GNU Scientific Library – Design document

Mark Galassi
Los Alamos National Laboratory
James Theiler
Astrophysics and Radiation Measurements Group, Los Alamos National Laboratory
Brian Gough
Network Theory Limited

Table of Contents

About GSL

The *GNU Scientific Library* is a library of scientific subroutines. It aims to provide a convenient interface to routines that do standard (and not so standard) tasks that arise in scientific research. More than that, it also provides the source code. Users are welcome to alter, adjust, modify, and improve the interfaces and/or implementations of whichever routines might be needed for a particular purpose.

GSL is intended to provide a free equivalent to existing proprietary numerical libraries written in C or Fortran, such as NAG, IMSL's CNL, IBM's ESSL, and SGI's SCSL.

The target platform is a low-end desktop workstation. The goal is to provide something which is generally useful, and the library is aimed at general users rather than specialists.

1 Motivation

There is a need for scientists and engineers to have a numerical library that:

- is free (in the sense of freedom, not in the sense of gratis; see the GNU General Public License), so that people can use that library, redistribute it, modify it ...
- is written in C using modern coding conventions, calling conventions, scoping ...
- is clearly and pedagogically documented; preferably with TeXinfo, so as to allow online info, WWW and TeX output.
- uses top quality state-of-the-art algorithms.
- is portable and configurable using *autoconf* and *automake*.
- basically, is GNUlitically correct.

There are strengths and weaknesses with existing libraries:

Netlib (http://www.netlib.org/) is probably the most advanced set of numerical algorithms available on the net, maintained by AT&T. Unfortunately most of the software is written in Fortran, with strange calling conventions in many places. It is also not very well collected, so it is a lot of work to get started with netlib.

GAMS (http://gams.nist.gov/) is an extremely well organized set of pointers to scientific software, but like netlib, the individual routines vary in their quality and their level of documentation.

Numerical Recipes (http://www.nr.com, http://cfata2.harvard.edu/nr/) is an excellent book: it explains the algorithms in a very clear way. Unfortunately the authors released the source code under a license which allows you to use it, but prevents you from re-distributing it. Thus Numerical Recipes is not *free* in the sense of *freedom*. On top of that, the implementation suffers from *fortranitis* and other limitations. [http://www.lysator.liu.se/c/num-recipes-in-c.html]

SLATEC is a large public domain collection of numerical routines in Fortran written under a Department of Energy program in the 1970's. The routines are well tested and have a reasonable overall design (given the limitations of that era). GSL should aim to be a modern version of SLATEC.

NSWC is the Naval Surface Warfare Center numerical library. It is a large public-domain Fortran library, containing a lot of high-quality code. Documentation for the library is hard to find, only a few photocopies of the printed manual are still in circulation.

NAG and *IMSL* both sell high-quality libraries which are proprietary. The NAG library is more advanced and has wider scope than IMSL. The IMSL library leans more towards ease-of-use and makes extensive use of variable length argument lists to emulate "default arguments".

ESSL and *SCSL* are proprietary libraries from IBM and SGI.

Forth Scientific Library [see the URL http://www.taygeta.com/fsl/sciforth.html]. Mainly of interest to Forth users.

Numerical Algorithms with C G. Engeln-Mullges, F. Uhlig. A nice numerical library written in ANSI C with an accompanying textbook. Source code is available but the library is not free software.

NUMAL A C version of the NUMAL library has been written by H.T. Lau and is published as a book and disk with the title "A Numerical Library in C for Scientists and Engineers". Source code is available but the library is not free software.

C Mathematical Function Handbook by Louis Baker. A library of function approximations and methods corresponding to those in the "Handbook of Mathematical Functions" by Abramowitz and Stegun. Source code is available but the library is not free software.

CCMATH by Daniel A. Atkinson. A C numerical library covering similar areas to GSL. The code is quite terse. Earlier versions were under the GPL but unfortunately it has changed to the LGPL in recent versions.

CEPHES A useful collection of high-quality special functions written in C. Not GPL'ed.

WNLIB A small collection of numerical routines written in C by Will Naylor. Public domain.

MESHACH A comprehensive matrix-vector linear algebra library written in C. Freely available but not GPL'ed (non-commercial license).

CERNLIB is a large high-quality Fortran library developed at CERN over many years. It was originally non-free software but has recently been released under the GPL.

COLT is a free numerical library in Java developed at CERN by Wolfgang Hoschek. It is under a BSD-style license.

The long-term goal will be to provide a framework to which the real numerical experts (or their graduate students) will contribute.

