Funtools: FITS Users Need Tools

Summary

This document is the Table of Contents for Funtools.

Description

Funtools, is a "minimal buy-in" FITS library and utility package developed at the the High Energy Astrophysics Division of SAO. The Funtools library provides simplified access to a wide array of file types: standard astronomical FITS images and binary tables, raw arrays and binary event lists, and even tables of ASCII column data. A sophisticated region filtering library (compatible with ds9) filters images and tables using boolean operations between geometric shapes, support world coordinates, etc. Funtools also supports advanced capabilities such as optimized data searching using index files. The main goal of the Funtools project has been to develop a minimal buy-in FITS library for researchers who are occasional (but serious) coders. In this case, "minimal buy-in" means "easy to learn, easy to use, and easy to re-learn next month". We have tried to achieve this goal by emphasizing two essential capabilities. The first is the ability to develop FITS programs without knowing much about FITS, i.e., without having to deal with the arcane rules for generating a properly formatted FITS file. The second is to support the use of already-familiar C/Unix facilities, especially C structs and Unix stdio. Taken together, these two capabilities should allow researchers to leverage their existing programming expertise while minimizing the need to learn new and complex coding rules.

Choose from the following topics:

- Funtools User Programs 1
  - funcalc: Funtools calculator (for binary tables) 1
  - funcen: find centroid (for binary tables) 8
  - funcnts: count photons in specified regions 9
  - funcone: cone search on RA, Dec columns 19
  - fundisp: display data in a Funtools data file 21
  - funhead: display a header in a Funtools file 28
  - funhist: create a 1D histogram of a column 30
  - funimage: create a FITS image from a Funtools data file 33
  - funindex: create an index on a column in a binary table 37
  - funjoin: join two or more FITS binary tables on specified columns 38
  - funmerge: merge one or more Funtools table files 41
  - funsky: convert between image and sky coordinates, using WCS info from a FITS header 42
  - funtable: copy selected rows from a Funtools file to a FITS binary table 45
  - funtbl: extract a table from Funtools ASCII output 48
  - funtools and ds9 image display 51
- Funtools Programming 53
  - Funtools Programming Summary . . . . . ??
  - Funtools Programming Tutorial 54

- A Short Digression on Subroutine Order . . . 56
- Compiling and Linking . . . 56
- The Funtools Reference Handle . . . 87
- The Funtools Programming Reference Manual . . . 57
  - FunOpen: open a Funtools file . . . 58
  - FunImageGet: retrieve image data . . . 60
  - FunImagePut: output image data . . . 62
  - FunImageRowGet: retrieve image data by row . . . 64
  - FunImageRowPut: output image data by row . . . 65
  - FunTableRowGet: retrieve rows from a table . . . 76
  - FunTableRowPut: output rows to a table . . . 77
  - FunColumnSelect: select columns in a table for access . . . 66
  - FunColumnActivate: activate columns in a table for read/write . . . 73
  - FunColumnLookup: lookup info about the columns in a table . . . 75
  - FunInfoGet: get info about an image or table . . . 82
  - FunInfoPut: put info about an image or table . . . 85
  - FunParamGet: get header param . . . 78
  - FunParamPut: put header param . . . 80
  - FunFlush: flush I/O in a Funtools file . . . 86
  - FunClose: close a Funtools file . . . 87
- Funtools Programming Examples . . . 57
  - evmerge: merge new columns with existing columns
  - evcols: add column and rows to binary tables
  - imblank: blank out image values below a threshold

- Funtools Data Files . . . 91
  - Supported Data Formats . . . 91
    - FITS File and Extensions . . . 92
    - Non-FITS Raw Event Files . . . 92
    - Non-FITS Array Files . . . 94
    - Column-based Text (ASCII) Files . . . 102
    - Database Views of Tables . . . 111
  - Image Sections and Blocking . . . 95
  - Binning FITS Binary Tables and Non-FITS Event Files
  - Disk Files and Other Supported File Types . . . 96
- Funtools Data Filtering
  - Table Filtering . . . 100
  - Fast Table Filtering using Indexes . . . 117
  - Spatial Region Filtering . . . 123
    - Region Geometry . . . 127
    - Region Algebra . . . 136
    - Region Coordinates . . . 152
    - Region Boundaries . . . 157
    - Differences Between Funtools and IRAF Regions . . . 161

- Combining Table and Region Filters 164
- Miscellaneous
  - Funtools Environment Variables 165
  - Funtools ChangeLog 167

Last updated: January 6, 2006

Funtools Programs

Summary

funcalc [-n] [-a argstr] [-e expr] [-f file] [-l link] [-p prog] [-u] <iname> [oname [columns]]

funcen [-i] [-n iter] [-t tol] [-v lev] <iname> <region>

funcnts [switches] <source_file> [source_region] [bkgd_file] [bkgd_region|bkgd_cnts]

funcone [-n] [-x|-X|-j|-J] [[-l|-L] list] [-r ra_col] [-d dec_col] <iname> <oname> <ra[hdr]> <dec[hdr]> <radius[dr'"]> [columns]

fundisp [-f format] [-l] [-n] [-T] <iname> [columns|bitpix=n]

funhead [-a] [-l] [-s] [-t] [-L] <iname> [oname ename]

funhist [-n|-w|-T] <iname> [column] [[lo_edge:hi_edge:]bins]

funimage [-a] [-l] [-p x|y] <iname> <oname> [bitpix=n]

funindex <iname> <key> [oname]

funjoin [switches] <ifile1> <ifile2> ... <ifilen> <ofile>

funmerge <iname1> <iname2> ... <oname>

funsky [switches] <iname1> [<lname2> <col1> <col2>]

funtable [-a] [-i|-z] [-m] [-s cols] <iname> <oname> [columns]

funtbl [-c cols] [-h] [-n table] [-p prog] [-s sep] [-T] <iname>

funcalc - Funtools calculator (for binary tables)

```
funcalc [-n] [-a argstr] [-e expr] [-f file] [-l link] [-p prog] <iname> [oname [columns]]

  -a argstr    # user arguments to pass to the compiled program
  -e expr      # funcalc expression
  -f file      # file containing funcalc expression
  -l libs      # libs to add to link command
  -n           # output generated code instead of compiling and executing
  -p prog      # generate named program, no execution
  -u           # die if any variable is undeclared (don't auto-declare)
```

funcalc is a calculator program that allows arbitrary expressions to be constructed, compiled, and executed on columns in a Funtools table (FITS binary table or raw event file). It works by integrating user-supplied expression(s) into a template C program, then compiling and executing the program. **funcalc** expressions are C statements, although some important simplifications (such as automatic declaration of variables) are supported.

funcalc expressions can be specified in three ways: on the command line using the **-e [expression]** switch, in a file using the **-f [file]** switch, or from stdin (if neither **-e** nor **-f** is specified). Of course a file containing **funcalc** expressions can be read from stdin.

Each invocation of **funcalc** requires an input Funtools table file to be specified as the first command line argument. The output Funtools table file is the second optional argument. It is needed only if an output FITS file is being created (i.e., in cases where the **funcalc** expression only prints values, no output file is needed). If input and output file are both specified, a third optional argument can specify the list of columns to activate (using FunColumnActivate()). Note that **funcalc** determines whether or not to generate code for writing an output file based on the presence or absence of an output file argument.

A **funcalc** expression executes on each row of a table and consists of one or more C statements that operate on the columns of that row (possibly using temporary variables). Within an expression, reference is made to a column of the **current** row using the C struct syntax **cur->[colname]**, e.g. cur->x, cur->pha, etc. Local scalar variables can be defined using C declarations at very the beginning of the expression, or else they can be defined automatically by **funcalc** (to be of type double). Thus, for example, a swap of columns x and y in a table can be performed using either of the following equivalent **funcalc** expressions:

```
double temp;
temp = cur->x;
cur->x = cur->y;
cur->y = temp;
```

or:

```
temp = cur->x;
cur->x = cur->y;
cur->y = temp;
```

When this expression is executed using a command such as:

```
funcalc -f swap.expr itest.ev otest.ev
```

the resulting file will have values of the x and y columns swapped.

By default, the data type of the variable for a column is the same as the data type of the column as stored in the file. This can be changed by appending ":[dtype]" to the first reference to that column. In the example above, to force x and y to be output as doubles, specify the type 'D' explicitly:

```
temp = cur->x:D;
cur->x = cur->y:D;
cur->y = temp;
```

Data type specifiers follow standard FITS table syntax for defining columns using TFORM:

- A: ASCII characters
- B: unsigned 8-bit char
- I: signed 16-bit int
- U: unsigned 16-bit int (not standard FITS)
- J: signed 32-bit int
- V: unsigned 32-bit int (not standard FITS)
- E: 32-bit float
- D: 64-bit float
- X: bits (treated as an array of chars)

Note that only the first reference to a column should contain the explicit data type specifier.

Of course, it is important to handle the data type of the columns correctly. One of the most frequent cause of error in **funcalc** programming is the implicit use of the wrong data type for a column in expression. For example, the calculation:

```
dx = (cur->x - cur->y)/(cur->x + cur->y);
```

usually needs to be performed using floating point arithmetic. In cases where the x and y columns are integers, this can be done by reading the columns as doubles using an explicit type specification:

```
dx = (cur->x:D - cur->y:D)/(cur->x + cur->y);
```

Alternatively, it can be done using C type-casting in the expression:

```
dx = ((double)cur->x - (double)cur->y)/((double)cur->x + (double)cur->y);
```

In addition to accessing columns in the current row, reference also can be made to the **previous** row using **prev->[colname]**, and to the **next** row using **next->[colname]**. Note that if **prev->[colname]** is specified in the **funcalc** expression, the very first row is not processed. If **next->[colname]** is specified in the **funcalc** expression, the very last row is not processed. In this way, **prev** and **next** are guaranteed always to point to valid rows. For example, to print out the values of the current x column and the previous y column, use the C fprintf function in a **funcalc** expression:

```
fprintf(stdout, "%d %d\n", cur->x, prev->y);
```

New columns can be specified using the same **cur->[colname]** syntax by appending the column type (and optional tlmin/tlmax/binsiz specifiers), separated by colons. For example, cur->avg:D will define a new column of type double. Type specifiers are the same those used above to specify new data types for existing columns.

For example, to create and output a new column that is the average value of the x and y columns, a new "avg" column can be defined:

```
cur->avg:D = (cur->x + cur->y)/2.0
```

Note that the final ';' is not required for single-line expressions.

As with FITS TFORM data type specification, the column data type specifier can be preceded by a numeric count to define an array, e.g., "10I" means a vector of 10 short ints, "2E" means two single precision floats, etc. A new column only needs to be defined once in a **funcalc** expression, after which it can be used without re-specifying the type. This includes reference to elements of a column array:

```
cur->avg[0]:2D = (cur->x + cur->y)/2.0;
cur->avg[1] = (cur->x - cur->y)/2.0;
```

The 'X' (bits) data type is treated as a char array of dimension (numeric_count/8), i.e., 16X is processed as a 2-byte char array. Each 8-bit array element is accessed separately:

```
cur->stat[0]:16X  = 1;
cur->stat[1]      = 2;
```

Here, a 16-bit column is created with the MSB is set to 1 and the LSB set to 2.

By default, all processed rows are written to the specified output file. If you want to skip writing certain rows, simply execute the C "continue" statement at the end of the **funcalc** expression, since the writing of the row is performed immediately after the expression is executed. For example, to skip writing rows

whose average is the same as the current x value:

```
cur->avg[0]:2D = (cur->x + cur->y)/2.0;
cur->avg[1] = (cur->x - cur->y)/2.0;
if( cur->avg[0] == cur->x )
  continue;
```

If no output file argument is specified on the **funcalc** command line, no output file is opened and no rows are written. This is useful in expressions that simply print output results instead of generating a new file:

```
fpv = (cur->av3:D-cur->av1:D)/(cur->av1+cur->av2:D+cur->av3);
fbv =  cur->av2/(cur->av1+cur->av2+cur->av3);
fpu = ((double)cur->au3-cur->au1)/((double)cur->au1+cur->au2+cur->au3);
fbu =  cur->au2/(double)(cur->au1+cur->au2+cur->au3);
fprintf(stdout, "%f\t%f\t%f\t%f\n", fpv, fbv, fpu, fbu);
```

In the above example, we use both explicit type specification (for "av" columns) and type casting (for "au" columns) to ensure that all operations are performed in double precision.

When an output file is specified, the selected input table is processed and output rows are copied to the output file. Note that the output file can be specified as "stdout" in order to write the output rows to the standard output. If the output file argument is passed, an optional third argument also can be passed to specify which columns to process.

In a FITS binary table, it sometimes is desirable to copy all of the other FITS extensions to the output file as well. This can be done by appending a '+' sign to the name of the extension in the input file name. See **funtable** for a related example.

funcalc works by integrating the user-specified expression into a template C program called tabcalc.c. The completed program then is compiled and executed. Variable declarations that begin the **funcalc** expression are placed in the local declaration section of the template main program. All other lines are placed in the template main program's inner processing loop. Other details of program generation are handled automatically. For example, column specifiers are analyzed to build a C struct for processing rows, which is passed to FunColumnSelect() and used in FunTableRowGet(). If an unknown variable is used in the expression, resulting in a compilation error, the program build is retried after defining the unknown variable to be of type double.

