave looks for a next occurrence of file `javaclasspath.txt` in the m-files directory where Octave Java functions live. This is where `javaclasspath.m` resides, usually something like `OCTAVE_HOME/share/octave/OCTAVE_VERSION/m/java/`. You can find this directory by executing the command

  ```
  which javaclasspath
  ```

  If this file exists here, its contents are also appended to the static classpath. Note that the archives and class directories defined in this last step will affect all users.

Classes which are used only by a specific script should be placed in the *dynamic classpath*. This portion of the classpath can be modified at runtime using the `javaaddpath` and `javarmpath` functions.

Example:

```
octave> base_path = "C:/Octave/java_files";

octave> % add two JARchives to the dynamic classpath
octave> javaaddpath ([base_path, "/someclasses.jar"]);
octave> javaaddpath ([base_path, "/moreclasses.jar"]);

octave> % check the dynamic classpath
octave> p = javaclasspath;
octave> disp (p{1});
C:/Octave/java_files/someclasses.jar
octave> disp (p{2});
C:/Octave/java_files/moreclasses.jar

octave> % remove the first element from the classpath
octave> javarmpath ([base_path, "/someclasses.jar"]);
octave> p = javaclasspath;
octave> disp (p{1});
C:/Octave/java_files/moreclasses.jar

octave> % provoke an error
octave> disp (p{2});
error: A(I): Index exceeds matrix dimension.
```

Another way to add files to the *dynamic classpath* exclusively for your user account is to use the file .octaverc which is stored in your home directory. All Octave commands in this file are executed each time you start a new instance of Octave. The following example adds the directory octave to Octave's search path and the archive myclasses.jar in this directory to the Java search path.

```
% contents of .octaverc:
addpath ("~/octave");
javaaddpath ("~/octave/myclasses.jar");
```

37.3.3 How to create an instance of a Java class?

The function javaObject can be used to create Java objects..

Example:

```
Passenger = javaObject ("package.FirstClass", row, seat);
```

37.3.4 How can I handle memory limitations?

In order to execute Java code Octave creates a Java Virtual Machine (JVM). Such a JVM allocates a fixed amount of initial memory and may expand this pool up to a fixed maximum memory limit. The default values depend on the Java version (see [javamem], page 797). The memory pool is shared by all Java objects running in the JVM. This strict memory limit is intended mainly to avoid that runaway applications inside web browsers or in enterprise servers can consume all memory and crash the system. When the maximum memory limit is hit, Java code will throw exceptions so that applications will fail or behave unexpectedly.

You can specify options for the creation of the JVM inside a file named java.opts. This is a text file where you can enter lines containing -X and -D options handed to the JVM during initialization.

The directory where the Java options file is located is specified by the environment variable OCTAVE_JAVA_DIR. If unset the directory where javaclasspath.m resides is used instead (typically OCTAVE_HOME/share/octave/OCTAVE_VERSION/m/java/). You can find this directory by executing

```
which javaclasspath
```

The -X options allow you to increase the maximum amount of memory available to the JVM. The following example allows up to 256 Megabytes to be used by adding the following line to the java.opts file:

```
-Xmx256m
```

The maximum possible amount of memory depends on your system. On a Windows system with 2 Gigabytes main memory you should be able to set this maximum to about 1 Gigabyte.

If your application requires a large amount of memory from the beginning, you can also specify the initial amount of memory allocated to the JVM. Adding the following line to the java.opts file starts the JVM with 64 Megabytes of initial memory:

```
-Xms64m
```

For more details on the available -X options of your Java Virtual Machine issue the command 'java -X' at the operating system command prompt and consult the Java documentation.

The `-D` options can be used to define system properties which can then be used by Java classes inside Octave. System properties can be retrieved by using the `getProperty()` methods of the `java.lang.System` class. The following example line defines the property *MyProperty* and assigns it the string `12.34`.

```
-DMyProperty=12.34
```

The value of this property can then be retrieved as a string by a Java object or in Octave:

```
octave> javaMethod ("getProperty", "java.lang.System", "MyProperty");
ans = 12.34
```

See also: javamem.

37.3.5 Which TEX symbols are implemented in dialog functions?

The dialog functions contain a translation table for TEX like symbol codes. Thus messages and labels can be tailored to show some common mathematical symbols or Greek characters. No further TEX formatting codes are supported. The characters are translated to their Unicode equivalent. However, not all characters may be displayable on your system. This depends on the font used by the Java system on your computer.

Each TEX symbol code must be terminated by a space character to make it distinguishable from the surrounding text. Therefore the string '`\alpha =12.0`' will produce the desired result, whereas '`\alpha=12.0`' would produce the literal text '*\alpha=12.0*'.

See also: errordlg, helpdlg, inputdlg, listdlg, msgbox, questdlg, warndlg.

The table below shows each TEX character code and the corresponding Unicode character:

TEX code	**Symbol**	**TEX code**	**Symbol**	**TEX code**	**Symbol**
\alpha	'α'	\beta	'β'	\gamma	'γ'
\delta	'δ'	\epsilon	'ϵ'	\zeta	'ζ'
\eta	'η'	\theta	'θ'	\vartheta	'ϑ'
\iota	'ι'	\kappa	'κ'	\lambda	'λ'
\mu	'μ'	\nu	'ν'	\xi	'ξ'
\pi	'π'	\rho	'ρ'	\sigma	'σ'
\varsigma	'ς'	\tau	'τ'	\phi	'ϕ'
\chi	'χ'	\psi	'ψ'	\omega	'ω'
\upsilon	'υ'	\Gamma	'Γ'	\Delta	'Δ'
\Theta	'Θ'	\Lambda	'Λ'	\Pi	'Π'
\Xi	'Ξ'	\Sigma	'Σ'	\Upsilon	'Υ'
\Phi	'Φ'	\Psi	'Ψ'	\Omega	'Ω'
\Im	'$\Im$'	\Re	'$\Re$'	\leq	'$\leq$'
\geq	'$\geq$'	\neq	'$\neq$'	\pm	'$\pm$'
\infty	'∞'	\partial	'∂'	\approx	'$\approx$'
\circ	'$\circ$'	\bullet	'$\bullet$'	\times	'$\times$'
\sim	'$\sim$'	\nabla	'∇'	\ldots	'$\ldots$'
\exists	'$\exists$'	\neg	'$\neg$'	\aleph	'$\aleph$'
\forall	'$\forall$'	\cong	'$\cong$'	\wp	'$\wp$'
\propto	'$\propto$'	\otimes	'$\otimes$'	\oplus	'$\oplus$'
\oslash	'$\oslash$'	\cap	'$\cap$'	\cup	'$\cup$'
\ni	'$\ni$'	\in	'$\in$'	\div	'$\div$'
\equiv	'$\equiv$'	\int	'$\int$'	\perp	'$\perp$'
\wedge	'$\wedge$'	\vee	'$\vee$'	\supseteq	'$\supseteq$'
\supset	'$\supset$'	\subseteq	'$\subseteq$'	\subset	'$\subset$'
\clubsuit	'$\clubsuit$'	\spadesuit	'$\spadesuit$'	\heartsuit	'$\heartsuit$'
\diamondsuit	'$\diamondsuit$'	\copyright	'©'	\leftarrow	'$\leftarrow$'
\uparrow	'$\uparrow$'	\rightarrow	'$\rightarrow$'	\downarrow	'$\downarrow$'
\leftrightarrow	'$\leftrightarrow$'	\updownarrow	'$\updownarrow$'		

Table: TEX character codes and the resulting symbols.

38 Packages

Since Octave is Free Software users are encouraged to share their programs amongst each other. To aid this sharing Octave supports the installation of extra packages. The 'Octave-Forge' project is a community-maintained set of packages that can be downloaded and installed in Octave. At the time of writing the 'Octave-Forge' project can be found online at `http://octave.sourceforge.net`, but since the Internet is an ever-changing place this may not be true at the time of reading. Therefore it is recommended to see the Octave website for an updated reference.

38.1 Installing and Removing Packages

Assuming a package is available in the file `image-1.0.0.tar.gz` it can be installed from the Octave prompt with the command

```
pkg install image-1.0.0.tar.gz
```

If the package is installed successfully nothing will be printed on the prompt, but if an error occurred during installation it will be reported. It is possible to install several packages at once by writing several package files after the `pkg install` command. If a different version of the package is already installed it will be removed prior to installing the new package. This makes it easy to upgrade and downgrade the version of a package, but makes it impossible to have several versions of the same package installed at once.

To see which packages are installed type

```
pkg list
⊣ Package Name  | Version | Installation directory
⊣ --------------+---------+-----------------------
⊣        image *|   1.0.0 | /home/jwe/octave/image-1.0.0
```

In this case only version 1.0.0 of the `image` package is installed. The `'*'` character next to the package name shows that the image package is loaded and ready for use.

It is possible to remove a package from the system using the `pkg uninstall` command like this

```
pkg uninstall image
```

If the package is removed successfully nothing will be printed in the prompt, but if an error occurred it will be reported. It should be noted that the package file used for installation is not needed for removal, and that only the package name as reported by `pkg list` should be used when removing a package. It is possible to remove several packages at once by writing several package names after the `pkg uninstall` command.

To minimize the amount of code duplication between packages it is possible that one package depends on another one. If a package depends on another, it will check if that package is installed during installation. If it is not, an error will be reported and the package will not be installed. This behavior can be disabled by passing the `-nodeps` flag to the `pkg install` command

```
pkg install -nodeps my_package_with_dependencies.tar.gz
```

Since the installed package expects its dependencies to be installed it may not function correctly. Because of this it is not recommended to disable dependency checking.

`pkg` *command pkg_name* [Command]
`pkg` *command option pkg_name* [Command]

Manage packages (groups of add-on functions) for Octave.

Different actions are available depending on the value of *command*.

Available commands:

'`install`' Install named packages. For example,

```
pkg install image-1.0.0.tar.gz
```

installs the package found in the file `image-1.0.0.tar.gz`.

The *option* variable can contain options that affect the manner in which a package is installed. These options can be one or more of

`-nodeps` The package manager will disable dependency checking. With this option it is possible to install a package even when it depends on another package which is not installed on the system. **Use this option with care.**

`-noauto` The package manager will not automatically load the installed package when starting Octave. This overrides any setting within the package.

`-auto` The package manager will automatically load the installed package when starting Octave. This overrides any setting within the package.

`-local` A local installation (package available only to current user) is forced, even if the user has system privileges.

`-global` A global installation (package available to all users) is forced, even if the user doesn't normally have system privileges.

`-forge` Install a package directly from the Octave-Forge repository. This requires an internet connection and the cURL library.

`-verbose` The package manager will print the output of all commands as they are performed.

'`update`' Check installed Octave-Forge packages against repository and update any outdated items. This requires an internet connection and the cURL library. Usage:

```
pkg update
```

'`uninstall`'

Uninstall named packages. For example,

```
pkg uninstall image
```

removes the `image` package from the system. If another installed package depends on the `image` package an error will be issued. The package can be uninstalled anyway by using the `-nodeps` option.

'`load`' Add named packages to the path. After loading a package it is possible to use the functions provided by the package. For example,

```
pkg load image
```

adds the `image` package to the path. It is possible to load all installed packages at once with the keyword '`all`'. Usage:

```
pkg load all
```

'`unload`' Remove named packages from the path. After unloading a package it is no longer possible to use the functions provided by the package. It is possible to unload all installed packages at once with the keyword '`all`'. Usage:

```
pkg unload all
```

'`list`' Show the list of currently installed packages. For example,

```
pkg list
```

will produce a short report with the package name, version, and installation directory for each installed package. Supply a package name to limit reporting to a particular package. For example:

```
pkg list image
```

If a single return argument is requested then `pkg` returns a cell array where each element is a structure with information on a single package.

```
installed_packages = pkg ("list")
```

If two output arguments are requested `pkg` splits the list of installed packages into those which were installed by the current user, and those which were installed by the system administrator.

```
[user_packages, system_packages] = pkg ("list")
```

The `"-forge"` option lists packages available at the Octave-Forge repository. This requires an internet connection and the cURL library. For example:

```
oct_forge_pkgs = pkg ("list", "-forge")
```

'`describe`'

Show a short description of the named installed packages, with the option `"-verbose"` also list functions provided by the package. For example,

```
pkg describe -verbose all
```

will describe all installed packages and the functions they provide. If one output is requested a cell of structure containing the description and list of functions of each package is returned as output rather than printed on screen:

```
desc = pkg ("describe", "secs1d", "image")
```

If any of the requested packages is not installed, `pkg` returns an error, unless a second output is requested:

```
[desc, flag] = pkg ("describe", "secs1d", "image")
```

flag will take one of the values `"Not installed"`, `"Loaded"`, or `"Not loaded"` for each of the named packages.

'`prefix`' Set the installation prefix directory. For example,

```
pkg prefix ~/my_octave_packages
```

sets the installation prefix to `~/my_octave_packages`. Packages will be installed in this directory.

It is possible to get the current installation prefix by requesting an output argument. For example:

```
pfx = pkg ("prefix")
```

The location in which to install the architecture dependent files can be independently specified with an addition argument. For example:

```
pkg prefix ~/my_octave_packages ~/my_arch_dep_pkgs
```

'`local_list`'
Set the file in which to look for information on locally installed packages. Locally installed packages are those that are available only to the current user. For example:

```
pkg local_list ~/.octave_packages
```

It is possible to get the current value of local_list with the following

```
pkg local_list
```

'`global_list`'
Set the file in which to look for information on globally installed packages. Globally installed packages are those that are available to all users. For example:

```
pkg global_list /usr/share/octave/octave_packages
```

It is possible to get the current value of global_list with the following

```
pkg global_list
```

'`build`' Build a binary form of a package or packages. The binary file produced will itself be an Octave package that can be installed normally with `pkg`. The form of the command to build a binary package is

```
pkg build builddir image-1.0.0.tar.gz ...
```

where `builddir` is the name of a directory where the temporary installation will be produced and the binary packages will be found. The options `-verbose` and `-nodeps` are respected, while all other options are ignored.

'`rebuild`' Rebuild the package database from the installed directories. This can be used in cases where the package database has been corrupted. It can also take the `-auto` and `-noauto` options to allow the autoloading state of a package to be changed. For example,

```
pkg rebuild -noauto image
```

will remove the autoloading status of the image package.

See also: [ver], page 790, [news], page 21.

38.2 Using Packages

By default installed packages are not available from the Octave prompt, but it is possible to control this using the `pkg load` and `pkg unload` commands. The functions from a package can be added to the Octave path by typing

```
pkg load package_name
```

where package_name is the name of the package to be added to the path.

In much the same way a package can be removed from the Octave path by typing

```
pkg unload package_name
```

38.3 Administrating Packages

On UNIX-like systems it is possible to make both per-user and system-wide installations of a package. If the user performing the installation is root the packages will be installed in a system-wide directory that defaults to OCTAVE_HOME/share/octave/packages/. If the user is not root the default installation directory is ~/octave/. Packages will be installed in a subdirectory of the installation directory that will be named after the package. It is possible to change the installation directory by using the pkg prefix command

```
pkg prefix new_installation_directory
```

The current installation directory can be retrieved by typing

```
current_installation_directory = pkg prefix
```

To function properly the package manager needs to keep some information about the installed packages. For per-user packages this information is by default stored in the file ~/.octave_packages and for system-wide installations it is stored in OCTAVE_HOME/share/octave/octave_packages. The path to the per-user file can be changed with the pkg local_list command

```
pkg local_list /path/to/new_file
```

For system-wide installations this can be changed in the same way using the pkg global_list command. If these commands are called without a new path, the current path will be returned.

38.4 Creating Packages

Internally a package is simply a gzipped tar file that contains a top level directory of any given name. This directory will in the following be referred to as package and may contain the following files:

package/CITATION
: This is am optional file describing instructions on how to cite the package for publication. It will be displayed verbatim by the function citation.

package/COPYING
: This is a required file containing the license of the package. No restrictions is made on the license in general. If however the package contains dynamically linked functions the license must be compatible with the GNU General Public License.

package/DESCRIPTION
: This is a required file containing information about the package. See Section 38.4.1 [The DESCRIPTION File], page 813, for details on this file.

package/ChangeLog
: This is an optional file describing all the changes made to the package source files.

`package/INDEX`
: This is an optional file describing the functions provided by the package. If this file is not given then one with be created automatically from the functions in the package and the `Categories` keyword in the `DESCRIPTION` file. See Section 38.4.2 [The INDEX File], page 815, for details on this file.

`package/NEWS`
: This is an optional file describing all user-visible changes worth mentioning. As this file increases on size, old entries can be moved into `package/ONEWS`.

`package/ONEWS`
: This is an optional file describing old entries from the `NEWS` file.

`package/PKG_ADD`
: An optional file that includes commands that are run when the package is added to the users path. Note that `PKG_ADD` directives in the source code of the package will also be added to this file by the Octave package manager. Note that symbolic links are to be avoided in packages, as symbolic links do not exist on some file systems, and so a typical use for this file is the replacement of the symbolic link

```
ln -s foo.oct bar.oct
```

with an autoload directive like

```
autoload ('bar', which ('foo'));
```

See Section 38.4.3 [PKG_ADD and PKG_DEL Directives], page 816, for details on `PKG_ADD` directives.

`package/PKG_DEL`
: An optional file that includes commands that are run when the package is removed from the users path. Note that `PKG_DEL` directives in the source code of the package will also be added to this file by the Octave package manager. See Section 38.4.3 [PKG_ADD and PKG_DEL Directives], page 816, for details on `PKG_DEL` directives.

`package/pre_install.m`
: This is an optional function that is run prior to the installation of a package. This function is called with a single argument, a struct with fields names after the data in the `DESCRIPTION`, and the paths where the package functions will be installed.

`package/post_install.m`
: This is an optional function that is run after the installation of a package. This function is called with a single argument, a struct with fields names after the data in the `DESCRIPTION`, and the paths where the package functions were installed.

`package/on_uninstall.m`
: This is an optional function that is run prior to the removal of a package. This function is called with a single argument, a struct with fields names after the data in the `DESCRIPTION`, the paths where the package functions are installed, and whether the package is currently loaded.

Besides the above mentioned files, a package can also contain one or more of the following directories:

`package/inst`
: An optional directory containing any files that are directly installed by the package. Typically this will include any `m`-files.

`package/src`
: An optional directory containing code that must be built prior to the packages installation. The Octave package manager will execute `./configure` in this directory if this script exists, and will then call `make` if a file `Makefile` exists in this directory. `make install` will however not be called. The environment variables `MKOCTFILE`, `OCTAVE_CONFIG`, and `OCTAVE` will be set to the full paths of the programs `mkoctfile`, `octave-config`, and `octave`, respectively, of the correct version when `configure` and `make` are called. If a file called `FILES` exists all files listed there will be copied to the `inst` directory, so they also will be installed. If the `FILES` file doesn't exist, `src/*.m` and `src/*.oct` will be copied to the `inst` directory.

`package/doc`
: An optional directory containing documentation for the package. The files in this directory will be directly installed in a sub-directory of the installed package for future reference.

`package/bin`
: An optional directory containing files that will be added to the Octave `EXEC_PATH` when the package is loaded. This might contain external scripts, etc., called by functions within the package.

38.4.1 The DESCRIPTION File

The `DESCRIPTION` file contains various information about the package, such as its name, author, and version. This file has a very simple format

- Lines starting with '`#`' are comments.
- Lines starting with a blank character are continuations from the previous line.
- Everything else is of the form `NameOfOption: ValueOfOption`.

The following is a simple example of a `DESCRIPTION` file

```
Name: The name of my package
Version: 1.0.0
Date: 2007-18-04
Author: The name (and possibly email) of the package author.
Maintainer: The name (and possibly email) of the current
 package maintainer.
Title: The title of the package
Description: A short description of the package.  If this
 description gets too long for one line it can continue
 on the next by adding a space to the beginning of the
 following lines.
License: GPLv3+
```

The package manager currently recognizes the following keywords

`Name` Name of the package.

`Version` Version of the package. A package version must be 3 numbers separated by dots.

`Date` Date of last update.

`Author` Original author of the package.

`Maintainer`
Maintainer of the package.

`Title` A one line description of the package.

`Description`
A one paragraph description of the package.

`Categories`
Optional keyword describing the package (if no `INDEX` file is given this is mandatory).

`Problems` Optional list of known problems.

`Url` Optional list of homepages related to the package.

`Autoload` Optional field that sets the default loading behavior for the package. If set to `yes`, `true` or `on`, then Octave will automatically load the package when starting. Otherwise the package must be manually loaded with the pkg load command. This default behavior can be overridden when the package is installed.

`Depends` A list of other Octave packages that this package depends on. This can include dependencies on particular versions, with a format

```
Depends: package (>= 1.0.0)
```

Possible operators are `<`, `<=`, `==`, `>=` or `>`. If the part of the dependency in `()` is missing, any version of the package is acceptable. Multiple dependencies can be defined either as a comma separated list or on separate `Depends` lines.

`License` An optional short description of the used license (e.g., GPL version 3 or newer). This is optional since the file `COPYING` is mandatory.

`SystemRequirements`
These are the external install dependencies of the package and are not checked by the package manager. This is here as a hint to the distribution packager. They follow the same conventions as the `Depends` keyword.

`BuildRequires`
These are the external build dependencies of the package and are not checked by the package manager. This is here as a hint to the distribution packager. They follow the same conventions as the `Depends` keyword. Note that in general, packaging systems such as `rpm` or `deb` and autoprobe the install dependencies from the build dependencies, and therefore the often a `BuildRequires` dependency removes the need for a `SystemRequirements` dependency.

The developer is free to add additional arguments to the `DESCRIPTION` file for their own purposes. One further detail to aid the packager is that the `SystemRequirements` and `BuildRequires` keywords can have a distribution dependent section, and the automatic build process will use these. An example of the format of this is

```
BuildRequires: libtermcap-devel [Mandriva] libtermcap2-devel
```

where the first package name will be used as a default and if the RPMs are built on a Mandriva distribution, then the second package name will be used instead.

38.4.2 The INDEX File

The optional `INDEX` file provides a categorical view of the functions in the package. This file has a very simple format

- Lines beginning with '`#`' are comments.
- The first non-comment line should look like this

  ```
  toolbox >> Toolbox name
  ```
- Lines beginning with an alphabetical character indicates a new category of functions.
- Lines starting with a white space character indicate that the function names on the line belong to the last mentioned category.

The format can be summarized with the following example:

```
# A comment
toolbox >> Toolbox name
Category Name 1
 function1 function2 function3
 function4
Category Name 2
 function2 function5
```

If you wish to refer to a function that users might expect to find in your package but is not there, providing a work around or pointing out that the function is available elsewhere, you can use:

```
fn = workaround description
```

This workaround description will not appear when listing functions in the package with `pkg describe` but they will be published in the HTML documentation online. Workaround descriptions can use any HTML markup, but keep in mind that it will be enclosed in a bold-italic environment. For the special case of:

```
fn = use <code>alternate expression</code>
```

the bold-italic is automatically suppressed. You will need to use `<code>` even in references:

```
fn = use <a href="someothersite.html"><code>fn</code></a>
```

Sometimes functions are only partially compatible, in which case you can list the non-compatible cases separately. To refer to another function in the package, use `<f>fn</f>`. For example:

```
eig (a, b) = use <f>qz</f>
```

Since sites may have many missing functions, you can define a macro rather than typing the same link over and again.

```
$id = expansion
```

defines the macro id. You can use `$id` anywhere in the description and it will be expanded. For example:

```
$TSA = see <a href="link_to_spctools">SPC Tools</a>
arcov = $TSA <code>armcv</code>
```

id is any string of letters, numbers and `_`.

38.4.3 PKG_ADD and PKG_DEL Directives

If the package contains files called `PKG_ADD` or `PKG_DEL` the commands in these files will be executed when the package is added or removed from the users path. In some situations such files are a bit cumbersome to maintain, so the package manager supports automatic creation of such files. If a source file in the package contains a `PKG_ADD` or `PKG_DEL` directive they will be added to either the `PKG_ADD` or `PKG_DEL` files.

In `m`-files a `PKG_ADD` directive looks like this

```
## PKG_ADD: some_octave_command
```

Such lines should be added before the `function` keyword. In C++ files a `PKG_ADD` directive looks like this

```
// PKG_ADD: some_octave_command
```

In both cases `some_octave_command` should be replaced by the command that should be placed in the `PKG_ADD` file. `PKG_DEL` directives work in the same way, except the `PKG_ADD` keyword is replaced with `PKG_DEL` and the commands get added to the `PKG_DEL` file.