2 Contributing

This design document was originally written in 1996. As of 2004, GSL itself is essentially feature complete, the developers are not actively working on any major new functionality.

The main emphasis is now on ensuring the stability of the existing functions, improving consistency, tidying up a few problem areas and fixing any bugs that are reported. Potential contributors are encouraged to gain familiarity with the library by investigating and fixing known problems listed in the '`BUGS`' file in the CVS repository.

Adding large amounts of new code is difficult because it leads to differences in the maturity of different parts of the library. To maintain stability, any new functionality is encouraged as *packages*, built on top of GSL and maintained independently by the author, as in other free software projects (such as the Perl CPAN archive and TeX CTAN archive, etc).

2.1 Packages

The design of GSL permits extensions to be used alongside the existing library easily by simple linking. For example, additional random number generators can be provided in a separate library:

```
$ tar xvfz rngextra-0.1.tar.gz
$ cd rngextra-0.1
$ ./configure; make; make check; make install
$ ...
$ gcc -Wall main.c -lrngextra -lgsl -lgslcblas -lm
```

The points below summarise the package design guidelines. These are intended to ensure that packages are consistent with GSL itself, to make life easier for the end-user and make it possible to distribute popular well-tested packages as part of the core GSL in future.

- Follow the GSL and GNU coding standards described in this document

  This means using the standard GNU packaging tools, such as Automake, providing documentation in Texinfo format, and a test suite. The test suite should run using '`make check`', and use the test functions provided in GSL to produce the output with `PASS:/FAIL:` lines. It is not essential to use libtool since packages are likely to be small, a static library is sufficient and simpler to build.

- Use a new unique prefix for the package (do not use '`gsl_`' – this is reserved for internal use).

  For example, a package of additional random number generators might use the prefix `rngextra`.

  ```
  #include <rngextra.h>

  gsl_rng * r = gsl_rng_alloc (rngextra_lsfr32);
  ```

- Use a meaningful version number which reflects the state of development

  Generally, `0.x` are alpha versions, which provide no guarantees. Following that, `0.9.x` are beta versions, which should be essentially complete, subject only to minor changes and bug fixes. The first major release is `1.0`. Any version number of `1.0` or higher should be suitable for production use with a well-defined API.

The API must not change in a major release and should be backwards-compatible in its behavior (excluding actual bug-fixes), so that existing code do not have to be modified. Note that the API includes all exported definitions, including data-structures defined with `struct`. If you need to change the API in a package, it requires a new major release (e.g. `2.0`).

- Use the GNU General Public License (GPL)

  Follow the normal procedures of obtaining a copyright disclaimer if you would like to have the package considered for inclusion in GSL itself in the future (see Section 3.36 [Legal issues], page 18).

Post announcements of your package releases to `gsl-discuss@sources.redhat.com` so that information about them can be added to the GSL webpages.

An example package '`rngextra`' containing two additional random number generators can be found at `http://www.network-theory.co.uk/download/rngextra/`.

3 Design

3.1 Language for implementation

One language only (C)

Advantages: simpler, compiler available and quite universal.

3.2 Interface to other languages

Wrapper packages are supplied as "extra" packages; not as part of the "core". They are maintained separately by independent contributors.

Use standard tools to make wrappers: swig, g-wrap

3.3 What routines are implemented

Anything which is in any of the existing libraries. Obviously it makes sense to prioritize and write code for the most important areas first.

3.4 What routines are not implemented

- anything which already exists as a high-quality GPL'ed package.
- anything which is too big – i.e. an application in its own right rather than a subroutine

  For example, partial differential equation solvers are often huge and very specialized applications (since there are so many types of PDEs, types of solution, types of grid, etc). This sort of thing should remain separate. It is better to point people to the good applications which exist.
- anything which is independent and useful separately.

  Arguably functions for manipulating date and time, or financial functions might be included in a "scientific" library. However, these sorts of modules could equally well be used independently in other programs, so it makes sense for them to be separate libraries.

3.5 Design of Numerical Libraries

In writing a numerical library there is a unavoidable conflict between completeness and simplicity. Completeness refers to the ability to perform operations on different objects so that the group is "closed". In mathematics objects can be combined and operated on in an infinite number of ways. For example, I can take the derivative of a scalar field with respect to a vector and the derivative of a vector field wrt a scalar (along a path).

There is a definite tendency to unconsciously try to reproduce all these possibilities in a numerical library, by adding new features one by one. After all, it is always easy enough to support just one more feature.... so why not?