Normally, **funcalc** expression code is added to **funcalc** row processing loop. It is possible to add code to other parts of the program by placing this code inside special directives of the form:

```
[directive name]
  ... code goes here ...
end
```

The directives are:

- **global** add code and declarations in global space, before the main routine.
- **local** add declarations (and code) just after the local declarations in main
- **before** add code just before entering the main row processing loop
- **after** add code just after exiting the main row processing loop

Thus, the following **funcalc** expression will declare global variables and make subroutine calls just before and just after the main processing loop:

```
global
  double v1, v2;
  double init(void);
  double finish(double v);
end
before
  v1  = init();
end
... process rows, with calculations using v1 ...
after
  v2 = finish(v1);
  if( v2 < 0.0 ){
    fprintf(stderr, "processing failed %g -> %g\n", v1, v2);
    exit(1);
  }
end
```

Routines such as init() and finish() above are passed to the generated program for linking using the **-l [link directives ...]** switch. The string specified by this switch will be added to the link line used to build the program (before the funtools library). For example, assuming that init() and finish() are in the library libmysubs.a in the /opt/special/lib directory, use:

```
funcalc  -l "-L/opt/special/lib -lmysubs" ...
```

User arguments can be passed to a compiled funcalc program using a string argument to the "-a" switch. The string should contain all of the user arguments. For example, to pass the integers 1 and 2, use:

```
funcalc -a "1 2" ...
```

The arguments are stored in an internal array and are accessed as strings via the ARGV(n) macro. For example, consider the following expression:

```
local
  int pmin, pmax;
end

before
  pmin=atoi(ARGV(0));
  pmax=atoi(ARGV(1));
end

if( (cur->pha >= pmin) && (cur->pha <= pmax) )
  fprintf(stderr, "%d %d %d\n", cur->x, cur->y, cur->pha);
```

This expression will print out x, y, and pha values for all rows in which the pha value is between the two user-input values:

```
funcalc -a '1 12' -f foo snr.ev'[cir 512 512 .1]'
512 512 6
512 512 8
512 512 5
512 512 5
512 512 8

funcalc -a '5 6' -f foo snr.ev'[cir 512 512 .1]'
512 512 6
512 512 5
512 512 5
```

Note that it is the user's responsibility to ensure that the correct number of arguments are passed. The ARGV(n) macro returns a NULL if a requested argument is outside the limits of the actual number of args, usually resulting in a SEGV if processed blindly. To check the argument count, use the ARGC macro:

```
local
  long int seed=1;
  double limit=0.8;
end

before
  if( ARGC >= 1 ) seed = atol(ARGV(0));
  if( ARGC >= 2 ) limit = atof(ARGV(1));
  srand48(seed);
end

if ( drand48() > limit ) continue;
```

The macro WRITE_ROW expands to the FunTableRowPut() call that writes the current row. It can be used to write the row more than once. In addition, the macro NROW expands to the row number currently being processed. Use of these two macros is shown in the following example:

```
if( cur->pha:I == cur->pi:I ) continue;
a = cur->pha;
cur->pha = cur->pi;
cur->pi = a;
cur->AVG:E  = (cur->pha+cur->pi)/2.0;
cur->NR:I = NROW;
if( NROW < 10 ) WRITE_ROW;
```

If the **-p [prog]** switch is specified, the expression is not executed. Rather, the generated executable is saved with the specified program name for later use.

If the **-n** switch is specified, the expression is not executed. Rather, the generated code is written to stdout. This is especially useful if you want to generate a skeleton file and add your own code, or if you need to check compilation errors. Note that the comment at the start of the output gives the compiler command needed to build the program on that platform. (The command can change from platform to platform because of the use of different libraries, compiler switches, etc.)

As mentioned previously, **funcalc** will declare a scalar variable automatically (as a double) if that variable has been used but not declared. This facility is implemented using a sed script named funcalc.sed, which processes the compiler output to sense an undeclared variable error. This script has been seeded with the appropriate error information for gcc, and for cc on Solaris, DecAlpha, and SGI platforms. If you find that automatic declaration of scalars is not working on your platform, check this sed script; it might be necessary to add to or edit some of the error messages it senses.

In order to keep the lexical analysis of **funcalc** expressions (reasonably) simple, we chose to accept some limitations on how accurately C comments, spaces, and new-lines are placed in the generated program. In particular, comments associated with local variables declared at the beginning of an expression (i.e., not in a **local...end** block) will usually end up in the inner loop, not with the local declarations:

```
/* this comment will end up in the wrong place (i.e, inner loop) */
double a; /* also in wrong place */
/* this will be in the the right place (inner loop) */
if( cur->x:D == cur->y:D ) continue; /* also in right place */
a = cur->x;
cur->x = cur->y;
cur->y = a;
cur->avg:E  = (cur->x+cur->y)/2.0;
```

Similarly, spaces and new-lines sometimes are omitted or added in a seemingly arbitrary manner. Of course, none of these stylistic blemishes affect the correctness of the generated code.

Because **funcalc** must analyze the user expression using the data file(s) passed on the command line, the input file(s) must be opened and read twice: once during program generation and once during execution. As a result, it is not possible to use stdin for the input file: **funcalc** cannot be used as a filter. We will consider removing this restriction at a later time.

Along with C comments, **funcalc** expressions can have one-line internal comments that are not passed on to the generated C program. These internal comment start with the **#** character and continue up to the new-line:

```
double a; # this is not passed to the generated C file
# nor is this
a = cur->x;
cur->x = cur->y;
cur->y = a;
/* this comment is passed to the C file */
cur->avg:E  = (cur->x+cur->y)/2.0;
```

As previously mentioned, input columns normally are identified by their being used within the inner event loop. There are rare cases where you might want to read a column and process it outside the main loop. For example, qsort might use a column in its sort comparison routine that is not processed inside the inner loop (and therefore not implicitly specified as a column to be read). To ensure that such a column is read by the event loop, use the **explicit** keyword. The arguments to this keyword specify columns that should be read into the input record structure even though they are not mentioned in the inner loop. For example:

```
  explicit pi pha
```

will ensure that the pi and pha columns are read for each row, even if they are not processed in the inner event loop. The **explicit** statement can be placed anywhere.

Finally, note that **funcalc** currently works on expressions involving FITS binary tables and raw event files. We will consider adding support for image expressions at a later point, if there is demand for such support from the community.

funcen - find centroid (for binary tables)

```
funcen [-i] [-n iter] [-t tol] [-v lev] <iname> <region>

  -i            # use image filtering (default: event filtering)
  -n iter       # max number of iterations (default: 0)
  -t tol        # pixel tolerance distance (default: 1.0)
  -v [0,1,2,3]  # output verbosity level (default: 0)
```

funcen iteratively calculates the centroid position within one or more regions of a Funtools table (FITS binary table or raw event file). Starting with an input table, an initial region specification, and an iteration count, the program calculates the average x and y position within the region and then uses this new position as the region center for the next iteration. Iteration terminates when the maximum number of iterations is reached or when the input tolerance distance is met for that region. A count of events in the final region is then output, along with the pixel position value (and, where available, WCS position).

The first argument to the program specifies the Funtools table file to process. Since the file must be read repeatedly, a value of "stdin" is not permitted when the number of iterations is non-zero. Use Funtools Bracket Notation to specify FITS extensions and filters.

The second required argument is the initial region descriptor. Multiple regions are permitted. However, compound regions (accelerators, variable argument regions and regions connected via boolean algebra) are not permitted. Points and polygons also are illegal. These restrictions might be lifted in a future version, if warranted.

The **-n** (iteration number) switch specifies the maximum number of iterations to perform. The default is 0, which means that the program will simply count and display the number of events in the initial region(s). Note that when iterations is 0, the data can be input via stdin.

The **-t** (tolerance) switch specifies a floating point tolerance value. If the distance between the current centroid position value and the last position values is less than this value, iteration terminates. The default value is 1 pixel.

The **-v** (verbosity) switch specifies the verbosity level of the output. The default is 0, which results in a single line of output for each input region consisting of the following values:

```
  counts x y [ra dec coordsys]
```

The last 3 WCS values are output if WCS information is available in the data file header. Thus, for example:

```
[sh] funcen -n 0 snr.ev "cir 505 508 5"
915 505.00 508.00 345.284038 58.870920 j2000

[sh] funcen -n 3 snr.ev "cir 505 508 5"
1120 504.43 509.65 345.286480 58.874587 j2000
```

The first example simply counts the number of events in the initial region. The second example iterates the centroid calculation three times to determine a final "best" position.

Higher levels of verbosity obviously imply more verbose output. At level 1, the output essentially contains the same information as level 0, but with keyword formatting:

```
[sh] funcen -v 1 -n 3 snr.ev "cir 505 508 5"
event_file:     snr.ev
initial_region: cir 505 508 5
tolerance:      1.0000
iterations:     1
events:         1120
x,y(physical):  504.43 509.65
ra,dec(j2000):  345.286480 58.874587
final_region1:  cir 504.43 509.65 5
```

Level 2 outputs results from intermediate calculations as well.

Ordinarily, region filtering is performed using analytic (event) filtering, i.e. that same style of filtering as is performed by **fundisp** and **funtable**. Use the **-i** switch to specify image filtering, i.e. the same style filtering as is performed by **funcnts**. Thus, you can perform a quick calculation of counts in regions, using either the analytic or image filtering method, by specifying the **-n 0** and optional **-i** switches. These two method often give different results because of how boundary events are processed:

```
[sh] funcen  snr.ev "cir 505 508 5"
915 505.00 508.00 345.284038 58.870920 j2000

[sh] funcen -i snr.ev "cir 505 508 5"
798 505.00 508.00 345.284038 58.870920 j2000
```

See Region Boundaries for more information about how boundaries are calculated using these two methods.

funcnts - count photons in specified regions, with bkgd subtraction

```
funcnts  [switches] <source_file> [source_region] [bkgd_file] [bkgd_region|bkgd_value]

  -e "source_exposure[;bkgd_exposure]"
               # source (bkgd) FITS exposure image using matching files
  -w "source_exposure[;bkgd_exposure]"
               # source (bkgd) FITS exposure image using WCS transform
  -t "source_timecorr[;bkgd_timecorr]"
               # source (bkgd) time correction value or header parameter name
  -g           # output using nice g format
  -G           # output using %.14g format (maximum precision)
  -i "[column;]int1;int2..." # column-based intervals
```

```
-m            # match individual source and bkgd regions
-p            # output in pixels, even if wcs is present
-r            # output inner/outer radii (and angles) for annuli (and pandas)
-s            # output summed values
-v "scol[;bcol]" # src and bkgd value columns for tables
-T            # output in starbase/rdb format
-z            # output regions with zero area
```

funcnts counts photons in the specified source regions and reports the results for each region. Regions are specified using the Spatial Region Filtering mechanism. Photons are also counted in the specified bkgd regions applied to the same data file or a different data file. (Alternatively, a constant background value in counts/pixel**2 can be specified.) The bkgd regions are either paired one-to-one with source regions or pooled and normalized by area, and then subtracted from the source counts in each region. Displayed results include the bkgd-subtracted counts in each region, as well as the error on the counts, the area in each region, and the surface brightness (cnts/area**2) calculated for each region.

The first argument to the program specifies the FITS input image, array, or raw event file to process. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions, image sections, and filters.

The optional second argument is the source region descriptor. If no region is specified, the entire field is used.

The background arguments can take one of two forms, depending on whether a separate background file is specified. If the source file is to be used for background as well, the third argument can be either the background region, or a constant value denoting background cnts/pixel. Alternatively, the third argument can be a background data file, in which case the fourth argument is the background region. If no third argument is specified, a constant value of 0 is used (i.e., no background).

In summary, the following command arguments are valid:

```
[sh] funcnts sfile                        # counts in source file
[sh] funcnts sfile sregion                # counts in source region
[sh] funcnts sfile sregion bregion        # bkgd reg. is from source file
[sh] funcnts sfile sregion bvalue         # bkgd reg. is constant
[sh] funcnts sfile sregion bfile bregion  # bkgd reg. is from separate file
```

NB: unlike other Funtools programs, source and background regions are specified as separate arguments on the command line, rather than being placed inside brackets as part of the source and background filenames. This is because regions in funcnts are not simply used as data filters, but also are used to calculate areas, exposure, etc. If you put the source region inside the brackets (i.e. use it simply as a filter) rather than specifying it as argument two, the program still will only count photons that pass the region filter. However, the area calculation will be performed on the whole field, since field() is the default source region. This rarely is the desired behavior. On the other hand, with FITS binary tables, it often is useful to put a column filter in the filename brackets, so that only events matching the column filter are counted inside the region.

For example, to extract the counts within a radius of 22 pixels from the center of the FITS binary table snr.ev and subtract the background determined from the same image within an annulus of radii 50-100 pixels:

```
[sh] funcnts snr.ev "circle(502,512,22)" "annulus(502,512,50,100)"
# source
#   data file:        snr.ev
#   degrees/pix:      0.00222222
# background
#   data file:        snr.ev
# column units
#   area:             arcsec**2
#   surf_bri:         cnts/arcsec**2
#   surf_err:         cnts/arcsec**2

# background-subtracted results
 reg   net_counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1     3826.403    66.465      555.597     5.972  96831.98     0.040     0.001


# the following source and background components were used:
source region(s)
----------------
circle(502,512,22)

 reg       counts    pixels
---- ------------ ---------
   1     4382.000      1513

background region(s)
--------------------
annulus(502,512,50,100)

 reg       counts    pixels
---- ------------ ---------
all      8656.000     23572
```

The area units for the output columns labeled "area", "surf_bri" (surface brightness) and "surf_err" will be given either in arc-seconds (if appropriate WCS information is in the data file header(s)) or in pixels. If the data file has WCS info, but you do not want arc-second units, use the **-p** switch to force output in pixels. Also, regions having zero area are not normally included in the primary (background-subtracted) table, but are included in the secondary source and bkgd tables. If you want these regions to be included in the primary table, use the **-z** switch.

Note that a simple sed command will extract the background-subtracted results for further analysis:

```
[sh] cat funcnts.sed
1,/---- .*/d
/^$/,$d

[sh] sed -f funcnts.sed funcnts.out
1     3826.403    66.465      555.597     5.972  96831.98     0.040     0.001
```

If separate source and background files are specified, **funcnts** will attempt to normalize the the background area so that the background pixel size is the same as the source pixel size. This normalization can only take place if the appropriate WCS information is contained in both files (e.g. degrees/pixel values in CDELT). If either file does not contain the requisite size information, the normalization is not

performed. In this case, it is the user's responsibility to ensure that the pixel sizes are the same for the two files.

Normally, if more than one background region is specified, **funcnts** will combine them all into a single region and use this background region to produce the background-subtracted results for each source region. The **-m** (match multiple backgrounds) switch tells **funcnts** to make a one to one correspondence between background and source regions, instead of using a single combined background region. For example, the default case is to combine 2 background regions into a single region and then apply that region to each of the source regions:

```
[sh] funcnts snr.ev "annulus(502,512,0,22,n=2)" "annulus(502,512,50,100,n=2)"
# source
#   data file:        snr.ev
#   degrees/pix:      0.00222222
# background
#   data file:        snr.ev
# column units
#   area:             arcsec**2
#   surf_bri:         cnts/arcsec**2
#   surf_err:         cnts/arcsec**2

# background-subtracted results
 reg   net_counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1     3101.029    56.922      136.971     1.472  23872.00     0.130     0.002
   2      725.375    34.121      418.625     4.500  72959.99     0.010     0.000


# the following source and background components were used:
source region(s)
----------------
annulus(502,512,0,22,n=2)

 reg       counts    pixels
---- ------------ ---------
   1     3238.000       373
   2     1144.000      1140

background region(s)
--------------------
annulus(502,512,50,100,n=2)

 reg       counts    pixels
---- ------------ ---------
all      8656.000     23572
```

Note that the basic region filter rule "each photon is counted once and no photon is counted more than once" still applies when using The **-m** to match background regions. That is, if two background regions overlap, the overlapping pixels will be counted in only one of them. In a worst-case scenario, if two background regions are the same region, the first will get all the counts and area and the second will get none.