38.4.4 Missing Components

If a package relies on a component, such as another Octave package, that may not be present it may be useful to install a function which informs users what to do when a particular component is missing. The function must be written by the package maintainer and registered with Octave using `missing_component_hook`.

val = `missing_component_hook ()` [Built-in Function]
old_val = `missing_component_hook (`*new_val*`)` [Built-in Function]
`missing_component_hook (`*new_val*, "*local*"`)` [Built-in Function]

Query or set the internal variable that specifies the function to call when a component of Octave is missing.

This can be useful for packagers that may split the Octave installation into multiple sub-packages, for example, to provide a hint to users for how to install the missing components.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

The hook function is expected to be of the form

```
fcn (component)
```

Octave will call *fcn* with the name of the function that requires the component and a string describing the missing component. The hook function should return an error message to be displayed.

See also: [missing_function_hook], page 916.

Appendix A External Code Interface

"The sum of human wisdom is not contained in any one language" —Ezra Pound

Octave is a fantastic language for solving many problems in science and engineering. However, it is not the only computer language and there are times when you may want to use code written in other languages. Good reasons for doing so include: 1) not re-inventing the wheel; existing function libraries which have been thoroughly tested and debugged or large scale simulation codebases are a good example, 2) accessing unique capabilities of a different language; for example the well-known regular expression functions of Perl (but don't do that because `regexp` already exists in Octave).

Performance should generally **not** be a reason for using compiled extensions. Although compiled extensions can run faster, particularly if they replace a loop in Octave code, this is almost never the best path to take. First, there are many techniques to speed up Octave performance while remaining within the language. Second, Octave is a high-level language that makes it easy to perform common mathematical tasks. Giving that up means shifting the focus from solving the real problem to solving a computer programming problem. It means returning to low-level constructs such as pointers, memory management, mathematical overflow/underflow, etc. Because of the low level nature, and the fact that the compiled code is executed outside of Octave, there is the very real possibility of crashing the interpreter and losing work.

Before going further, you should first determine if you really need to bother writing code outside of Octave.

- Can I get the same functionality using the Octave scripting language alone?

  Even when a function already exists outside the language, it may be better to simply reproduce the behavior in an m-file rather than attempt to interface to the outside code.

- Is the code thoroughly optimized for Octave?

  If performance is an issue you should always start with the in-language techniques for getting better performance. Chief among these is vectorization (see Chapter 19 [Vectorization and Faster Code Execution], page 489) which not only makes the code concise and more understandable but improves performance (10X-100X). If loops must be used, make sure that the allocation of space for variables takes place outside the loops using an assignment to a matrix of the right size, or zeros.

- Does the code make as much use as possible of existing built-in library routines?

  These routines are highly optimized and many do not carry the overhead of being interpreted.

- Does writing a dynamically linked function represent a useful investment of your time, relative to staying in Octave?

  It will take time to learn Octave's interface for external code and there will inevitably be issues with tools such as compilers.

With that said, Octave offers a versatile interface for including chunks of compiled code as dynamically linked extensions. These dynamically linked functions can be called from the interpreter in the same manner as any ordinary function. The interface is bi-directional and external code can call Octave functions (like `plot`) which otherwise might be very difficult to develop.

The interface is centered around supporting the languages C++, C, and Fortran. Octave itself is written in C++ and can call external C++/C code through its native oct-file interface. The C language is also supported through the mex-file interface for compatibility with MATLAB. Fortran code is easiest to reach through the oct-file interface.

Because many other languages provide C or C++ APIs it is relatively simple to build bridges between Octave and other languages. This is also a way to bridge to hardware resources which often have device drivers written in C.

A.1 Oct-Files

A.1.1 Getting Started with Oct-Files

Oct-files are pieces of C++ code that have been compiled with the Octave API into a dynamically loadable object. They take their name from the file which contains the object which has the extension `.oct`.

Finding a C++ compiler, using the correct switches, adding the right include paths for header files, etc. is a difficult task. Octave automates this by providing the `mkoctfile` command with which to build oct-files. The command is available from within Octave or at the shell command line.

`mkoctfile` [-*options*] *file* . . . [Command]
[*output*, *status*] = `mkoctfile` (. . .) [Function File]

The `mkoctfile` function compiles source code written in C, C++, or Fortran. Depending on the options used with `mkoctfile`, the compiled code can be called within Octave or can be used as a stand-alone application.

`mkoctfile` can be called from the shell prompt or from the Octave prompt. Calling it from the Octave prompt simply delegates the call to the shell prompt. The output is stored in the *output* variable and the exit status in the *status* variable.

`mkoctfile` accepts the following options, all of which are optional except for the file name of the code you wish to compile:

'`-I DIR`' Add the include directory DIR to compile commands.

'`-D DEF`' Add the definition DEF to the compiler call.

'`-l LIB`' Add the library LIB to the link command.

'`-L DIR`' Add the library directory DIR to the link command.

'`-M`'
'`--depend`'
Generate dependency files (.d) for C and C++ source files.

'`-R DIR`' Add the run-time path to the link command.

'`-Wl,...`' Pass flags though the linker like "-Wl,-rpath=...". The quotes are needed since commas are interpreted as command separators.

'`-W...`' Pass flags though the compiler like "-Wa,OPTION".

'-c' Compile but do not link.

'-g' Enable debugging options for compilers.

'-o FILE'
'--output FILE'
: Output file name. Default extension is .oct (or .mex if '--mex' is specified) unless linking a stand-alone executable.

'-p VAR'
'--print VAR'
: Print the configuration variable VAR. Recognized variables are:

ALL_CFLAGS	INCFLAGS
ALL_CXXFLAGS	INCLUDEDIR
ALL_FFLAGS	LAPACK_LIBS
ALL_LDFLAGS	LD_CXX
AR	LDFLAGS
BLAS_LIBS	LD_STATIC_FLAG
CC	LFLAGS
CFLAGS	LIBDIR
CPICFLAG	LIBOCTAVE
CPPFLAGS	LIBOCTINTERP
CXX	LIBS
CXXFLAGS	OCTAVE_HOME
CXXPICFLAG	OCTAVE_LIBS
DEPEND_EXTRA_SED_PATTERN	OCTAVE_LINK_DEPS
DEPEND_FLAGS	OCTAVE_LINK_OPTS
DL_LD	OCTAVE_PREFIX
DL_LDFLAGS	OCTINCLUDEDIR
F77	OCTLIBDIR
F77_INTEGER8_FLAG	OCT_LINK_DEPS
FFLAGS	OCT_LINK_OPTS
FFTW3F_LDFLAGS	RANLIB
FFTW3F_LIBS	RDYNAMIC_FLAG
FFTW3_LDFLAGS	READLINE_LIBS
FFTW3_LIBS	SED
FFTW_LIBS	SPECIAL_MATH_LIB
FLIBS	XTRA_CFLAGS
FPICFLAG	XTRA_CXXFLAGS

'--link-stand-alone'
: Link a stand-alone executable file.

'--mex' Assume we are creating a MEX file. Set the default output extension to ".mex".

'-s'
'--strip' Strip the output file.

'-v'
'--verbose'
Echo commands as they are executed.

'file' The file to compile or link. Recognized file types are

```
.c    C source
.cc   C++ source
.C    C++ source
.cpp  C++ source
.f    Fortran source (fixed form)
.F    Fortran source (fixed form)
.f90  Fortran source (free form)
.F90  Fortran source (free form)
.o    object file
.a    library file
```

Consider the following short example which introduces the basics of writing a C++ function that can be linked to Octave.

```
#include <octave/oct.h>

DEFUN_DLD (helloworld, args, nargout,
           "Hello World Help String")
{
  int nargin = args.length ();

  octave_stdout << "Hello World has "
                << nargin << " input arguments and "
                << nargout << " output arguments.\n";

  return octave_value_list ();
}
```

The first critical line is `#include <octave/oct.h>` which makes available most of the definitions necessary for a C++ oct-file. Note that `octave/oct.h` is a C++ header and cannot be directly `#include`'ed in a C source file, nor any other language.

Included by `oct.h` is a definition for the macro `DEFUN_DLD` which creates a dynamically loaded function. This macro takes four arguments:

1. The function name as it will be seen in Octave,
2. The list of arguments to the function of type `octave_value_list`,
3. The number of output arguments, which can and often is omitted if not used, and
4. The string to use for the help text of the function.

The return type of functions defined with `DEFUN_DLD` is always `octave_value_list`.

There are a couple of important considerations in the choice of function name. First, it must be a valid Octave function name and so must be a sequence of letters, digits, and underscores not starting with a digit. Second, as Octave uses the function name to define the filename it attempts to find the function in, the function name in the DEFUN_DLD macro must match the filename of the oct-file. Therefore, the above function should be in a file helloworld.cc, and would be compiled to an oct-file using the command

```
mkoctfile helloworld.cc
```

This will create a file called helloworld.oct that is the compiled version of the function. It should be noted that it is perfectly acceptable to have more than one DEFUN_DLD function in a source file. However, there must either be a symbolic link to the oct-file for each of the functions defined in the source code with the DEFUN_DLD macro or the autoload (Section 11.9 [Function Files], page 187) function should be used.

The rest of the function shows how to find the number of input arguments, how to print through the Octave pager, and return from the function. After compiling this function as above, an example of its use is

```
helloworld (1, 2, 3)
 ⊣ Hello World has 3 input arguments and 0 output arguments.
```

Subsequent sections show how to use specific classes from Octave's core internals. Base classes like dMatrix (a matrix of double values) are found in the directory liboctave/array. The definitive reference for how to use a particular class is the header file itself. However, it is often enough just to study the examples in the manual in order to be able to use the class.

A.1.2 Matrices and Arrays in Oct-Files

Octave supports a number of different array and matrix classes, the majority of which are based on the Array class. The exception is the sparse matrix types discussed separately below. There are three basic matrix types

Matrix
: A double precision matrix class defined in dMatrix.h,

ComplexMatrix
: A complex matrix class defined in CMatrix.h, and

BoolMatrix
: A boolean matrix class defined in boolMatrix.h.

These are the basic two-dimensional matrix types of Octave. In addition there are a number of multi-dimensional array types including

NDArray
: A double precision array class defined in dNDArray.h

ComplexNDarray
: A complex array class defined in CNDArray.h

boolNDArray
: A boolean array class defined in boolNDArray.h

int8NDArray
int16NDArray
int32NDArray
int64NDArray
: 8, 16, 32, and 64-bit signed array classes defined in int8NDArray.h, int16NDArray.h, etc.

`uint8NDArray`
`uint16NDArray`
`uint32NDArray`
`uint64NDArray`
: 8, 16, 32, and 64-bit unsigned array classes defined in `uint8NDArray.h`, `uint16NDArray.h`, etc.

There are several basic ways of constructing matrices or multi-dimensional arrays. Using the class `Matrix` as an example one can

- Create an empty matrix or array with the empty constructor. For example:

  ```
  Matrix a;
  ```

  This can be used for all matrix and array types.
- Define the dimensions of the matrix or array with a dim_vector which has the same characteristics as the vector returned from `size`. For example:

  ```
  dim_vector dv (2, 3);  // 2 rows, 3 columns
  Matrix a (dv);
  ```

  This can be used on all matrix and array types.
- Define the number of rows and columns in the matrix. For example:

  ```
  Matrix a (2, 2)
  ```

  However, this constructor can only be used with matrix types.

These types all share a number of basic methods and operators. Many bear a resemblance to functions that exist in the interpreter. A selection of useful methods include

`T& operator` () (*octave_idx_type*) [Method]
`T& elem` (*octave_idx_type*) [Method]
: The `()` operator or `elem` method allow the values of the matrix or array to be read or set. These can take a single argument, which is of type `octave_idx_type`, that is the index into the matrix or array. Additionally, the matrix type allows two argument versions of the `()` operator and elem method, giving the row and column index of the value to obtain or set.

Note that these functions do significant error checking and so in some circumstances the user might prefer to access the data of the array or matrix directly through the fortran_vec method discussed below.

`octave_idx_type` *numel* (*void*) *const* [Method]
: The total number of elements in the matrix or array.

`size_t byte_size` (*void*) *const* [Method]
: The number of bytes used to store the matrix or array.

`dim_vector dims` (*void*) *const* [Method]
: The dimensions of the matrix or array in value of type dim_vector.

`int ndims` (*void*) *const* [Method]
: The number of dimensions of the matrix or array. Matrices are 2-D, but arrays can be N-dimensional.

`void resize` (*const dim_vector&*) [Method]

A method taking either an argument of type `dim_vector`, or in the case of a matrix two arguments of type `octave_idx_type` defining the number of rows and columns in the matrix.

`T* fortran_vec` (*void*) [Method]

This method returns a pointer to the underlying data of the matrix or array so that it can be manipulated directly, either within Octave or by an external library.

Operators such an `+`, `-`, or `*` can be used on the majority of the matrix and array types. In addition there are a number of methods that are of interest only for matrices such as `transpose`, `hermitian`, `solve`, etc.

The typical way to extract a matrix or array from the input arguments of `DEFUN_DLD` function is as follows

```
#include <octave/oct.h>

DEFUN_DLD (addtwomatrices, args, , "Add A to B")
{
  int nargin = args.length ();

  if (nargin != 2)
    print_usage ();
  else
    {
      NDArray A = args(0).array_value ();
      NDArray B = args(1).array_value ();
      if (! error_state)
        return octave_value (A + B);
    }

  return octave_value_list ();
}
```

To avoid segmentation faults causing Octave to abort this function explicitly checks that there are sufficient arguments available before accessing these arguments. It then obtains two multi-dimensional arrays of type `NDArray` and adds these together. Note that the array_value method is called without using the `is_matrix_type` type, and instead the error_state is checked before returning `A + B`. The reason to prefer this is that the arguments might be a type that is not an `NDArray`, but it would make sense to convert it to one. The `array_value` method allows this conversion to be performed transparently if possible, and sets `error_state` if it is not.

`A + B`, operating on two `NDArray`'s returns an `NDArray`, which is cast to an `octave_value` on the return from the function. An example of the use of this demonstration function is

```
addtwomatrices (ones (2, 2), eye (2, 2))
      ⇒  2  1
          1  2
```

A list of the basic Matrix and Array types, the methods to extract these from an octave_value, and the associated header file is listed below.

Type	Function	Source Code
RowVector	row_vector_value	dRowVector.h
ComplexRowVector	complex_row_vector_value	CRowVector.h
ColumnVector	column_vector_value	dColVector.h
ComplexColumnVector	complex_column_vector_value	CColVector.h
Matrix	matrix_value	dMatrix.h
ComplexMatrix	complex_matrix_value	CMatrix.h
boolMatrix	bool_matrix_value	boolMatrix.h
charMatrix	char_matrix_value	chMatrix.h
NDArray	array_value	dNDArray.h
ComplexNDArray	complex_array_value	CNDArray.h
boolNDArray	bool_array_value	boolNDArray.h
charNDArray	char_array_value	charNDArray.h
int8NDArray	int8_array_value	int8NDArray.h
int16NDArray	int16_array_value	int16NDArray.h
int32NDArray	int32_array_value	int32NDArray.h
int64NDArray	int64_array_value	int64NDArray.h
uint8NDArray	uint8_array_value	uint8NDArray.h
uint16NDArray	uint16_array_value	uint16NDArray.h
uint32NDArray	uint32_array_value	uint32NDArray.h
uint64NDArray	uint64_array_value	uint64NDArray.h

A.1.3 Character Strings in Oct-Files

A character string in Octave is just a special Array class. Consider the example:

```
#include <octave/oct.h>

DEFUN_DLD (stringdemo, args, , "String Demo")
{
  octave_value_list retval;
  int nargin = args.length ();

  if (nargin != 1)
    print_usage ();
  else
    {
      charMatrix ch = args(0).char_matrix_value ();

      if (! error_state)
        {
          retval(1) = octave_value (ch, '\'');  // Single Quote String

          octave_idx_type nr = ch.rows ();
          for (octave_idx_type i = 0; i < nr / 2; i++)
```

```
              {
                std::string tmp = ch.row_as_string (i);
                ch.insert (ch.row_as_string (nr-i-1).c_str (), i, 0);
                ch.insert (tmp.c_str (), nr-i-1, 0);
              }
            retval(0) = octave_value (ch, '"');  // Double Quote String
          }
      }
    return retval;
  }
```

An example of the use of this function is

```
s0 = ["First String"; "Second String"];
[s1,s2] = stringdemo (s0)
⇒ s1 = Second String
        First String

⇒ s2 = First String
        Second String

typeinfo (s2)
⇒ sq_string
typeinfo (s1)
⇒ string
```

One additional complication of strings in Octave is the difference between single quoted and double quoted strings. To find out if an `octave_value` contains a single or double quoted string use one of the predicate tests shown below.

```
if (args(0).is_sq_string ())
  octave_stdout << "First argument is a single quoted string\n";
else if (args(0).is_dq_string ())
  octave_stdout << "First argument is a double quoted string\n";
```

Note, however, that both types of strings are represented by the `charNDArray` type, and so when assigning to an `octave_value`, the type of string should be specified. For example:

```
octave_value_list retval;
charNDArray ch;
...
// Create single quoted string
retval(1) = octave_value (ch);        // default constructor is sq_string
            OR
retval(1) = octave_value (ch, '\'');  // explicitly create sq_string

// Create a double quoted string
retval(0) = octave_value (ch, '"');
```

A.1.4 Cell Arrays in Oct-Files

Octave's cell type is also available from within oct-files. A cell array is just an array of `octave_values`, and thus each element of the cell array can be treated just like any other `octave_value`. A simple example is

```
#include <octave/oct.h>
#include <octave/Cell.h>

DEFUN_DLD (celldemo, args, , "Cell Demo")
{
  octave_value_list retval;
  int nargin = args.length ();

  if (nargin != 1)
    print_usage ();
  else
    {
      Cell c = args(0).cell_value ();
      if (! error_state)
        for (octave_idx_type i = 0; i < c.numel (); i++)
          {
            retval(i) = c(i);          // using operator syntax
            //retval(i) = c.elem (i);  // using method syntax
          }
    }

  return retval;
}
```

Note that cell arrays are used less often in standard oct-files and so the `Cell.h` header file must be explicitly included. The rest of the example extracts the `octave_values` one by one from the cell array and returns them as individual return arguments. For example:

```
[b1, b2, b3] = celldemo ({1, [1, 2], "test"})
⇒
b1 =  1
b2 =

   1   2

b3 = test
```

A.1.5 Structures in Oct-Files

A structure in Octave is a map between a number of fields represented and their values. The Standard Template Library `map` class is used, with the pair consisting of a `std::string` and an Octave `Cell` variable.

A simple example demonstrating the use of structures within oct-files is

```
#include <octave/oct.h>
#include <octave/ov-struct.h>

DEFUN_DLD (structdemo, args, , "Struct Demo")
{
  octave_value retval;
  int nargin = args.length ();

  if (args.length () == 2)
    {
      octave_scalar_map arg0 = args(0).scalar_map_value ();
      //octave_map arg0 = args(0).map_value ();

      if (! error_state)
        {
          std::string arg1 = args(1).string_value ();

          if (! error_state)
            {
              octave_value tmp = arg0.contents (arg1);
              //octave_value tmp = arg0.contents (arg1)(0);

              if (tmp.is_defined ())
                {
                  octave_scalar_map st;

                  st.assign ("selected", tmp);

                  retval = octave_value (st);
                }
              else
                error ("structdemo: struct does not have a field named '%s'\n",
                       arg1.c_str ());
            }
          else
            error ("structdemo: ARG2 must be a character string");
        }
      else
        error ("structdemo: ARG1 must be a struct");
    }
  else
    print_usage ();

  return retval;
}
```

An example of its use is

```
x.a = 1; x.b = "test"; x.c = [1, 2];
structdemo (x, "b")
⇒ selected = test
```

The example above specifically uses the `octave_scalar_map` class which is for representing a single struct. For structure arrays the `octave_map` class is used instead. The commented code shows how the demo could be modified to handle a structure array. In that case the `contents` method returns a `Cell` which may have more than one element. Therefore, to obtain the underlying `octave_value` in this single-struct example we write

```
octave_value tmp = arg0.contents (arg1)(0);
```

where the trailing (0) is the () operator on the `Cell` object. If this were a true structure array with multiple elements we could iterate over the elements using the () operator.

Structures are a relatively complex data container and there are more functions available in `oct-map.h` which make coding with them easier than relying on just `contents`.

A.1.6 Sparse Matrices in Oct-Files

There are three classes of sparse objects that are of interest to the user.

`SparseMatrix`
: A double precision sparse matrix class

`SparseComplexMatrix`
: A complex sparse matrix class

`SparseBoolMatrix`
: A boolean sparse matrix class

All of these classes inherit from the `Sparse<T>` template class, and so all have similar capabilities and usage. The `Sparse<T>` class was based on Octave's `Array<T>` class, and so users familiar with Octave's `Array` classes will be comfortable with the use of the sparse classes.

The sparse classes will not be entirely described in this section, due to their similarity with the existing `Array` classes. However, there are a few differences due the different nature of sparse objects, and these will be described. First, although it is fundamentally possible to have N-dimensional sparse objects, the Octave sparse classes do not allow them at this time; All instances of the sparse classes must be 2-dimensional. This means that `SparseMatrix` is actually more similar to Octave's `Matrix` class than its `NDArray` class.

A.1.6.1 Array and Sparse Class Differences

The number of elements in a sparse matrix is considered to be the number of nonzero elements rather than the product of the dimensions. Therefore

```
SparseMatrix sm;
...
int nel = sm.nelem ();
```

returns the number of nonzero elements. If the user really requires the number of elements in the matrix, including the nonzero elements, they should use `numel` rather than `nelem`. Note that for very large matrices, where the product of the two dimensions is larger than the representation of an unsigned int, then `numel` can overflow. An example is `speye (1e6)` which will create a matrix with a million rows and columns, but only a million nonzero elements. Therefore the number of rows by the number of columns in this case is more than two hundred times the maximum value that can be represented by an unsigned int. The use of `numel` should therefore be avoided useless it is known it won't overflow.

Extreme care must be take with the elem method and the `"()"` operator, which perform basically the same function. The reason is that if a sparse object is non-const, then Octave will assume that a request for a zero element in a sparse matrix is in fact a request to create this element so it can be filled. Therefore a piece of code like

```
SparseMatrix sm;
...
for (int j = 0; j < nc; j++)
  for (int i = 0; i < nr; i++)
    std::cerr << " (" << i << "," << j << "): " << sm(i,j) << std::endl;
```

is a great way of turning the sparse matrix into a dense one, and a very slow way at that since it reallocates the sparse object at each zero element in the matrix.

An easy way of preventing the above from happening is to create a temporary constant version of the sparse matrix. Note that only the container for the sparse matrix will be copied, while the actual representation of the data will be shared between the two versions of the sparse matrix. So this is not a costly operation. For example, the above would become

```
SparseMatrix sm;
...
const SparseMatrix tmp (sm);
for (int j = 0; j < nc; j++)
  for (int i = 0; i < nr; i++)
    std::cerr << " (" << i << "," << j << "): " << tmp(i,j) << std::endl;
```

Finally, as the sparse types aren't represented by a contiguous block of memory, the `fortran_vec` method of the `Array<T>` is not available. It is, however, replaced by three separate methods `ridx`, `cidx` and `data`, that access the raw compressed column format that Octave sparse matrices are stored in. These methods can be used in a manner similar to `elem` to allow the matrix to be accessed or filled. However, in that case it is up to the user to respect the sparse matrix compressed column format.

A.1.6.2 Creating Sparse Matrices in Oct-Files

There are several useful alternatives for creating a sparse matrix. The first is to create three vectors representing the row index, column index, and data values, and from these create the matrix. The second alternative is to create a sparse matrix with the appropriate amount of space and then fill in the values. Both techniques have their advantages and disadvantages.

Below is an example of creating a small sparse matrix using the first technique

```
int nz, nr, nc;
nz = 4, nr = 3, nc = 4;

ColumnVector ridx (nz);
ColumnVector cidx (nz);
ColumnVector data (nz);

ridx(0) = 1; cidx(0) = 1; data(0) = 1;
ridx(1) = 2; cidx(1) = 2; data(1) = 2;
ridx(2) = 2; cidx(2) = 4; data(2) = 3;
ridx(3) = 3; cidx(3) = 4; data(3) = 4;
SparseMatrix sm (data, ridx, cidx, nr, nc);
```

which creates the matrix given in section Section 22.1.1 [Storage of Sparse Matrices], page 519. Note that the compressed matrix format is not used at the time of the creation of the matrix itself, but is used internally.

As discussed in the chapter on Sparse Matrices, the values of the sparse matrix are stored in increasing column-major ordering. Although the data passed by the user need not respect this requirement, pre-sorting the data will significantly speed up creation of the sparse matrix.

The disadvantage of this technique for creating a sparse matrix is that there is a brief time when two copies of the data exist. For extremely memory constrained problems this may not be the best technique for creating a sparse matrix.