Looking at the big picture, no-one would start out by saying "I want to be able to represent every possible mathematical object and operation using C structs" – this is a strategy which is doomed to fail. There is a limited amount of complexity which can be represented in a programming language like C. Attempts to reproduce the complexity of mathematics within such a language would just lead to a morass of unmaintainable code. However, it's easy to go down that road if you don't think about it ahead of time.

It is better to choose simplicity over completeness. In designing new parts of the library keep modules independent where possible. If interdependencies between modules are introduced be sure about where you are going to draw the line.

3.6 Code Reuse

It is useful if people can grab a single source file and include it in their own programs without needing the whole library. Try to allow standalone files like this whenever it is reasonable. Obviously the user might need to define a few macros, such as GSL_ERROR, to compile the file but that is ok. Examples where this can be done: grabbing a single random number generator.

3.7 Standards and conventions

The people who kick off this project should set the coding standards and conventions. In order of precedence the standards that we follow are,

- We follow the GNU Coding Standards.
- We follow the conventions of the ANSI Standard C Library.
- We follow the conventions of the GNU C Library.
- We follow the conventions of the glib GTK support Library.

The references for these standards are the *GNU Coding Standards* document, Harbison and Steele *C: A Reference Manual*, the *GNU C Library Manual* (version 2), and the Glib source code.

For mathematical formulas, always follow the conventions in Abramowitz & Stegun, the *Handbook of Mathematical Functions*, since it is the definitive reference and also in the public domain.

If the project has a philosophy it is to "Think in C". Since we are working in C we should only do what is natural in C, rather than trying to simulate features of other languages. If there is something which is unnatural in C and has to be simulated then we avoid using it. If this means leaving something out of the library, or only offering a limited version then so be it. It is not worthwhile making the library over-complicated. There are numerical libraries in other languages, and if people need the features of those languages it would be sensible for them to use the corresponding libraries, rather than coercing a C library into doing that job.

It should be borne in mind at all time that C is a macro-assembler. If you are in doubt about something being too complicated ask yourself the question "Would I try to write this in macro-assembler?" If the answer is obviously "No" then do not try to include it in GSL. [BJG]

It will be useful to read the following paper,

> Kiem-Phong Vo, "The Discipline and Method Architecture for Reusable Libraries", Software - Practice & Experience, v.30, pp.107-128, 2000.

It is available from `http://www.research.att.com/sw/tools/sfio/dm-spe.ps` or the earlier technical report Kiem-Phong Vo, "An Architecture for Reusable Libraries" `http://citeseer.nj.nec.com/48973.html`.

There are associated papers on Vmalloc, SFIO, and CDT which are also relevant to the design of portable C libraries.

> Kiem-Phong Vo, "Vmalloc: A General and Efficient Memory Allocator". Software Practice & Experience, 26:1–18, 1996.
>
> `http://www.research.att.com/sw/tools/vmalloc/vmalloc.ps`
>
> Kiem-Phong Vo. "Cdt: A Container Data Type Library". Soft. Prac. & Exp., 27:1177–1197, 1997
>
> `http://www.research.att.com/sw/tools/cdt/cdt.ps`
>
> David G. Korn and Kiem-Phong Vo, "Sfio: Safe/Fast String/File IO", Proceedings of the Summer '91 Usenix Conference, pp. 235-256, 1991.
>
> `http://citeseer.nj.nec.com/korn91sfio.html`

Source code should be indented according to the GNU Coding Standards, with spaces not tabs. For example, by using the `indent` command:

```
indent -gnu -nut *.c *.h
```

The `-nut` option converts tabs into spaces.

3.8 Background and Preparation

Before implementing something be sure to research the subject thoroughly! This will save a lot of time in the long-run. The two most important steps are,

1. to determine whether there is already a free library (GPL or GPL-compatible) which does the job. If so, there is no need to reimplement it. Carry out a search on Netlib, GAMs, na-net, sci.math.num-analysis and the web in general. This should also provide you with a list of existing proprietary libraries which are relevant, keep a note of these for future reference in step 2.
2. make a comparative survey of existing implementations in the commercial/free libraries. Examine the typical APIs, methods of communication between program and subroutine, and classify them so that you are familiar with the key concepts or features that an implementation may or may not have, depending on the relevant tradeoffs chosen. Be sure to review the documentation of existing libraries for useful references.
3. read up on the subject and determine the state-of-the-art. Find the latest review papers. A search of the following journals should be undertaken.