Using the **-m** switch causes **funcnts** to use each of the two background regions independently with each of the two source regions:

```
[sh] funcnts -m snr.ev "annulus(502,512,0,22,n=2)" "ann(502,512,50,100,n=2)"
# source
#   data file:        snr.ev
#   degrees/pix:      0.00222222
# background
#   data file:        snr.ev
# column units
#   area:             arcsec**2
#   surf_bri:         cnts/arcsec**2
#   surf_err:         cnts/arcsec**2

# background-subtracted results
 reg   net_counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1     3087.015    56.954      150.985     2.395  23872.00     0.129     0.002
   2      755.959    34.295      388.041     5.672  72959.99     0.010     0.000


# the following source and background components were used:
source region(s)
----------------
annulus(502,512,0,22,n=2)

 reg       counts    pixels
---- ------------ ---------
   1     3238.000       373
   2     1144.000      1140

background region(s)
--------------------
ann(502,512,50,100,n=2)

 reg       counts    pixels
---- ------------ ---------
   1     3975.000      9820
   2     4681.000     13752
```

Note that most floating point quantities are displayed using "f" format. You can change this to "g" format using the **-g** switch. This can be useful when the counts in each pixel is very small or very large. If you want maximum precision and don't care about the columns lining up nicely, use **-G**, which outputs all floating values as %.14g.

When counting photons using the annulus and panda (pie and annuli) shapes, it often is useful to have access to the radii (and panda angles) for each separate region. The **-r** switch will add radii and angle columns to the output table:

```
[sh] funcnts -r snr.ev "annulus(502,512,0,22,n=2)" "ann(502,512,50,100,n=2)"
# source
#   data file:        snr.ev
#   degrees/pix:      0.00222222
# background
#   data file:        snr.ev
# column units
#   area:             arcsec**2
```

```
#   surf_bri:         cnts/arcsec**2
#   surf_err:         cnts/arcsec**2
#   radii:            arcsecs
#   angles:           degrees

# background-subtracted results
 reg   net_counts     error   background    berror      area  surf_bri  surf_err   radius1   radius2    angle1    angle2
---- ------------ --------- ------------ --------- --------- --------- --------- --------- --------- --------- ---------
   1     3101.029    56.922      136.971     1.472  23872.00     0.130     0.002      0.00     88.00        NA        NA
   2      725.375    34.121      418.625     4.500  72959.99     0.010     0.000     88.00    176.00        NA        NA


# the following source and background components were used:
source region(s)
----------------
annulus(502,512,0,22,n=2)

 reg       counts    pixels
---- ------------ ---------
   1     3238.000       373
   2     1144.000      1140

background region(s)
--------------------
ann(502,512,50,100,n=2)

 reg       counts    pixels
---- ------------ ---------
all      8656.000     23572
```

Radii are given in units of pixels or arc-seconds (depending on the presence of WCS info), while the angle values (when present) are in degrees. These columns can be used to plot radial profiles. For example, the script **funcnts.plot** in the funtools distribution) will plot a radial profile using gnuplot (version 3.7 or above). A simplified version of this script is shown below:

```
#!/bin/sh

if [ x"$1" = xgnuplot ]; then
  if [ x`which gnuplot 2>/dev/null` = x ]; then
    echo "ERROR: gnuplot not available"
    exit 1
  fi
  awk '
  BEGIN{HEADER=1; DATA=0; FILES=""; XLABEL="unknown"; YLABEL="unknown"}
  HEADER==1{
    if( $1 == "#" && $2 == "data" && $3 == "file:" ){
      if( FILES != "" ) FILES = FILES ","
      FILES = FILES $4
    }
    else if( $1 == "#" && $2 == "radii:" ){
      XLABEL = $3
    }
    else if( $1 == "#" && $2 == "surf_bri:" ){
      YLABEL = $3
    }
    else if( $1 == "----" ){
      printf "set nokey; set title \"funcnts(%s)\"\n", FILES
      printf "set xlabel \" radius(%s)\"\n", XLABEL
      printf "set ylabel \"surf_bri(%s)\"\n", YLABEL
      print  "plot \"-\" using 3:4:6:7:8 with boxerrorbars"
      HEADER = 0
      DATA = 1
      next
    }
```

```
  }
  DATA==1{
    if( NF == 12 ){
      print $9, $10, ($9+$10)/2, $7, $8, $7-$8, $7+$8, $10-$9
    }
    else{
      exit
    }
  }
  ' | gnuplot -persist - 1>/dev/null 2>&1

elif [ x"$1" = xds9 ]; then
  awk '
  BEGIN{HEADER=1; DATA=0; XLABEL="unknown"; YLABEL="unknown"}
  HEADER==1{
    if( $1 == "#" && $2 == "data" && $3 == "file:" ){
      if( FILES != "" ) FILES = FILES ","
      FILES = FILES $4
    }
    else if( $1 == "#" && $2 == "radii:" ){
      XLABEL = $3
    }
    else if( $1 == "#" && $2 == "surf_bri:" ){
      YLABEL = $3
    }
    else if( $1 == "----" ){
      printf "funcnts(%s) radius(%s) surf_bri(%s) 3\n", FILES, XLABEL, YLABEL
      HEADER = 0
      DATA = 1
      next
    }
  }
  DATA==1{
    if( NF == 12 ){
      print $9, $7, $8
    }
    else{
      exit
    }
  }
  '
else
  echo "funcnts -r ... | funcnts.plot [ds9|gnuplot]"
  exit 1
fi
```

Thus, to run **funcnts** and plot the results using gnuplot (version 3.7 or above), use:

```
funcnts -r snr.ev "annulus(502,512,0,50,n=5)" ...  | funcnts.plot gnuplot
```

The **-s** (sum) switch causes **funcnts** to produce an additional table of summed (integrated) background subtracted values, along with the default table of individual values:

```
[sh] funcnts -s snr.ev "annulus(502,512,0,50,n=5)" "annulus(502,512,50,100)"
# source
#   data file:        snr.ev
#   degrees/pix:      0.00222222
# background
#   data file:        snr.ev
# column units
#   area:             arcsec**2
#   surf_bri:         cnts/arcsec**2
#   surf_err:         cnts/arcsec**2

# summed background-subtracted results
upto   net_counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1     2880.999    54.722      112.001     1.204  19520.00     0.148     0.003
   2     3776.817    65.254      457.183     4.914  79679.98     0.047     0.001
   3     4025.492    71.972     1031.508    11.087 179775.96     0.022     0.000
   4     4185.149    80.109     1840.851    19.786 320831.94     0.013     0.000
   5     4415.540    90.790     2873.460    30.885 500799.90     0.009     0.000


# background-subtracted results
 reg       counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1     2880.999    54.722      112.001     1.204  19520.00     0.148     0.003
   2      895.818    35.423      345.182     3.710  60159.99     0.015     0.001
   3      248.675    29.345      574.325     6.173 100095.98     0.002     0.000
   4      159.657    32.321      809.343     8.699 141055.97     0.001     0.000
   5      230.390    37.231     1032.610    11.099 179967.96     0.001     0.000


# the following source and background components were used:
source region(s)
----------------
annulus(502,512,0,50,n=5)

 reg       counts    pixels      sumcnts    sumpix
---- ------------ --------- ------------ ---------
   1     2993.000       305     2993.000       305
   2     1241.000       940     4234.000      1245
   3      823.000      1564     5057.000      2809
   4      969.000      2204     6026.000      5013
   5     1263.000      2812     7289.000      7825

background region(s)
--------------------
annulus(502,512,50,100)

 reg       counts    pixels
---- ------------ ---------
all      8656.000     23572
```

The **-t** and **-e** switches can be used to apply timing and exposure corrections, respectively, to the data. Please note that these corrections are meant to be used qualitatively, since application of more accurate correction factors is a complex and mission-dependent effort. The algorithm for applying these simple corrections is as follows:

```
C =  Raw Counts in Source Region
Ac=  Area of Source Region
Tc=  Exposure time for Source Data
Ec=  Average exposure in Source Region, from exposure map

B=   Raw Counts in Background Region
Ab=  Area of Background Region
Tb=  (Exposure) time for Background Data
Eb=  Average exposure in Background Region, from exposure map
```

Then, Net Counts in Source region is

```
Net=  C - B * (Ac*Tc*Ec)/(Ab*Tb*Eb)
```

with the standard propagation of errors for the Error on Net. The net rate would then be

```
Net Rate = Net/(Ac*Tc*Ec)
```

The average exposure in each region is calculated by summing up the pixel values in the exposure map for the given region and then dividing by the number of pixels in that region. Exposure maps often are generated at a block factor > 1 (e.g., block 4 means that each exposure pixel contains 4x4 pixels at full resolution) and **funcnts** will deal with the blocking automatically. Using the **-e** switch, you can supply both source and background exposure files (separated by ";"), if you have separate source and background data files. If you do not supply a background exposure file to go with a separate background data file, **funcnts** assumes that exposure already has been applied to the background data file. In addition, it assumes that the error on the pixels in the background data file is zero.

NB: The **-e** switch assumes that the exposure map overlays the image file **exactly**, except for the block factor. Each pixel in the image is scaled by the block factor to access the corresponding pixel in the exposure map. If your exposure map does not line up exactly with the image, **do not use** the **-e** exposure correction. In this case, it still is possible to perform exposure correction **if** both the image and the exposure map have valid WCS information: use the **-w** switch so that the transformation from image pixel to exposure pixel uses the WCS information. That is, each pixel in the image region will be transformed first from image coordinates to sky coordinates, then from sky coordinates to exposure coordinates. Please note that using **-w** can increase the time required to process the exposure correction considerably.

A time correction can be applied to both source and background data using the **-t** switch. The value for the correction can either be a numeric constant or the name of a header parameter in the source (or background) file:

```
[sh] funcnts -t 23.4 ...            # number for source
[sh] funcnts -t "LIVETIME;23.4" ... # param for source, numeric for bkgd
```

When a time correction is specified, it is applied to the net counts as well (see algorithm above), so that the units of surface brightness become cnts/area**2/sec.

The **-i** (interval) switch is used to run **funcnts** on multiple column-based intervals with only a single pass through the data. It is equivalent to running **funcnts** several times with a different column filter added to the source and background data each time. For each interval, the full **funcnts** output is generated, with a linefeed character (^L) inserted between each run. In addition, the output for each interval will contain the interval specification in its header. Intervals are very useful for generating X-ray hardness ratios

efficiently. Of course, they are only supported when the input data are contained in a table.

Two formats are supported for interval specification. The most general format is semi-colon-delimited list of filters to be used as intervals:

```
funcnts -i "pha=1:5;pha=6:10;pha=11:15" snr.ev "circle(502,512,22)" ...
```

Conceptually, this will be equivalent to running **funcnts** three times:

```
funcnts snr.ev'[pha=1:5]' "circle(502,512,22)"
funcnts snr.ev'[pha=6:10]' "circle(502,512,22)"
funcnts snr.ev'[pha=11:15]' "circle(502,512,22)"
```

However, using the **-i** switch will require only one pass through the data.

Note that complex filters can be used to specify intervals:

```
funcnts -i "pha=1:5&π=4;pha=6:10&π=5;pha=11:15&π=6" snr.ev ...
```

The program simply runs the data through each filter in turn and generates three **funcnts** outputs, separated by the line-feed character.

In fact, although the intent is to support intervals for hardness ratios, the specified filters do not have to be intervals at all. Nor does one "interval" filter have to be related to another. For example:

```
funcnts -i "pha=1:5;pi=6:10;energy=11:15" snr.ev "circle(502,512,22)" ...
```

is equivalent to running **funcnts** three times with unrelated filter specifications.

A second interval format is supported for the simple case in which a single column is used to specify multiple homogeneous intervals for that column. In this format, a column name is specified first, followed by intervals:

```
funcnts -i "pha;1:5;6:10;11:15" snr.ev "circle(502,512,22)" ...
```

This is equivalent to the first example, but requires less typing. The **funcnts** program will simply prepend "pha=" before each of the specified intervals. (Note that this format does not contain the "=" character in the column argument.)

Ordinarily, when **funcnts** is run on a FITS binary table (or a raw event table), one integral count is accumulated for each row (event) contained within a given region. The **-v "scol[;bcol]"** (value column) switch will accumulate counts using the value from the specified column for the given event. If only a single column is specified, it is used for both the source and background regions. Two separate columns, separated by a semi-colon, can be specified for source and background. The special token '$none' can be used to specify that a value column is to be used for one but not the other. For example, 'pha;$none' will use the pha column for the source but use integral counts for the background, while '$none;pha' will do the converse. If the value column is of type logical, then the value used will be 1 for T and 0 for F. Value columns are used, for example, to integrate probabilities instead of integral counts.

If the **-T** (rdb table) switch is used, the output will conform to starbase/rdb data base format: tabs will be inserted between columns rather than spaces and line-feed will be inserted between tables.

Finally, note that **funcnts** is an image program, even though it can be run directly on FITS binary tables. This means that image filtering is applied to the rows in order to ensure that the same results are obtained regardless of whether a table or the equivalent binned image is used. Because of this, however, the number of counts found using **funcnts** can differ from the number of events found using row-filter programs such as **fundisp** or **funtable** For more information about these difference, see the discussion of Region Boundaries.

funcone - cone search of a binary table containing RA, Dec columns

```
funcone <switches>  <iname> <oname> <ra[hdr]> <dec[hdr]> <radius[dr'"]> [columns]

  -d deccol:[hdr]  # Dec column name, units (def: DEC:d)
  -j               # join columns from list file
  -J               # join columns from list file, output all rows
  -l listfile      # read centers and radii from a list
  -L listfile      # read centers and radii from a list, output list rows
  -n               # don't use cone limits as a filter
  -r  racol:[hdr]  # RA column name, units (def: RA:h)
  -x               # append RA_CEN, DEC_CEN, RAD_CEN, CONE_KEY cols
  -X               # append RA_CEN, DEC_CEN, RAD_CEN, CONE_KEY cols, output all rows
```

Funcone performs a cone search on the RA and Dec columns of a FITS binary table. The distance from the center RA, Dec position to the RA, Dec in each row in the table is calculated. Rows whose distance is less than the specified radius are output.

The first argument to the program specifies the FITS file, raw event file, or raw array file. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions, and filters. The second argument is the output FITS file. If "stdout" is specified, the FITS binary table is written to the standard output.

The third and fourth required arguments are the RA and Dec center position. By default, RA is specified in hours while Dec is specified in degrees. You can change the units of either of these by appending the character "d" (degrees), "h" (hours) or "r" (radians). Sexagesimal notation is supported, with colons or spaces separating hms and dms. (When using spaces, please ensure that the entire string is quoted.)

The fifth required argument is the radius of the cone search. By default, the radius value is given in degrees. The units can be changed by appending the character "d" (degrees), "r" (radians), "'" (arc minutes) or '"' (arc seconds).