The alternative is to first create a sparse matrix with the desired number of nonzero elements and then later fill those elements in. Sample code:

```
int nz, nr, nc;
nz = 4, nr = 3, nc = 4;
SparseMatrix sm (nr, nc, nz);
sm(0,0) = 1; sm(0,1) = 2; sm(1,3) = 3; sm(2,3) = 4;
```

This creates the same matrix as previously. Again, although not strictly necessary, it is significantly faster if the sparse matrix is created and the elements are added in column-major ordering. The reason for this is that when elements are inserted at the end of the current list of known elements then no element in the matrix needs to be moved to allow the new element to be inserted; Only the column indexes need to be updated.

There are a few further points to note about this method of creating a sparse matrix. First, it is possible to create a sparse matrix with fewer elements than are actually inserted in the matrix. Therefore,

```
int nr, nc;
nr = 3, nc = 4;
SparseMatrix sm (nr, nc, 0);
sm(0,0) = 1; sm(0,1) = 2; sm(1,3) = 3; sm(2,3) = 4;
```

is perfectly valid. However, it is a very bad idea because as each new element is added to the sparse matrix the matrix needs to request more space and reallocate memory. This is an expensive operation, that will significantly slow this means of creating a sparse matrix. Furthermore, it is possible to create a sparse matrix with too much storage, so having *nz* greater than 4 is also valid. The disadvantage is that the matrix occupies more memory than strictly needed.

It is not always possible to know the number of nonzero elements prior to filling a matrix. For this reason the additional unused storage of a sparse matrix can be removed after its creation with the `maybe_compress` function. In addition, `maybe_compress` can deallocate the unused storage, but it can also remove zero elements from the matrix. The removal of zero elements from the matrix is controlled by setting the argument of the `maybe_compress` function to be `true`. However, the cost of removing the zeros is high because it implies re-sorting the elements. If possible, it is better if the user does not add the unnecessary zeros in the first place. An example of the use of `maybe_compress` is

```
int nz, nr, nc;
nz = 6, nr = 3, nc = 4;

SparseMatrix sm1 (nr, nc, nz);
sm1(0,0) = 1; sm1(0,1) = 2; sm1(1,3) = 3; sm1(2,3) = 4;
sm1.maybe_compress ();  // No zero elements were added

SparseMatrix sm2 (nr, nc, nz);
sm2(0,0) = 1; sm2(0,1) = 2; sm(0,2) = 0; sm(1,2) = 0;
sm1(1,3) = 3; sm1(2,3) = 4;
sm2.maybe_compress (true);  // Zero elements were added
```

The use of the `maybe_compress` function should be avoided if possible as it will slow the creation of the matrix.

A third means of creating a sparse matrix is to work directly with the data in compressed row format. An example of this technique might be

```
octave_value arg;
...
int nz, nr, nc;
nz = 6, nr = 3, nc = 4;   // Assume we know the max # nz
SparseMatrix sm (nr, nc, nz);
Matrix m = arg.matrix_value ();

int ii = 0;
sm.cidx (0) = 0;
for (int j = 1; j < nc; j++)
  {
    for (int i = 0; i < nr; i++)
      {
        double tmp = foo (m(i,j));
        if (tmp != 0.)
          {
            sm.data(ii) = tmp;
            sm.ridx(ii) = i;
            ii++;
          }
      }
```

```
        sm.cidx(j+1) = ii;
     }
    sm.maybe_compress ();  // If don't know a priori the final # of nz.
```

which is probably the most efficient means of creating a sparse matrix.

Finally, it might sometimes arise that the amount of storage initially created is insufficient to completely store the sparse matrix. Therefore, the method `change_capacity` exists to reallocate the sparse memory. The above example would then be modified as

```
octave_value arg;
...
int nz, nr, nc;
nz = 6, nr = 3, nc = 4;   // Assume we know the max # nz
SparseMatrix sm (nr, nc, nz);
Matrix m = arg.matrix_value ();

int ii = 0;
sm.cidx (0) = 0;
for (int j = 1; j < nc; j++)
  {
    for (int i = 0; i < nr; i++)
      {
        double tmp = foo (m(i,j));
        if (tmp != 0.)
          {
            if (ii == nz)
              {
                nz += 2;   // Add 2 more elements
                sm.change_capacity (nz);
              }
            sm.data(ii) = tmp;
            sm.ridx(ii) = i;
            ii++;
          }
      }
    sm.cidx(j+1) = ii;
 }
sm.maybe_mutate ();  // If don't know a priori the final # of nz.
```

Note that both increasing and decreasing the number of nonzero elements in a sparse matrix is expensive as it involves memory reallocation. Also as parts of the matrix, though not its entirety, exist as old and new copies at the same time, additional memory is needed. Therefore, if possible this should be avoided.

A.1.6.3 Using Sparse Matrices in Oct-Files

Most of the same operators and functions on sparse matrices that are available from the Octave command line are also available within oct-files. The basic means of extracting a sparse matrix from an `octave_value` and returning it as an `octave_value`, can be seen in the following example.

```
octave_value_list retval;

SparseMatrix sm = args(0).sparse_matrix_value ();
SparseComplexMatrix scm =
    args(1).sparse_complex_matrix_value ();
SparseBoolMatrix sbm = args(2).sparse_bool_matrix_value ();
...
retval(2) = sbm;
retval(1) = scm;
retval(0) = sm;
```

The conversion to an `octave_value` is handled by the sparse `octave_value` constructors, and so no special care is needed.

A.1.7 Accessing Global Variables in Oct-Files

Global variables allow variables in the global scope to be accessed. Global variables can be accessed within oct-files by using the support functions `get_global_value` and `set_global_value`. `get_global_value` takes two arguments, the first is a string representing the variable name to obtain. The second argument is a boolean argument specifying what to do if no global variable of the desired name is found. An example of the use of these two functions is

```
#include <octave/oct.h>

DEFUN_DLD (globaldemo, args, , "Global Demo")
{
  octave_value retval;
  int nargin = args.length ();

  if (nargin != 1)
    print_usage ();
  else
    {
      std::string s = args(0).string_value ();
      if (! error_state)
        {
          octave_value tmp = get_global_value (s, true);
          if (tmp.is_defined ())
            retval = tmp;
          else
            retval = "Global variable not found";

          set_global_value ("a", 42.0);
        }
    }
  return retval;
}
```

An example of its use is

```
global a b
b = 10;
globaldemo ("b")
⇒ 10
globaldemo ("c")
⇒ "Global variable not found"
num2str (a)
⇒ 42
```

A.1.8 Calling Octave Functions from Oct-Files

There is often a need to be able to call another Octave function from within an oct-file, and there are many examples of such within Octave itself. For example, the `quad` function is an oct-file that calculates the definite integral by quadrature over a user supplied function.

There are also many ways in which a function might be passed. It might be passed as one of

1. Function Handle
2. Anonymous Function Handle
3. Inline Function
4. String

The example below demonstrates an example that accepts all four means of passing a function to an oct-file.

```
#include <octave/oct.h>
#include <octave/parse.h>

DEFUN_DLD (funcdemo, args, nargout, "Function Demo")
{
  octave_value_list retval;
  int nargin = args.length ();

  if (nargin < 2)
    print_usage ();
  else
    {
      octave_value_list newargs;
      for (octave_idx_type i = nargin - 1; i > 0; i--)
        newargs(i-1) = args(i);
      if (args(0).is_function_handle () || args(0).is_inline_function ())
        {
          octave_function *fcn = args(0).function_value ();
          if (! error_state)
            retval = feval (fcn, newargs, nargout);
        }
      else if (args(0).is_string ())
        {
```

```
            std::string fcn = args(0).string_value ();
            if (! error_state)
              retval = feval (fcn, newargs, nargout);
          }
        else
          error ("funcdemo: INPUT must be string, inline, or function handle");
      }
    return retval;
  }
```

The first argument to this demonstration is the user-supplied function and the remaining arguments are all passed to the user function.

```
funcdemo (@sin,1)
⇒ 0.84147
funcdemo (@(x) sin (x), 1)
⇒ 0.84147
funcdemo (inline ("sin (x)"), 1)
⇒ 0.84147
funcdemo ("sin",1)
⇒ 0.84147
funcdemo (@atan2, 1, 1)
⇒ 0.78540
```

When the user function is passed as a string the treatment of the function is different. In some cases it is necessary to have the user supplied function as an `octave_function` object. In that case the string argument can be used to create a temporary function as demonstrated below.

```
std::octave fcn_name = unique_symbol_name ("__fcn__");
std::string fcode = "function y = ";
fcode.append (fcn_name);
fcode.append ("(x) y = ");
fcn = extract_function (args(0), "funcdemo", fcn_name,
                        fcode, "; endfunction");
...
if (fcn_name.length ())
  clear_function (fcn_name);
```

There are two important things to know in this case. First, the number of input arguments to the user function is fixed, and in the above example is a single argument. Second, to avoid leaving the temporary function in the Octave symbol table it should be cleared after use. Also, by convention internal function names begin and end with the character sequence '`__`'.

A.1.9 Calling External Code from Oct-Files

Linking external C code to Octave is relatively simple, as the C functions can easily be called directly from C++. One possible issue is that the declarations of the external C functions may need to be explicitly defined as C functions to the compiler. If the declarations of the external C functions are in the header `foo.h`, then the tactic to ensure that the C++ compiler treats these declarations as C code is

```
#ifdef __cplusplus
extern "C"
{
#endif
#include "foo.h"
#ifdef __cplusplus
}  /* end extern "C" */
#endif
```

Calling Fortran code, however, can pose more difficulties. This is due to differences in the manner in which compilers treat the linking of Fortran code with C or C++ code. Octave supplies a number of macros that allow consistent behavior across a number of compilers.

The underlying Fortran code should use the `XSTOPX` function to replace the Fortran `STOP` function. `XSTOPX` uses the Octave exception handler to treat failing cases in the Fortran code explicitly. Note that Octave supplies its own replacement BLAS `XERBLA` function, which uses `XSTOPX`.

If the code calls `XSTOPX`, then the `F77_XFCN` macro should be used to call the underlying Fortran function. The Fortran exception state can then be checked with the global variable `f77_exception_encountered`. If `XSTOPX` will not be called, then the `F77_FCN` macro should be used instead to call the Fortran code.

There is no great harm in using `F77_XFCN` in all cases, except that for Fortran code that is short running and executes a large number of times, there is potentially an overhead in doing so. However, if `F77_FCN` is used with code that calls `XSTOP`, Octave can generate a segmentation fault.

An example of the inclusion of a Fortran function in an oct-file is given in the following example, where the C++ wrapper is

```
#include <octave/oct.h>
#include <octave/f77-fcn.h>

extern "C"
{
  F77_RET_T
  F77_FUNC (fortransub, FORTSUB)
    (const int&, double*, F77_CHAR_ARG_DECL F77_CHAR_ARG_LEN_DECL);
}

DEFUN_DLD (fortrandemo, args, , "Fortran Demo")
{
  octave_value_list retval;
  int nargin = args.length ();

  if (nargin != 1)
    print_usage ();
  else
```

```
    {
      NDArray a = args(0).array_value ();
      if (! error_state)
        {
          double *av = a.fortran_vec ();
          octave_idx_type na = a.numel ();
          OCTAVE_LOCAL_BUFFER (char, ctmp, 128);

          F77_XFCN (fortransub, FORTSUB,
                    (na, av, ctmp F77_CHAR_ARG_LEN (128)));

          retval(1) = std::string (ctmp);
          retval(0) = a;
        }
    }
  return retval;
}
```

and the Fortran function is

```
      subroutine fortransub (n, a, s)
      implicit none
      character*(*) s
      real*8 a(*)
      integer*4 i, n, ioerr
      do i = 1, n
        if (a(i) .eq. 0d0) then
          call xstopx ('fortransub: divide by zero')
        else
          a(i) = 1d0 / a(i)
        endif
      enddo
      write (unit = s, fmt = '(a,i3,a,a)', iostat = ioerr)
     $       'There are ', n,
     $       ' values in the input vector', char(0)
      if (ioerr .ne. 0) then
        call xstopx ('fortransub: error writing string')
      endif
      return
      end
```

This example demonstrates most of the features needed to link to an external Fortran function, including passing arrays and strings, as well as exception handling. Both the Fortran and C++ files need to be compiled in order for the example to work.

```
mkoctfile fortrandemo.cc fortransub.f
[b, s] = fortrandemo (1:3)
⇒
  b = 1.00000   0.50000   0.33333
  s = There are   3 values in the input vector
[b, s] = fortrandemo (0:3)
error: fortrandemo: fortransub: divide by zero
```

A.1.10 Allocating Local Memory in Oct-Files

Allocating memory within an oct-file might seem easy as the C++ new/delete operators can be used. However, in that case great care must be taken to avoid memory leaks. The preferred manner in which to allocate memory for use locally is to use the `OCTAVE_LOCAL_BUFFER` macro. An example of its use is

```
OCTAVE_LOCAL_BUFFER (double, tmp, len)
```

that returns a pointer `tmp` of type `double *` of length `len`.

In this case Octave itself will worry about reference counting and variable scope and will properly free memory without programmer intervention.

A.1.11 Input Parameter Checking in Oct-Files

As oct-files are compiled functions they open up the possibility of crashing Octave through careless function calls or memory faults. It is quite important that each and every function have a sufficient level of parameter checking to ensure that Octave behaves well.

The minimum requirement, as previously discussed, is to check the number of input arguments before using them to avoid referencing a nonexistent argument. However, in some cases this might not be sufficient as the underlying code imposes further constraints. For example, an external function call might be undefined if the input arguments are not integers, or if one of the arguments is zero, or if the input is complex and a real value was expected. Therefore, oct-files often need additional input parameter checking.

There are several functions within Octave that can be useful for the purposes of parameter checking. These include the methods of the octave_value class like `is_real_matrix`, `is_numeric_type`, etc. Often, with a knowledge of the Octave m-file language, you can guess at what the corresponding C++ routine will. In addition there are some more specialized input validation functions of which a few are demonstrated below.

```
#include <octave/oct.h>

DEFUN_DLD (paramdemo, args, nargout, "Parameter Check Demo")
{
  octave_value retval;
  int nargin = args.length ();

  if (nargin != 1)
    print_usage ();
  else if (nargout != 0)
    error ("paramdemo: OUTPUT argument required");
  else
```

```
    {
      NDArray m = args(0).array_value ();
      double min_val = -10.0;
      double max_val = 10.0;
      octave_stdout << "Properties of input array:\n";
      if (m.any_element_is_negative ())
        octave_stdout << "  includes negative values\n";
      if (m.any_element_is_inf_or_nan ())
        octave_stdout << "  includes Inf or NaN values\n";
      if (m.any_element_not_one_or_zero ())
        octave_stdout << "  includes other values than 1 and 0\n";
      if (m.all_elements_are_int_or_inf_or_nan ())
        octave_stdout << "  includes only int, Inf or NaN values\n";
      if (m.all_integers (min_val, max_val))
        octave_stdout << "  includes only integers in [-10,10]\n";
    }
  return retval;
}
```

An example of its use is:

```
paramdemo ([1, 2, NaN, Inf])
⇒ Properties of input array:
     includes Inf or NaN values
     includes other values than 1 and 0
     includes only int, Inf or NaN values
```

A.1.12 Exception and Error Handling in Oct-Files

Another important feature of Octave is its ability to react to the user typing `Control-C` even during calculations. This ability is based on the C++ exception handler, where memory allocated by the C++ new/delete methods are automatically released when the exception is treated. When writing an oct-file, to allow Octave to treat the user typing `Control-C`, the `OCTAVE_QUIT` macro is supplied. For example:

```
for (octave_idx_type i = 0; i < a.nelem (); i++)
  {
    OCTAVE_QUIT;
    b.elem (i) = 2. * a.elem (i);
  }
```

The presence of the `OCTAVE_QUIT` macro in the inner loop allows Octave to treat the user request with the `Control-C`. Without this macro, the user must either wait for the function to return before the interrupt is processed, or press `Control-C` three times to force Octave to exit.

The `OCTAVE_QUIT` macro does impose a very small speed penalty, and so for loops that are known to be small it might not make sense to include `OCTAVE_QUIT`.

When creating an oct-file that uses an external libraries, the function might spend a significant portion of its time in the external library. It is not generally possible to use the `OCTAVE_QUIT` macro in this case. The alternative in this case is

```
BEGIN_INTERRUPT_IMMEDIATELY_IN_FOREIGN_CODE;
...  some code that calls a "foreign" function ...
END_INTERRUPT_IMMEDIATELY_IN_FOREIGN_CODE;
```

The disadvantage of this is that if the foreign code allocates any memory internally, then this memory might be lost during an interrupt, without being deallocated. Therefore, ideally Octave itself should allocate any memory that is needed by the foreign code, with either the fortran_vec method or the `OCTAVE_LOCAL_BUFFER` macro.

The Octave unwind_protect mechanism (Section 10.8 [The unwind_protect Statement], page 168) can also be used in oct-files. In conjunction with the exception handling of Octave, it is important to enforce that certain code is run to allow variables, etc. to be restored even if an exception occurs. An example of the use of this mechanism is

```
#include <octave/oct.h>
#include <octave/unwind-prot.h>

void
my_err_handler (const char *fmt, ...)
{
  // Do nothing!!
}

void
my_err_with_id_handler (const char *id, const char *fmt, ...)
{
  // Do nothing!!
}

DEFUN_DLD (unwinddemo, args, nargout, "Unwind Demo")
{
  octave_value retval;
  int nargin = args.length ();

  if (nargin < 2)
    print_usage ();
  else
    {
      NDArray a = args(0).array_value ();
      NDArray b = args(1).array_value ();

      if (! error_state)
        {
          // Declare unwind_protect frame which lasts as long as
          // the variable frame has scope.
          unwind_protect frame;
          frame.add_fcn (set_liboctave_warning_handler,
                         current_liboctave_warning_handler);
```

```
            frame.add_fcn (set_liboctave_warning_with_id_handler,
                           current_liboctave_warning_with_id_handler);

            set_liboctave_warning_handler (my_err_handler);
            set_liboctave_warning_with_id_handler (my_err_with_id_handler);

            retval = octave_value (quotient (a, b));
          }
      }

    return retval;
  }
```

As can be seen in the example:

```
unwinddemo (1, 0)
⇒ Inf
1 / 0
⇒ warning: division by zero
   Inf
```

The warning for division by zero (and in fact all warnings) are disabled in the `unwinddemo` function.

A.1.13 Documentation and Test of Oct-Files

The documentation of an oct-file is the fourth string parameter of the `DEFUN_DLD` macro. This string can be formatted in the same manner as the help strings for user functions (see Section C.4 [Documentation Tips], page 871), however there are some issue that are particular to the formatting of help strings within oct-files.

The major issue is that the help string will typically be longer than a single line of text, and so the formatting of long help strings needs to be taken into account. There are several possible solutions, but the most common is illustrated in the following example,

```
DEFUN_DLD (do_what_i_want, args, nargout,
  "-*- texinfo -*-\n\
@deftypefn {Function File} {} do_what_i_say (@var{n})\n\
A function that does what the user actually wants rather\n\
than what they requested.\n\
@end deftypefn")
{
...
}
```

where, as can be seen, each line of text is terminated by `\n\` which is an embedded new-line in the string together with a C++ string continuation character. Note that the final `\` must be the last character on the line.

Octave also includes the ability to embed test and demonstration code for a function within the code itself (see Appendix B [Test and Demo Functions], page 857). This can be used from within oct-files (or in fact any file) with certain provisos. First, the test and demo functions of Octave look for `%!` as the first two characters of a line to identify test and demonstration code. This is a requirement for oct-files as well. In addition, the test and demonstration code must be wrapped in a comment block to avoid it being interpreted by the compiler. Finally, the Octave test and demonstration code must have access to the original source code of the oct-file and not just the compiled code as the tests are stripped from the compiled code. An example in an oct-file might be

```
/*
%!assert (sin ([1,2]), [sin(1),sin(2)])
%!error (sin ())
%!error (sin (1,1))
*/
```

A.2 Mex-Files

Octave includes an interface to allow legacy mex-files to be compiled and used with Octave. This interface can also be used to share code between Octave and MATLAB users. However, as mex-files expose MATLAB's internal API, and the internal structure of Octave is different, a mex-file can never have the same performance in Octave as the equivalent oct-file. In particular, to support the manner in which variables are passed to mex functions there are a significant number of additional copies of memory blocks when calling or returning from a mex-file function. For this reason, it is recommended that any new code be written with the oct-file interface previously discussed.

A.2.1 Getting Started with Mex-Files

The basic command to build a mex-file is either `mkoctfile --mex` or `mex`. The first command can be used either from within Octave or from the command line. However, to avoid issues with MATLAB's own `mex` command, the use of the command `mex` is limited to within Octave. Compiled mex-files have the extension `.mex`.

`mex` [*options*] *file* ... [Command]

Compile source code written in C, C++, or Fortran, to a MEX file.

This is equivalent to `mkoctfile --mex [options] file`.

See also: [mkoctfile], page 818, [mexext], page 842.

`mexext` () [Function File]

Return the filename extension used for MEX files.

See also: [mex], page 842.

Consider the following short example:

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[],
             int nrhs, const mxArray *prhs[])
{
  mexPrintf ("Hello, World!\n");

  mexPrintf ("I have %d inputs and %d outputs\n", nrhs, nlhs);
}
```

The first line `#include "mex.h"` makes available all of the definitions necessary for a mex-file. One important difference between Octave and MATLAB is that the header file `"matrix.h"` is implicitly included through the inclusion of `"mex.h"`. This is necessary to avoid a conflict with the Octave file `"Matrix.h"` for operating systems and compilers that don't distinguish between filenames in upper and lower case.

The entry point into the mex-file is defined by `mexFunction`. The function takes four arguments:

1. The number of return arguments (# of left-hand side args).
2. An array of pointers to return arguments.
3. The number of input arguments (# of right-hand side args).
4. An array of pointers to input arguments.

Note that the function name definition is not explicitly included in `mexFunction` and so there can only be a single `mexFunction` entry point per file. Instead, the name of the function as seen in Octave is determined by the name of the mex-file itself minus the extension. Therefore, if the above function is in the file `myhello.c`, it can be compiled with

```
mkoctfile --mex myhello.c
```

which creates a file `myhello.mex`. The function can then be run from Octave as

```
myhello (1,2,3)
⇒ Hello, World!
⇒ I have 3 inputs and 0 outputs
```

It should be noted that the mex-file contains no help string for the functions it contains. To document mex-files, there should exist an m-file in the same directory as the mex-file itself. Taking the above as an example, we would therefore have a file `myhello.m` that might contain the text

```
%MYHELLO Simple test of the functionality of a mex-file.
```

In this case, the function that will be executed within Octave will be given by the mex-file, while the help string will come from the m-file. This can also be useful to allow a sample implementation of the mex-file within the Octave language itself for testing purposes.

Although there cannot be multiple entry points in a single mex-file, one can use the `mexFunctionName` function to determine what name the mex-file was called with. This can be used to alter the behavior of the mex-file based on the function name. For example, if

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[],
             int nrhs, const mxArray *prhs[])
{
  const char *nm;

  nm = mexFunctionName ();
  mexPrintf ("You called function: %s\n", nm);
  if (strcmp (nm, "myfunc") == 0)
    mexPrintf ("This is the principal function\n", nm);

  return;
}
```

is in file `myfunc.c`, and it is compiled with

```
mkoctfile --mex myfunc.c
ln -s myfunc.mex myfunc2.mex
```

then as can be seen by

```
myfunc ()
⇒ You called function: myfunc
    This is the principal function
myfunc2 ()
⇒ You called function: myfunc2
```

the behavior of the mex-file can be altered depending on the functions name.

Although the user should only include `mex.h` in their code, Octave declares additional functions, typedefs, etc., available to the user to write mex-files in the headers `mexproto.h` and `mxarray.h`.

A.2.2 Working with Matrices and Arrays in Mex-Files

The basic mex type of all variables is `mxArray`. Any object, such as a matrix, cell array, or structure is stored in this basic type. As such, `mxArray` serves basically the same purpose as the octave_value class in oct-files in that it acts as a container for the more specialized types.

The `mxArray` structure contains at a minimum, the name of the variable it represents, its dimensions, its type, and whether the variable is real or complex. It can also contain a number of additional fields depending on the type of the `mxArray`. There are a number of functions to create `mxArray` structures, including `mxCreateDoubleMatrix`, `mxCreateCellArray`, `mxCreateSparse`, and the generic `mxCreateNumericArray`.