    ACM Transactions on Mathematical Software

    Numerische Mathematik

    Journal of Computation and Applied Mathematics

    Computer Physics Communications

SIAM Journal of Numerical Analysis
SIAM Journal of Scientific Computing

Keep in mind that GSL is not a research project. Making a good implementation is difficult enough, without also needing to invent new algorithms. We want to implement existing algorithms whenever possible. Making minor improvements is ok, but don't let it be a time-sink.

3.9 Choice of Algorithms

Whenever possible choose algorithms which scale well and always remember to handle asymptotic cases. This is particularly relevant for functions with integer arguments. It is tempting to implement these using the simple $O(n)$ algorithms used to define the functions, such as the many recurrence relations found in Abramowitz and Stegun. While such methods might be acceptable for $n = O(10 - 100)$ they will not be satisfactory for a user who needs to compute the same function for $n = 1000000$.

Similarly, do not make the implicit assumption that multivariate data has been scaled to have components of the same size or O(1). Algorithms should take care of any necessary scaling or balancing internally, and use appropriate norms (e.g. `|Dx|` where D is a diagonal scaling matrix, rather than `|x|`).

3.10 Documentation

Documentation: the project leaders should give examples of how things are to be documented. High quality documentation is absolutely mandatory, so documentation should introduce the topic, and give careful reference for the provided functions. The priority is to provide reference documentation for each function. It is not necessary to provide tutorial documentation.

Use free software, such as GNU Plotutils, to produce the graphs in the manual.

Some of the graphs have been made with gnuplot which is not truly free (or GNU) software, and some have been made with proprietary programs. These should be replaced with output from GNU plotutils.

When citing references be sure to use the standard, definitive and best reference books in the field, rather than lesser known text-books or introductory books which happen to be available (e.g. from undergraduate studies). For example, references concerning algorithms should be to Knuth, references concerning statistics should be to Kendall & Stuart, references concerning special functions should be to Abramowitz & Stegun (Handbook of Mathematical Functions AMS-55), etc.

The standard references have a better chance of being available in an accessible library for the user. If they are not available and the user decides to buy a copy in order to look up the reference then this also gives them the best quality book which should also cover the largest number of other references in the GSL Manual. If many different books were to be referenced this would be an expensive and inefficient use of resources for a user who needs to look up the details of the algorithms. Reference books also stay in print much longer than text books, which are often out-of-print after a few years.

Similarly, cite original papers wherever possible. Be sure to keep copies of these for your own reference (e.g. when dealing with bug reports) or to pass on to future maintainers.

If you need help in tracking down references, ask on the `gsl-discuss` mailing list. There is a group of volunteers with access to good libraries who have offered to help GSL developers get copies of papers.

[JT section: written by James Theiler

And we furthermore promise to try as hard as possible to document the software: this will ideally involve discussion of why you might want to use it, what precisely it does, how precisely to invoke it, how more-or-less it works, and where we learned about the algorithm, and (unless we wrote it from scratch) where we got the code. We do not plan to write this entire package from scratch, but to cannibalize existing mathematical freeware, just as we expect our own software to be cannibalized.]

3.11 Namespace

Use `gsl_` as a prefix for all exported functions and variables.

Use `GSL_` as a prefix for all exported macros.

All exported header files should have a filename with the prefix `gsl_`.

All installed libraries should have a name like libgslhistogram.a

Any installed executables (utility programs etc) should have the prefix `gsl-` (with a hyphen, not an underscore).

All function names, variables, etc should be in lower case. Macros and preprocessor variables should be in upper case.

3.12 Header files

Installed header files should be idempotent, i.e. surround them by the preprocessor conditionals like the following,

```
#ifndef __GSL_HISTOGRAM_H__
#define __GSL_HISTOGRAM_H__
...
#endif /* __GSL_HISTOGRAM_H__ */
```

3.13 Target system

The target system is ANSI C, with a full Standard C Library, and IEEE arithmetic.

3.14 Function Names

Each module has a name, which prefixes any function names in that module, e.g. the module gsl_fft has function names like gsl_fft_init. The modules correspond to subdirectories of the library source tree.

3.15 Object-orientation

The algorithms should be object oriented, but only to the extent that is easy in portable ANSI C. The use of casting or other tricks to simulate inheritance is not desirable, and the user should not have to be aware of anything like that. This means many types of patterns are ruled out. However, this is not considered a problem – they are too complicated for the library.

Note: it is possible to define an abstract base class easily in C, using function pointers. See the rng directory for an example.

When reimplementing public domain fortran code, please try to introduce the appropriate object concepts as structs, rather than translating the code literally in terms of arrays. The structs can be useful just within the file, you don't need to export them to the user.