By default, all columns of the input file are copied to the output file. Selected columns can be output using an optional sixth argument in the form:

```
  "column1 column1 ... columnN"
```

A seventh argument allows you to output selected columns from the list file when **-j** switch is used. Note

that the RA and Dec columns used in the cone calculation must not be de-selected.

Also by default, the RA and Dec column names are named "RA" and "Dec", and are given in units of hours and degrees respectively. You can change both the name and the units using the -r [RA] and/or -d [Dec] switches. Once again, one of "h", "d", or "r" is appended to the column name to specify units but in this case, there must be a colon ":" between the name and the unit specification.

If the **-l [listfile]** switch is used, then one or more of the center RA, center Dec, and radius can be taken from a list file (which can be a FITS table or an ASCII column text file). In this case, the third (center RA), fourth (center Dec), and fifth (radius) command line arguments can either be a column name in the list file (if that parameter varies) or else a numeric value (if that parameter is static). When a column name is specified for the RA, Dec, or radius, you can append a colon followed by "h", "d", or "r" to specify units (also ’ and " for radius). The cone search algorithm is run once for each row in the list, taking RA, Dec, and radius values from the specified columns or from static numeric values specified on the command line.

When using a list, all valid rows from each iteration are written to a single output file. Use the **-x** switch to help delineate which line of the list file was used to produce the given output row(s). This switch causes the values for the center RA, Dec, radius, and row number to be appended to the output file, in columns called RA_CEN, DEC_CEN, RAD_CEN and CONE_KEY, respectively. Alternatively, the **-j** (join) switch will append all columns from the list row to the output row (essentially a join of the list row and input row), along with the CONE_KEY row number. These two switches are mutually exclusive.

The **-X** and **-J** switches write out the same data as their lower case counterparts for each row satisfying a cone search. In addition, these switches also write out rows from the event file that do not satisfy any cone search. In such cases, that CONE_KEY column will be given a value of -1 and the center and list position information will be set to zero for the given row. Thus, all rows of the input event file are guaranteed to be output, with rows satisfying at least one cone search having additional search information.

The **-L** switch acts similarly to the **-l** switch in that it takes centers from a list file. However, it also implicitly sets the -j switch, so that output rows are the join of the input event row and the center position row. In addition, this switch also writes out all center position rows for which no event satisfies the cone search criteria of that row. The CONE_KEY column will be given a value of -2 for center rows that were not close to any data row and the event columns will be zeroed out for such rows. In this way, all centers rows are guaranteed to be output at least once.

If any of "all row" switches (**-X**, **-J**, or **-L**) are specified, then a new column named JSTAT is added to the output table. The positive values in this column indicate the center position row number (starting from 1) in the list file that this data row successful matched in a cone search. A value of -1 means that the data row did not match any center position. A value of -2 means that the center position was not matched by any data row.

Given a center position and radius, the cone search algorithm calculates limit parameters for a box enclosing the specified cone, and only tests rows whose positions values lie within those limits. For small files, the overhead associated with this cone limit filtering can cause the program to run more slowly than if all events were tested. You can turn off cone limit filtering using the **-n** switch to see if this speeds up the processing (especially useful when processing a large list of positions).

For example, the default cone search uses columns "RA" and "Dec" in hours and degrees (respectively) and RA position in hours, Dec and radius in degrees:

```
funone in.fits out.fits 23.45 34.56 0.01
```

To specify the RA position in degrees:

```
funcone in.fits out.fits 23.45d 34.56 0.01
```

To get RA and Dec from a list but use a static value for radius (and also write identifying info for each row in the list):

```
funcone -x -l list.txt in.fits out.fits MYRA MYDec 0.01
```

User specified columns in degrees, RA position in hours (sexagesimal notation), Dec position in degrees (sexagesimal notation) and radius in arc minutes:

```
funcone -r myRa:d -d myDec in.fits out.fits 12:30:15.5 30:12 15'
```

fundisp - display data in a Funtools data file

```
fundisp  [-f format] [-l] [-n] [-T] <iname> [columns|bitpix=n]

  -f      # format string for display
  -l      # display image as a list containing the columns X, Y, VAL
  -n      # don't output header
  -F [c]  # use specified character as column separator (def: space)
  -T      # output in rdb/starbase format (tab separators)
```

fundisp displays the data in the specified FITS Extension and/or Image Section of a FITS file, or in a Section of a non-FITS array or raw event file.

The first argument to the program specifies the FITS input image, array, or raw event file to display. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions, image sections, and filters.

If the data being displayed are columns (either in a FITS binary table or a raw event file), the individual rows are listed. Filters can be added using bracket notation. Thus:

```
[sh] fundisp "test.ev[time-(int)time>.15]"
       X       Y     PHA        PI              TIME         DX         DY
 ------- ------- ------- --------- ----------------- ---------- ----------
      10       8      10         8           17.1600       8.50      10.50
       9       9       9         9           17.1600       9.50       9.50
      10       9      10         9           18.1600       9.50      10.50
      10       9      10         9           18.1700       9.50      10.50
       8      10       8        10           17.1600      10.50       8.50
       9      10       9        10           18.1600      10.50       9.50
       9      10       9        10           18.1700      10.50       9.50
      10      10      10        10           19.1600      10.50      10.50
      10      10      10        10           19.1700      10.50      10.50
      10      10      10        10           19.1800      10.50      10.50
```

[NB: The FITS binary table test file test.ev, as well as the FITS image test.fits, are contained in the funtools funtest directory.]

When a table is being displayed using **fundisp**, a second optional argument can be used to specify the columns to display. For example:

```
[sh] fundisp "test.ev[time-(int)time>=.99]" "x y time"
        X        Y                   TIME
 -------- -------- ---------------------
        5       -6           40.99000000
        4       -5           59.99000000
       -1        0          154.99000000
       -2        1          168.99000000
       -3        2          183.99000000
       -4        3          199.99000000
       -5        4          216.99000000
       -6        5          234.99000000
       -7        6          253.99000000
```

The special column **$REGION** can be specified to display the region id of each row:

```
[sh $] fundisp "test.ev[time-(int)time>=.99&&annulus(0 0 0 10 n=3)]" 'x y time $REGION'
        X        Y                   TIME     REGION
 -------- -------- --------------------- ----------
        5       -6           40.99000000          3
        4       -5           59.99000000          2
       -1        0          154.99000000          1
       -2        1          168.99000000          1
       -3        2          183.99000000          2
       -4        3          199.99000000          2
       -5        4          216.99000000          2
       -6        5          234.99000000          3
       -7        6          253.99000000          3
```

Here only rows with the proper fractional time and whose position also is within one of the three annuli are displayed.

Columns can be excluded from display using a minus sign before the column:

```
[sh $] fundisp "test.ev[time-(int)time>=.99]" "-time"
        X        Y      PHA         PI           DX           DY
 -------- -------- -------- ---------- ------------ ------------
        5       -6        5         -6         5.50        -6.50
        4       -5        4         -5         4.50        -5.50
       -1        0       -1          0        -1.50         0.50
       -2        1       -2          1        -2.50         1.50
       -3        2       -3          2        -3.50         2.50
       -4        3       -4          3        -4.50         3.50
       -5        4       -5          4        -5.50         4.50
       -6        5       -6          5        -6.50         5.50
       -7        6       -7          6        -7.50         6.50
```

All columns except the time column are displayed.

The special column **$N** can be specified to display the ordinal value of each row. Thus, continuing the previous example:

```
fundisp "test.ev[time-(int)time>=.99]" '-time $n'
       X        Y      PHA         PI          DX          DY          N
 ------- -------- -------- ---------- ----------- ----------- ----------
       5       -6        5         -6        5.50       -6.50         337
       4       -5        4         -5        4.50       -5.50         356
      -1        0       -1          0       -1.50        0.50         451
      -2        1       -2          1       -2.50        1.50         465
      -3        2       -3          2       -3.50        2.50         480
      -4        3       -4          3       -4.50        3.50         496
      -5        4       -5          4       -5.50        4.50         513
      -6        5       -6          5       -6.50        5.50         531
      -7        6       -7          6       -7.50        6.50         550
```

Note that the column specification is enclosed in single quotes to protect '$n' from begin expanded by the shell.

In general, the rules for activating and de-activating columns are:

- If only exclude columns are specified, then all columns but the exclude columns will be activated.
- If only include columns are specified, then only the specified columns are activated.
- If a mixture of include and exclude columns are specified, then all but the exclude columns will be active; this last case is ambiguous and the rule is arbitrary.

In addition to specifying columns names explicitly, the special symbols **+** and **-** can be used to activate and de-activate **all** columns. This is useful if you want to activate the $REGION column along with all other columns. According to the rules, the syntax "$REGION" only activates the region column and de-activates the rest. Use "+ $REGION" to activate all columns as well as the region column.

If the data being displayed are image data (either in a FITS primary image, a FITS image extension, or an array file), an mxn pixel display is produced, where m and n are the dimensions of the image. By default, pixel values are displayed using the same data type as in the file. However, for integer data where the BSCALE and BZERO header parameters are present, the data is displayed as floats. In either case, the display data type can be overridden using an optional second argument of the form:

```
bitpix=n
```

where n is 8,16,32,-32,-64, for unsigned char, short, int, float and double, respectively.

Of course, running **fundisp** on anything but the smallest image usually results in a display whose size makes it unreadable. Therefore, one can uses bracket notation (see below) to apply section and/or blocking to the image before generating a display. For example:

```
[sh] fundisp "test.fits[2:6,2:7]" bitpix=-32
                  2          3          4          5          6
          ---------- ---------- ---------- ---------- ----------
       2:       3.00       4.00       5.00       6.00       7.00
       3:       4.00       5.00       6.00       7.00       8.00
       4:       5.00       6.00       7.00       8.00       9.00
       5:       6.00       7.00       8.00       9.00      10.00
       6:       7.00       8.00       9.00      10.00      11.00
       7:       8.00       9.00      10.00      11.00      12.00
```

Note that is is possible to display a FITS binary table as an image simply by passing the table through **funimage** first:

```
[sh] ./funimage test.ev stdout | fundisp "stdin[2:6,2:7]" bitpix=8
                2       3       4       5       6
        ------- ------- ------- ------- -------
     2:       3       4       5       6       7
     3:       4       5       6       7       8
     4:       5       6       7       8       9
     5:       6       7       8       9      10
     6:       7       8       9      10      11
     7:       8       9      10      11      12
```

If the **-l** (list) switch is used, then an image is displayed as a list containing the columns: X, Y, VAL. For example:

```
fundisp -l "test1.fits[2:6,2:7]" bitpix=-32
          X          Y          VAL
 ---------- ---------- -----------
          2          2        6.00
          3          2        1.00
          4          2        1.00
          5          2        1.00
          6          2        1.00
          2          3        1.00
          3          3        5.00
          4          3        1.00
          5          3        1.00
          6          3        1.00
          2          4        1.00
          3          4        1.00
          4          4        4.00
          5          4        1.00
          6          4        1.00
          2          5        1.00
          3          5        1.00
          4          5        1.00
          5          5        3.00
          6          5        1.00
          2          6        1.00
          3          6        1.00
          4          6        1.00
          5          6        1.00
          6          6        2.00
          2          7        1.00
```

```
3          7        1.00
4          7        1.00
5          7        1.00
6          7        1.00
```

If the **-n** (nohead) switch is used, then no header is output for tables. This is useful, for example, when fundisp output is being directed into gnuplot.

The **fundisp** program uses a default set of display formats:

```
datatype      TFORM   format
--------      -----   --------
double        D       "%21.8f"
float         E       "%11.2f"
int           J       "%10d"
short         I       "%8d"
byte          B       "%6d"
string        A       "%12.12s"
bits          X       "%8x"
logical       L       "%1x"
```

Thus, the default display of 1 double and 2 shorts gives:

```
[sh] fundisp snr.ev "time x y"

                  TIME        X        Y
 --------------------- -------- --------
     79494546.56818075      546      201
     79488769.94469175      548      201
     ...
```

You can change the display format for individual columns or for all columns of a given data types by means of the -f switch. The format string that accompanies -f is a space-delimited list of keyword=format values. The keyword values can either be column names (in which case the associated format pertains only to that column) or FITS table TFORM specifiers (in which case the format pertains to all columns having that data type). For example, you can change the double and short formats for all columns like this:

```
[sh] fundisp -f "D=%22.11f I=%3d" snr.ev "time x y"

                  TIME   X   Y
---------------------- --- ---
  79494546.56818075478 546 201
  79488769.94469174743 548 201
  ...
```

Alternatively, you can change the format of the time and x columns like this:

```
[sh] fundisp -f "time=%22.11f x=%3d" snr.ev "time x y"

                  TIME   X        Y
---------------------- --- --------
  79494546.56818075478 546      201
  79488769.94469174743 548      201
  ...
```

Note that there is a potential conflict if a column has the same name as one of the TFORM specifiers. In the examples above, the the "X" column in the table has the same name as the X (bit) datatype. To resolve this conflict, the format string is processed such that TFORM datatype specifiers are checked for first, using a case-sensitive comparison. If the specified format value is not an upper case TFORM value, then a case-insensitive check is made on the column name. This means that, in the examples above, "X=%3d" will refer to the X (bit) datatype, while "x=%3d" will refer to the X column:

```
[sh] fundisp -f "X=%3d" snr.ev "x y"

       X        Y
-------- --------
     546      201
     548      201
     ...

[sh] fundisp -f "x=%3d" snr.ev "x y"

  X        Y
--- --------
546      201
548      201
...
```

As a rule, therefore, it is best always to specify the column name in lower case and TFORM data types in upper case.

The **-f [format]** will change the format for a single execution of fundisp. You also can use the **FUN_FORMAT** envronment variable to change the format for all invocations of fundisp. The format of this environment variable's value is identical to that used with the **-f** switch. This global value can be overridden in individual cases by use of the **-f [format]** switch.

Caveats: Please also note that it is the user's responsibility to match the format specifier to the column data type correctly. Also note that, in order to maintain visual alignment between names and columns, the column name will be truncated (on the left) if the format width is less than the length of the name. However, truncation is not performed if the output is in RDB format (using the -T switch).

[An older-style format string is supported but deprecated. It consists of space-delimited C format statements for all data types, specified in the following order:

```
double float int short byte string bit.
```

This order of the list is based on the assumption that people generally will want to change the float formats.

If "-" is entered instead of a format statement for a given data type, the default format is used. Also, the format string can be terminated without specifying all formats, and defaults will be used for the rest of the list. Note that you must supply a minimum field width, i.e., "%6d" and "%-6d" are legal, "%d" is not legal. By using -f [format], you can change the double and short formats like this:

```
[sh] fundisp -f "22.11f - - 3d" snr.ev "time x y"

                     TIME   X   Y
 ---------------------- --- ---
   79494546.56818075478 546 201
   79488769.94469174743 548 201
   ...
```

NB: This format is deprecated and will be removed in a future release.]