The basic function to access the data contained in an array is `mxGetPr`. As the mex interface assumes that real and imaginary parts of a complex array are stored separately, there is an equivalent function `mxGetPi` that gets the imaginary part. Both of these functions are only for use with double precision matrices. The generic functions `mxGetData` and `mxGetImagData` perform the same operation on all matrix types. For example:

```
mxArray *m;
mwSize *dims;
UINT32_T *pr;

dims = (mwSize *) mxMalloc (2 * sizeof (mwSize));
dims[0] = 2; dims[1] = 2;
m = mxCreateNumericArray (2, dims, mxUINT32_CLASS, mxREAL);
pr = (UINT32_T *) mxGetData (m);
```

There are also the functions `mxSetPr`, etc., that perform the inverse, and set the data of an array to use the block of memory pointed to by the argument of `mxSetPr`.

Note the type `mwSize` used above, and also `mwIndex`, are defined as the native precision of the indexing in Octave on the platform on which the mex-file is built. This allows both 32- and 64-bit platforms to support mex-files. `mwSize` is used to define array dimensions and the maximum number or elements, while `mwIndex` is used to define indexing into arrays.

An example that demonstrates how to work with arbitrary real or complex double precision arrays is given by the file `mypow2.c` shown below.

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[],
             int nrhs, const mxArray* prhs[])
{
  mwSize n;
  mwIndex i;
  double *vri, *vro;

  if (nrhs != 1 || ! mxIsDouble (prhs[0]))
    mexErrMsgTxt ("ARG1 must be a double matrix");

  n = mxGetNumberOfElements (prhs[0]);
  plhs[0] = mxCreateNumericArray (mxGetNumberOfDimensions (prhs[0]),
                                  mxGetDimensions (prhs[0]),
                                  mxGetClassID (prhs[0]),
                                  mxIsComplex (prhs[0]));
  vri = mxGetPr (prhs[0]);
  vro = mxGetPr (plhs[0]);

  if (mxIsComplex (prhs[0]))
    {
      double *vii, *vio;
      vii = mxGetPi (prhs[0]);
      vio = mxGetPi (plhs[0]);

      for (i = 0; i < n; i++)
        {
          vro[i] = vri[i] * vri[i] - vii[i] * vii[i];
```

```
              vio[i] = 2 * vri[i] * vii[i];
            }
        }
      else
        {
          for (i = 0; i < n; i++)
            vro[i] = vri[i] * vri[i];
        }
    }
```

with an example of its use

```
b = randn (4,1) + 1i * randn (4,1);
all (b.^2 == mypow2 (b))
⇒ 1
```

The example above uses the functions `mxGetDimensions`, `mxGetNumberOfElements`, and `mxGetNumberOfDimensions` to work with the dimensions of multi-dimensional arrays. The functions `mxGetM`, and `mxGetN` are also available to find the number of rows and columns in a 2-D matrix.

A.2.3 Character Strings in Mex-Files

As mex-files do not make the distinction between single and double quoted strings within Octave, there is perhaps less complexity in the use of strings and character matrices in mex-files. An example of their use that parallels the demo in `stringdemo.cc` is given in the file `mystring.c`, as shown below.

```
#include <string.h>
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[],
             int nrhs, const mxArray *prhs[])
{
  mwSize m, n;
  mwIndex i, j;
  mxChar *pi, *po;

  if (nrhs != 1 || ! mxIsChar (prhs[0])
      || mxGetNumberOfDimensions (prhs[0]) > 2)
    mexErrMsgTxt ("ARG1 must be a char matrix");

  m = mxGetM (prhs[0]);
  n = mxGetN (prhs[0]);
  pi = mxGetChars (prhs[0]);
  plhs[0] = mxCreateNumericMatrix (m, n, mxCHAR_CLASS, mxREAL);
  po = mxGetChars (plhs[0]);

  for (j = 0; j < n; j++)
```

```
    for (i = 0; i < m; i++)
      po[j*m + m - 1 - i] = pi[j*m + i];
}
```

An example of its expected output is

```
mystring (["First String"; "Second String"])
⇒ Second String
   First String
```

Other functions in the mex interface for handling character strings are `mxCreateString`, `mxArrayToString`, and `mxCreateCharMatrixFromStrings`. In a mex-file, a character string is considered to be a vector rather than a matrix. This is perhaps an arbitrary distinction as the data in the mxArray for the matrix is consecutive in any case.

A.2.4 Cell Arrays with Mex-Files

One can perform exactly the same operations on Cell arrays in mex-files as in oct-files. An example that reduplicates the function of the `celldemo.cc` oct-file in a mex-file is given by `mycell.c` as shown below.

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[],
             int nrhs, const mxArray* prhs[])
{
  mwSize n;
  mwIndex i;

  if (nrhs != 1 || ! mxIsCell (prhs[0]))
    mexErrMsgTxt ("ARG1 must be a cell");

  n = mxGetNumberOfElements (prhs[0]);
  n = (n > nlhs ? nlhs : n);

  for (i = 0; i < n; i++)
    plhs[i] = mxDuplicateArray (mxGetCell (prhs[0], i));
}
```

The output is identical to the oct-file version as well.

```
[b1, b2, b3] = mycell ({1, [1, 2], "test"})
⇒
b1 =  1
b2 =

   1   2

b3 = test
```

Note in the example the use of the mxDuplicateArray function. This is needed as the mxArray pointer returned by mxGetCell might be deallocated. The inverse function to mxGetCell, used for setting Cell values, is mxSetCell and is defined as

```
void mxSetCell (mxArray *ptr, int idx, mxArray *val);
```

Finally, to create a cell array or matrix, the appropriate functions are

```
mxArray *mxCreateCellArray (int ndims, const int *dims);
mxArray *mxCreateCellMatrix (int m, int n);
```

A.2.5 Structures with Mex-Files

The basic function to create a structure in a mex-file is mxCreateStructMatrix which creates a structure array with a two dimensional matrix, or mxCreateStructArray.

```
mxArray *mxCreateStructArray (int ndims, int *dims,
                              int num_keys,
                              const char **keys);
mxArray *mxCreateStructMatrix (int rows, int cols,
                               int num_keys,
                               const char **keys);
```

Accessing the fields of the structure can then be performed with mxGetField and mxSetField or alternatively with the mxGetFieldByNumber and mxSetFieldByNumber functions.

```
mxArray *mxGetField (const mxArray *ptr, mwIndex index,
                     const char *key);
mxArray *mxGetFieldByNumber (const mxArray *ptr,
                             mwIndex index, int key_num);
void mxSetField (mxArray *ptr, mwIndex index,
                 const char *key, mxArray *val);
void mxSetFieldByNumber (mxArray *ptr, mwIndex index,
                         int key_num, mxArray *val);
```

A difference between the oct-file interface to structures and the mex-file version is that the functions to operate on structures in mex-files directly include an index over the elements of the arrays of elements per field; Whereas, the oct-file structure includes a Cell Array per field of the structure.

An example that demonstrates the use of structures in a mex-file can be found in the file mystruct.c shown below.

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[],
             int nrhs, const mxArray* prhs[])
{
  int i;
  mwIndex j;
  mxArray *v;
  const char *keys[] = { "this", "that" };
```

```
  if (nrhs != 1 || ! mxIsStruct (prhs[0]))
    mexErrMsgTxt ("ARG1 must be a struct");

  for (i = 0; i < mxGetNumberOfFields (prhs[0]); i++)
    for (j = 0; j < mxGetNumberOfElements (prhs[0]); j++)
      {
        mexPrintf ("field %s(%d) = ", mxGetFieldNameByNumber (prhs[0], i), j);
        v = mxGetFieldByNumber (prhs[0], j, i);
        mexCallMATLAB (0, NULL, 1, &v, "disp");
      }

  v = mxCreateStructMatrix (2, 2, 2, keys);

  mxSetFieldByNumber (v, 0, 0, mxCreateString ("this1"));
  mxSetFieldByNumber (v, 0, 1, mxCreateString ("that1"));
  mxSetFieldByNumber (v, 1, 0, mxCreateString ("this2"));
  mxSetFieldByNumber (v, 1, 1, mxCreateString ("that2"));
  mxSetFieldByNumber (v, 2, 0, mxCreateString ("this3"));
  mxSetFieldByNumber (v, 2, 1, mxCreateString ("that3"));
  mxSetFieldByNumber (v, 3, 0, mxCreateString ("this4"));
  mxSetFieldByNumber (v, 3, 1, mxCreateString ("that4"));

  if (nlhs)
    plhs[0] = v;
}
```

An example of the behavior of this function within Octave is then

```
a(1).f1 = "f11"; a(1).f2 = "f12";
a(2).f1 = "f21"; a(2).f2 = "f22";
b = mystruct (a);
⇒  field f1(0) = f11
    field f1(1) = f21
    field f2(0) = f12
    field f2(1) = f22
b
⇒ 2x2 struct array containing the fields:

     this
     that

b(3)
⇒ scalar structure containing the fields:

     this = this3
     that = that3
```

A.2.6 Sparse Matrices with Mex-Files

The Octave format for sparse matrices is identical to the mex format in that it is a compressed column sparse format. Also in both, sparse matrices are required to be two-dimensional. The only difference is that the real and imaginary parts of the matrix are stored separately.

The mex-file interface, in addition to using `mxGetM`, `mxGetN`, `mxSetM`, `mxSetN`, `mxGetPr`, `mxGetPi`, `mxSetPr`, and `mxSetPi`, also supplies the following functions.

```
mwIndex *mxGetIr (const mxArray *ptr);
mwIndex *mxGetJc (const mxArray *ptr);
mwSize mxGetNzmax (const mxArray *ptr);

void mxSetIr (mxArray *ptr, mwIndex *ir);
void mxSetJc (mxArray *ptr, mwIndex *jc);
void mxSetNzmax (mxArray *ptr, mwSize nzmax);
```

`mxGetNzmax` gets the maximum number of elements that can be stored in the sparse matrix. This is not necessarily the number of nonzero elements in the sparse matrix. `mxGetJc` returns an array with one additional value than the number of columns in the sparse matrix. The difference between consecutive values of the array returned by `mxGetJc` define the number of nonzero elements in each column of the sparse matrix. Therefore,

```
mwSize nz, n;
mwIndex *Jc;
mxArray *m;
...
n = mxGetN (m);
Jc = mxGetJc (m);
nz = Jc[n];
```

returns the actual number of nonzero elements stored in the matrix in `nz`. As the arrays returned by `mxGetPr` and `mxGetPi` only contain the nonzero values of the matrix, we also need a pointer to the rows of the nonzero elements, and this is given by `mxGetIr`. A complete example of the use of sparse matrices in mex-files is given by the file `mysparse.c` shown below.

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray *plhs[],
             int nrhs, const mxArray *prhs[])
{
  mwSize m, n, nz;
  mxArray *v;
  mwIndex i;
  double *pr, *pi;
  double *pr2, *pi2;
  mwIndex *ir, *jc;
  mwIndex *ir2, *jc2;
```

```
if (nrhs != 1 || ! mxIsSparse (prhs[0]))
  mexErrMsgTxt ("ARG1 must be a sparse matrix");

m = mxGetM (prhs[0]);
n = mxGetN (prhs[0]);
nz = mxGetNzmax (prhs[0]);

if (mxIsComplex (prhs[0]))
  {
    mexPrintf ("Matrix is %d-by-%d complex sparse matrix", m, n);
    mexPrintf (" with %d elements\n", nz);

    pr = mxGetPr (prhs[0]);
    pi = mxGetPi (prhs[0]);
    ir = mxGetIr (prhs[0]);
    jc = mxGetJc (prhs[0]);

    i = n;
    while (jc[i] == jc[i-1] && i != 0) i--;

    mexPrintf ("last nonzero element (%d, %d) = (%g, %g)\n",
               ir[nz-1]+ 1, i, pr[nz-1], pi[nz-1]);

    v = mxCreateSparse (m, n, nz, mxCOMPLEX);
    pr2 = mxGetPr (v);
    pi2 = mxGetPi (v);
    ir2 = mxGetIr (v);
    jc2 = mxGetJc (v);

    for (i = 0; i < nz; i++)
      {
        pr2[i] = 2 * pr[i];
        pi2[i] = 2 * pi[i];
        ir2[i] = ir[i];
      }
    for (i = 0; i < n + 1; i++)
      jc2[i] = jc[i];

    if (nlhs > 0)
      plhs[0] = v;
  }
else if (mxIsLogical (prhs[0]))
  {
    mxLogical *pbr, *pbr2;
    mexPrintf ("Matrix is %d-by-%d logical sparse matrix", m, n);
    mexPrintf (" with %d elements\n", nz);
```

```
      pbr = mxGetLogicals (prhs[0]);
      ir = mxGetIr (prhs[0]);
      jc = mxGetJc (prhs[0]);

      i = n;
      while (jc[i] == jc[i-1] && i != 0) i--;
      mexPrintf ("last nonzero element (%d, %d) = %d\n",
                 ir[nz-1]+ 1, i, pbr[nz-1]);

      v = mxCreateSparseLogicalMatrix (m, n, nz);
      pbr2 = mxGetLogicals (v);
      ir2 = mxGetIr (v);
      jc2 = mxGetJc (v);

      for (i = 0; i < nz; i++)
        {
          pbr2[i] = pbr[i];
          ir2[i] = ir[i];
        }
      for (i = 0; i < n + 1; i++)
        jc2[i] = jc[i];

      if (nlhs > 0)
        plhs[0] = v;
    }
  else
    {
      mexPrintf ("Matrix is %d-by-%d real sparse matrix", m, n);
      mexPrintf (" with %d elements\n", nz);

      pr = mxGetPr (prhs[0]);
      ir = mxGetIr (prhs[0]);
      jc = mxGetJc (prhs[0]);

      i = n;
      while (jc[i] == jc[i-1] && i != 0) i--;
      mexPrintf ("last nonzero element (%d, %d) = %g\n",
                 ir[nz-1]+ 1, i, pr[nz-1]);

      v = mxCreateSparse (m, n, nz, mxREAL);
      pr2 = mxGetPr (v);
      ir2 = mxGetIr (v);
      jc2 = mxGetJc (v);

      for (i = 0; i < nz; i++)
        {
          pr2[i] = 2 * pr[i];
```

```
     ir2[i] = ir[i];
    }
   for (i = 0; i < n + 1; i++)
    jc2[i] = jc[i];

   if (nlhs > 0)
    plhs[0] = v;
  }
}
```

A sample usage of `mysparse` is

```
sm = sparse ([1, 0; 0, pi]);
mysparse (sm)
⇒
Matrix is 2-by-2 real sparse matrix with 2 elements
last nonzero element (2, 2) = 3.14159
```

A.2.7 Calling Other Functions in Mex-Files

It is possible to call other Octave functions from within a mex-file using `mexCallMATLAB`. An example of the use of `mexCallMATLAB` can be see in the example below.

```
#include "mex.h"

void
mexFunction (int nlhs, mxArray* plhs[],
             int nrhs, const mxArray* prhs[])
{
  char *str;

  mexPrintf ("Starting file myfeval.mex\n");

  mexPrintf ("I have %d inputs and %d outputs\n", nrhs, nlhs);

  if (nrhs < 1 || ! mxIsChar (prhs[0]))
    mexErrMsgTxt ("ARG1 must be a function name");

  str = mxArrayToString (prhs[0]);

  mexPrintf ("I'm going to call the function %s\n", str);

  if (nlhs == 0)
    nlhs = 1;  // Octave's automatic 'ans' variable

  /* Cast prhs just to get rid of 'const' qualifier and stop compile warning */
  mexCallMATLAB (nlhs, plhs, nrhs-1, (mxArray**)prhs+1, str);

  mxFree (str);
```

```
}
```

If this code is in the file `myfeval.c`, and is compiled to `myfeval.mex`, then an example of its use is

```
a = myfeval ("sin", 1)
⇒ Starting file myfeval.mex
   I have 2 inputs and 1 outputs
   I'm going to call the interpreter function sin
   a =  0.84147
```

Note that it is not possible to use function handles or inline functions within a mex-file.

A.3 Standalone Programs

The libraries Octave itself uses can be utilized in standalone applications. These applications then have access, for example, to the array and matrix classes, as well as to all of the Octave algorithms. The following C++ program, uses class Matrix from `liboctave.a` or `liboctave.so`.

```
#include <iostream>
#include <octave/oct.h>

int
main (void)
{
  std::cout << "Hello Octave world!\n";

  int n = 2;
  Matrix a_matrix = Matrix (n, n);

  for (octave_idx_type i = 0; i < n; i++)
    for (octave_idx_type j = 0; j < n; j++)
      a_matrix(i,j) = (i + 1) * 10 + (j + 1);

  std::cout << a_matrix;

  return 0;
}
```

mkoctfile can be used to build a standalone application with a command like

```
$ mkoctfile --link-stand-alone standalone.cc -o standalone
$ ./standalone
Hello Octave world!
  11 12
  21 22
$
```

Note that the application `standalone` will be dynamically linked against the Octave libraries and any Octave support libraries. The above allows the Octave math libraries to be used by an application. It does not, however, allow the script files, oct-files, or built-in functions of Octave to be used by the application. To do that the Octave interpreter needs to be initialized first. An example of how to do this can then be seen in the code

```
#include <iostream>
#include <octave/oct.h>
#include <octave/octave.h>
#include <octave/parse.h>
#include <octave/toplev.h>

int
main (void)
{
  string_vector argv (2);
  argv(0) = "embedded";
  argv(1) = "-q";

  octave_main (2, argv.c_str_vec (), 1);

  octave_idx_type n = 2;
  octave_value_list in;

  for (octave_idx_type i = 0; i < n; i++)
    in(i) = octave_value (5 * (i + 2));

  octave_value_list out = feval ("gcd", in, 1);

  if (! error_state && out.length () > 0)
    std::cout << "GCD of ["
              << in(0).int_value ()
              << ", "
              << in(1).int_value ()
              << "] is " << out(0).int_value ()
              << std::endl;
  else
    std::cout << "invalid\n";

  clean_up_and_exit (0);
}
```

which, as before, is compiled and run as a standalone application with

```
$ mkoctfile --link-stand-alone embedded.cc -o embedded
$ ./embedded
GCD of [10, 15] is 5
$
```

It is worth noting that, if only built-in functions are to be called from a C++ standalone program, then it does not need to initialize the interpreter to do so. The general rule is that, for a built-in function named `function_name` in the interpreter, there will be a C++ function named `Ffunction_name` (note the prepended capital `F`) accessible in the C++ API. The declarations for all built-in functions are collected in the header file `builtin-defun-decls.h`. This feature should be used with care as the list of built-in functions can change. No guarantees can be made that a function that is currently built in won't be implemented as a .m file or as a dynamically linked function in the future. An example of how to call built-in functions from C++ can be seen in the code

```
#include <iostream>
#include <octave/oct.h>
#include <octave/builtin-defun-decls.h>

int
main (void)
{
  int n = 2;
  Matrix a_matrix = Matrix (n, n);

  for (octave_idx_type i = 0; i < n; i++)
    for (octave_idx_type j = 0; j < n; j++)
      a_matrix(i,j) = (i + 1) * 10 + (j + 1);

  std::cout << "This is a matrix:" << std::endl
            << a_matrix            << std::endl;

  octave_value_list in;
  in(0) = a_matrix;

  octave_value_list out = Fnorm (in, 1);
  double norm_of_the_matrix = out(0).double_value ();

  std::cout << "This is the norm of the matrix:" << std::endl
            << norm_of_the_matrix                << std::endl;

  return 0;
}
```

which, again, is compiled and run as a standalone application with

```
$ mkoctfile --link-stand-alone standalonebuiltin.cc -o standalonebuiltin
$ ./standalonebuiltin
This is a matrix:
 11 12
 21 22

This is the norm of the matrix:
34.4952
$
```

Appendix B Test and Demo Functions

Octave includes a number of functions to allow the integration of testing and demonstration code in the source code of the functions themselves.

B.1 Test Functions

`test` *name* [Command]
`test` *name quiet|normal|verbose* [Command]
`test ("`*name*`", "`*quiet|normal|verbose*`",` *fid*`)` [Function File]
`test ("`*name*`", "`*quiet|normal|verbose*`",` *fname*`)` [Function File]
success `= test (...)` [Function File]
`[`*n*`,` *nmax*`,` *nxfail*`,` *nskip*`] = test (...)` [Function File]
`[`*code*`,` *idx*`] = test ("`*name*`", "`*grabdemo*`")` [Function File]
`test ([], "`*explain*`",` *fid*`)` [Function File]
`test ([], "`*explain*`",` *fname*`)` [Function File]

Perform built-in self-tests from the first file in the loadpath matching *name*.

`test` can be called in either command or functional form. The exact operation of test is determined by a combination of mode (interactive or batch), reporting level (`"quiet"`, `"normal"`, `"verbose"`), and whether a logfile or summary output variable is used.

The default mode when `test` is called from the command line is interactive. In this mode, tests will be run until the first error is encountered, or all tests complete successfully. In batch mode, all tests are run regardless of any failures, and the results are collected for reporting. Tests which require user interaction, i.e., demo blocks, are never run in batch mode.

Batch mode is enabled by either 1) specifying a logfile using the third argument *fname* or *fid*, or 2) requesting an output argument such as *success*, *n*, etc.

The optional second argument determines the amount of output to generate and which types of tests to run. The default value is `"normal"`. Requesting an output argument will suppress printing the final summary message and any intermediate warnings, unless verbose reporting is enabled.

`"quiet"` Print a summary message when all tests pass, or print an error with the results of the first bad test when a failure occurs. Don't run tests which require user interaction.

`"normal"` Display warning messages about skipped tests or failing xtests during test execution. Print a summary message when all tests pass, or print an error with the results of the first bad test when a failure occurs. Don't run tests which require user interaction.

`"verbose"`

Display tests before execution. Print all warning messages. In interactive mode, run all tests including those which require user interaction.

The optional third input argument specifies a logfile where results of the tests should be written. The logfile may be a character string (*fname*) or an open file descriptor

ID (*fid*). To enable batch processing, but still print the results to the screen, use `stdout` for *fid*.

When called with just a single output argument *success*, `test` returns true if all of the tests were successful. If called with more than one output argument then the number of successful tests (*n*), the total number of tests in the file (*nmax*), the number of xtest failures (*nxfail*), and the number of skipped tests (*nskip* are returned.

Example

```
test sind
⇒
PASSES 5 out of 5 tests

[n, nmax] = test ("sind")
⇒
n =  5
nmax =  5
```

Additional Calling Syntaxes

If the second argument is the string `"grabdemo"`, the contents of any built-in demo blocks are extracted but not executed. The text for all code blocks is concatenated and returned as *code* with *idx* being a vector of positions of the ends of each demo block. For an easier way to extract demo blocks from files, See [example], page 865.

If the second argument is `"explain"` then *name* is ignored and an explanation of the line markers used in `test` output reports is written to the file specified by *fname* or *fid*.

See also: [assert], page 862, [fail], page 863, [demo], page 864, [example], page 865, [error], page 205.

`test` scans the named script file looking for lines which start with the identifier '`%!`'. The prefix is stripped off and the rest of the line is processed through the Octave interpreter. If the code generates an error, then the test is said to fail.

Since `eval()` will stop at the first error it encounters, you must divide your tests up into blocks, with anything in a separate block evaluated separately. Blocks are introduced by valid keywords like `test`, `function`, or `assert` immediately following '`%!`'. A block is defined by indentation as in Python. Lines beginning with '`%!<whitespace>`' are part of the preceeding block.

For example:

```
%!test error ("this test fails!");
%!test "test doesn't fail. it doesn't generate an error";
```

When a test fails, you will see something like:

```
  ***** test error ("this test fails!")
!!!!! test failed
this test fails!
```

Generally, to test if something works, you want to assert that it produces a correct value. A real test might look something like

```
%!test
%! a = [1, 2, 3; 4, 5, 6]; B = [1; 2];
%! expect = [ a ; 2*a ];
%! get = kron (b, a);
%! if (any (size (expect) != size (get)))
%!   error ("wrong size: expected %d,%d but got %d,%d",
%!          size (expect), size (get));
%! elseif (any (any (expect != get)))
%!   error ("didn't get what was expected.");
%! endif
```

To make the process easier, use the `assert` function. For example, with `assert` the previous test is reduced to:

```
%!test
%! a = [1, 2, 3; 4, 5, 6]; b = [1; 2];
%! assert (kron (b, a), [ a; 2*a ]);
```

`assert` can accept a tolerance so that you can compare results absolutely or relatively. For example, the following all succeed:

```
%!test assert (1+eps, 1, 2*eps)          # absolute error
%!test assert (100+100*eps, 100, -2*eps) # relative error
```

You can also do the comparison yourself, but still have assert generate the error:

```
%!test assert (isempty ([]))
%!test assert ([1, 2; 3, 4] > 0)
```

Because `assert` is so frequently used alone in a test block, there is a shorthand form:

```
%!assert (...)
```

which is equivalent to:

```
%!test assert (...)
```

Occasionally a block of tests will depend on having optional functionality in Octave. Before testing such blocks the availability of the required functionality must be checked. A `%!testif HAVE_XXX` block will only be run if Octave was compiled with functionality '`HAVE_XXX`'. For example, the sparse single value decomposition, `svds()`, depends on having the ARPACK library. All of the tests for `svds` begin with

```
%!testif HAVE_ARPACK
```

Review `config.h` or `octave_config_info ("features")` to see some of the possible values to check.