For example, if a fortran program repeatedly uses a subroutine like,

```
SUBROUTINE  RESIZE (X, K, ND, K1)
```

where X(K,D) represents a grid to be resized to X(K1,D) you can make this more readable by introducing a struct,

```
struct grid {
    int nd;  /* number of dimensions */
    int k;   /* number of bins */
    double * x;   /* partition of axes, array of size x[k][nd] */
}

void
resize_grid (struct grid * g, int k_new)
{
...
}
```

Similarly, if you have a frequently recurring code fragment within a single file you can define a static or static inline function for it. This is typesafe and saves writing out everything in full.

3.16 Comments

Follow the GNU Coding Standards. A relevant quote is,

"Please write complete sentences and capitalize the first word. If a lower-case identifier comes at the beginning of a sentence, don't capitalize it! Changing the spelling makes it a different identifier. If you don't like starting a sentence with a lower case letter, write the sentence differently (e.g., "The identifier lower-case is ...")."

3.17 Minimal structs

We prefer to make structs which are *minimal*. For example, if a certain type of problem can be solved by several classes of algorithm (e.g. with and without derivative information) it is better to make separate types of struct to handle those cases. i.e. run time type identification is not desirable.

3.18 Algorithm decomposition

Iterative algorithms should be decomposed into an INITIALIZE, ITERATE, TEST form, so that the user can control the progress of the iteration and print out intermediate results. This is better than using call-backs or using flags to control whether the function prints out intermediate results. In fact, call-backs should not be used – if they seem necessary then it's a sign that the algorithm should be broken down further into individual components so that the user has complete control over them.

For example, when solving a differential equation the user may need to be able to advance the solution by individual steps, while tracking a realtime process. This is only possible if the algorithm is broken down into step-level components. Higher level decompositions would not give sufficient flexibility.

3.19 Memory allocation and ownership

Functions which allocate memory on the heap should end in _alloc (e.g. gsl_foo_alloc) and be deallocated by a corresponding _free function (gsl_foo_free).

Be sure to free any memory allocated by your function if you have to return an error in a partially initialized object.

Don't allocate memory 'temporarily' inside a function and then free it before the function returns. This prevents the user from controlling memory allocation. All memory should be allocated and freed through separate functions and passed around as a "workspace" argument. This allows memory allocation to be factored out of tight loops.

3.20 Memory layout

We use flat blocks of memory to store matrices and vectors, not C-style pointer-to-pointer arrays. The matrices are stored in row-major order – i.e. the column index (second index) moves continuously through memory.

3.21 Linear Algebra Levels

Functions using linear algebra are divided into two levels:

For purely "1d" functions we use the C-style arguments (double *, stride, size) so that it is simpler to use the functions in a normal C program, without needing to invoke all the gsl_vector machinery.

The philosophy here is to minimize the learning curve. If someone only needs to use one function, like an fft, they can do so without having to learn about gsl_vector.

This leads to the question of why we don't do the same for matrices. In that case the argument list gets too long and confusing, with (size1, size2, tda) for each matrix and potential ambiguities over row vs column ordering. In this case, it makes sense to use gsl_vector and gsl_matrix, which take care of this for the user.

So really the library has two levels – a lower level based on C types for 1d operations, and a higher level based on gsl_matrix and gsl_vector for general linear algebra.

Of course, it would be possible to define a vector version of the lower level functions too. So far we have not done that because it was not essential – it could be done but it is easy enough to get by using the C arguments, by typing v->data, v->stride, v->size instead. A gsl_vector version of low-level functions would mainly be a convenience.

Please use BLAS routines internally within the library whenever possible for efficiency.

3.22 Exceptions and Error handling

The basic error handling procedure is the return code (see gsl_errno.h for a list of allowed values). Use the GSL_ERROR macro to mark an error. The current definition of this macro is not ideal but it can be changed at compile time.

You should always use the GSL_ERROR macro to indicate an error, rather than just returning an error code. The macro allows the user to trap errors using the debugger (by setting a breakpoint on the function gsl_error).

The only circumstances where GSL_ERROR should not be used are where the return value is "indicative" rather than an error – for example, the iterative routines use the return code to indicate the success or failure of an iteration. By the nature of an iterative algorithm "failure" (a return code of GSL_CONTINUE) is a normal occurrence and there is no need to use GSL_ERROR there.

Be sure to free any memory allocated by your function if you return an error (in particular for errors in partially initialized objects).