The **-F[c]** switch can be used to specify a (single-character) column separator (where the default is a space). Note that column formatting will almost certainly also add spaces to pad individual columns to the required width. These can be removed with a program such as sed, at the cost of generating unaligned columns. For example:

```
fundisp -F',' snr.ev'[cir 512 512 .1]'
       X,       Y,     PHA,      PI,                 TIME,      DX,      DY
--------,--------,--------,--------,--------------------,--------,--------
     512,     512,       6,       7,    79493997.45854475,     578,     574
     512,     512,       8,       9,    79494575.58943175,     579,     573
     512,     512,       5,       6,    79493631.03866175,     578,     575
     512,     512,       5,       5,    79493290.86521725,     578,     575
     512,     512,       8,       9,    79493432.00990875,     579,     573

fundisp -F',' snr.ev'[cir 512 512 .1]' | sed 's/ *, */,/g'
       X,Y,PHA,PI,TIME,DX,DY
--------,--------,--------,--------,--------------------,--------,--------
     512,512,6,7,79493997.45854475,578,574
     512,512,8,9,79494575.58943175,579,573
     512,512,5,6,79493631.03866175,578,575
     512,512,5,5,79493290.86521725,578,575
     512,512,8,9,79493432.00990875,579,573

fundisp -f "x=%3d y=%3d pi=%1d pha=%1d time=%20.11f dx=%3d dy=%3d" -F',' snr.ev'[cir 512 512 .1]' | sed 's/ *, */,/g'
  X,Y,A,I,TIME,DX,DY
---,---,-,-,--------------------,---,---
512,512,6,7,79493997.45854474604,578,574
512,512,8,9,79494575.58943174779,579,573
512,512,5,6,79493631.03866174817,578,575
512,512,5,5,79493290.86521725357,578,575
512,512,8,9,79493432.00990875065,579,573
```

If the **-T** (rdb table) switch is used, the output will conform to starbase/rdb data base format: tabs will be inserted between columns rather than spaces. This format is not available when displaying image pixels (except in conjunction with the **-l** switch).

Finally, note that **fundisp** can be used to create column filters from the auxiliary tables in a FITS file. For example, the following shell code will generate a good-time interval (GTI) filter for X-ray data files that contain a standard GTI extension:

```
#!/bin/sh
sed '1,/---- .*/d
/^$/,$d' | awk 'tot>0{printf "||"};{printf "time="$1":"$2; tot++}'
```

If this script is placed in a file called "mkgti", it can be used in a command such as:

```
fundisp foo.fits"[GTI]" | mkgti > gti.filter
```

The resulting filter file can then be used in various funtools programs:

```
funcnts foo.fits"[@gti.filter]" ...
```

to process only the events in the good-time intervals.

funhead - display a header in a Funtools file

```
funhead  [-a] [-s] [-t] [-L] <iname> [oname ename]

  -a    # display all extension headers
  -s    # display 79 chars instead of 80 before the new-line
  -t    # prepend data type char to each line of output
  -L    # output in rdb/starbase list format
```

funhead displays the FITS header parameters in the specified FITS Extension.

The first argument to the program specifies the Funtools input file to display. If "stdin" is specified, data are read from the standard input. Funtools Bracket Notation is used to specify particular FITS extension to process. Normally, the full 80 characters of each header card is output, followed by a new-line.

If the **-a** switch is specified, the header from each FITS extensions in the file is displayed. Note, however, that the **-a** switch does not work with FITS files input via stdin. We hope to remove this restriction in a future release.

If the **-s** switch is specified, only 79 characters are output before the new-line. This helps the display on 80 character terminals.

If the **-t** switch is specified, the data type of the parameter is output as a one character prefix, followed by 77 characters of the param. The parameter data types are defined as: FUN_PAR_UNKNOWN ('u'), FUN_PAR_COMMENT ('c'), FUN_PAR_LOGICAL ('l'), FUN_PAR_INTEGER ('i'), FUN_PAR_STRING ('s'), FUN_PAR_REAL ('r'), FUN_PAR_COMPLEX ('x').

If the **-L** (rdb table) switch is used, the output will conform to starbase/rdb data base list format.

For example to display the EVENTS extension (binary table):

```
[sh] funhead "foo.fits[EVENTS]"
XTENSION= 'BINTABLE'            /  FITS 3D BINARY TABLE
BITPIX  =                    8  /  Binary data
NAXIS   =                    2  /  Table is a matrix
NAXIS1  =                   20  /  Width of table in bytes
NAXIS2  =                30760  /  Number of entries in table
PCOUNT  =                    0  /  Random parameter count
GCOUNT  =                    1  /  Group count
TFIELDS =                    7  /  Number of fields in each row
EXTNAME = 'EVENTS  '            /  Table name
EXTVER  =                    1  /  Version number of table
TFORM1  = '1I      '            /  Data type for field
TTYPE1  = 'X       '            /  Label for field
TUNIT1  = '        '            /  Physical units for field
TFORM2  = '1I      '            /  Data type for field
  etc. ...
END
```

To display the third header:

```
[sh] funhead "foo.fits[3]"
XTENSION= 'BINTABLE'            /  FITS 3D BINARY TABLE
BITPIX  =                    8  /  Binary data
NAXIS   =                    2  /  Table is a matrix
NAXIS1  =                   32  /  Width of table in bytes
NAXIS2  =                   40  /  Number of entries in table
PCOUNT  =                    0  /  Random parameter count
GCOUNT  =                    1  /  Group count
TFIELDS =                    7  /  Number of fields in each row
EXTNAME = 'TGR     '            /  Table name
EXTVER  =                    1  /  Version number of table
TFORM1  = '1D      '            /  Data type for field
  etc. ...
END
```

To display the primary header (i.e., extension 0):

```
sh> funhead "coma.fits[0]"
SIMPLE  =                    T /STANDARD FITS FORMAT
BITPIX  =                   16 /2-BYTE TWOS-COMPL INTEGER
NAXIS   =                    2 /NUMBER OF AXES
NAXIS1  =                  800 /
NAXIS2  =                  800 /
DATATYPE= 'INTEGER*2'          /SHORT INTEGER
END
```

The funhead program also can edit (i.e. add, delete, or modify) or display individual headers parameters. Edit mode is signalled by the presence of two additional command-line arguments: output file and edit command file, in that order. Edit mode acts as a filter: the output file will contain the entire input FITS file, including other extensions. The edit command file can be "stdin", in which case edit command are read from the standard input.

The edit command file contains parameter comments (having '#' in the first column) and delete and assignment(modify or add) operations. A delete operation is specified by preceding the parameter name with a minus sign "-". A display operation (very useful in interactive sessions, i.e., where the edit commands are taken from stdin) is specified by preceding the parameter name with a question mark "?". In either case, a parameter value need not be specified. An assignment operation is specified in the same two ways that a parameter is specified in a text header (but without the comment character that precedes header params), i.e.:

- FITS-style comments have an equal sign "=" between the keyword and value and an optional slash "/" to signify a comment. The strict FITS rules on column positions are not enforced.
- Free-form comments can have an optional colon separator between the keyword and value. In the absence of quote, all tokens after the keyword are part of the value, i.e. no comment is allowed.

For example, the following interactive session checks for the existence of parameters, adds new parameters, modifies them, and modifies and deletes existing parameters:

```
sh$ ./funhead snr.ev foo.fits -
# look for FOO1
? FOO1
WARNING: FOO1 not found
# add new foo1
FOO1 = 100
# add foo2
FOO2 = 200
# reset foo1 to a different value
FOO1 -1
# delete foo2
-FOO2
# change existing value
EXTVER 2
? XS-SORT
XS-SORT = 'EOF     '            /  type of event sort
# delete existing value
-XS-SORT
# exit
^D
```

See Column-based Text Files for more information about header parameter format.

funhist - create a 1D histogram of a column (from a FITS binary table or raw event file) or an image

```
funhist  [-n|-w|-T] <iname> [column] [[lo:hi:]bins]

  -n    # normalize bin value by the width of each bin
  -w    # specify bin width instead of number of bins in arg3
  -T    # output in rdb/starbase format (tab separators)
```

funhist creates a one-dimensional histogram from the specified columns of a FITS Extension binary table of a FITS file (or from a non-FITS raw event file), or from a FITS image or array, and writes that histogram as an ASCII table. Alternatively, the program can perform a 1D projection of one of the image axes.

The first argument to the program is required, and specifies the Funtools file: FITS table or image, raw event file, or array. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions, and filters.

For a table, the second argument also is required. It specifies the column to use in generating the histogram. If the data file is of type image (or array), the column is optional: if "x" (or "X"), "y" (or "Y") is specified, then a projection is performed over the x (dim1) or y (dim2) axes, respectively. (That is, this projection will give the same results as a histogram performed on a table containing the equivalent x,y event rows.) If no column name is specified or "xy" (or "XY") is specified for the image, then a histogram is performed on the values contained in the image pixels.

The argument that follows is optional and specifies the number of bins to use in creating the histogram and, if desired, the range of bin values. For image and table histograms, the range should specify the min and max data values. For image histograms on the x and y axes, the range should specify the min and max

image bin values. If this argument is omitted, the number of output bins for a table is calculated either from the TLMIN/TLMAX headers values (if these exist in the table FITS header for the specified column) or by going through the data to calculate the min and max value. For an image, the number of output bins is calculated either from the DATAMIN/DATAMAX header values, or by going through the data to calculate min and max value. (Note that this latter calculation might fail if the image cannot be fit in memory.) If the data are floating point (table or image) and the number of bins is not specified, an arbitrary default of 128 is used.

For binary table processing, the **-w** (bin width) switch can be used to specify the width of each bin rather than the number of bins. Thus:

```
funhist test.ev pha 1:100:5
```

means that 5 bins of width 20 are used in the histogram, while:

```
funhist -w test.ev pha 1:100:5
```

means that 20 bins of width 5 are used in the histogram.

The data are divvied up into the specified number of bins and the resulting 1D histogram (or projection) is output in ASCII table format. For a table, the output displays the low_edge (inclusive) and hi_edge (exclusive) values for the data. For example, a 15-row table containing a "pha" column whose values range from -7.5 to 7.5 can be processed thus:

```
[sh] funhist test.ev pha
# data file:        /home/eric/data/test.ev
# column:           pha
# min,max,bins:     -7.5 7.5 15

   bin     value               lo_edge               hi_edge
------ --------- -------------------- --------------------
     1        22          -7.50000000          -6.50000000
     2        21          -6.50000000          -5.50000000
     3        20          -5.50000000          -4.50000000
     4        19          -4.50000000          -3.50000000
     5        18          -3.50000000          -2.50000000
     6        17          -2.50000000          -1.50000000
     7        16          -1.50000000          -0.50000000
     8        30          -0.50000000           0.50000000
     9        16           0.50000000           1.50000000
    10        17           1.50000000           2.50000000
    11        18           2.50000000           3.50000000
    12        19           3.50000000           4.50000000
    13        20           4.50000000           5.50000000
    14        21           5.50000000           6.50000000
    15        22           6.50000000           7.50000000

[sh] funhist test.ev pha 1:6
# data file:          /home/eric/data/test.ev
# column:             pha
# min,max,bins:       0.5 6.5 6

   bin     value               lo_edge               hi_edge
------ --------- -------------------- --------------------
```

```
     1        16            0.50000000            1.50000000
     2        17            1.50000000            2.50000000
     3        18            2.50000000            3.50000000
     4        19            3.50000000            4.50000000
     5        20            4.50000000            5.50000000
     6        21            5.50000000            6.50000000

[sh] funhist test.ev pha 1:6:3
# data file:          /home/eric/data/test.ev
# column:             pha
# min,max,bins:       0.5 6.5 3

   bin     value               lo_edge               hi_edge
------ --------- --------------------- ---------------------
     1        33            0.50000000            2.50000000
     2        37            2.50000000            4.50000000
     3        41            4.50000000            6.50000000
```

For a table histogram, the **-n**(normalize) switch can be used to normalize the bin value by the width of the bin (i.e., hi_edge-lo_edge):

```
[sh] funhist -n test.ev pha 1:6:3
# data file:          test.ev
# column:             pha
# min,max,bins:       0.5 6.5 3
# width normalization (val/(hi_edge-lo_edge)) is applied

   bin                 value               lo_edge               hi_edge
------ --------------------- --------------------- ---------------------
     1           16.50000000            0.50000000            2.50000000
     2            6.16666667            2.50000000            4.50000000
     3            4.10000000            4.50000000            6.50000000
```

This could used, for example, to produce a light curve with values having units of counts/second instead of counts.

For an image histogram, the output displays the low and high image values (both inclusive) used to generate the histogram. For example, in the following example, 184 pixels had a value of 1, 31 had a value of 2, while only 2 had a value of 3,4,5,6, or 7:

```
[sh] funhist test.fits
# data file:          /home/eric/data/test.fits
# min,max,bins:       1 7 7

   bin                 value                lo_val                hi_val
------ --------------------- --------------------- ---------------------
     1          184.00000000            1.00000000            1.00000000
     2           31.00000000            2.00000000            2.00000000
     3            2.00000000            3.00000000            3.00000000
     4            2.00000000            4.00000000            4.00000000
     5            2.00000000            5.00000000            5.00000000
     6            2.00000000            6.00000000            6.00000000
     7            2.00000000            7.00000000            7.00000000
```

For the axis projection of an image, the output displays the low and high image bins (both inclusive) used to generate the projection. For example, in the following example, 21 counts had their X bin value of 2, etc.:

```
[sh] funhist test.fits x 2:7
# data file:            /home/eric/data/test.fits
# column:               X
# min,max,bins: 2 7 6

   bin                 value                lo_bin                hi_bin
------ -------------------- -------------------- --------------------
     1           21.00000000           2.00000000           2.00000000
     2           20.00000000           3.00000000           3.00000000
     3           19.00000000           4.00000000           4.00000000
     4           18.00000000           5.00000000           5.00000000
     5           17.00000000           6.00000000           6.00000000
     6           16.00000000           7.00000000           7.00000000

[sh] funhist test.fits x 2:7:2
# data file:            /home/eric/data/test.fits
# column:               X
# min,max,bins: 2 7 2

   bin                 value                lo_bin                hi_bin
------ -------------------- -------------------- --------------------
     1           60.00000000           2.00000000           4.00000000
     2           51.00000000           5.00000000           7.00000000
```

You can use gnuplot or other plotting programs to graph the results, using a script such as:

```
#!/bin/sh
sed -e '1,/---- .*/d
/^$/,$d' | \
awk '\
BEGIN{print "set nokey; set title \"funhist\"; set xlabel \"bin\"; set ylabel \"counts\"; plot \"-\" with boxes"}   \
{print $3, $2, $4-$3}'        | \
gnuplot -persist - 1>/dev/null 2>&1
```

Similar plot commands are supplied in the script **funhist.plot**:

```
funhist test.ev pha ...  | funhist.plot gnuplot
```

funimage - create a FITS image from a Funtools data file

```
funimage [-a] <iname> <oname> [bitpix=n]
funimage [-l] <iname> <oname> <xcol:xdims> <ycol:ydims> <vcol> [bitpix=n]
funimage [-p x|y] <iname> <oname> [bitpix=n]

  -a       # append to existing output file as an image extension
  -l       # input is a list file containing xcol, ycol, value
  -p [x|y] # project along x or y axis to create a 1D image
```

funimage creates a primary FITS image from the specified FITS Extension and/or Image Section of a FITS file, or from an Image Section of a non-FITS array, or from a raw event file.