Sometimes during development there is a test that should work but is known to fail. You still want to leave the test in because when the final code is ready the test should pass, but you may not be able to fix it immediately. To avoid unnecessary bug reports for these known failures, mark the block with `xtest` rather than `test`:

```
%!xtest assert (1==0)
%!xtest fail ("success=1", "error")
```

In this case, the test will run and any failure will be reported. However, testing is not aborted and subsequent test blocks will be processed normally. Another use of `xtest` is for statistical tests which should pass most of the time but are known to fail occasionally.

Each block is evaluated in its own function environment, which means that variables defined in one block are not automatically shared with other blocks. If you do want to share variables, then you must declare them as `shared` before you use them. For example, the following declares the variable *a*, gives it an initial value (default is empty), and then uses it in several subsequent tests.

```
%!shared a
%! a = [1, 2, 3; 4, 5, 6];
%!assert (kron ([1; 2], a), [ a; 2*a ]);
%!assert (kron ([1, 2], a), [ a, 2*a ]);
%!assert (kron ([1,2; 3,4], a), [ a,2*a; 3*a,4*a ]);
```

You can share several variables at the same time:

```
%!shared a, b
```

You can also share test functions:

```
%!function a = fn (b)
%!  a = 2*b;
%!endfunction
%!assert (fn(2), 4);
```

Note that all previous variables and values are lost when a new shared block is declared.

Remember that `%!function` begins a new block and that `%!endfunction` ends this block. Be aware that until a new block is started, lines starting with '`%!<space>`' will be discarded as comments. The following is nearly identical to the example above, but does nothing.

```
%!function a = fn (b)
%!  a = 2*b;
%!endfunction
%! assert (fn(2), 4);
```

Because there is a space after '`%!`' the `assert` statement does not begin a new block and this line is treated as a comment.

Error and warning blocks are like test blocks, but they only succeed if the code generates an error. You can check the text of the error is correct using an optional regular expression `<pattern>`. For example:

```
%!error <passes!> error ("this test passes!");
```

If the code doesn't generate an error, the test fails. For example:

```
%!error "this is an error because it succeeds.";
```

produces

```
  ***** error "this is an error because it succeeds.";
!!!!! test failed: no error
```

It is important to automate the tests as much as possible, however some tests require user interaction. These can be isolated into demo blocks, which if you are in batch mode, are only run when called with `demo` or the `verbose` option to `test`. The code is displayed before it is executed. For example,

```
%!demo
%! t = [0:0.01:2*pi]; x = sin (t);
%! plot (t, x);
%! # you should now see a sine wave in your figure window
```

produces

```
funcname example 1:
 t = [0:0.01:2*pi]; x = sin (t);
 plot (t, x);
 # you should now see a sine wave in your figure window

Press <enter> to continue:
```

Note that demo blocks cannot use any shared variables. This is so that they can be executed by themselves, ignoring all other tests.

If you want to temporarily disable a test block, put # in place of the block type. This creates a comment block which is echoed in the log file but not executed. For example:

```
%!#demo
%! t = [0:0.01:2*pi]; x = sin (t);
%! plot (t, x);
%! # you should now see a sine wave in your figure window
```

The following trivial code snippet provides examples for the use of fail, assert, error and xtest:

```
function output = must_be_zero (input)
  if (input != 0)
    error ("Nonzero input!")
  endif
  output = input;
endfunction

%!fail ("must_be_zero (1)");
%!assert (must_be_zero (0), 0);
%!error <Nonzero> must_be_zero (1);
%!xtest error ("This code generates an error");
```

When putting this a file `must_be_zero.m`, and running the test, we see

```
test must_be_zero verbose

⇒
>>>>> /path/to/must_be_zero.m
  ***** fail ("must_be_zero (1)");
  ***** assert (must_be_zero (0), 0);
  ***** error <Nonzero> must_be_zero (1);
  ***** xtest error ("This code generates an error");
!!!!! known failure
This code generates an error
PASSES 4 out of 4 tests (1 expected failures)
```

Block type summary:

`%!test` check that entire block is correct

`%!testif HAVE_XXX`
check block only if Octave was compiled with feature HAVE_XXX.

`%!xtest` check block, report a test failure but do not abort testing.

`%!error` check for correct error message

`%!warning`
check for correct warning message

`%!demo` demo only executes in interactive mode

`%!#` comment: ignore everything within the block

`%!shared x,y,z`
declare variables for use in multiple tests

`%!function`
define a function for use in multiple tests

`%!endfunction`
close a function definition

`%!assert (x, y, tol)`
shorthand for `%!test assert (x, y, tol)`

You can also create test scripts for built-in functions and your own C++ functions. To do so, put a file with the bare function name (no .m extension) in a directory in the load path and it will be discovered by the `test` function. Alternatively, you can embed tests directly in your C++ code:

```
/*
%!test disp ("this is a test")
*/
```

or

```
#if 0
%!test disp ("this is a test")
#endif
```

However, in this case the raw source code will need to be on the load path and the user will have to remember to type `test ("funcname.cc")`.

`assert` (*cond*) [Function File]
`assert` (*cond*, *errmsg*) [Function File]
`assert` (*cond*, *errmsg*, ...) [Function File]
`assert` (*cond*, *msg_id*, *errmsg*, ...) [Function File]
`assert` (*observed*, *expected*) [Function File]
`assert` (*observed*, *expected*, *tol*) [Function File]

Produce an error if the specified condition is not met.

`assert` can be called in three different ways.

`assert (`*cond*`)`
`assert (`*cond*`, `*errmsg*`)`
`assert (`*cond*`, `*errmsg*`, ...)`
`assert (`*cond*`, `*msg_id*`, `*errmsg*`, ...)`

Called with a single argument *cond*, `assert` produces an error if *cond* is false (numeric zero).

Any additional arguments are passed to the `error` function for processing.

`assert (`*observed*`, `*expected*`)`

Produce an error if observed is not the same as expected.

Note that *observed* and *expected* can be scalars, vectors, matrices, strings, cell arrays, or structures.

`assert (`*observed*`, `*expected*`, `*tol*`)`

Produce an error if observed is not the same as expected but equality comparison for numeric data uses a tolerance *tol*.

If *tol* is positive then it is an absolute tolerance which will produce an error if `abs (`*observed* `-` *expected*`) > abs (`*tol*`)`.

If *tol* is negative then it is a relative tolerance which will produce an error if `abs (`*observed* `-` *expected*`) > abs (`*tol* `*` *expected*`)`.

If *expected* is zero *tol* will always be interpreted as an absolute tolerance.

If *tol* is not scalar its dimensions must agree with those of *observed* and *expected* and tests are performed on an element-by-element basis.

See also: [fail], page 863, [test], page 857, [error], page 205, [isequal], page 146.

`fail (`*code*`)` [Function File]
`fail (`*code*`, `*pattern*`)` [Function File]
`fail (`*code*`, "`*warning*`")` [Function File]
`fail (`*code*`, "`*warning*`", `*pattern*`)` [Function File]

Return true if *code* fails with an error message matching *pattern*, otherwise produce an error.

code must be in the form of a string that is passed to the Octave interpreter via the `evalin` function, i.e., a (quoted) string constant or a string variable.

Note that if *code* runs successfully, rather than failing, the error printed is:

```
expected error <.> but got none
```

If called with two arguments, the return value will be true only if *code* fails with an error message containing *pattern* (case sensitive). If the code fails with a different error than the one specified in *pattern* then the message produced is:

```
expected <pattern>
but got <text of actual error>
```

The angle brackets are not part of the output.

When called with the `"warning"` option `fail` will produce an error if executing the code produces no warning.

See also: [assert], page 862, [error], page 205.

B.2 Demonstration Functions

`demo name` [Command]
`demo name n` [Command]
`demo ("name")` [Function File]
`demo ("name", n)` [Function File]

Run example code block *n* associated with the function *name*.

If *n* is not specified, all examples are run.

The preferred location for example code blocks is embedded within the script m-file immediately following the code that it exercises. Alternatively, the examples may be stored in a file with the same name but no extension located on Octave's load path. To separate examples from regular script code all lines are prefixed by `%!`. Each example must also be introduced by the keyword `"demo"` flush left to the prefix with no intervening spaces. The remainder of the example can contain arbitrary Octave code. For example:

```
%!demo
%! t = 0:0.01:2*pi;
%! x = sin (t);
%! plot (t, x);
%! title ("one cycle of a sine wave");
%! #-------------------------------------------------
%! # the figure window shows one cycle of a sine wave
```

Note that the code is displayed before it is executed so that a simple comment at the end suffices for labeling what is being shown. For plots, labeling can also be done with `title` or `text`. It is generally **not** necessary to use `disp` or `printf` within the demo.

Demos are run in a stand-alone function environment with no access to external variables. This means that every demo must have separate initialization code. Alternatively, all demos can be combined into a single large demo with the code

```
%! input ("Press <enter> to continue: ", "s");
```

between the sections, but this usage is discouraged. Other techniques to avoid multiple initialization blocks include using multiple plots with a new `figure` command between each plot, or using `subplot` to put multiple plots in the same window.

Finally, because `demo` evaluates within a function context it is not possible to define new functions within the code. Anonymous functions make a good substitute in most instances. If function blocks **must** be used then the code `eval (example ("function", n))` will allow Octave to see them. This has its own problems, however, as `eval` only evaluates one line or statement at a time. In this case the function declaration must be wrapped with `"if 1 <demo stuff> endif"` where `"if"` is on the same line as `"demo"`. For example:

```
%!demo if 1
%!  function y = f(x)
%!    y = x;
%!  endfunction
%!  f(3)
%! endif
```

See also: [rundemos], page 865, [example], page 865, [test], page 857.

`example name` [Command]
`example name n` [Command]
`example ("name")` [Function File]
`example ("name", n)` [Function File]
`[s, idx] = example (...)` [Function File]

Display the code for example *n* associated with the function *name*, but do not run it.

If *n* is not specified, all examples are displayed.

When called with output arguments, the examples are returned in the form of a string *s*, with *idx* indicating the ending position of the various examples.

See `demo` for a complete explanation.

See also: [demo], page 864, [test], page 857.

`rundemos ()` [Function File]
`rundemos (directory)` [Function File]

Execute built-in demos for all m-files in the specified *directory*.

Demo blocks in any C++ source files (`*.cc`) will also be executed for use with dynamically linked oct-file functions.

If no directory is specified, operate on all directories in Octave's search path for functions.

See also: [demo], page 864, [runtests], page 865, [path], page 190.

`runtests ()` [Function File]
`runtests (directory)` [Function File]

Execute built-in tests for all m-files in the specified *directory*.

Test blocks in any C++ source files (`*.cc`) will also be executed for use with dynamically linked oct-file functions.

If no directory is specified, operate on all directories in Octave's search path for functions.

See also: [rundemos], page 865, [test], page 857, [path], page 190.

`speed (f, init, max_n, f2, tol)` [Function File]
`[order, n, T_f, T_f2] = speed (...)` [Function File]

Determine the execution time of an expression (*f*) for various input values (*n*).

The *n* are log-spaced from 1 to *max_n*. For each *n*, an initialization expression (*init*) is computed to create any data needed for the test. If a second expression (*f2*) is given then the execution times of the two expressions are compared. When called without output arguments the results are printed to stdout and displayed graphically.

f The code expression to evaluate.

max_n The maximum test length to run. The default value is 100. Alternatively, use `[min_n, max_n]` or specify the *n* exactly with `[n1, n2, ..., nk]`.

init Initialization expression for function argument values. Use *k* for the test number and *n* for the size of the test. This should compute values for all variables used by *f*. Note that *init* will be evaluated first for $k = 0$, so things which are constant throughout the test series can be computed once. The default value is `x = randn (n, 1)`.

f2 An alternative expression to evaluate, so that the speed of two expressions can be directly compared. The default is `[]`.

tol Tolerance used to compare the results of expression *f* and expression *f2*. If *tol* is positive, the tolerance is an absolute one. If *tol* is negative, the tolerance is a relative one. The default is `eps`. If *tol* is `Inf`, then no comparison will be made.

order The time complexity of the expression $O(a*n^p)$. This is a structure with fields `a` and `p`.

n The values *n* for which the expression was calculated **AND** the execution time was greater than zero.

T_f The nonzero execution times recorded for the expression *f* in seconds.

T_f2 The nonzero execution times recorded for the expression *f2* in seconds. If required, the mean time ratio is simply `mean (T_f ./ T_f2)`.

The slope of the execution time graph shows the approximate power of the asymptotic running time $O(n^p)$. This power is plotted for the region over which it is approximated (the latter half of the graph). The estimated power is not very accurate, but should be sufficient to determine the general order of an algorithm. It should indicate if, for example, the implementation is unexpectedly $O(n^2)$ rather than $O(n)$ because it extends a vector each time through the loop rather than pre-allocating storage. In the current version of Octave, the following is not the expected $O(n)$.

```
speed ("for i = 1:n, y{i} = x(i); endfor", "", [1000, 10000])
```

But it is if you preallocate the cell array `y`:

```
speed ("for i = 1:n, y{i} = x(i); endfor", ...
       "x = rand (n, 1); y = cell (size (x));", [1000, 10000])
```

An attempt is made to approximate the cost of individual operations, but it is wildly inaccurate. You can improve the stability somewhat by doing more work for each `n`. For example:

```
speed ("airy(x)", "x = rand (n, 10)", [10000, 100000])
```

When comparing two different expressions (*f*, *f2*), the slope of the line on the speedup ratio graph should be larger than 1 if the new expression is faster. Better algorithms have a shallow slope. Generally, vectorizing an algorithm will not change the slope of the execution time graph, but will shift it relative to the original. For example:

```
speed ("sum (x)", "", [10000, 100000], ...
       "v = 0; for i = 1:length (x), v += x(i); endfor")
```

The following is a more complex example. If there was an original version of `xcorr` using for loops and a second version using an FFT, then one could compare the run speed for various lags as follows, or for a fixed lag with varying vector lengths as follows:

```
speed ("xcorr (x, n)", "x = rand (128, 1);", 100,
       "xcorr_orig (x, n)", -100*eps)
speed ("xcorr (x, 15)", "x = rand (20+n, 1);", 100,
       "xcorr_orig (x, n)", -100*eps)
```

Assuming one of the two versions is in xcorr_orig, this would compare their speed and their output values. Note that the FFT version is not exact, so one must specify an acceptable tolerance on the comparison `100*eps`. In this case, the comparison should be computed relatively, as `abs ((x - y) ./ y)` rather than absolutely as `abs (x - y)`.

Type *`example ("speed")`* to see some real examples or *`demo ("speed")`* to run them.

Appendix C Tips and Standards

This chapter describes no additional features of Octave. Instead it gives advice on making effective use of the features described in the previous chapters.

C.1 Writing Clean Octave Programs

Here are some tips for avoiding common errors in writing Octave code intended for widespread use:

- Since all global variables share the same name space, and all functions share another name space, you should choose a short word to distinguish your program from other Octave programs. Then take care to begin the names of all global variables, constants, and functions with the chosen prefix. This helps avoid name conflicts.

  If you write a function that you think ought to be added to Octave under a certain name, such as `fiddle_matrix`, don't call it by that name in your program. Call it `mylib_fiddle_matrix` in your program, and send mail to `maintainers@octave.org` suggesting that it be added to Octave. If and when it is, the name can be changed easily enough.

  If one prefix is insufficient, your package may use two or three alternative common prefixes, so long as they make sense.

  Separate the prefix from the rest of the symbol name with an underscore '`_`'. This will be consistent with Octave itself and with most Octave programs.
- When you encounter an error condition, call the function `error` (or `usage`). The `error` and `usage` functions do not return. See Section 2.5 [Errors], page 35.
- Please put a copyright notice on the file if you give copies to anyone. Use the same lines that appear at the top of the function files distributed with Octave. If you have not signed papers to assign the copyright to anyone else, then place your name in the copyright notice.

C.2 Tips on Writing Comments

Here are the conventions to follow when writing comments.

'`#`' Comments that start with a single sharp-sign, '`#`', should all be aligned to the same column on the right of the source code. Such comments usually explain how the code on the same line does its job. In the Emacs mode for Octave, the *M-;* (`indent-for-comment`) command automatically inserts such a '`#`' in the right place, or aligns such a comment if it is already present.

'`##`' Comments that start with a double sharp-sign, '`##`', should be aligned to the same level of indentation as the code. Such comments usually describe the purpose of the following lines or the state of the program at that point.

The indentation commands of the Octave mode in Emacs, such as *M-;* (`indent-for-comment`) and *TAB* (`octave-indent-line`) automatically indent comments according to these conventions, depending on the number of semicolons. See Section "Manipulating Comments" in *The GNU Emacs Manual*.

C.3 Conventional Headers for Octave Functions

Octave has conventions for using special comments in function files to give information such as who wrote them. This section explains these conventions.

The top of the file should contain a copyright notice, followed by a block of comments that can be used as the help text for the function. Here is an example:

```
## Copyright (C) 1996, 1997, 2007 John W. Eaton
##
## This file is part of Octave.
##
## Octave is free software; you can redistribute it and/or
## modify it under the terms of the GNU General Public
## License as published by the Free Software Foundation;
## either version 3 of the License, or (at your option) any
## later version.
##
## Octave is distributed in the hope that it will be useful,
## but WITHOUT ANY WARRANTY; without even the implied
## warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
## PURPOSE.  See the GNU General Public License for more
## details.
##
## You should have received a copy of the GNU General Public
## License along with Octave; see the file COPYING.  If not,
## see <http://www.gnu.org/licenses/>.

## usage: [IN, OUT, PID] = popen2 (COMMAND, ARGS)
##
## Start a subprocess with two-way communication.  COMMAND
## specifies the name of the command to start.  ARGS is an
## array of strings containing options for COMMAND.  IN and
## OUT are the file ids of the input and streams for the
## subprocess, and PID is the process id of the subprocess,
## or -1 if COMMAND could not be executed.
##
## Example:
##
##  [in, out, pid] = popen2 ("sort", "-nr");
##  fputs (in, "these\nare\nsome\nstrings\n");
##  fclose (in);
##  while (ischar (s = fgets (out)))
##    fputs (stdout, s);
##  endwhile
##  fclose (out);
```

Octave uses the first block of comments in a function file that do not appear to be a copyright notice as the help text for the file. For Octave to recognize the first comment block as a copyright notice, it must start with the word 'Copyright' after stripping the leading comment characters.

After the copyright notice and help text come several *header comment* lines, each beginning with '`## header-name:`'. For example,

```
## Author: jwe
## Keywords: subprocesses input-output
## Maintainer: jwe
```

Here is a table of the conventional possibilities for *header-name*:

'`Author`' This line states the name and net address of at least the principal author of the library.

```
## Author: John W. Eaton <jwe@octave.org>
```

'`Maintainer`'
This line should contain a single name/address as in the Author line, or an address only, or the string '`jwe`'. If there is no maintainer line, the person(s) in the Author field are presumed to be the maintainers. The example above is mildly bogus because the maintainer line is redundant.

The idea behind the '`Author`' and '`Maintainer`' lines is to make possible a function to "send mail to the maintainer" without having to mine the name out by hand.

Be sure to surround the network address with '`<...>`' if you include the person's full name as well as the network address.

'`Created`' This optional line gives the original creation date of the file. For historical interest only.

'`Version`' If you wish to record version numbers for the individual Octave program, put them in this line.

'`Adapted-By`'
In this header line, place the name of the person who adapted the library for installation (to make it fit the style conventions, for example).

'`Keywords`'
This line lists keywords. Eventually, it will be used by an apropos command to allow people will find your package when they're looking for things by topic area. To separate the keywords, you can use spaces, commas, or both.

Just about every Octave function ought to have the '`Author`' and '`Keywords`' header comment lines. Use the others if they are appropriate. You can also put in header lines with other header names—they have no standard meanings, so they can't do any harm.

C.4 Tips for Documentation Strings

As noted above, documentation is typically in a commented header block on an Octave function following the copyright statement. The help string shown above is an unformatted string and will be displayed as is by Octave. Here are some tips for the writing of documentation strings.

- Every command, function, or variable intended for users to know about should have a documentation string.

- An internal variable or subroutine of an Octave program might as well have a documentation string.
- The first line of the documentation string should consist of a summary of the function.

  Subsequent lines may expand the general nature of the function.

  After the introduction there should be paragraphs describing the meaning and usage of each input, followed by the meaning and usage of each output.

  Finally, there may be more general information such as notes about the algorithm used, references to scientific papers, notes about any incompatibilities with MATLAB, etc.
- For consistency, phrase the verb in the first sentence of a documentation string as an infinitive with "to" omitted. For instance, use "Return the frob of A and B." in preference to "Returns the frob of A and B." Usually it looks good to do likewise for the rest of the first paragraph. Subsequent paragraphs usually look better if they have proper subjects.
- Write documentation strings in the active voice, not the passive, and in the present tense, not the future. For instance, use "Return a list containing A and B." instead of "A list containing A and B will be returned."
- Avoid using the word "cause" (or its equivalents) unnecessarily. Instead of, "Cause Octave to display text in boldface," just write "Display text in boldface."
- Use two spaces between the period marking the end of a sentence and the word which opens the next sentence. This convention has no effect for typeset formats like TeX, but improves the readability of the documentation in fixed-width environments such as the Info reader.
- Do not start or end a documentation string with whitespace.
- Format the documentation string so that it fits within an 80-column screen. It is a good idea for most lines to be no wider than 60 characters.

  However, rather than simply filling the entire documentation string, you can make it much more readable by choosing line breaks with care. Use blank lines between topics if the documentation string is long.
- **Do not** indent subsequent lines of a documentation string so that the text is lined up in the source code with the text of the first line. This looks nice in the source code, but looks bizarre when users view the documentation. Remember that the indentation before the starting double-quote is not part of the string!
- When choosing variable names try to adhere to the following guidelines.

  vectors : x,y,z,t,w

  matrices : A,B,M

  strings : str,s

  filenames : fname

  cells,cellstrs :
  c,cstr
- The documentation string for a variable that is a yes-or-no flag should start with words such as "Nonzero means...", to make it clear that all nonzero values are equivalent and indicate explicitly what zero and nonzero mean.

- When a function's documentation string mentions the value of an argument of the function, use the argument name in capital letters as if it were a name for that value. Thus, the documentation string of the operator `/` refers to its second argument as '`DIVISOR`', because the actual argument name is `divisor`.

  Also use all caps for meta-syntactic variables, such as when you show the decomposition of a list or vector into subunits, some of which may vary.

Octave also allows extensive formatting of the help string of functions using Texinfo. The effect on the online documentation is relatively small, but makes the help string of functions conform to the help of Octave's own functions. However, the effect on the appearance of printed or online documentation will be greatly improved.

The fundamental building block of Texinfo documentation strings is the Texinfo-macro `@deftypefn`, which takes three arguments: The class the function is in, its output arguments, and the function's signature. Typical classes for functions include `Function File` for standard Octave functions, and `Loadable Function` for dynamically linked functions. A skeletal Texinfo documentation string therefore looks like this

```
-*- texinfo -*-
@deftypefn {Function File} {@var{ret} =} fn (...)
@cindex index term
Help text in Texinfo format.  Code samples should be marked
like @code{sample of code} and variables should be marked
as @var{variable}.
@seealso{fn2, fn3}
@end deftypefn
```

This help string must be commented in user functions, or in the help string of the `DEFUN_DLD` macro for dynamically loadable functions. The important aspects of the documentation string are

-*- texinfo -*-
: This string signals Octave that the following text is in Texinfo format, and should be the first part of any help string in Texinfo format.

@deftypefn {class} ... @end deftypefn
: The entire help string should be enclosed within the block defined by deftypefn.

@cindex index term
: This generates an index entry, and can be useful when the function is included as part of a larger piece of documentation. It is ignored within Octave's help viewer. Only one index term may appear per line but multiple @cindex lines are valid if the function should be filed under different terms.

@var{variable}
: All variables should be marked with this macro. The markup of variables is then changed appropriately for display.

@code{sample of code}
: All samples of code should be marked with this macro for the same reasons as the @var macro.

@qcode{"sample_code"}
@qcode{'sample_code'}

All samples of code which are quoted should use this more specialized macro. This happens frequently when discussing graphics properties such as "position" or options such as "on"/"off".

@seealso{function2, function3}

This is a comma separated list of function names that allows cross referencing from one function documentation string to another.