3.23 Persistence

If you make an object foo which uses blocks of memory (e.g. vector, matrix, histogram) you can provide functions for reading and writing those blocks,

```
int gsl_foo_fread (FILE * stream, gsl_foo * v);
int gsl_foo_fwrite (FILE * stream, const gsl_foo * v);
int gsl_foo_fscanf (FILE * stream, gsl_foo * v);
int gsl_foo_fprintf (FILE * stream, const gsl_foo * v, const char *format);
```

Only dump out the blocks of memory, not any associated parameters such as lengths. The idea is for the user to build higher level input/output facilities using the functions the library provides. The fprintf/fscanf versions should be portable between architectures, while the binary versions should be the "raw" version of the data. Use the functions

```
int gsl_block_fread (FILE * stream, gsl_block * b);
int gsl_block_fwrite (FILE * stream, const gsl_block * b);
int gsl_block_fscanf (FILE * stream, gsl_block * b);
int gsl_block_fprintf (FILE * stream, const gsl_block * b, const char *format);
```

or

```
int gsl_block_raw_fread (FILE * stream, double * b, size_t n, size_t stride);
int gsl_block_raw_fwrite (FILE * stream, const double * b, size_t n, size_t stri
de);
int gsl_block_raw_fscanf (FILE * stream, double * b, size_t n, size_t stride);
int gsl_block_raw_fprintf (FILE * stream, const double * b, size_t n, size_t str
ide, const char *format);
```

to do the actual reading and writing.

3.24 Using Return Values

Always assign a return value to a variable before using it. This allows easier debugging of the function, and inspection and modification of the return value. If the variable is only needed temporarily then enclose it in a suitable scope.

For example, instead of writing,

```
a = f(g(h(x,y)))
```

use temporary variables to store the intermediate values,

```
{
  double u = h(x,y);
  double v = g(u);
  a = f(v);
}
```

These can then be inspected more easily in the debugger, and breakpoints can be placed more precisely. The compiler will eliminate the temporary variables automatically when the program is compiled with optimization.

3.25 Variable Names

Try to follow existing conventions for variable names,

`dim` number of dimensions

`w` pointer to workspace

`state` pointer to state variable (use `s` if you need to save characters)

`result` pointer to result (output variable)

`abserr` absolute error

`relerr` relative error

`epsabs` absolute tolerance

`epsrel` relative tolerance

`size` the size of an array or vector e.g. double array[size]

`stride` the stride of a vector

`size1` the number of rows in a matrix

`size2` the number of columns in a matrix

`n` general integer number, e.g. number of elements of array, in fft, etc

`r` random number generator (gsl_rng)

3.26 Datatype widths

Be aware that in ANSI C the type `int` is only guaranteed to provide 16-bits. It may provide more, but is not guaranteed to. Therefore if you require 32 bits you must use `long int`, which will have 32 bits or more. Of course, on many platforms the type `int` does have 32 bits instead of 16 bits but we have to code to the ANSI standard rather than a specific platform.

3.27 size_t

All objects (blocks of memory, etc) should be measured in terms of a `size_t` type. Therefore any iterations (e.g. `for(i=0; i<N; i++)`) should also use an index of type `size_t`.

Don't mix `int` and `size_t`. They are *not* interchangeable.

If you need to write a descending loop you have to be careful because `size_t` is unsigned, so instead of

```
for (i = N - 1; i >= 0; i--) { ... } /* DOESN'T WORK */
```

use something like

```
for (i = N; i > 0 && i--;) { ... }
```

to avoid problems with wrap-around at `i=0`.

If you really want to avoid confusion use a separate variable to invert the loop order,

```
for (i = 0; i < N; i++) { j = N - i; ... }
```

3.28 Arrays vs Pointers

A function can be declared with either pointer arguments or array arguments. The C standard considers these to be equivalent. However, it is useful to distinguish between the case of a pointer, representing a single object which is being modified, and an array which represents a set of objects with unit stride (that are modified or not depending on the presence of `const`). For vectors, where the stride is not required to be unity, the pointer form is preferred.

```
/* real value, set on output */
int foo (double * x);

/* real vector, modified */
int foo (double * x, size_t stride, size_t n);

/* constant real vector */
int foo (const double * x, size_t stride, size_t n);

/* real array, modified */
int bar (double x[], size_t n);

/* real array, not modified */
int baz (const double x[], size_t n);
```

3.29 Pointers

Avoid dereferencing pointers on the right-hand side of an expression where possible. It's better to introduce a temporary variable. This is easier for the compiler to optimise and also more readable since it avoids confusion between the use of * for multiplication and dereferencing.

```
while (fabs (f) < 0.5)
{
  *e = *e - 1;
  f  *= 2;
}
```

is better written as,

```
{
  int p = *e;

  while (fabs(f) < 0.5)
    {
     p--;
     f *= 2;
    }

  *e = p;
}
```

3.30 Constness

Use `const` in function prototypes wherever an object pointed to by a pointer is constant (obviously). For variables which are meaningfully constant within a function/scope use `const` also. This prevents you from accidentally modifying a variable which should be constant (e.g. length of an array, etc). It can also help the compiler do optimization. These comments also apply to arguments passed by value which should be made `const` when that is meaningful.