The first argument to the program specifies the FITS input image, array, or raw event file to process. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions, image sections, and filters. The second argument is the output FITS file. If "stdout" is specified, the FITS image is written to the standard output. By default, the output pixel values are of the same data type as those of the input file (or type "int" when binning a table), but this can be overridden using an optional third argument of the form:

```
bitpix=n
```

where n is 8,16,32,-32,-64, for unsigned char, short, int, float and double, respectively.

If the input data are of type image, the appropriate section is extracted and blocked (based on how the Image Section is specified), and the result is written to the FITS primary image. When an integer image containing the BSCALE and BZERO keywords is converted to float, the pixel values are scaled and the scaling keywords are deleted from the output header. When converting integer scaled data to integer (possibly of a different size), the pixels are not scaled and the scaling keywords are retained.

If the input data is a binary table or raw event file, these are binned into an image, from which a section is extracted and blocked, and written to a primary FITS image. In this case, it is necessary to specify the two columns that will be used in the 2D binning. This can be done on the command line using the **bincols=(x,y)** keyword:

```
funcnts "foo.ev[EVENTS,bincols=(detx,dety)]"
```

The full form of the **bincols=** specifier is:

```
bincols=([xname[:tlmin[:tlmax:[binsiz]]]],[yname[:tlmin[:tlmax[:binsiz]]]])
```

where the tlmin, tlmax, and binsiz specifiers determine the image binning dimensions:

```
dim = (tlmax - tlmin)/binsiz     (floating point data)
dim = (tlmax - tlmin)/binsiz + 1 (integer data)
```

Using this syntax, it is possible to bin any two columns of a binary table at any bin size. Note that the tlmin, tlmax, and binsiz specifiers can be omitted if TLMIN, TLMAX, and TDBIN header parameters (respectively) are present in the FITS binary table header for the column in question. Note also that if only one parameter is specified, it is assumed to be tlmax, and tlmin defaults to 1. If two parameters are specified, they are assumed to be tlmin and tlmax. See Binning FITS Binary Tables and Non-FITS Event Files for more information about binning parameters.

By default, a new 2D FITS image file is created and the image is written to the primary HDU. If the **-a** (append) switch is specified, the image is appended to an existing FITS file as an IMAGE extension. (If the output file does not exist, the switch is effectively ignored and the image is written to the primary HDU.) This can be useful in a shell programming environment when processing multiple FITS images that you want to combine into a single final FITS file.

funimage also can take input from a table containing columns of x, y, and value (e.g., the output from **fundisp -l** which displays each image x and y and the number of counts at that position.) When the **-l** (list) switch is used, the input file is taken to be a FITS or ASCII table containing (at least) three columns that specify the x and y image coordinates and the value of that image pixel. In this case, **funimage** requires

four extra arguments: xcolumn:xdims, ycolumn:ydims, vcolumn and bitpix=n. The x and y col:dim information takes the form:

```
name:dim               # values range from 1 to dim
name:min:max           # values range from min to max
name:min:max:binsiz    # dimensions scaled by binsize
```

In particular, the min value should be used whenever the minimum coordinate value is something other than one. For example:

```
funimage -l foo.lst foo.fits xcol:0:512 ycol:0:512 value bitpix=-32
```

The list feature also can be used to read unnamed columns from standard input: simply replace the column name with a null string. Note that the dimension information is still required:

```
funimage -l stdin foo.fits "":0:512 "":0:512 "" bitpix=-32
240 250 1
255 256 2
...
^D
```

The list feature provides a simple way to generate a blank image. If you pass a Column-based Text File to funimage in which the text header contains the required image information, then funimage will correctly make a blank image. For example, consider the following text file (called foo.txt):

```
x:I:1:10  y:I:1:10
------    ------
0         0
```

This text file defines two columns, x and y, each of data type 32-bit int and image dimension 10. The command:

```
funimage foo.txt foo.fits bitpix=8
```

will create an empty FITS image called foo.fits containing a 10x10 image of unsigned char:

```
fundisp foo.fits
         1      2      3      4      5      6      7      8      9     10
    ------ ------ ------ ------ ------ ------ ------ ------ ------ ------
10:      0      0      0      0      0      0      0      0      0      0
 9:      0      0      0      0      0      0      0      0      0      0
 8:      0      0      0      0      0      0      0      0      0      0
 7:      0      0      0      0      0      0      0      0      0      0
 6:      0      0      0      0      0      0      0      0      0      0
 5:      0      0      0      0      0      0      0      0      0      0
 4:      0      0      0      0      0      0      0      0      0      0
 3:      0      0      0      0      0      0      0      0      0      0
 2:      0      0      0      0      0      0      0      0      0      0
 1:      1      0      0      0      0      0      0      0      0      0
```

Note that the text file must contain at least one row of data. However, in the present example, event position 0,0 is outside the limits of the image and will be ignored. (You can, of course, use real x,y values to seed the image with data.)

Furthermore, you can use the TEXT filter specification to obviate the need for an input text file altogether. The following command will create the same 10x10 char image without an actual input file:

```
  funimage stdin'[TEXT(x:I:10,y:I:10)]' foo.fits bitpix=8 < /dev/null
or
  funimage /dev/null'[TEXT(x:I:10,y:I:10)]' foo.fits bitpix=8
```

You also can use either of these methods to generate a region mask simply by appending a region inside the filter brackets and specfying **mask=all** along with the bitpix. For example, the following command will generate a 10x10 char mask using 3 regions:

```
  funimage stdin'[TEXT(x:I:10,y:I:10),cir(5,5,4),point(10,1),-cir(5,5,2)]' \
  foo.fits bitpix=8,mask=all < /dev/null
```

The resulting mask looks like this:

```
  fundisp foo.fits
          1      2      3      4      5      6      7      8      9     10
     ------ ------ ------ ------ ------ ------ ------ ------ ------ ------
 10:      0      0      0      0      0      0      0      0      0      0
  9:      0      0      0      0      0      0      0      0      0      0
  8:      0      0      1      1      1      1      1      0      0      0
  7:      0      1      1      1      1      1      1      1      0      0
  6:      0      1      1      0      0      0      1      1      0      0
  5:      0      1      1      0      0      0      1      1      0      0
  4:      0      1      1      0      0      0      1      1      0      0
  3:      0      1      1      1      1      1      1      1      0      0
  2:      0      0      1      1      1      1      1      0      0      0
  1:      0      0      0      0      0      0      0      0      0      2
```

You can use **funimage** to create 1D image projections along the x or y axis using the **-p [x|y]** switch. This capability works for both images and tables. For example consider a FITS table named ev.fits containing the following rows:

```
        X        Y
 -------- --------
        1        1
        1        2
        1        3
        1        4
        1        5
        2        2
        2        3
        2        4
        2        5
        3        3
        3        4
        3        5
        4        4
        4        5
        5        5
```

A corresponding 5x5 image, called dim2.fits, would therefore contain:

```
            1          2          3          4          5
   ---------- ---------- ---------- ---------- ----------
5:          1          1          1          1          1
4:          1          1          1          1          0
3:          1          1          1          0          0
2:          1          1          0          0          0
1:          1          0          0          0          0
```

A projection along the y axis can be performed on either the table or the image:

```
funimage -p y ev.fits stdout | fundisp stdin
            1          2          3          4          5
   ---------- ---------- ---------- ---------- ----------
1:          1          2          3          4          5

funimage -p y dim2.fits stdout | fundisp stdin
            1          2          3          4          5
   ---------- ---------- ---------- ---------- ----------
1:          1          2          3          4          5
```

Furthermore, you can create a 1D image projection along any column of a table by using the **bincols=[column]** filter specification and specifying a single column. For example, the following command projects the same 1D image along the y axis of a table as use of the **-p y** switch:

```
funimage ev.fits'[bincols=y]' stdout | fundisp stdin
            1          2          3          4          5
   ---------- ---------- ---------- ---------- ----------
1:          1          2          3          4          5
```

Examples:

Create a FITS image from a FITS binary table:

```
[sh] funimage test.ev test.fits
```

Display the FITS image generated from a blocked section of FITS binary table:

```
[sh]  funimage "test.ev[2:8,3:7,2]" stdout | fundisp stdin
                 1         2         3
          --------- --------- ---------
       1:        20        28        36
       2:        28        36        44
```

funindex - create an index for a column of a FITS binary table

```
funindex <switches>  <iname> <key> [oname]

  NB: these options are not compatible with Funtools processing. Please
  use the defaults instead.
  -c        # compress output using gzip"
  -a        # ASCII output, ignore -c (default: FITS table)"
  -f        # FITS table output (default: FITS table)"
  -l        # long output, i.e. with key value(s) (default: long)"
  -s        # short output, i.e. no key value(s) (default: long)"
```

The funindex script creates an index for the specified column (key) by running funtable -s (sort) and then saving the column value and the record number for each sorted row. This index will be used automatically by funtools filtering of that column, provided the index file's modification date is later than that of the data file.

The first required argument is the name of the FITS binary table to index. Please note that text files cannot be indexed at this time. The second required argument is the column (key) name to index. While multiple keys can be specified in principle, the funtools index processing assume a single key and will not recognize files containing multiple keys.

By default, the output index file name is [root]_[key].idx, where [root] is the root of the input file. Funtools looks for this specific file name when deciding whether to use an index for faster filtering. Therefore, the optional third argument (output file name) should not be used for funtools processing.

For example, to create an index on column Y for a given FITS file, use:

```
  funindex foo.fits Y
```

This will generate an index named foo_y.idx, which will be used by funtools for filters involving the Y column.

funjoin - join two or more FITS binary tables on specified columns

```
funjoin [switches] <ifile1> <ifile2> ... <ifilen> <ofile>

  -a  cols             # columns to activate in all files
  -a1 cols ... an cols # columns to activate in each file
  -b  'c1:bv1,c2:bv2'  # blank values for common columns in all files
  -bn 'c1:bv1,c2:bv2'  # blank values for columns in specific files
  -j  col              # column to join in all files
  -j1 col ... jn col   # column to join in each file
  -m min               # min matches to output a row
  -M max               # max matches to output a row
  -s                   # add 'jfiles' status column
  -S col               # add col as status column
  -t tol               # tolerance for joining numeric cols [2 files only]
```

funjoin joins rows from two or more (up to 32) FITS Binary Table files, based on the values of specified join columns in each file. NB: the join columns must have an index file associated with it. These files are generated using the **funindex** program.

The first argument to the program specifies the first input FITS table or raw event file. If "stdin" is specified, data are read from the standard input. Subsequent arguments specify additional event files and tables to join. The last argument is the output FITS file.

NB: Do **not** use Funtools Bracket Notation to specify FITS extensions and row filters when running funjoin or you will get wrong results. Rows are accessed and joined using the index files directly, and this bypasses all filtering.

The join columns are specified using the **-j col** switch (which specifies a column name to use for all files) or with **-j1 col1**, **-j2 col2**, ... **-jn coln** switches (which specify a column name to use for each file). A join column must be specified for each file. If both **-j col** and **-jn coln** are specified for a given file, then the latter is used. Join columns must either be of type string or type numeric; it is illegal to mix numeric and string columns in a given join. For example, to join three files using the same key column for each file, use:

```
funjoin -j key in1.fits in2.fits in3.fits out.fits
```

A different key can be specified for the third file in this way:

```
funjoin -j key -j3 otherkey in1.fits in2.fits in3.fits out.fits
```

The **-a "cols"** switch (and **-a1 "col1"**, **-a2 "cols2"** counterparts) can be used to specify columns to activate (i.e. write to the output file) for each input file. By default, all columns are output.

If two or more columns from separate files have the same name, the second (and subsequent) columns are renamed to have an underscore and a numeric value appended.

The **-m min** and **-M max** switches specify the minimum and maximum number of joins required to write out a row. The default minimum is 0 joins (i.e. all rows are written out) and the default maximum is 63 (the maximum number of possible joins with a limit of 32 input files). For example, to write out only those rows in which exactly two files have columns that match (i.e. one join):

```
funjoin -j key -m 1 -M 1 in1.fits in2.fits in3.fits ... out.fits
```

A given row can have the requisite number of joins without all of the files being joined (e.g. three files are being joined but only two have a given join key value). In this case, all of the columns of the non-joined file are written out, by default, using blanks (zeros or NULLs). The **-b c1:bv1,c2:bv2** and **-b1 'c1:bv1,c2:bv2' -b2 'c1:bv1,c2:bv2' ...** switches can be used to set the blank value for columns common to all files and/or columns in a specified file, respectively. Each blank value string contains a comma-separated list of column:blank_val specifiers. For floating point values (single or double), a case-insensitive string value of "nan" means that the IEEE NaN (not-a-number) should be used. Thus, for example:

```
funjoin -b "AKEY:???" -b1 "A:-1" -b3 "G:NaN,E:-1,F:-100" ...
```

means that a non-joined AKEY column in any file will contain the string "???", the non-joined A column of file 1 will contain a value of -1, the non-joined G column of file 3 will contain IEEE NaNs, while the non-joined E and F columns of the same file will contain values -1 and -100, respectively. Of course, where common and specific blank values are specified for the same column, the specific blank value is used.

To distinguish which files are non-blank components of a given row, the **-s** (status) switch can be used to add a bitmask column named "JFILES" to the output file. In this column, a bit is set for each non-blank file composing the given row, with bit 0 corresponds to the first file, bit 1 to the second file, and so on. The file names themselves are stored in the FITS header as parameters named JFILE1, JFILE2, etc. The **-S col** switch allows you to change the name of the status column from the default "JFILES".

A join between rows is the Cartesian product of all rows in one file having a given join column value with all rows in a second file having the same value for its join column and so on. Thus, if file1 has 2 rows with join column value 100, file2 has 3 rows with the same value, and file3 has 4 rows, then the join results in 2*3*4=24 rows being output.

The join algorithm directly processes the index file associated with the join column of each file. The smallest value of all the current columns is selected as a base, and this value is used to join equal-valued columns in the other files. In this way, the index files are traversed exactly once.