Texinfo format has been designed to generate output for online viewing with text terminals as well as generating high-quality printed output. To these ends, Texinfo has commands which control the diversion of parts of the document into a particular output processor. Three formats are of importance: info, HTML, and TEX. These are selected with

```
@ifinfo
Text area for info only
@end ifinfo

@ifhtml
Text area for HTML only
@end ifhtml

@tex
Text area for TeX only
@end tex
```

Note that often TEX output can be used in HTML documents and so often the `@ifhtml` blocks are unnecessary. If no specific output processor is chosen, by default, the text goes into all output processors. It is usual to have the above blocks in pairs to allow the same information to be conveyed in all output formats, but with a different markup. Currently, most Octave documentation only makes a distinction between TEX and all other formats. Therefore, the following construct is seen repeatedly.

```
@tex
text for TeX only
@end tex
@ifnottex
text for info, HTML, plaintext
@end ifnottex
```

Another important feature of Texinfo that is often used in Octave help strings is the `@example` environment. An example of its use is

```
@example
@group
@code{2 * 2}
@result{} 4
@end group
@end example
```

which produces

```
2 * 2
⇒ 4
```

The `@group` block prevents the example from being split across a page boundary, while the `@result{}` macro produces a right arrow signifying the result of a command. If your example is larger than 20 lines it is better *NOT* to use grouping so that a reasonable page boundary can be calculated.

In many cases a function has multiple ways in which it can be called, and the `@deftypefnx` macro can be used to give alternatives. For example

```
-*- texinfo -*-
@deftypefn  {Function File} {@var{a} =} fn (@var{x}, ...)
@deftypefnx {Function File} {@var{a} =} fn (@var{y}, ...)
Help text in Texinfo format.
@end deftypefn
```

Many complete examples of Texinfo documentation can be taken from the help strings for the Octave functions themselves. A relatively complete example of which is the `nchoosek` function. The Texinfo documentation string for `nchoosek` is

```
-*- texinfo -*-
@deftypefn  {Function File} {@var{c} =} nchoosek (@var{n}, @var{k})
@deftypefnx {Function File} {@var{c} =} nchoosek (@var{set}, @var{k})

Compute the binomial coefficient of @var{n} or list all possible
combinations of a @var{set} of items.

If @var{n} is a scalar then calculate the binomial coefficient
of @var{n} and @var{k} which is defined as
@tex
$$
 {n \choose k} = {n (n-1) (n-2) \cdots (n-k+1) \over k!}
               = {n! \over k! (n-k)!}
$$
@end tex
@ifnottex

@example
@group
 /   \
 | n |    n (n-1) (n-2) @dots{} (n-k+1)       n!
 |   |  = ------------------------- =  ---------
 | k |               k!                k! (n-k)!
 \   /
@end group
@end example

@end ifnottex
@noindent
This is the number of combinations of @var{n} items taken in groups of
size @var{k}.
```

```
If the first argument is a vector, @var{set}, then generate all
combinations of the elements of @var{set}, taken @var{k} at a time, with
one row per combination.  The result @var{c} has @var{k} columns and
@w{@code{nchoosek (length (@var{set}), @var{k})}} rows.

For example:

How many ways can three items be grouped into pairs?

@example
@group
nchoosek (3, 2)
   @result{} 3
@end group
@end example

What are the possible pairs?

@example
@group
nchoosek (1:3, 2)
   @result{}  1   2
       1   3
       2   3
@end group
@end example

Programming Note: When calculating the binomial coefficient @code{nchoosek}
works only for non-negative, integer arguments.  Use @code{bincoeff} for
non-integer and negative scalar arguments, or for computing many binomial
coefficients at once with vector inputs for @var{n} or @var{k}.

@seealso{bincoeff, perms}
@end deftypefn
```

which demonstrates most of the concepts discussed above. This documentation string renders in Info format as

```
 -- Function File: C = nchoosek (N, K)
 -- Function File: C = nchoosek (SET, K)
     Compute the binomial coefficient of N or list all possible
     combinations of a SET of items.

     If N is a scalar then calculate the binomial coefficient of N and
     K which is defined as

           /   \
           | n |    n (n-1) (n-2) ... (n-k+1)       n!
```

```
| |  = ------------------------ =  ---------
| k |              k!               k! (n-k)!
\   /
```

```
This is the number of combinations of N items taken in groups of
size K.

If the first argument is a vector, SET, then generate all
combinations of the elements of SET, taken K at a time, with one
row per combination.  The result C has K columns and
‘nchoosek (length (SET), K)’ rows.

For example:

How many ways can three items be grouped into pairs?

     nchoosek (3, 2)
        => 3

What are the possible pairs?

     nchoosek (1:3, 2)
        =>  1   2
            1   3
            2   3

Programming Note: When calculating the binomial coefficient
‘nchoosek’ works only for non-negative, integer arguments.  Use
‘bincoeff’ for non-integer and negative scalar arguments, or for
computing many binomial coefficients at once with vector inputs
for N or K.

See also: bincoeff, perms.
```

whereas in printed documentation using TEX it will appear as

c = `nchoosek` (*n*, *k*) [Function File]
c = `nchoosek` (*set*, *k*) [Function File]

Compute the binomial coefficient of n or list all possible combinations of a *set* of items.

If n is a scalar then calculate the binomial coefficient of n and k which is defined as

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k!} = \frac{n!}{k!(n-k)!}$$

This is the number of combinations of n items taken in groups of size k.

If the first argument is a vector, *set*, then generate all combinations of the elements of *set*, taken k at a time, with one row per combination. The result c has k columns and `nchoosek (length (set), k)` rows.

For example:

How many ways can three items be grouped into pairs?

```
nchoosek (3, 2)
   ⇒ 3
```

What are the possible pairs?

```
nchoosek (1:3, 2)
   ⇒  1   2
       1   3
       2   3
```

Programming Note: When calculating the binomial coefficient `nchoosek` works only for non-negative, integer arguments. Use `bincoeff` for non-integer and negative scalar arguments, or for computing many binomial coefficients at once with vector inputs for *n* or *k*.

See also: bincoeff, perms.

Appendix D Contributing Guidelines

This chapter is dedicated to those who wish to contribute code to Octave.

D.1 How to Contribute

The mailing list for Octave development discussions is `maintainers@octave.org`. Patches should be submitted to Octave's patch tracker (`https://savannah.gnu.org/patch/?func=additem&group=octave`). This concerns the development of Octave core, i.e., code that goes in to Octave directly. You may consider developing and publishing a package instead; a great place for this is the allied Octave-Forge project (`http://octave.sourceforge.net`). Note that the Octave core project is inherently more conservative and follows narrower rules.

D.2 Building the Development Sources

The directions for building from the development sources change from time to time, so you should read the resources for developers on the web or in the development sources archive. Start here: `http://www.octave.org/get-involved.html`.

D.3 Basics of Generating a Changeset

The best way to contribute is to create a Mercurial changeset and submit it to the bug (`http://savannah.gnu.org/bugs/?group=octave`) or patch (`http://savannah.gnu.org/patch/?func=additem&group=octave`) trackers[1]. Mercurial is the source code management system currently used to develop Octave. Other forms of contributions (e.g., simple diff patches) are also acceptable, but they slow down the review process. If you want to make more contributions, you should really get familiar with Mercurial. A good place to start is `http://www.selenic.com/mercurial/wiki/index.cgi/Tutorial`. There you will also find help about how to install Mercurial.

A simple contribution sequence could look like this:

```
hg clone http://www.octave.org/hg/octave
                             # make a local copy of the octave
                             # source repository
cd octave
# change some sources...
hg commit -m "make Octave the coolest software ever"
                             # commit the changeset into your
                             # local repository
hg export -o ../cool.diff tip
                             # export the changeset to a diff
                             # file
# attach ../cool.diff to your bug report
```

[1] Please use the patch tracker only for patches which add new features. If you have a patch to submit that fixes a bug, you should use the bug tracker instead.

You may want to get familiar with Mercurial queues to manage your changesets. To work with queues you must activate the extension mq with the following entry in Mercurial's configuration file `.hgrc` (or `Mercurial.ini` on Windows):

```
[extensions]
mq=
```

Here is a slightly more complex example using Mercurial queues, where work on two unrelated changesets is done in parallel and one of the changesets is updated after discussion on the bug tracker:

```
hg qnew nasty_bug           # create a new patch
# change sources...
hg qref                     # save the changes into the patch
# change even more...
hg qref -m "solution to nasty bug!"
                            # save again with commit message
hg export -o ../nasty.diff tip
                            # export the patch
# attach ../nasty.diff to your bug report
hg qpop                     # undo the application of the patch
                            # and remove the changes from the
                            # source tree
hg qnew doc_improvements    # create an unrelated patch
# change doc sources...
hg qref -m "could not find myfav.m in the doc"
                            # save the changes into the patch
hg export -o ../doc.diff tip
                            # export the second patch
# attach ../doc.diff to your bug report
hg qpop
# discussion in the bug tracker ...
hg qpush nasty_bug          # apply the patch again
# change sources yet again ...
hg qref
hg export -o ../nasty2.diff tip
# attach ../nasty2.diff to your bug report
```

Mercurial has a few more useful extensions that really should be enabled. They are not enabled by default due to a number of factors (mostly because they don't work in all terminal types).

The following entries in the `.hgrc` are recommended

```
[extensions]
graphlog=
color=
progress=
pager=
```

For the color extension, default color and formatting of `hg status` can be modified by

```
[color]
status.modified = magenta bold
status.added = green bold
status.removed = red bold
status.deleted = cyan bold
status.unknown = black  bold
status.ignored = black bold
```

Sometimes a few further improvements for the pager extension are necessary. The following options should not be enabled unless paging is not working correctly.

```
[pager]
# Some options for the less pager, see less(1) for their meaning.
pager = LESS='FSRX' less

# Some commands that aren't paged by default; also enable paging
# for them
attend = tags, help, annotate, cat, diff, export, status, \
         outgoing, incoming
```

Enabling the described extensions should immediately lead to a difference when using the command line version of hg. Of these options, the only one that enables a new command is graphlog. It is recommanded that to use the command `hg glog`, instead of `hg log`, for a better feel about what commits are being based on.

D.4 General Guidelines

All Octave's sources are distributed under the GNU General Public License (GPL). Currently, Octave uses GPL version 3. For details about this license, see `http://www.gnu.org/licenses/gpl.html`. Therefore, whenever you create a new source file, it should have the following comment header (use appropriate year, name and comment marks):

```
## Copyright (C) 1996-2015 John W. Eaton <jwe@octave.org>
##
## This file is part of Octave.
##
## Octave is free software; you can redistribute it and/or modify it
## under the terms of the GNU General Public License as published by
## the Free Software Foundation; either version 3 of the License, or
## (at your option) any later version.
##
## Octave is distributed in the hope that it will be useful, but
## WITHOUT ANY WARRANTY; without even the implied warranty of
## MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
## GNU General Public License for more details.
##
## You should have received a copy of the GNU General Public License
## along with Octave; see the file COPYING.  If not,
## see <http://www.gnu.org/licenses/>.
```

Always include commit messages in changesets. After making your source changes, record and briefly describe the changes in your commit message. You should have previously configured your `.hgrc` (or `Mercurial.ini` on Windows) with your name and email, which will be automatically added to your commit message. Your commit message should have a brief one-line explanation of what the commit does. If you are patching a bug, this one-line explanation should mention the bug number at the end. If your change is small and only touches one file then this is typically sufficient. If you are modifying several files, or several parts of one file, you should enumerate your changes roughly following the GNU coding standards for changelogs, as in the following example:

```
look for methods before constructors

* symtab.cc (symbol_table::fcn_info::fcn_info_rep::find):
Look for class methods before constructors, contrary to MATLAB
documentation.

* test/ctor-vs-method: New directory of test classes.
* test/test_ctor_vs_method.m: New file.
* test/Makefile.am: Include ctor-vs-method/module.mk.
(FCN_FILES): Include test_ctor_vs_method.m in the list.
```

In this example, the names of the file changed is listed first, and in parentheses the name of the function in that file that was modified. There is no need to mention the function for m-files that only contain one function. The commit message should describe what was changed, not why it was changed. Any explanation for why a change is needed should appear as comments in the code, particularly if there is something that might not be obvious to someone reading it later.

When submitting code which addresses a known bug on the Octave bug tracker (`http://bugs.octave.org`), please add '(bug #XXXXX)' to the first line of the commit messages. For example:

```
Fix bug for complex input for gradient (bug #34292).
```

The preferred comment mark for places that may need further attention is `FIXME:`.

D.5 Octave Sources (m-files)

Don't use tabs. Tabs cause trouble. If you are used to them, set up your editor so that it converts tabs to spaces. Indent the bodies of statement blocks. The recommended indent is 2 spaces. When calling functions, put spaces after commas and before the calling parentheses, like this:

```
x = max (sin (y+3), 2);
```

An exception are matrix or cell constructors:

```
[sin(x), cos(x)]
{sin(x), cos(x)}
```

Here, putting spaces after `sin`, `cos` would result in a parse error. For an indexing expression, do not put a space after the identifier (this differentiates indexing and function calls nicely). The space after a comma is not necessary if index expressions are simple, i.e., you may write

```
A(:,i,j)
```

but

```
A([1:i-1;i+1:n], XI(:,2:n-1))
```

Use lowercase names if possible. Uppercase is acceptable for variable names consisting of 1-2 letters. Do not use mixed case names. Function names must be lowercase. Function names are global, so choose them wisely.

Always use a specific end-of-block statement (like `endif`, `endswitch`) rather than the generic `end`.

Enclose the `if`, `while`, `until`, and `switch` conditions in parentheses, as in C:

```
if (isvector (a))
  s = sum (a);
endif
```

Do not do this, however, with the iteration counter portion of a `for` statement. Write:

```
for i = 1:n
  b(i) = sum (a(:,i));
endfor
```

D.6 C++ Sources

Don't use tabs. Tabs cause trouble. If you are used to them, set up your editor so that it converts tabs to spaces. Format function headers like this:

```
static bool
matches_patterns (const string_vector& patterns, int pat_idx,
                  int num_pat, const std::string& name)
```

The function name should start in column 1, and multi-line argument lists should be aligned on the first char after the open parenthesis. You should put a space before the left open parenthesis and after commas, for both function definitions and function calls.

The recommended indent is 2 spaces. When indenting, indent the statement after control structures (like `if`, `while`, etc.). If there is a compound statement, indent *both* the curly braces and the body of the statement (so that the body gets indented by *two* indents). Example:

```
if (have_args)
  {
    idx.push_back (first_args);
    have_args = false;
  }
else
  idx.push_back (make_value_list (*p_args, *p_arg_nm, &tmp));
```

If you have nested `if` statements, use extra braces for extra clarification.

Split long expressions in such a way that a continuation line starts with an operator rather than identifier. If the split occurs inside braces, continuation should be aligned with the first char after the innermost braces enclosing the split. Example:

```
SVD::type type = ((nargout == 0 || nargout == 1)
                  ? SVD::sigma_only
                  : (nargin == 2) ? SVD::economy : SVD::std);
```

Consider putting extra braces around a multi-line expression to make it more readable, even if they are not necessary. Also, do not hesitate to put extra braces anywhere if it improves clarity.

Declare variables just before they are needed. Use local variables of blocks—it helps optimization. Don't write a multi-line variable declaration with a single type specification and multiple variables. If the variables don't fit on single line, repeat the type specification. Example:

```
octave_value retval;

octave_idx_type nr = b.rows ();
octave_idx_type nc = b.cols ();

double d1, d2;
```

Use lowercase names if possible. Uppercase is acceptable for variable names consisting of 1-2 letters. Do not use mixed case names.

Use Octave's types and classes if possible. Otherwise, use the C++ standard library. Use of STL containers and algorithms is encouraged. Use templates wisely to reduce code duplication. Avoid comma expressions, labels and gotos, and explicit typecasts. If you need to typecast, use the modern C++ casting operators. In functions, minimize the number of `return` statements—use nested `if` statements if possible.

D.7 Other Sources

Apart from C++ and Octave language (m-files), Octave's sources include files written in C, Fortran, M4, Perl, Unix shell, AWK, Texinfo, and TeX. There are not many rules to follow when using these other languages; some of them are summarized below. In any case, the golden rule is: if you modify a source file, try to follow any conventions you can detect in the file or other similar files.

For C you should obviously follow all C++ rules that can apply.

If you modify a Fortran file, you should stay within Fortran 77 with common extensions like `END DO`. Currently, we want all sources to be compilable with the f2c and g77 compilers, without special flags if possible. This usually means that non-legacy compilers also accept the sources.

The M4 macro language is mainly used for Autoconf configuration files. You should follow normal M4 rules when contributing to these files. Some M4 files come from external source, namely the Autoconf archive `http://autoconf-archive.cryp.to`.

If you give a code example in the documentation written in Texinfo with the `@example` environment, you should be aware that the text within such an environment will not be wrapped. It is recommended that you keep the lines short enough to fit on pages in the generated pdf or ps documents. Here is a ruler (in an `@example` environment) for finding the appropriate line width:

```
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
```

Appendix E Obsolete Functions

After being marked as deprecated for two major releases, the following functions have been removed from Octave. The third column of the table shows the version of Octave in which the function was removed. Prior to removal, each function in the list was marked as deprecated for at least two major releases. All deprecated functions issue warnings explaining that they will be removed in a future version of Octave, and which function should be used instead.

Replacement functions do not always accept precisely the same arguments as the obsolete function, but should provide equivalent functionality.

Obsolete Function	**Replacement**	**Version**
beta_cdf	betacdf	3.4.0
beta_inv	betainv	3.4.0
beta_pdf	betapdf	3.4.0
beta_rnd	betarnd	3.4.0
binomial_cdf	binocdf	3.4.0
binomial_inv	binoinv	3.4.0
binomial_pdf	binopdf	3.4.0
binomial_rnd	binornd	3.4.0
chisquare_cdf	chi2cdf	3.4.0
chisquare_inv	chi2inv	3.4.0
chisquare_pdf	chi2pdf	3.4.0
chisquare_rnd	chi2rnd	3.4.0
clearplot	clf	3.4.0
com2str	num2str	3.4.0
exponential_cdf	expcdf	3.4.0
exponential_inv	expinv	3.4.0
exponential_pdf	exppdf	3.4.0
exponential_rnd	exprnd	3.4.0
f_cdf	fcdf	3.4.0
f_inv	finv	3.4.0
f_pdf	fpdf	3.4.0
f_rnd	frnd	3.4.0
gamma_cdf	gamcdf	3.4.0
gamma_inv	gaminv	3.4.0
gamma_pdf	gampdf	3.4.0
gamma_rnd	gamrnd	3.4.0
geometric_cdf	geocdf	3.4.0
geometric_inv	geoinv	3.4.0
geometric_pdf	geopdf	3.4.0
geometric_rnd	geornd	3.4.0
hypergeometric_cdf	hygecdf	3.4.0
hypergeometric_inv	hygeinv	3.4.0
hypergeometric_pdf	hygepdf	3.4.0
hypergeometric_rnd	hygernd	3.4.0

intersection	intersect	3.4.0
is_bool	isbool	3.4.0
is_complex	iscomplex	3.4.0
is_list	islist	3.4.0
is_matrix	ismatrix	3.4.0
is_scalar	isscalar	3.4.0
is_square	issquare	3.4.0
is_stream	isstream	3.4.0
is_struct	isstruct	3.4.0
is_symmetric	issymmetric	3.4.0
is_vector	isvector	3.4.0
lognormal_cdf	logncdf	3.4.0
lognormal_inv	logninv	3.4.0
lognormal_pdf	lognpdf	3.4.0
lognormal_rnd	lognrnd	3.4.0
meshdom	meshgrid	3.4.0
normal_cdf	normcdf	3.4.0
normal_inv	norminv	3.4.0
normal_pdf	normpdf	3.4.0
normal_rnd	normrnd	3.4.0
pascal_cdf	nbincdf	3.4.0
pascal_inv	nbininv	3.4.0
pascal_pdf	nbinpdf	3.4.0
pascal_rnd	nbinrnd	3.4.0
poisson_cdf	poisscdf	3.4.0
poisson_inv	poissinv	3.4.0
poisson_pdf	poisspdf	3.4.0
poisson_rnd	poissrnd	3.4.0
polyinteg	polyint	3.4.0
struct_contains	isfield	3.4.0
struct_elements	fieldnames	3.4.0
t_cdf	tcdf	3.4.0
t_inv	tinv	3.4.0
t_pdf	tpdf	3.4.0
t_rnd	trnd	3.4.0
uniform_cdf	unifcdf	3.4.0
uniform_inv	unifinv	3.4.0
uniform_pdf	unifpdf	3.4.0
uniform_rnd	unifrnd	3.4.0
weibull_cdf	wblcdf	3.4.0
weibull_inv	wblinv	3.4.0
weibull_pdf	wblpdf	3.4.0
weibull_rnd	wblrnd	3.4.0
wiener_rnd	wienrnd	3.4.0
create_set	unique	3.6.0
dmult	diag (A) * B	3.6.0
iscommand	None	3.6.0

`israwcommand`	None	3.6.0
`lchol`	`chol (..., "lower")`	3.6.0
`loadimage`	`load` or `imread`	3.6.0
`mark_as_command`	None	3.6.0
`mark_as_rawcommand`	None	3.6.0
`spatan2`	`atan2`	3.6.0
`spchol`	`chol`	3.6.0
`spchol2inv`	`chol2inv`	3.6.0
`spcholinv`	`cholinv`	3.6.0
`spcumprod`	`cumprod`	3.6.0
`spcumsum`	`cumsum`	3.6.0
`spdet`	`det`	3.6.0
`spdiag`	`sparse (diag (...))`	3.6.0
`spfind`	`find`	3.6.0
`sphcat`	`horzcat`	3.6.0
`spinv`	`inv`	3.6.0
`spkron`	`kron`	3.6.0
`splchol`	`chol (..., "lower")`	3.6.0
`split`	`char (strsplit (s, t))`	3.6.0
`splu`	`lu`	3.6.0
`spmax`	`max`	3.6.0
`spmin`	`min`	3.6.0
`spprod`	`prod`	3.6.0
`spqr`	`qr`	3.6.0
`spsum`	`sum`	3.6.0
`spsumsq`	`sumsq`	3.6.0
`spvcat`	`vertcat`	3.6.0
`str2mat`	`char`	3.6.0
`unmark_command`	None	3.6.0
`unmark_rawcommand`	None	3.6.0
`autocor`	Octave-Forge signal pkg, `xcor`	3.8.0
`autocov`	Octave-Forge signal pkg, `xcov`	3.8.0
`betai`	`betainc`	3.8.0
`cellidx`	`ismember`	3.8.0
`cquad`	`quadcc`	3.8.0
`dispatch`	None	3.8.0
`fstat`	`stat`	3.8.0
`gammai`	`gammainc`	3.8.0
`glpkmex`	`glpk`	3.8.0
`is_duplicate_entry`	`unique`	3.8.0
`is_global`	`isglobal`	3.8.0
`krylovb`	`[Uret, ~, Ucols] = krylov (...)`	3.8.0

`perror`	None	3.8.0
`replot`	`refresh`	3.8.0
`saveimage`	`imwrite`	3.8.0
`setstr`	`char`	3.8.0
`strerror`	None	3.8.0
`values`	`unique`	3.8.0
`cut`	`histc`	4.0.0
`cor`	`corr`	4.0.0
`corrcoef`	`corr`	4.0.0
`__error_text__`	`lasterr`	4.0.0
`error_text`	`lasterr`	4.0.0
`polyderiv`	`polyder`	4.0.0
`shell_cmd`	`system`	4.0.0
`studentize`	`zscore`	4.0.0
`sylvester_matrix`	`hadamard (2^k)`	4.0.0

Appendix F Known Causes of Trouble

This section describes known problems that affect users of Octave. Most of these are not Octave bugs per se—if they were, we would fix them. But the result for a user may be like the result of a bug.

Some of these problems are due to bugs in other software, some are missing features that are too much work to add, and some are places where people's opinions differ as to what is best.

F.1 Actual Bugs We Haven't Fixed Yet

- Output that comes directly from Fortran functions is not sent through the pager and may appear out of sequence with other output that is sent through the pager. One way to avoid this is to force pending output to be flushed before calling a function that will produce output from within Fortran functions. To do this, use the command

  ```
  fflush (stdout)
  ```

  Another possible workaround is to use the command

  ```
  page_screen_output (false);
  ```

  to turn the pager off.

A list of ideas for future enhancements is distributed with Octave. See the file `PROJECTS` in the top level directory in the source distribution.

F.2 Reporting Bugs

Your bug reports play an essential role in making Octave reliable.

When you encounter a problem, the first thing to do is to see if it is already known. See Appendix F [Trouble], page 889. If it isn't known, then you should report the problem.