3.31 Pseudo-templates

There are some pseudo-template macros available in '`templates_on.h`' and '`templates_off.h`'. See a directory link '`block`' for details on how to use them. Use sparingly, they are a bit of a nightmare, but unavoidable in places.

In particular, the convention is: templates are used for operations on "data" only (vectors, matrices, statistics, sorting). This is intended to cover the case where the program must interface with an external data-source which produces a fixed type. e.g. a big array of char's produced by an 8-bit counter.

All other functions can use double, for floating point, or the appropriate integer type for integers (e.g. unsigned long int for random numbers). It is not the intention to provide a fully templated version of the library.

That would be "putting a quart into a pint pot". To summarize, almost everything should be in a "natural type" which is appropriate for typical usage, and templates are there to handle a few cases where it is unavoidable that other data-types will be encountered.

For floating point work "double" is considered a "natural type". This sort of idea is a part of the C language.

3.32 Arbitrary Constants

Avoid arbitrary constants.

For example, don't hard code "small" values like '1e-30', '1e-100' or `10*GSL_DBL_EPSILON` into the routines. This is not appropriate for a general purpose library.

Compute values accurately using IEEE arithmetic. If errors are potentially significant then error terms should be estimated reliably and returned to the user, by analytically deriving an error propagation formula, not using guesswork.

A careful consideration of the algorithm usually shows that arbitrary constants are unnecessary, and represent an important parameter which should be accessible to the user.

For example, consider the following code:

```
if (residual < 1e-30) {
   return 0.0;  /* residual is zero within round-off error */
}
```

This should be rewritten as,

```
   return residual;
```

in order to allow the user to determine whether the residual is significant or not.

The only place where it is acceptable to use constants like `GSL_DBL_EPSILON` is in function approximations, (e.g. taylor series, asymptotic expansions, etc). In these cases it is not an arbitrary constant, but an inherent part of the algorithm.

3.33 Test suites

The implementor of each module should provide a reasonable test suite for the routines.

The test suite should be a program that uses the library and checks the result against known results, or invokes the library several times and does a statistical analysis on the results (for example in the case of random number generators).

Ideally the one test program per directory should aim for 100% path coverage of the code. Obviously it would be a lot of work to really achieve this, so prioritize testing on the critical parts and use inspection for the rest. Test all the error conditions by explicitly provoking them, because we consider it a serious defect if the function does not return an error for an invalid parameter. N.B. Don't bother to test for null pointers – it's sufficient for the library to segfault if the user provides an invalid pointer.

The tests should be deterministic. Use the `gsl_test` functions provided to perform separate tests for each feature with a separate output PASS/FAIL line, so that any failure can be uniquely identified.

Use realistic test cases with 'high entropy'. Tests on simple values such as 1 or 0 may not reveal bugs. For example, a test using a value of $x = 1$ will not pick up a missing factor of x in the code. Similarly, a test using a value of $x = 0$ will not pick any missing terms involving x in the code. Use values like 2.385 to avoid silent failures.

If your test uses multiple values make sure there are no simple relations between them that could allow bugs to be missed through silent cancellations.

If you need some random floats to put in the test programs use `od -f /dev/random` as a source of inspiration.

Don't use `sprintf` to create output strings in the tests. It can cause hard to find bugs in the test programs themselves. The functions `gsl_test_...` support format string arguments so use these instead.

3.34 Compilation

Make sure everything compiles cleanly. Use the strict compilation options for extra checking.

```
make CFLAGS="-ansi -pedantic -Werror -W -Wall -Wtraditional -Wconversion
  -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings
  -Wstrict-prototypes -fshort-enums -fno-common -Wmissing-prototypes
  -Wnested-externs -Dinline= -g -O4"
```

Also use `checkergcc` to check for memory problems on the stack and the heap. It's the best memory checking tool. If checkergcc isn't available then Electric Fence will check the heap, which is better than no checking.

There is a new tool `valgrind` for checking memory access. Test the code with this as well.

Make sure that the library will also compile with C++ compilers (g++). This should not be too much of a problem if you have been writing in ANSI C.