The **-t tol** switch specifies a tolerance value for numeric columns. At present, a tolerance value can join only two files at a time. (A completely different algorithm is required to join more than two files using a tolerance, somethng we might consider implementing in the future.)

The following example shows many of the features of funjoin. The input files t1.fits, t2.fits, and t3.fits contain the following columns:

```
  [sh] fundisp t1.fits
        AKEY    KEY       A       B
 ------------ ------ ------ ------
         aaa       0       0       1
         bbb       1       3       4
         ccc       2       6       7
         ddd       3       9      10
         eee       4      12      13
         fff       5      15      16
         ggg       6      18      19
         hhh       7      21      22

fundisp t2.fits
        AKEY    KEY       C       D
 ------------ ------ ------ ------
         iii       8      24      25
         ggg       6      18      19
         eee       4      12      13
         ccc       2       6       7
         aaa       0       0       1

fundisp t3.fits
        AKEY    KEY         E         F            G
------------ ------ -------- -------- ------------
         ggg       6        18        19       100.10
         jjj       9        27        28       200.20
         aaa       0         0         1       300.30
         ddd       3         9        10       400.40
```

Given these input files, the following funjoin command:

```
  funjoin -s -a1 "-B" -a2 "-D" -a3 "-E" -b \
  "AKEY:???" -b1 "AKEY:XXX,A:255" -b3 "G:NaN,E:-1,F:-100" \
  -j key t1.fits t2.fits t3.fits foo.fits
```

will join the files on the KEY column, outputting all columns except B (in t1.fits), D (in t2.fits) and E (in t3.fits), and setting blank values for AKEY (globally, but overridden for t1.fits) and A (in file 1) and G, E, and F (in file 3). A JFILES column will be output to flag which files were used in each row:

```
      AKEY    KEY      A       AKEY_2  KEY_2      C       AKEY_3  KEY_3        F           G   JFILES
---------- ------ ------ ------------ ------ ------ ------------ ------ -------- ---------- --------
       aaa      0      0          aaa      0      0          aaa      0        1     300.30        7
       bbb      1      3          ???      0      0          ???      0     -100        nan        1
       ccc      2      6          ccc      2      6          ???      0     -100        nan        3
       ddd      3      9          ???      0      0          ddd      3       10     400.40        5
       eee      4     12          eee      4     12          ???      0     -100        nan        3
       fff      5     15          ???      0      0          ???      0     -100        nan        1
       ggg      6     18          ggg      6     18          ggg      6       19     100.10        7
       hhh      7     21          ???      0      0          ???      0     -100        nan        1
       XXX      0    255          iii      8     24          ???      0     -100        nan        2
       XXX      0    255          ???      0      0          jjj      9       28     200.20        4
```

funmerge - merge one or more Funtools table files

```
funmerge  [-w|-x] -f [colname] <iname1> <iname2>  ... <oname>

  -f    # output a column specifying file from which this event came
  -w    # adjust position values using WCS info
  -x    # adjust position values using WCS info and save old values
```

funmerge merges FITS data from one or more FITS Binary Table files or raw event files.

The first argument to the program specifies the first input FITS table or raw event file. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions and row filters. Subsequent arguments specify additional event files and tables to merge. (NB: Stdin cannot not be used for any of these additional input file arguments.) The last argument is the output FITS file. The columns in each input table must be identical.

If an input file begins with the '@' character, it is processed as an include file, i.e., as a text file containing event file names (as well as blank lines and/or comment lines starting with the '#' sign). If standard input is specified as an include file ('@stdin'), then file names are read from the standard input until EOF (^D). Event files and include files can be mixed on a command line.

Rows from each table are written sequentially to the output file. If the switch **-f [colname]** is specified on the command line, an additional column is added to each row containing the number of the file from which that row was taken (starting from one). In this case, the corresponding file names are stored in the header parameters having the prefix **FUNFIL**, i.e., FUNFIL01, FUNFIL02, etc.

Using the **-w** switch (or **-x** switch as described below), **funmerge** also can adjust the position column values using the WCS information in each file. (By position columns, we mean the columns that the table is binned on, i.e., those columns defined by the **bincols=** switch, or (X,Y) by default.) To perform WCS alignment, the WCS of the first file is taken as the base WCS. Each position in subsequent files is adjusted by first converting it to the sky coordinate in its own WCS coordinate system, then by converting this sky position to the sky position of the base WCS, and finally converting back to a pixel position in the base system. Note that in order to perform WCS alignment, the appropriate WCS and TLMIN/TLMAX keywords must already exist in each FITS file.

When performing WCS alignment, you can save the original positions in the output file by using the **-x** (for "xtra") switch instead of the **-w** switch (i.e., using this switch also implies using **-w**) The old positions are saved in columns having the same name as the original positional columns, with the added prefix "OLD_".

Examples:

Merge two tables, and preserve the originating file number for each row in the column called "FILE" (along with the corresponding file name in the header):

```
[sh] funmerge -f "FILE" test.ev test2.ev merge.ev
```

Merge two tables with WCS alignment, saving the old position values in 2 additional columns:

```
[sh] funmerge -x test.ev test2.ev merge.ev
```

This program only works on raw event files and binary tables. We have not yet implemented image and array merging.

funsky - convert between image and sky coordinates

```
funsky iname[ext]                # RA,Dec (deg) or image pix from stdin
funsky iname[ext] [lname]        # RA, Dec (deg) or image pix from list
funsky iname[ext] [col1] [col2]          # named cols:units from stdin
funsky iname[ext] [lname] [col1] [col2] # named cols:units from list

-d        # always use integer tlmin conversion (as ds9 does)
-r        # convert x,y to RA,Dec (default: convert RA,Dec to x,y)
-o        # include offset from the nominal target position (in arcsec)
-v        # display input values also (default: display output only)
-T        # output display in rdb format (w/header,tab delimiters)
```

Funsky converts input sky coordinates (RA, Dec) to image coordinates (or vice versa) using the WCS information contained in the specified FITS file. Several calling sequences are supported in order to make it easy to specify coordinate positions in different ways.

The first required argument is always the input FITS file (or extension) containing the WCS information in an extension header. Note that the data from this file is not used. By default, the program converts input RA and Dec values to X and Y using this WCS information. If the WCS is associated with a FITS image, then the X,Y values are image values. If the WCS is associated with a binary table, then the X, Y values are physical values. To convert X,Y to RA and Dec, use the **-r** (reverse) switch.

If no other command arguments are supplied, then the input positions are read from the standard input. Each line is assumed to contain a single coordinate position consisting of an RA in degrees (or X in pixels) followed by a Dec in degrees (or Y in pixels). The usual delimiters are supported (spaces, commas, tabs). For example:

```
# read from stdin, default column names and units
[sh] funsky snr.ev
22.982695    58.606523   # input RA (hrs), Dec(deg)
   510.00       510.00
22.982127    58.607634   # input
   512.00       510.50
22.981700    58.614301   # input
   513.50       513.50
^D                       # end of input
```

If a second argument is supplied, this argument is assumed to be a file containing RA (X) and Dec (Y) positions. The file can either be an ASCII table or a FITS binary table. The order of columns is unimportant, if the table has a column header. In this case, the names of the columns must be one of "RA", "DEC", or "X", "Y" for sky to image and image to sky conversions, respectively. If the table has no header, then once again, RA (X) is assumed to first, followed by DEC (Y). For example:

```
# read from file, default column names and units
[sh] cat hd.in
       RA          DEC
---------    ---------
22.982695    58.606523
22.982127    58.607634
22.981700    58.614301

[sh] funsky snr.ev hd.in
      510.00       510.00
      512.00       510.50
      513.50       513.50
```

If three arguments are supplied, then the input positions again are read from the standard input. Each line is assumed to contain a single coordinate position consisting of an RA (or X in pixels) followed by a Dec (or Y in pixels), with the usual delimiters supported. However, the second and third arguments now specify the column names and/or sky units using a colon-delimited syntax:

```
[colname]:[h|d|r]
```

If the colname is omitted, the names default to "RA", "DEC", "X", "Y", "COL1", or "COL2" as above. If the units are omitted, the default is degrees for both RA and Dec. When the -r switch is used (convert from image to sky) the units are applied to the output instead of the input. The following examples will serve to illustrate the options:

```
# read from stdin, specifying column names (def. units: degrees)
[sh] cat hd.in
     MYRA        MYDEC
---------    ---------
22.982695    58.606523
22.982127    58.607634
22.981700    58.614301

[sh] funsky snr.ev MYRA MYDEC < hd.in
      510.00       510.00
      512.00       510.50
      513.50       513.50

# read from stdin, specifying column names and units
[sh] cat dd.in
     MYRA        MYDEC
---------    ---------
344.740432    58.606523
344.731900    58.607634
344.725500    58.614301

[sh] funsky snr.ev MYRA:d MYDEC:d < dd.in
      510.00       510.00
```

```
      512.00       510.50
      513.50       513.50

# read stdin, convert image to sky, specifying output sky units
[sh] cat im.in
      510.00       510.00
      512.00       510.50
      513.50       513.50

[sh] cat im.in | funsky -r snr.ev :d :d
344.740432    58.606523
344.731900    58.607634
344.725500    58.614301
```

Finally, four command arguments specify both and input file and column names and/or units:

```
[sh] cat dd.in
     MYRA        MYDEC
---------    ---------
344.740432    58.606523
344.731900    58.607634
344.725500    58.614301

[sh] funsky snr.ev dd.in MYRA:d MYDEC:d
      510.00       510.00
      512.00       510.50
      513.50       513.50

# read file, convert image to sky, specifying output sky units
[sh] cat im.in
      510.00       510.00
      512.00       510.50
      513.50       513.50

[sh] funsky -r snr.ev im.in :d :d
  344.740432    58.606523
  344.731900    58.607634
  344.725500    58.614301
```

By default, the output of funsky consists only of the converted coordinate position(s), one per output line. This makes parsing in shell scripts easy. Use the **-v** (verbose) switch to specify that the input coordinates should be pre-pended to each line. For example:

```
[sh] cat dd.in
     MYRA        MYDEC
---------    ---------
344.740432    58.606523
344.731900    58.607634
344.725500    58.614301

[sh] funsky snr.ev dd.in MYRA:d MYDEC:d
      510.00       510.00
      512.00       510.50
      513.50       513.50
```

```
[sh] funsky -v snr.ev dd.in MYRA:d MYDEC:d
  344.740432    58.606523       510.00       510.00
  344.731900    58.607634       512.00       510.50
  344.725500    58.614301       513.50       513.50
```

In addition, a full starbase table can be output using the **-T** (table) switch. This switch can be used with or without the -v switch. If the -T and -v are both specified, then a descriptive header parameters are output before the table (mainly to remind you of the sky units):

```
# output table in non-verbose mode
[sh] funsky -T snr.ev dd.in MYRA:d MYDEC:d
           X               Y
------------    ------------
      510.00          510.00
      512.00          510.50
      513.50          513.50

# output table in verbose mode
[sh] funsky -T -v snr.ev dd.in MYRA:d MYDEC:d
# IFILE = /Users/eric/data/snr.ev
# ICOL1 = MYRA
# ICOL2 = MYDEC
# IUNITS1 = d
# IUNITS2 = d
# OCOL1 = X
# OCOL2 = Y

        MYRA           MYDEC               X               Y
------------    ------------    ------------    ------------
  344.740432       58.606523          510.00          510.00
  344.731900       58.607634          512.00          510.50
  344.725500       58.614301          513.50          513.50
```

Finally, the **-d** (ds9) switch mimicks ds9's use of integer TLMIN and TLMAX values for all coordinate transformations. FITS conventions seem to call for use of floating point TLMIN and TLMAX when the data are floats. This convention is followed by funsky but results in a small discrepancy with ds9's converted values for floating point data. We will remedy this conflict in the future, maybe.

funtable - copy selected rows from a Funtools file to a FITS binary table

```
funtable [-a] [-i|-z] [-m] [-s cols] <iname> <oname> [columns]

  -a    # append to existing output file as a table extension
  -i    # for image data, only generate X and Y columns
  -m    # for tables, write a separate file for each region
  -s "col1 ..." # columns on which to sort
  -z    # for image data, output zero-valued pixels
```

funtable selects rows from the specified FITS Extension (binary table only) of a FITS file, or from a non-FITS raw event file, and writes those rows to a FITS binary table file. It also will create a FITS binary table from an image or a raw array file.

The first argument to the program specifies the FITS file, raw event file, or raw array file. If "stdin" is specified, data are read from the standard input. Use Funtools Bracket Notation to specify FITS extensions, and filters. The second argument is the output FITS file. If "stdout" is specified, the FITS binary table is written to the standard output. By default, all columns of the input file are copied to the output file. Selected columns can be output using an optional third argument in the form:

```
"column1 column1 ... columnN"
```

The **funtable** program generally is used to select rows from a FITS binary table using Table Filters and/or Spatial Region Filters. For example, you can copy only selected rows (and output only selected columns) by executing in a command such as:

```
[sh] funtable "test.ev[pha==1&π==10]" stdout "x y pi pha" | fundisp stdin
       X        Y     PHA        PI
 ------- ------- ------- ---------
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
       1       10       1         10
```

The special column **$REGION** can be specified to write the region id of each row:

```
[sh $] funtable "test.ev[time-(int)time>=.99&&annulus(0 0 0 10 n=3)]" stdout 'x y time $REGION' | fundisp stdin
       X        Y                  TIME     REGION
 -------- -------- -------------------- ----------
       5       -6          40.99000000          3
       4       -5          59.99000000          2
      -1        0         154.99000000          1
      -2        1         168.99000000          1
      -3        2         183.99000000          2
      -4        3         199.99000000          2
      -5        4         216.99000000          2
      -6        5         234.99000000          3
      -7        6         253.99000000          3
```

Here only rows with the proper fractional time and whose position also is within one of the three annuli are written.

Columns can be excluded from display using a minus sign before the column:

```
[sh $] funtable "test.ev[time-(int)time>=.99]" stdout "-time" | fundisp stdin
        X         Y      PHA          PI           DX           DY
 -------- --------- -------- ---------- ------------ ------------
        5        -6        5         -6         5.50        -6.50
        4        -5        4         -5         4.50        -5.50
       -1         0       -1          0        -1.50         0.50
       -2         1       -2          1        -2.50         1.50
       -3         2       -3          2        -3.50         2.50
```

```
-4        3        -4          3        -4.50        3.50
-5        4        -5          4        -5.50        4.50
-6        5        -6          5        -6.50        5.50
-7        6        -7          6        -7.50        6.50
```

All columns except the time column are written.

In general, the rules for activating and de-activating columns are:

- If only exclude columns are specified, then all columns but the exclude columns will be activated.
- If only include columns are specified, then only the specified columns are activated.
- If a mixture of include and exclude columns are specified, then all but the exclude columns will be active; this last case is ambiguous and the rule is arbitrary.