Reporting a bug may help you by bringing a solution to your problem, or it may not. In any case, the principal function of a bug report is to help the entire community by making the next version of Octave work better. Bug reports are your contribution to the maintenance of Octave.

In order for a bug report to serve its purpose, you must include the information that makes it possible to fix the bug.

F.2.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If Octave gets a fatal signal, for any input whatever, that is a bug. Reliable interpreters never crash.
- If Octave produces incorrect results, for any input whatever, that is a bug.
- Some output may appear to be incorrect when it is in fact due to a program whose behavior is undefined, which happened by chance to give the desired results on another system. For example, the range operator may produce different results because of differences in the way floating point arithmetic is handled on various systems.
- If Octave produces an error message for valid input, that is a bug.

- If Octave does not produce an error message for invalid input, that is a bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".
- If you are an experienced user of programs like Octave, your suggestions for improvement are welcome in any case.

F.2.2 Where to Report Bugs

To report a bug in Octave, submit a bug report to the Octave bug tracker `http://bugs.octave.org`.

Do not send bug reports to '`help-octave`'. Most users of Octave do not want to receive bug reports.

F.2.3 How to Report Bugs

Submit bug reports for Octave to the Octave bug tracker `http://bugs.octave.org`.

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to state a fact or leave it out, state it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the interpreter into doing the right thing despite the bug. Play it safe and give a specific, complete example.

Keep in mind that the purpose of a bug report is to enable someone to fix the bug if it is not known. Always write your bug reports on the assumption that the bug is not known.

Sometimes people give a few sketchy facts and ask, "Does this ring a bell?" This cannot help us fix a bug. It is better to send a complete bug report to begin with.

Try to make your bug report self-contained. If we have to ask you for more information, it is best if you include all the previous information in your response, as well as the information that was missing.

To enable someone to investigate the bug, you should include all these things:

- The version of Octave. You can get this by noting the version number that is printed when Octave starts, or running it with the '`-v`' option.
- A complete input file that will reproduce the bug.

  A single statement may not be enough of an example—the bug might depend on other details that are missing from the single statement where the error finally occurs.
- The command arguments you gave Octave to execute that example and observe the bug. To guarantee you won't omit something important, list all the options.

  If we were to try to guess the arguments, we would probably guess wrong and then we would not encounter the bug.
- The type of machine you are using, and the operating system name and version number.
- The command-line arguments you gave to the `configure` command when you installed the interpreter.

- A complete list of any modifications you have made to the interpreter source.

  Be precise about these changes—show a context diff for them.
- Details of any other deviations from the standard procedure for installing Octave.
- A description of what behavior you observe that you believe is incorrect. For example, "The interpreter gets a fatal signal," or, "The output produced at line 208 is incorrect."

  Of course, if the bug is that the interpreter gets a fatal signal, then one can't miss it. But if the bug is incorrect output, we might not notice unless it is glaringly wrong.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the interpreter is out of sync, or you have encountered a bug in the C library on your system. Your copy might crash and the copy here would not. If you said to expect a crash, then when the interpreter here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening. We would not be able to draw any conclusion from our observations.

  Often the observed symptom is incorrect output when your program is run. Unfortunately, this is not enough information unless the program is short and simple. It is very helpful if you can include an explanation of the expected output, and why the actual output is incorrect.
- If you wish to suggest changes to the Octave source, send them as context diffs. If you even discuss something in the Octave source, refer to it by context, not by line number, because the line numbers in the development sources probably won't match those in your sources.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it. Such information is usually not necessary to enable us to fix bugs in Octave, but if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc. Most Octave bugs involve just one function, so the most straightforward way to simplify an example is to delete all the function definitions except the one in which the bug occurs.

  However, simplification is not vital; if you don't want to do this, report the bug anyway and send the entire test case you used.
- A patch for the bug. Patches can be helpful, but if you find a bug, you should report it, even if you cannot send a fix for the problem.

F.2.4 Sending Patches for Octave

If you would like to write bug fixes or improvements for Octave, that is very helpful. When you send your changes, please follow these guidelines to avoid causing extra work for us in studying the patches.

If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining Octave is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a bug fix, just include a copy of the bug report, and explain why the change fixes the bug.
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is right, we might have trouble judging it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one.
- Use '`diff -c`' to make your diffs. Diffs without context are hard for us to install reliably. More than that, they make it hard for us to study the diffs to decide whether we want to install them. Unified diff format is better than contextless diffs, but not as easy to read as '`-c`' format.

  If you have GNU diff, use '`diff -cp`', which shows the name of the function that each change occurs in.
- Write the change log entries for your changes.

  Read the `ChangeLog` file to see what sorts of information to put in, and to learn the style that we use. The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was made.

  On the other hand, once you have shown people where to find the change, you need not explain its purpose. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but the explanation will be much more useful if you put it in comments in the code.

  If you would like your name to appear in the header line for who made the change, send us the header line.

F.3 How To Get Help with Octave

The mailing list `help@octave.org` exists for the discussion of matters related to using and installing Octave. If would like to join the discussion, please send a short note to `help-request@octave.org`.

Please do not send requests to be added or removed from the mailing list, or other administrative trivia to the list itself.

If you think you have found a bug in Octave or in the installation procedure, however, you should submit a complete bug report to the Octave bug tracker at `http://bugs.octave.org`. But before you submit a bug report, please read `http://www.octave.org/bugs.html` to learn how to submit a useful bug report.

Appendix G Installing Octave

The procedure for installing Octave from source on a Unix-like system is described next. Building on other platforms will follow similar steps. Note that this description applies to Octave releases. Building the development sources from the Mercurial archive requires additional steps as described in Section D.2 [Building the Development Sources], page 879.

G.1 Build Dependencies

Octave is a fairly large program with many build dependencies. You may be able to find pre-packaged versions of the dependencies distributed as part of your system, or you may have to build some or all of them yourself.

G.1.1 Obtaining the Dependencies Automatically

On some systems you can obtain many of Octave's build dependencies automatically. The commands for doing this vary by system. Similarly, the names of pre-compiled packages vary by system and do not always match exactly the names listed in Section G.1.2 [Build Tools], page 893, and Section G.1.3 [External Packages], page 894.

You will usually need the development version of an external dependency so that you get the libraries and header files for building software, not just for running already compiled programs. These packages typically have names that end with the suffix `-dev` or `-devel`.

On systems with `apt-get` (Debian, Ubuntu, etc.), you may be able to install most of the tools and external packages using a command similar to

```
apt-get build-dep octave
```

The specific package name may be `octave3.2` or `octave3.4`. The set of required tools and external dependencies does not change frequently, so it is not important that the version match exactly, but you should use the most recent one available.

On systems with `yum` (Fedora, Red Hat, etc.), you may be able to install most of the tools and external packages using a command similar to

```
yum-builddep octave
```

The `yum-builddep` utility is part of the `yum-utils` package.

For either type of system, the package name may include a version number. The set of required tools and external dependencies does not change frequently, so it is not important that the version exactly match the version you are installing, but you should use the most recent one available.

G.1.2 Build Tools

The following tools are required:

C++, C, and Fortran compilers

The Octave sources are primarily written in C++, but some portions are also written in C and Fortran. The Octave sources are intended to be portable. Recent versions of the GNU compiler collection (GCC) should work (`http://gcc.gnu.org`). If you use GCC, you should avoid mixing versions. For example, be sure that you are not using the obsolete `g77` Fortran compiler with modern versions of `gcc` and `g++`.

GNU Make
: Tool for building software (`http://www.gnu.org/software/make`). Octave's build system requires GNU Make. Other versions of Make will not work. Fortunately, GNU Make is highly portable and easy to install.

AWK, sed, and other Unix utilities
: Basic Unix system utilities are required for building Octave. All will be available with any modern Unix system and also on Windows with either Cygwin or MinGW and MSYS.

Additionally, the following tools may be needed:

Bison
: Parser generator (`http://www.gnu.org/software/bison`). You will need Bison if you modify the `oct-parse.yy` source file or if you delete the files that are generated from it.

Flex
: Lexer analyzer (`http://www.gnu.org/software/flex`). You will need Flex if you modify the `lex.ll` source file or if you delete the files that are generated from it.

Autoconf
: Package for software configuration (`http://www.gnu.org/software/autoconf`). Autoconf is required if you modify Octave's `configure.ac` file or other files that it requires.

Automake
: Package for Makefile generation (`http://www.gnu.org/software/automake`). Automake is required if you modify Octave's `Makefile.am` files or other files that they depend on.

Libtool
: Package for building software libraries (`http://www.gnu.org/software/libtool`). Libtool is required by Automake.

gperf
: Perfect hash function generator (`http://www.gnu.org/software/gperf`). You will need gperf if you modify the `octave.gperf` file or if you delete the file that is generated from it.

Texinfo
: Package for generating online and print documentation (`http://www.gnu.org/software/texinfo`). You will need Texinfo to build Octave's documentation or if you modify the documentation source files or the docstring of any Octave function.

G.1.3 External Packages

The following external packages are required:

BLAS
: Basic Linear Algebra Subroutine library. Accelerated BLAS libraries such as OpenBLAS (`http://www.openblas.net/`) or ATLAS (`http://math-atlas.sourceforge.net`) are recommended for best performance. The reference implementation (`http://www.netlib.org/blas`) is slow, unmaintained, and suffers from certain bugs in corner case inputs.

LAPACK
: Linear Algebra Package (`http://www.netlib.org/lapack`).

PCRE
: The Perl Compatible Regular Expression library (`http://www.pcre.org`).

The following external package is optional but strongly recommended:

GNU Readline
: Command-line editing library (`www.gnu.org/s/readline`).

If you wish to build Octave without GNU readline installed, you must use the `--disable-readline` option when running the configure script.

The following external software packages are optional but recommended:

ARPACK
: Library for the solution of large-scale eigenvalue problems (`http://forge.scilab.org/index.php/p/arpack-ng`). ARPACK is required to provide the functions `eigs` and `svds`.

cURL
: Library for transferring data with URL syntax (`http://curl.haxx.se`). cURL is required to provide the `urlread` and `urlwrite` functions and the `ftp` class.

FFTW3
: Library for computing discrete Fourier transforms (`http://www.fftw.org`). FFTW3 is used to provide better performance for functions that compute discrete Fourier transforms (`fft`, `ifft`, `fft2`, etc.)

FLTK
: Portable GUI toolkit (`http://www.fltk.org`). FLTK is currently used to provide windows for Octave's OpenGL-based graphics functions.

fontconfig
: Library for configuring and customizing font access (`http://www.freedesktop.org/wiki/Software/fontconfig`). Fontconfig is used to manage fonts for Octave's OpenGL-based graphics functions.

FreeType
: Portable font engine (`http://www.freetype.org`). FreeType is used to perform font rendering for Octave's OpenGL-based graphics functions.

GLPK
: GNU Linear Programming Kit (`http://www.gnu.org/software/glpk`). GPLK is required for the function `glpk`.

gl2ps
: OpenGL to PostScript printing library (`http://www.geuz.org/gl2ps/`). gl2ps is required for printing when using the FLTK toolkit.

gnuplot
: Interactive graphics program (`http://www.gnuplot.info`). gnuplot is currently the default graphics renderer for Octave.

GraphicsMagick++
: Image processing library (`http://www.graphicsmagick.org`). GraphicsMagick++ is used to provide the `imread` and `imwrite` functions.

HDF5
: Library for manipulating portable data files (`http://www.hdfgroup.org/HDF5`). HDF5 is required for Octave's `load` and `save` commands to read and write HDF data files.

Java Development Kit
: Java programming language compiler and libraries. The OpenJDK free software implementation is recommended (`http://openjdk.java.net/`), although other JDK implementations may work. Java is required to be able to call Java functions from within Octave.

LLVM
: Compiler framework, (`http://www.llvm.org`). LLVM is required for Octave's experimental just-in-time (JIT) compilation for speeding up the interpreter.

OpenGL
: API for portable 2-D and 3-D graphics (`http://www.opengl.org`). An OpenGL implementation is required to provide Octave's OpenGL-based graphics functions. Octave's OpenGL-based graphics functions usually outperform the gnuplot-based graphics functions because plot data can be rendered directly instead of sending data and commands to gnuplot for interpretation and rendering.

Qhull
: Computational geometry library (`http://www.qhull.org`). Qhull is required to provide the functions `convhull`, `convhulln`, `delaunay`, `delaunayn`, `voronoi`, and `voronoin`.

QRUPDATE
: QR factorization updating library (`http://sourceforge.net/projects/qrupdate`). QRUPDATE is used to provide improved performance for the functions `qrdelete`, `qrinsert`, `qrshift`, and `qrupdate`.

QScintilla
: Source code highlighter and manipulator; a Qt port of Scintilla (`http://www.riverbankcomputing.co.uk/software/qscintilla`). QScintilla is used for syntax highlighting and code completion in the GUI.

Qt
: GUI and utility libraries (). Qt is required for building the GUI. It is a large framework, but the only components required are the GUI, core, and network modules.

SuiteSparse
: Sparse matrix factorization library (`http://www.cise.ufl.edu/research/sparse/SuiteSparse`). SuiteSparse is required to provide sparse matrix factorizations and solution of linear equations for sparse systems.

zlib
: Data compression library (`http://zlib.net`). The zlib library is required for Octave's `load` and `save` commands to handle compressed data, including MATLAB v5 MAT files.

G.2 Running Configure and Make

- Run the shell script `configure`. This will determine the features your system has (or doesn't have) and create a file named `Makefile` from each of the files named `Makefile.in`.

  Here is a summary of the configure options that are most frequently used when building Octave:

  `--help`
  : Print a summary of the options recognized by the configure script.

  `--prefix=`*`prefix`*
  : Install Octave in subdirectories below *prefix*. The default value of *prefix* is `/usr/local`.

  `--srcdir=`*`dir`*
  : Look for Octave sources in the directory *dir*.

  `--enable-64`
  : This is an **experimental** option to enable Octave to use 64-bit integers for array dimensions and indexing on 64-bit platforms. You probably don't

want to use this option unless you know what you are doing. See Section G.3 [Compiling Octave with 64-bit Indexing], page 900, for more details about building Octave with this option.

`--enable-bounds-check`
: Enable bounds checking for indexing operators in the internal array classes. This option is primarily used for debugging Octave. Building Octave with this option has a negative impact on performance and is not recommended for general use.

`--disable-docs`
: Disable building all forms of the documentation (Info, PDF, HTML). The default is to build documentation, but your system will need functioning Texinfo and TEX installs for this to succeed.

`--enable-float-truncate`
: This option allows for truncation of intermediate floating point results in calculations. It is only necessary for certain platforms.

`--enable-readline`
: Use the readline library to provide for editing of the command line in terminal environments. This option is on by default.

`--enable-shared`
: Create shared libraries (this is the default). If you are planning to use the dynamic loading features, you will probably want to use this option. It will make your `.oct` files much smaller and on some systems it may be necessary to build shared libraries in order to use dynamically linked functions.

  You may also want to build a shared version of `libstdc++`, if your system doesn't already have one.

`--enable-dl`
: Use `dlopen` and friends to make Octave capable of dynamically linking externally compiled functions (this is the default if `--enable-shared` is specified). This option only works on systems that actually have these functions. If you plan on using this feature, you should probably also use `--enable-shared` to reduce the size of your `.oct` files.

`--with-blas=<lib>`
: By default, configure looks for the best BLAS matrix libraries on your system, including optimized implementations such as the free ATLAS 3.0, as well as vendor-tuned libraries. (The use of an optimized BLAS will generally result in several-times faster matrix operations.) Use this option to specify a particular BLAS library that Octave should use.

`--with-lapack=<lib>`
: By default, configure looks for the best LAPACK matrix libraries on your system, including optimized implementations such as the free ATLAS 3.0, as well as vendor-tuned libraries. (The use of an optimized LAPACK will generally result in several-times faster matrix operations.) Use this option to specify a particular LAPACK library that Octave should use.

--with-magick=<lib>
: Select the library to use for image I/O. The two possible values are "GraphicsMagick" (default) or "ImageMagick".

--with-sepchar=<char>
: Use <char> as the path separation character. This option can help when running Octave on non-Unix systems.

--without-amd
: Don't use AMD, disable some sparse matrix functionality.

--without-camd
: Don't use CAMD, disable some sparse matrix functionality.

--without-colamd
: Don't use COLAMD, disable some sparse matrix functionality.

--without-ccolamd
: Don't use CCOLAMD, disable some sparse matrix functionality.

--without-cholmod
: Don't use CHOLMOD, disable some sparse matrix functionality.

--without-curl
: Don't use the cURL library, disable the ftp objects, urlread and urlwrite functions.

--without-cxsparse
: Don't use CXSPARSE, disable some sparse matrix functionality.

--without-fftw3
: Use the included FFTPACK library for computing Fast Fourier Transforms instead of the FFTW3 library.

--without-fftw3f
: Use the included FFTPACK library for computing Fast Fourier Transforms instead of the FFTW3 library when operating on single precision (float) values.

--without-glpk
: Don't use the GLPK library for linear programming.

--without-hdf5
: Don't use the HDF5 library, disable reading and writing of HDF5 files.

--without-opengl
: Don't use OpenGL, disable native graphics toolkit for plotting. You will need gnuplot installed in order to make plots.

--without-qhull
: Don't use Qhull, disable delaunay, convhull, and related functions.

--without-qrupdate
: Don't use QRUPDATE, disable QR and Cholesky update functions.

--without-umfpack
: Don't use UMFPACK, disable some sparse matrix functionality.

`--without-zlib`
Don't use the zlib library, disable data file compression and support for recent MAT file formats.

`--without-framework-carbon`
Don't use framework Carbon headers, libraries, or specific source code even if the configure test succeeds (the default is to use Carbon framework if available). This is a platform specific configure option for Mac systems.

`--without-framework-opengl`
Don't use framework OpenGL headers, libraries, or specific source code even if the configure test succeeds. If this option is given then OpenGL headers and libraries in standard system locations are tested (the default value is `--with-framework-opengl`). This is a platform specific configure option for Mac systems.

See the file `INSTALL` for more general information about the command line options used by configure. That file also contains instructions for compiling in a directory other than the one where the source is located.

- Run make.

  You will need a recent version of GNU Make as Octave relies on certain features not generally available in all versions of make. Modifying Octave's makefiles to work with other make programs is probably not worth your time; instead, we simply recommend installing GNU Make.

  There are currently two options for plotting in Octave: (1) the external program gnuplot, or (2) the internal graphics engine using OpenGL and FLTK. Gnuplot is a command-driven interactive function plotting program. Gnuplot is copyrighted, but freely distributable. As of Octave release 3.4, gnuplot is the default option for plotting. But, the internal graphics engine is nearly 100% compatible, certainly for most ordinary plots, and users are encouraged to test it. It is anticipated that the internal engine will become the default option at the next major release of Octave.

  To compile Octave, you will need a recent version of `g++` or other ANSI C++ compiler. In addition, you will need a Fortran 77 compiler or `f2c`. If you use `f2c`, you will need a script like `fort77` that works like a normal Fortran compiler by combining `f2c` with your C compiler in a single script.

  If you plan to modify the parser you will also need GNU `bison` and `flex`. If you modify the documentation, you will need GNU Texinfo.

  GNU Make, `gcc` (and `libstdc++`), `gnuplot`, `bison`, `flex`, and Texinfo are all available from many anonymous ftp archives. The primary site is `ftp.gnu.org`, but it is often very busy. A list of sites that mirror the software on `ftp.gnu.org` is available by anonymous ftp from `ftp://ftp.gnu.org/pub/gnu/GNUinfo/FTP`.

  Octave requires approximately 1.4 GB of disk storage to unpack and compile from source (significantly less, 400 MB, if you don't compile with debugging symbols). To compile without debugging symbols try the command

  ```
  make CFLAGS=-O CXXFLAGS=-O LDFLAGS=
  ```

  instead of just `make`.

- If you encounter errors while compiling Octave, first see Section G.4 [Installation Problems], page 903, for a list of known problems and if there is a workaround or solution for your problem. If not, see Appendix F [Trouble], page 889, for information about how to report bugs.
- Once you have successfully compiled Octave, run `make install`.

  This will install a copy of Octave, its libraries, and its documentation in the destination directory. As distributed, Octave is installed in the following directories. In the table below, *prefix* defaults to `/usr/local`, *version* stands for the current version number of the interpreter, and *arch* is the type of computer on which Octave is installed (for example, '`i586-unknown-gnu`').

  `prefix/bin`
  : Octave and other binaries that people will want to run directly.

  `prefix/lib/octave-version`
  : Libraries like liboctave.a and liboctinterp.a.

  `prefix/octave-version/include/octave`
  : Include files distributed with Octave.

  `prefix/share`
  : Architecture-independent data files.

  `prefix/share/man/man1`
  : Unix-style man pages describing Octave.

  `prefix/share/info`
  : Info files describing Octave.

  `prefix/share/octave/version/m`
  : Function files distributed with Octave. This includes the Octave version, so that multiple versions of Octave may be installed at the same time.

  `prefix/libexec/octave/version/exec/arch`
  : Executables to be run by Octave rather than the user.

  `prefix/lib/octave/version/oct/arch`
  : Object files that will be dynamically loaded.

  `prefix/share/octave/version/imagelib`
  : Image files that are distributed with Octave.

G.3 Compiling Octave with 64-bit Indexing

Note: the following only applies to systems that have 64-bit pointers. Configuring Octave with `--enable-64` cannot magically make a 32-bit system have a 64-bit address space.

On 64-bit systems, Octave is limited to (approximately) the following array sizes when using the default 32-bit indexing mode:

```
double:         16 GB
single:          8 GB
uint64, int64:  16 GB
uint32, int32:   8 GB
uint16, int16:   4 GB
uint8, int8:     2 GB
```

In each case, the limit is really (approximately) 2^{31} elements because of the default type of the value used for indexing arrays (signed 32-bit integer, corresponding to the size of a Fortran INTEGER value).

Trying to create larger arrays will produce the following error:

```
octave:1> a = zeros (1024*1024*1024*3, 1, 'int8');
error: memory exhausted or requested size too large
       for range of Octave's index type --
       trying to return to prompt
```

You will obtain this error even if your system has enough memory to create this array (4 GB in the above case).

To use arrays larger than 2 GB, Octave has to be configured with the option `--enable-64`. This option is experimental and you are encouraged to submit bug reports if you find a problem. With this option, Octave will use 64-bit integers internally for array dimensions and indexing. However, all numerical libraries used by Octave will **also** need to use 64-bit integers for array dimensions and indexing. In most cases, this means they will need to be compiled from source since most (all?) distributions which package these libraries compile them with the default Fortran integer size, which is normally 32-bits wide.

The following instructions were tested with the development version of Octave and GCC 4.3.4 on an x86_64 Debian system.

The versions listed below are the versions used for testing. If newer versions of these packages are available, you should try to use them, although there may be some differences.

All libraries and header files will be installed in subdirectories of `$prefix64` (you must choose the location of this directory).

- BLAS and LAPACK (`http://www.netlib.org/lapack`)

  Reference versions for both libraries are included in the reference LAPACK 3.2.1 distribution from `netlib.org`.

  - Copy the file `make.inc.example` and name it `make.inc`. The options `-fdefault-integer-8` and `-fPIC` (on 64-bit CPU) have to be added to the variable `OPTS` and `NOOPT`.
  - Once you have compiled this library make sure that you use it for compiling Suite Sparse and Octave. In the following we assume that you installed the LAPACK library as $prefix64/lib/liblapack.a.

- QRUPDATE (`http://sourceforge.net/projects/qrupdate`)

  In the `Makeconf` file:

  - Add `-fdefault-integer-8` to `FFLAGS`.
  - Adjust the BLAS and LAPACK variables as needed if your 64-bit aware BLAS and LAPACK libraries are in a non-standard location.
  - Set `PREFIX` to the top-level directory of your install tree.
  - Run `make solib` to make a shared library.
  - Run `make install` to install the library.

- SuiteSparse (`http://www.cise.ufl.edu/research/sparse/SuiteSparse`)

  Pass the following options to `make` to enable 64-bit integers for BLAS library calls. On 64-bit Windows systems, use `-DLONGBLAS="long long"` instead.

```
CFLAGS='-DLONGBLAS=long'
CXXFLAGS='-DLONGBLAS=long'
```

The SuiteSparse makefiles don't generate shared libraries. On some systems, you can generate them by doing something as simple as

```
top=$(pwd)
for f in *.a; do
  mkdir tmp
  cd tmp
  ar vx ../$f
  gcc -shared -o ../${f%%.a}.so *.o
  cd $top
  rm -rf tmp
done
```

Other systems may require a different solution.