3.35 Thread-safety

The library should be usable in thread-safe programs. All the functions should be thread-safe, in the sense that they shouldn't use static variables.

We don't require everything to be completely thread safe, but anything that isn't should be obvious. For example, some global variables are used to control the overall behavior of the library (range-checking on/off, function to call on fatal error, etc). Since these are accessed directly by the user it is obvious to the multi-threaded programmer that they shouldn't be modified by different threads.

There is no need to provide any explicit support for threads (e.g. locking mechanisms etc), just to avoid anything which would make it impossible for someone to call a GSL routine from a multithreaded program.

3.36 Legal issues

- Each contributor must make sure her code is under the GNU General Public License (GPL). This means getting a disclaimer from your employer.

- We must clearly understand ownership of existing code and algorithms.
- Each contributor can retain ownership of their code, or sign it over to FSF as they prefer.

  There is a standard disclaimer in the GPL (take a look at it). The more specific you make your disclaimer the more likely it is that it will be accepted by an employer. For example,

  ```
  Yoyodyne, Inc., hereby disclaims all copyright interest in the software
  ‘GNU Scientific Library - Legendre Functions’ (routines for computing
  legendre functions numerically in C) written by James Hacker.

  <signature of Ty Coon>, 1 April 1989
  Ty Coon, President of Vice
  ```

- Obviously: don’t use or translate non-free code.

  In particular don’t copy or translate code from *Numerical Recipes* or *ACM TOMS*.

  Numerical Recipes is under a strict license and is not free software. The publishers Cambridge University Press claim copyright on all aspects of the book and the code, including function names, variable names and ordering of mathematical subexpressions. Routines in GSL should not refer to Numerical Recipes or be based on it in any way.

  The ACM algorithms published in TOMS (Transactions on Mathematical Software) are not public domain, even though they are distributed on the internet – the ACM uses a special non-commercial license which is not compatible with the GPL. The details of this license can be found on the cover page of ACM Transactions on Mathematical Software or on the ACM Website.

  Only use code which is explicitly under a free license: GPL or Public Domain. If there is no license on the code then this does not mean it is public domain – an explicit statement is required. If in doubt check with the author.

- I **think** one can reference algorithms from classic books on numerical analysis (BJG: yes, provided the code is an independent implementation and not copied from any existing software).

3.37 Non-UNIX portability

There is good reason to make this library work on non-UNIX systems. It is probably safe to ignore DOS and only worry about windows95/windowsNT portability (so filenames can be long, I think).

On the other hand, nobody should be forced to use non-UNIX systems for development.

The best solution is probably to issue guidelines for portability, like saying "don’t use XYZ unless you absolutely have to". Then the Windows people will be able to do their porting.

3.38 Compatibility with other libraries

We do not regard compatibility with other numerical libraries as a priority.

However, other libraries, such as Numerical Recipes, are widely used. If somebody writes the code to allow drop-in replacement of these libraries it would be useful to people. If it is done, it would be as a separate wrapper that can be maintained and shipped separately.

There is a separate issue of system libraries, such as BSD math library and functions like `expm1`, `log1p`, `hypot`. The functions in this library are available on nearly every platform (but not all).

In this case, it is best to write code in terms of these native functions to take advantage of the vendor-supplied system library (for example log1p is a machine instruction on the Intel x86). The library also provides portable implementations e.g. `gsl_hypot` which are used as an automatic fall back via autoconf when necessary. See the usage of `hypot` in '`gsl/complex/math.c`', the implementation of `gsl_hypot` and the corresponding parts of files '`configure.in`' and '`config.h.in`' as an example.

3.39 Parallelism

We don't intend to provide support for parallelism within the library itself. A parallel library would require a completely different design and would carry overhead that other applications do not need.

3.40 Precision

For algorithms which use cutoffs or other precision-related terms please express these in terms of GSL_DBL_EPSILON and GSL_DBL_MIN, or powers or combinations of these. This makes it easier to port the routines to different precisions.

3.41 Miscellaneous

Don't use the letter `l` as a variable name — it is difficult to distinguish from the number `1`. (This seems to be a favorite in old Fortran programs).

Final tip: one perfect routine is better than any number of routines containing errors.

Copying

The subroutines and source code in the *GNU Scientific Library* package are "free"; this means that everyone is free to use them and free to redistribute them on a free basis. The *GNU Scientific Library*-related programs are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these programs that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs that relate to *GNU Scientific Library*, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the *GNU Scientific Library*-related code, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to *GNU Scientific Library*. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to *GNU Scientific Library* are found in the General Public Licenses that accompany them.