- ARPACK (http://forge.scilab.org/index.php/p/arpack-ng/)
  - Add `-fdefault-integer-8` to `FFLAGS` when running configure.
  - Run `make` to build the library.
  - Run `make install` to install the library.
- ATLAS instead of reference BLAS and LAPACK

  Suggestions on how to compile ATLAS would be most welcome.
- GLPK
- Qhull (`http://www.qhull.org`)

  Both GLPK and Qhull use `int` internally so maximum problem sizes may be limited.
- Octave

  Octave's 64-bit index support is activated with the configure option `--enable-64`.

  ```
  ./configure \
    LD_LIBRARY_PATH="$prefix64/lib" \
    CPPFLAGS="-I$prefix64/include" LDFLAGS="-L$prefix64/lib" \
    --enable-64
  ```

  You must ensure that all Fortran sources except those in the `liboctave/cruft/ranlib` directory are compiled such that INTEGERS are 8-bytes wide. If you are using gfortran, the configure script should automatically set the Makefile variable `F77_INTEGER_8_FLAG` to `-fdefault-integer-8`. If you are using another compiler, you must set this variable yourself. You should NOT set this flag in `FFLAGS`, otherwise the files in `liboctave/cruft/ranlib` will be miscompiled.
- Other dependencies

  Probably nothing special needs to be done for the following dependencies. If you discover that something does need to be done, please submit a bug report.
  - pcre
  - zlib
  - hdf5
  - fftw3

- cURL
- GraphicsMagick++
- OpenGL
- freetype
- fontconfig
- fltk

G.4 Installation Problems

This section contains a list of problems (and some apparent problems that don't really mean anything is wrong) that may show up during installation of Octave.

- On some SCO systems, `info` fails to compile if `HAVE_TERMIOS_H` is defined in `config.h`. Simply removing the definition from `info/config.h` should allow it to compile.
- If `configure` finds `dlopen`, `dlsym`, `dlclose`, and `dlerror`, but not the header file `dlfcn.h`, you need to find the source for the header file and install it in the directory `usr/include`. This is reportedly a problem with Slackware 3.1. For Linux/GNU systems, the source for `dlfcn.h` is in the `ldso` package.
- Building `.oct` files doesn't work.

  You should probably have a shared version of `libstdc++`. A patch is needed to build shared versions of version 2.7.2 of `libstdc++` on the HP-PA architecture. You can find the patch at `ftp://ftp.cygnus.com/pub/g++/libg++-2.7.2-hppa-gcc-fix`.
- On some DEC alpha systems there may be a problem with the `libdxml` library, resulting in floating point errors and/or segmentation faults in the linear algebra routines called by Octave. If you encounter such problems, then you should modify the configure script so that `SPECIAL_MATH_LIB` is not set to `-ldxml`.
- On FreeBSD systems Octave may hang while initializing some internal constants. The fix appears to be to use

  ```
  options      GPL_MATH_EMULATE
  ```

  rather than

  ```
  options      MATH_EMULATE
  ```

  in the kernel configuration files (typically found in the directory `/sys/i386/conf`). After making this change, you'll need to rebuild the kernel, install it, and reboot.
- If you encounter errors like

  ```
  passing ‘void (*)()’ as argument 2 of
    ‘octave_set_signal_handler(int, void (*)(int))’
  ```

  or

  ```
  warning: ANSI C++ prohibits conversion from ‘(int)’
           to ‘(...)’
  ```

  while compiling `sighandlers.cc`, you may need to edit some files in the `gcc` include subdirectory to add proper prototypes for functions there. For example, Ultrix 4.2 needs proper declarations for the `signal` function and the `SIG_IGN` macro in the file `signal.h`.

On some systems the `SIG_IGN` macro is defined to be something like this:

```
#define  SIG_IGN  (void (*)())1
```

when it should really be something like:

```
#define  SIG_IGN  (void (*)(int))1
```

to match the prototype declaration for the `signal` function. This change should also be made for the `SIG_DFL` and `SIG_ERR` symbols. It may be necessary to change the definitions in `sys/signal.h` as well.

The `gcc` `fixincludes` and `fixproto` scripts should probably fix these problems when `gcc` installs its modified set of header files, but I don't think that's been done yet.

You should not change the files in `/usr/include`. You can find the `gcc` include directory tree by running the command

```
gcc -print-libgcc-file-name
```

The directory of `gcc` include files normally begins in the same directory that contains the file `libgcc.a`.

- Some of the Fortran subroutines may fail to compile with older versions of the Sun Fortran compiler. If you get errors like

```
zgemm.f:
        zgemm:
warning: unexpected parent of complex expression subtree
zgemm.f, line 245: warning: unexpected parent of complex
  expression subtree
warning: unexpected parent of complex expression subtree
zgemm.f, line 304: warning: unexpected parent of complex
  expression subtree
warning: unexpected parent of complex expression subtree
zgemm.f, line 327: warning: unexpected parent of complex
  expression subtree
pcc_binval: missing IR_CONV in complex op
make[2]: *** [zgemm.o] Error 1
```

  when compiling the Fortran subroutines in the `liboctave/cruft` subdirectory, you should either upgrade your compiler or try compiling with optimization turned off.

- On NeXT systems, if you get errors like this:

```
/usr/tmp/cc007458.s:unknown:Undefined local
      symbol LBB7656
/usr/tmp/cc007458.s:unknown:Undefined local
      symbol LBE7656
```

  when compiling `Array.cc` and `Matrix.cc`, try recompiling these files without `-g`.

- Some people have reported that calls to system() and the pager do not work on SunOS systems. This is apparently due to having `G_HAVE_SYS_WAIT` defined to be 0 instead of 1 when compiling `libg++`.

- On systems where the reference BLAS library is used the following matrix-by-vector multiplication incorrectly handles NaN values of the form `NaN * 0`.

```
[NaN, 1; 0, 0] * [0; 1]
⇒
[ 1
  0 ]

correct result ⇒
[ NaN
  0   ]
```

Install a different BLAS library such as OpenBLAS or ATLAS to correct this issue.

- On NeXT systems, linking to `libsys_s.a` may fail to resolve the following functions

  ```
  _tcgetattr
  _tcsetattr
  _tcflow
  ```

  which are part of `libposix.a`. Unfortunately, linking Octave with `-posix` results in the following undefined symbols.

  ```
  .destructors_used
  .constructors_used
  _objc_msgSend
  _NXGetDefaultValue
  _NXRegisterDefaults
  .objc_class_name_NXStringTable
  .objc_class_name_NXBundle
  ```

  One kluge around this problem is to extract `termios.o` from `libposix.a`, put it in Octave's `src` directory, and add it to the list of files to link together in the makefile. Suggestions for better ways to solve this problem are welcome!

- If Octave crashes immediately with a floating point exception, it is likely that it is failing to initialize the IEEE floating point values for infinity and NaN.

  If your system actually does support IEEE arithmetic, you should be able to fix this problem by modifying the function `octave_ieee_init` in the file `lo-ieee.cc` to correctly initialize Octave's internal infinity and NaN variables.

  If your system does not support IEEE arithmetic but Octave's configure script incorrectly determined that it does, you can work around the problem by editing the file `config.h` to not define `HAVE_ISINF`, `HAVE_FINITE`, and `HAVE_ISNAN`.

  In any case, please report this as a bug since it might be possible to modify Octave's configuration script to automatically determine the proper thing to do.

- If Octave is unable to find a header file because it is installed in a location that is not normally searched by the compiler, you can add the directory to the include search path by specifying (for example) `CPPFLAGS=-I/some/nonstandard/directory` as an argument to `configure`. Other variables that can be specified this way are `CFLAGS`, `CXXFLAGS`, `FFLAGS`, and `LDFLAGS`. Passing them as options to the configure script also records them in the `config.status` file. By default, `CPPFLAGS` and `LDFLAGS` are empty, `CFLAGS` and `CXXFLAGS` are set to `"-g -O2"` and `FFLAGS` is set to `"-O"`.

Appendix H Emacs Octave Support

The development of Octave code can greatly be facilitated using Emacs with Octave mode, a major mode for editing Octave files which can e.g. automatically indent the code, do some of the typing (with Abbrev mode) and show keywords, comments, strings, etc. in different faces (with Font-lock mode on devices that support it).

It is also possible to run Octave from within Emacs, either by directly entering commands at the prompt in a buffer in Inferior Octave mode, or by interacting with Octave from within a file with Octave code. This is useful in particular for debugging Octave code.

Finally, you can convince Octave to use the Emacs info reader for `help -i`.

All functionality is provided by the Emacs Lisp package EOS (for "Emacs Octave Support"). This chapter describes how to set up and use this package.

Please contact `Kurt.Hornik@wu-wien.ac.at` if you have any questions or suggestions on using EOS.

H.1 Installing EOS

The Emacs package EOS consists of the three files `octave-mod.el`, `octave-inf.el`, and `octave-hlp.el`. These files, or better yet their byte-compiled versions, should be somewhere in your Emacs load-path.

If you have GNU Emacs with a version number at least as high as 19.35, you are all set up, because EOS is respectively will be part of GNU Emacs as of version 19.35.

Otherwise, copy the three files from the `emacs` subdirectory of the Octave distribution to a place where Emacs can find them (this depends on how your Emacs was installed). Byte-compile them for speed if you want.

H.2 Using Octave Mode

If you are lucky, your sysadmins have already arranged everything so that Emacs automatically goes into Octave mode whenever you visit an Octave code file as characterized by its extension `.m`. If not, proceed as follows.

1. To begin using Octave mode for all `.m` files you visit, add the following lines to a file loaded by Emacs at startup time, typically your `~/.emacs` file:

   ```
   (autoload 'octave-mode "octave-mod" nil t)
   (setq auto-mode-alist
         (cons '("\\.m$" . octave-mode) auto-mode-alist))
   ```

2. Finally, to turn on the abbrevs, auto-fill and font-lock features automatically, also add the following lines to one of the Emacs startup files:

   ```
   (add-hook 'octave-mode-hook
             (lambda ()
               (abbrev-mode 1)
               (auto-fill-mode 1)
               (if (eq window-system 'x)
                   (font-lock-mode 1))))
   ```

   See the Emacs manual for more information about how to customize Font-lock mode.

In Octave mode, the following special Emacs commands can be used in addition to the standard Emacs commands.

C-h m Describe the features of Octave mode.

LFD Reindent the current Octave line, insert a newline and indent the new line (`octave-reindent-then-newline-and-indent`). An abbrev before point is expanded if `abbrev-mode` is non-`nil`.

TAB Indents current Octave line based on its contents and on previous lines (`indent-according-to-mode`).

; Insert an "electric" semicolon (`octave-electric-semi`). If `octave-auto-indent` is non-`nil`, reindent the current line. If `octave-auto-newline` is non-`nil`, automagically insert a newline and indent the new line.

`* Start entering an abbreviation (`octave-abbrev-start`). If Abbrev mode is turned on, typing *`C-h or *`?* lists all abbrevs. Any other key combination is executed normally. Note that all Octave abbrevs start with a grave accent.

M-LFD Break line at point and insert continuation marker and alignment (`octave-split-line`).

M-TAB Perform completion on Octave symbol preceding point, comparing that symbol against Octave's reserved words and built-in variables (`octave-complete-symbol`).

M-C-a Move backward to the beginning of a function (`octave-beginning-of-defun`). With prefix argument *N*, do it that many times if *N* is positive; otherwise, move forward to the *N*-th following beginning of a function.

M-C-e Move forward to the end of a function (`octave-end-of-defun`). With prefix argument *N*, do it that many times if *N* is positive; otherwise, move back to the *N*-th preceding end of a function.

M-C-h Puts point at beginning and mark at the end of the current Octave function, i.e., the one containing point or following point (`octave-mark-defun`).

M-C-q Properly indents the Octave function which contains point (`octave-indent-defun`).

M-; If there is no comment already on this line, create a code-level comment (started by two comment characters) if the line is empty, or an in-line comment (started by one comment character) otherwise (`octave-indent-for-comment`). Point is left after the start of the comment which is properly aligned.

C-c ; Puts the comment character '`#`' (more precisely, the string value of `octave-comment-start`) at the beginning of every line in the region (`octave-comment-region`). With just *C-u* prefix argument, uncomment each line in the region. A numeric prefix argument *N* means use *N* comment characters.

C-c : Uncomments every line in the region (`octave-uncomment-region`).

C-c C-p Move one line of Octave code backward, skipping empty and comment lines (`octave-previous-code-line`). With numeric prefix argument *N*, move that many code lines backward (forward if *N* is negative).

C-c C-n Move one line of Octave code forward, skipping empty and comment lines (`octave-next-code-line`). With numeric prefix argument *N*, move that many code lines forward (backward if *N* is negative).

C-c C-a Move to the 'real' beginning of the current line (`octave-beginning-of-line`). If point is in an empty or comment line, simply go to its beginning; otherwise, move backwards to the beginning of the first code line which is not inside a continuation statement, i.e., which does not follow a code line ending in '`...`' or '\', or is inside an open parenthesis list.

C-c C-e Move to the 'real' end of the current line (`octave-end-of-line`). If point is in a code line, move forward to the end of the first Octave code line which does not end in '`...`' or '\' or is inside an open parenthesis list. Otherwise, simply go to the end of the current line.

C-c M-C-n Move forward across one balanced begin-end block of Octave code (`octave-forward-block`). With numeric prefix argument *N*, move forward across *n* such blocks (backward if *N* is negative).

C-c M-C-p Move back across one balanced begin-end block of Octave code (`octave-backward-block`). With numeric prefix argument *N*, move backward across *N* such blocks (forward if *N* is negative).

C-c M-C-d Move forward down one begin-end block level of Octave code (`octave-down-block`). With numeric prefix argument, do it that many times; a negative argument means move backward, but still go down one level.

C-c M-C-u Move backward out of one begin-end block level of Octave code (`octave-backward-up-block`). With numeric prefix argument, do it that many times; a negative argument means move forward, but still to a less deep spot.

C-c M-C-h Put point at the beginning of this block, mark at the end (`octave-mark-block`). The block marked is the one that contains point or follows point.

C-c] Close the current block on a separate line (`octave-close-block`). An error is signaled if no block to close is found.

C-c f Insert a function skeleton, prompting for the function's name, arguments and return values which have to be entered without parentheses (`octave-insert-defun`).

C-c C-h Search the function, operator and variable indices of all info files with documentation for Octave for entries (`octave-help`). If used interactively, the entry is prompted for with completion. If multiple matches are found, one can cycle through them using the standard '`,`' (`Info-index-next`) command of the Info reader.

The variable `octave-help-files` is a list of files to search through and defaults to `'("octave")`. If there is also an Octave Local Guide with corresponding info file, say, `octave-LG`, you can have `octave-help` search both files by

```
(setq octave-help-files '("octave" "octave-LG"))
```

in one of your Emacs startup files.

A common problem is that the RET key does *not* indent the line to where the new text should go after inserting the newline. This is because the standard Emacs convention is that RET (aka *C-m*) just adds a newline, whereas LFD (aka *C-j*) adds a newline and indents it. This is particularly inconvenient for users with keyboards which do not have a special LFD key at all; in such cases, it is typically more convenient to use RET as the LFD key (rather than typing *C-j*).

You can make RET do this by adding

```
(define-key octave-mode-map "\C-m"
  'octave-reindent-then-newline-and-indent)
```

to one of your Emacs startup files. Another, more generally applicable solution is

```
(defun RET-behaves-as-LFD ()
  (let ((x (key-binding "\C-j")))
    (local-set-key "\C-m" x)))
(add-hook 'octave-mode-hook 'RET-behaves-as-LFD)
```

(this works for all modes by adding to the startup hooks, without having to know the particular binding of RET in that mode!). Similar considerations apply for using M-RET as M-LFD. As Barry A. Warsaw `bwarsaw@cnri.reston.va.us` says in the documentation for his `cc-mode`, "This is a very common question. `:-)` If you want this to be the default behavior, don't lobby me, lobby RMS!"

The following variables can be used to customize Octave mode.

`octave-auto-indent`
: Non-`nil` means auto-indent the current line after a semicolon or space. Default is `nil`.

`octave-auto-newline`
: Non-`nil` means auto-insert a newline and indent after semicolons are typed. The default value is `nil`.

`octave-blink-matching-block`
: Non-`nil` means show matching begin of block when inserting a space, newline or '`;`' after an else or end keyword. Default is `t`. This is an extremely useful feature for automatically verifying that the keywords match—if they don't, an error message is displayed.

`octave-block-offset`
: Extra indentation applied to statements in block structures. Default is 2.

`octave-continuation-offset`
: Extra indentation applied to Octave continuation lines. Default is 4.

`octave-continuation-string`
: String used for Octave continuation lines. Normally '\'.

`octave-mode-startup-message`
: If `t` (default), a startup message is displayed when Octave mode is called.

If Font Lock mode is enabled, Octave mode will display

- strings in `font-lock-string-face`
- comments in `font-lock-comment-face`
- the Octave reserved words (such as all block keywords) and the text functions (such as '`cd`' or '`who`') which are also reserved using `font-lock-keyword-face`
- the built-in operators ('`&&`', '`==`', ...) using `font-lock-reference-face`
- and the function names in function declarations in `font-lock-function-name-face`.

There is also rudimentary support for Imenu (currently, function names can be indexed).

You can generate TAGS files for Emacs from Octave `.m` files using the shell script `octave-tags` that is installed alongside your copy of Octave.

Customization of Octave mode can be performed by modification of the variable `octave-mode-hook`. If the value of this variable is non-`nil`, turning on Octave mode calls its value.

If you discover a problem with Octave mode, you can conveniently send a bug report using *C-c C-b* (`octave-submit-bug-report`). This automatically sets up a mail buffer with version information already added. You just need to add a description of the problem, including a reproducible test case and send the message.

H.3 Running Octave from Within Emacs

The package `octave` provides commands for running an inferior Octave process in a special Emacs buffer. Use

```
M-x run-octave
```

to directly start an inferior Octave process. If Emacs does not know about this command, add the line

```
(autoload 'run-octave "octave-inf" nil t)
```

to your `.emacs` file.

This will start Octave in a special buffer the name of which is specified by the variable `inferior-octave-buffer` and defaults to `"*Inferior Octave*"`. From within this buffer, you can interact with the inferior Octave process 'as usual', i.e., by entering Octave commands at the prompt. The buffer is in Inferior Octave mode, which is derived from the standard Comint mode, a major mode for interacting with an inferior interpreter. See the documentation for `comint-mode` for more details, and use *C-h b* to find out about available special keybindings.

You can also communicate with an inferior Octave process from within files with Octave code (i.e., buffers in Octave mode), using the following commands.

C-c i l
: Send the current line to the inferior Octave process (`octave-send-line`). With positive prefix argument *N*, send that many lines. If `octave-send-line-auto-forward` is non-`nil`, go to the next unsent code line.

C-c i b
: Send the current block to the inferior Octave process (`octave-send-block`).

C-c i f
: Send the current function to the inferior Octave process (`octave-send-defun`).

C-c i r
: Send the region to the inferior Octave process (`octave-send-region`).

C-c i s Make sure that 'inferior-octave-buffer' is displayed (`octave-show-process-buffer`).

C-c i h Delete all windows that display the inferior Octave buffer (`octave-hide-process-buffer`).

C-c i k Kill the inferior Octave process and its buffer (`octave-kill-process`).

The effect of the commands which send code to the Octave process can be customized by the following variables.

`octave-send-echo-input`
: Non-`nil` means echo input sent to the inferior Octave process. Default is `t`.

`octave-send-show-buffer`
: Non-`nil` means display the buffer running the Octave process after sending a command (but without selecting it). Default is `t`.

If you send code and there is no inferior Octave process yet, it will be started automatically.

The startup of the inferior Octave process is highly customizable. The variable `inferior-octave-startup-args` can be used for specifying command lines arguments to be passed to Octave on startup as a list of strings. For example, to suppress the startup message and use 'traditional' mode, set this to `'("-q" "--traditional")`. You can also specify a startup file of Octave commands to be loaded on startup; note that these commands will not produce any visible output in the process buffer. Which file to use is controlled by the variable `inferior-octave-startup-file`. If this is `nil`, the file `~/.emacs-octave` is used if it exists.

And finally, `inferior-octave-mode-hook` is run after starting the process and putting its buffer into Inferior Octave mode. Hence, if you like the up and down arrow keys to behave in the interaction buffer as in the shell, and you want this buffer to use nice colors, add

```
(add-hook 'inferior-octave-mode-hook
          (lambda ()
            (turn-on-font-lock)
            (define-key inferior-octave-mode-map [up]
              'comint-previous-input)
            (define-key inferior-octave-mode-map [down]
              'comint-next-input)))
```

to your `.emacs` file. You could also swap the roles of *C*-`a` (`beginning-of-line`) and `C-c C-a` (`comint-bol`) using this hook.

> **Note** that if you set your Octave prompts to something different from the defaults, make sure that `inferior-octave-prompt` matches them. Otherwise, *nothing* will work, because Emacs will not know when Octave is waiting for input, or done sending output.

H.4 Using the Emacs Info Reader for Octave

You may also use the Emacs Info reader with Octave's `doc` function. For this, the package `gnuserv` needs to be installed.

If gnuserv is installed, add the lines

```
(autoload 'octave-help "octave-hlp" nil t)
(require 'gnuserv)
(gnuserv-start)
```

to your .emacs file.

You can use either 'plain' Emacs Info or the function octave-help as your Octave info reader (for 'help -i'). In the former case, use info_program ("info-emacs-info"). The latter is perhaps more attractive because it allows one to look up keys in the indices of *several* info files related to Octave (provided that the Emacs variable octave-help-files is set correctly). In this case, use info_program ("info-emacs-octave-help").

If you use Octave from within Emacs, it is best to add these settings to your ~/.emacs-octave startup file (or the file pointed to by the Emacs variable inferior-octave-startup-file).

Appendix I Grammar and Parser

This appendix will eventually contain a semi-formal description of Octave's language.

I.1 Keywords

The following identifiers are keywords, and may not be used as variable or function names:

`__FILE__`	`__LINE__`	`break`
`case`	`catch`	`classdef`
`continue`	`do`	`else`
`elseif`	`end`	`end_try_catch`
`end_unwind_protect`	`endclassdef`	`endenumeration`
`endevents`	`endfor`	`endfunction`
`endif`	`endmethods`	`endparfor`
`endproperties`	`endswitch`	`endwhile`
`enumeration`	`events`	`for`
`function`	`global`	`if`
`methods`	`otherwise`	`parfor`
`persistent`	`properties`	`return`
`switch`	`try`	`until`
`unwind_protect`	`unwind_protect_cleanup`	`while`

The function `iskeyword` can be used to quickly check whether an identifier is reserved by Octave.

`iskeyword ()` [Built-in Function]
`iskeyword (`*name*`)` [Built-in Function]

Return true if *name* is an Octave keyword.

If *name* is omitted, return a list of keywords.

See also: [isvarname], page 123, [exist], page 130.

I.2 Parser

The parser has a number of variables that affect its internal operation. These variables are generally documented in the manual alongside the code that they affect. For example, `allow_noninteger_range_as_index` is discussed in the section on index expressions.

In addition, there are three non-specific parser customization functions. `add_input_event_hook` can be used to schedule a user function for periodic evaluation. `remove_input_event_hook` will stop a user function from being evaluated periodically.

id `= add_input_event_hook (`*fcn*`)` [Built-in Function]
id `= add_input_event_hook (`*fcn*`,` *data*`)` [Built-in Function]

Add the named function or function handle *fcn* to the list of functions to call periodically when Octave is waiting for input.

The function should have the form

```
fcn (data)
```

If *data* is omitted, Octave calls the function without any arguments.

The returned identifier may be used to remove the function handle from the list of input hook functions.

See also: [remove_input_event_hook], page 916.

`remove_input_event_hook (`*name*`)` [Built-in Function]
`remove_input_event_hook (`*fcn_id*`)` [Built-in Function]

Remove the named function or function handle with the given identifier from the list of functions to call periodically when Octave is waiting for input.

See also: [add_input_event_hook], page 915.

Finally, when the parser cannot identify an input token it calls a particular function to handle this. By default, this is the internal function `"__unimplemented__"` which makes suggestions about possible Octave substitutes for MATLAB functions.

val `= missing_function_hook ()` [Built-in Function]
old_val `= missing_function_hook (`*new_val*`)` [Built-in Function]
`missing_function_hook (`*new_val*`, "`*local*`")` [Built-in Function]

Query or set the internal variable that specifies the function to call when an unknown identifier is requested.

When called from inside a function with the `"local"` option, the variable is changed locally for the function and any subroutines it calls. The original variable value is restored when exiting the function.

See also: [missing_component_hook], page 816.

Appendix J GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.
e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

   Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
   b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
   c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or
e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

Concept Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Function Index

A

B

C

D

E

F

G

H

I

J

K

L

M

Q

R

S

T

U

V

W

X

Y

Z

Operator Index

\

{

}

|

~