In addition to specifying columns names explicitly, the special symbols + and - can be used to activate and de-activate *all* columns. This is useful if you want to activate the $REGION column along with all other columns. According to the rules, the syntax "$REGION" only activates the region column and de-activates the rest. Use "+ $REGION" to activate all columns as well as the region column.

Ordinarily, only the selected table is copied to the output file. In a FITS binary table, it sometimes is desirable to copy all of the other FITS extensions to the output file as well. This can be done by appending a '+' sign to the name of the extension in the input file name. For example, the first command below copies only the EVENT table, while the second command copies other extensions as well:

```
[sh] funtable "/proj/rd/data/snr.ev[EVENTS]" events.ev
[sh] funtable "/proj/rd/data/snr.ev[EVENTS+]" eventsandmore.ev
```

If the input file is an image or a raw array file, then **funtable** will generate a FITS binary table from the pixel values in the image. Note that it is not possible to specify the columns to output (using command-line argument 3). Instead, there are two ways to create such a binary table from an image. By default, a 3-column table is generated, where the columns are "X", "Y", and "VALUE". For each pixel in the image, a single row (event) is generated with the "X" and "Y" columns assigned the dim1 and dim2 values of the image pixel, respectively and the "VALUE" column assigned the value of the pixel. With sort of table, running **funhist** on the "VALUE" column will give the same results as running **funhist** on the original image.

If the **-i** ("individual" rows) switch is specified, then only the "X" and "Y" columns are generated. In this case, each positive pixel value in the image generates n rows (events), where n is equal to the integerized value of that pixel (plus 0.5, for floating point data). In effect, **-i** approximately recreates the rows of a table that would have been binned into the input image. (Of course, this is only approximately correct, since the resulting x,y positions are integerized.)

If the **-s [col1 col2 ... coln]** ("sort") switch is specified, the output rows of a binary table will be sorted using the specified columns as sort keys. The sort keys must be scalar columns and also must be part of the output file (i.e. you cannot sort on a column but not include it in the output). This facility uses the **_sort** program (included with funtools), which must be accessible via your path.

For binary tables, the **-m** ("multiple files") switch will generate a separate file for each region in the filter specification i.e. each file contains only the rows from that region. Rows which pass the filter but are not in any region also are put in a separate file.

The separate output file names generated by the **-m** switch are produced automatically from the root output file to contain the region id of the associated region. (Note that region ids start at 1, so that the file name associated with id 0 contains rows that pass the filter but are not in any given region.) Output file names are generated as follows:

- A $n specification can be used anywhere in the root file name (suitably quoted to protect it from the shell) and will be expanded to be the id number of the associated region. For example:

  ```
  funtable -m input.fits'[cir(512,512,1);cir(520,520,1)...]' 'foo.goo_$n.fits'
  ```

  will generate files named foo.goo_0.fits (for rows not in any region but still passing the filter), foo.goo_1.fits (rows in region id #1, the first region), foo.goo_2.fits (rows in region id #2), etc. Note that single quotes in the output root are required to protect the '$' from the shell.
- If $n is not specified, then the region id will be placed before the first dot (.) in the filename. Thus:

  ```
  funtable -m input.fits'[cir(512,512,1);cir(520,520,1)...]' foo.evt.fits
  ```

  will generate files named foo0.evt.fits (for rows not in any region but still passing the filter), foo1.evt.fits (rows in region id #1), foo2.evt.fits (rows in region id #2), etc.
- If no dot is specified in the root output file name, then the region id will be appended to the filename. Thus:

  ```
  funtable -m input.fits'[cir(512,512,1);cir(520,520,1)...]' 'foo_evt'
  ```

  will generate files named foo_evt0 (for rows not in any region but still passing the filter), foo_evt1 (rows in region id #1), foo_evt2 (rows in region id #2), etc.

The multiple file mechanism provide a simple way to generate individual source data files with a single pass through the data.

By default, a new FITS file is created and the binary table is written to the first extension. If the **-a** (append) switch is specified, the table is appended to an existing FITS file as a BINTABLE extension. Note that the output FITS file must already exist.

If the **-z** ("zero" pixel values) switch is specified and **-i** is not specified, then pixels having a zero value will be output with their "VALUE" column set to zero. Obviously, this switch does not make sense when individual events are output.

funtbl - extract a table from Funtools ASCII output

```
funtable [-c cols] [-h] [-n table] [-p prog] [-s sep] <iname>
```

[NB: This program has been deprecated in favor of the ASCII text processing support in funtools. You can now perform fundisp on funtools ASCII output files (specifying the table using bracket notation) to extract tables and columns.] The **funtbl** script extracts a specified table (without the header and comments) from a

funtools ASCII output file and writes the result to the standard output. The first non-switch argument is the ASCII input file name (i.e. the saved output from funcnts, fundisp, funhist, etc.). If no filename is specified, stdin is read. The -n switch specifies which table (starting from 1) to extract. The default is to extract the first table. The -c switch is a space-delimited list of column numbers to output, e.g. -c "1 3 5" will extract the first three odd-numbered columns. The default is to extract all columns. The -s switch specifies the separator string to put between columns. The default is a single space. The -h switch specifies that column names should be added in a header line before the data is output. Without the switch, no header is prepended. The -p program switch allows you to specify an awk-like program to run instead of the default (which is host-specific and is determined at build time). The -T switch will output the data in rdb format (i.e., with a 2-row header of column names and dashes, and with data columns separated by tabs). The -help switch will print out a message describing program usage.

For example, consider the output from the following funcnts command:

```
[sh] funcnts -sr snr.ev "ann 512 512 0 9 n=3"
# source
#   data file:        /proj/rd/data/snr.ev
#   arcsec/pixel:     8
# background
#   constant value:   0.000000
# column units
#   area:             arcsec**2
#   surf_bri:         cnts/arcsec**2
#   surf_err:         cnts/arcsec**2

# summed background-subtracted results
upto   net_counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1      147.000    12.124        0.000     0.000   1600.00     0.092     0.008
   2      625.000    25.000        0.000     0.000   6976.00     0.090     0.004
   3     1442.000    37.974        0.000     0.000  15936.00     0.090     0.002


# background-subtracted results
 reg   net_counts     error   background    berror      area  surf_bri  surf_err
---- ------------ --------- ------------ --------- --------- --------- ---------
   1      147.000    12.124        0.000     0.000   1600.00     0.092     0.008
   2      478.000    21.863        0.000     0.000   5376.00     0.089     0.004
   3      817.000    28.583        0.000     0.000   8960.00     0.091     0.003


# the following source and background components were used:
source_region(s)
----------------
ann 512 512 0 9 n=3

 reg       counts    pixels      sumcnts    sumpix
---- ------------ --------- ------------ ---------
   1      147.000        25      147.000        25
   2      478.000        84      625.000       109
   3      817.000       140     1442.000       249
```

There are four tables in this output. To extract the last one, you can execute:

```
[sh] funcnts -s snr.ev "ann 512 512 0 9 n=3" | funtbl -n 4
1 147.000 25 147.000 25
2 478.000 84 625.000 109
3 817.000 140 1442.000 249
```

Note that the output has been re-formatted so that only a single space separates each column, with no extraneous header or comment information.

To extract only columns 1,2, and 4 from the last example (but with a header prepended and tabs between columns), you can execute:

```
[sh] funcnts -s snr.ev "ann 512 512 0 9 n=3" | funtbl -c "1 2 4" -h -n 4 -s "\t"
#reg    counts  sumcnts
1       147.000 147.000
2       478.000 625.000
3       817.000 1442.000
```

Of course, if the output has previously been saved in a file named foo.out, the same result can be obtained by executing:

```
[sh] funtbl -c "1 2 4" -h -n 4 -s "\t" foo.out
#reg    counts  sumcnts
1       147.000 147.000
2       478.000 625.000
3       817.000 1442.000
```

Go to Funtools Help Index

Last updated: April 1, 2007

FunDS9: Funtools and DS9 Image Display

Summary

Describes how funtools can be integrated into the ds9 Analysis menu.

Description

SAOImage/DS9 is an astronomical imaging and data visualization application used by astronomers around the world. DS9 can display standard astronomical FITS images and binary tables, but also has support for displaying raw array files, shared memory files, and data files automatically retrieved via FTP and HTTP. Standard functional capabilities include multiple frame buffers, colormap and region manipulation, and many data scaling algorithms. DS9's advanced features include TrueColor visuals, deep frame buffers, true PostScript printing, and display of image mosaics. The program's support of image tiling, "blinking", arbitrary zoom, rotation, and pan is unparalleled in astronomy. It also has innovative support for automatic retrieval and display of standard image data such as the Digital Sky Survey (using servers at SAO, StScI, or ESO).

DS9 can communicate with external programs such as Funtools using the XPA messaging system. In addition, programs can be integrated directly into the DS9 GUI by means of a configurable Analysis menu. By default, the DS9 Analysis menu contains algorithms deemed essential to the core functions of DS9, e.g., display cross-cuts of data, iso-intensity contours, and WCS grids. However, new programs can be added to DS9 by creating a set-up file which can be loaded into DS9 to reconfigure the Analysis menu.

```
The basic format of the analysis set-up file is:
  #
  # Analysis command descriptions:
  #   menu label/description
  #   file templates for this command
  #   "menu" (add to menu) |"bind" (bind to key)
  #   analysis command line
```

For example, the funcnts program can be specified in this way:

```
  Funcnts (counts in source/bkgd regions; options: none)
  *
  menu
  funcnts $filename $regions(source,,) $regions(background,,) | $text
```

As shown above, DS9 supports a macro facility to provide information as well as task support to command lines. For example, the $regions macro is expanded by DS9 to provide the current source and/or background region to the analysis command. The $text macro is expanded to generate a text window display. It also is possible to query for parameters using a $param macro, plot data using a $plot macro, etc. See the DS9 documentation for further details.

A set-up file called funtools.ds9 will load some useful Funtools applications (counts in regions, radial profile, X-ray light curve and energy spectrum, 1D histogram) into the DS9 Analysis menu (version 2.1 and above). The file resides in the bin directory where Funtools programs are installed. It can be manually loaded into DS9 from the **Load Analysis Commands ...** option of the **Analysis** menu. Alternatively, you

can tell DS9 to load this file automatically at start-up time by adding the pathname to the **Edit->Preferences->Analysis Menu->Analysis File** menu option. (NB: make sure you select **Edit->Preferences->Save Preferences** after setting the pathname.)

The tasks in this setup file generally process the original disk-based FITS file. Funcnts-based results (radial profile, counts in regions) are presented in WCS units, if present in the FITS header. For situations where a disk file is not available (e.g., image data generated and sent to DS9's 'fits' XPA access point), versions of the radial profile and counts in regions tasks also are also offered utilizing DS9's internal image data. Results are presented in pixels. Aside from the units, the results should be identical to the file-based results.

Go to Funtools Help Index

Last updated: November 16, 2005

FunLib: the Funtools Programming Interface

Summary

A description of the Funtools library.

Introduction to the Funtools Programming Interface

To create a Funtools application, you need to include the funtools.h definitions file in your code:

```
#include <funtools.h>
```

You then call Funtools subroutines and functions to access Funtools data. The most important routines are:

- FunOpen: open a Funtools file
- FunInfoGet: get info about an image or table
- FunImageGet: retrieve image data
- FunImageRowGet: retrieve image data by row
- FunImagePut: output image data
- FunImageRowPut: output image data by row
- FunColumnSelect: select columns in a table for access
- FunTableRowGet: retrieve rows from a table
- FunTableRowPut: output rows to a table
- FunClose: close a Funtools file

Your program must be linked against the libfuntools.a library, along with the math library. The following libraries also might be required on your system:

- -lsocket -lnsl for socket support
- -ldl for dynamic loading

For example, on a Solaris system using gcc, use the following link line:

```
gcc -o foo foo.c -lfuntools -lsocket -lnsl -ldl -lm
```

On a Solaris system using Solaris cc, use the following link line:

```
gcc -o foo foo.c -lfuntools -lsocket -lnsl -lm
```

On a Linux system using gcc, use the following link line:

```
gcc -o foo foo.c -lfuntools -ldl -lm
```

Once configure has built a Makefile on your platform, the required "extra" libraries (aside from -lm, which always is required) are specified in that file's EXTRA_LIBS variable. For example, under Linux you will find:

```
grep EXTRA_LIBS Makefile
EXTRA_LIBS      =  -ldl
...
```

The Funtools library contains both the zlib library (http://www.gzip.org/zlib/) and Doug Mink's WCS library (http://tdc-www.harvard.edu/software/wcstools/). It is not necessary to put these libraries on a Funtools link line. Include files necessary for using these libraries are installed in the Funtools include directory.

Funtools Programming Tutorial

The FunOpen() function is used to open a FITS file, an array, or a raw event file:

```
/* open the input FITS file for reading */
ifun = FunOpen(iname, "r", NULL);
/* open the output FITS file for writing, and connect it to the input file */
ofun = FunOpen(iname, "w", ifun);
```

A new output file can inherit header parameters automatically from existing input file by passing the input Funtools handle as the last argument to the new file's FunOpen() call as shown above.

For image data, you then can call FunImageGet() to read an image into memory.

```
float buf=NULL;
/* extract and bin the data section into an image buffer */
buf = FunImageGet(fun, NULL, "bitpix=-32");
```

If the (second) buf argument to this call is NULL, buffer space is allocated automatically. The (third) plist argument can be used to specify the return data type of the array. If NULL is specified, the data type of the input file is used.

To process an image buffer, you would generally make a call to FunInfoGet() to determine the dimensions of the image (which may have been changed from the original file dimensions due to Funtools image sectioning on the command line). In a FITS image, the index along
the dim1 axis varies most rapidly, followed by the dim2 axis, etc.
Thus, to access each pixel in an 2D image, use a double loop such as:

```
buf = FunImageGet(fun, NULL, "bitpix=-32");
FunInfoGet(fun, FUN_SECT_DIM1, &dim1, FUN_SECT_DIM2, &dim2, 0);
for(i=1; i<=dim2; i++){
  for(j=1; j<=dim1; j++){
    ... process buf[((i-1)*dim1)+(j-1)] ...
  }
}
```

or:

```
buf = FunImageGet(fun, NULL, "bitpix=-32");
FunInfoGet(fun, FUN_SECT_DIM1, &dim1, FUN_SECT_DIM2, &dim2, 0);
for(i=0; i<(dim1*dim2); i++){
  ... process buf[i] ...
}
```

Finally, you can write the resulting image to disk using FunImagePut():

```
FunImagePut(fun2, buf, dim1, dim2, -32, NULL);
```

Note that Funtools automatically takes care of book-keeping tasks such as reading and writing FITS headers (although you can, of course, write your own header or add your own parameters to a header).

For binary tables and raw event files, a call to FunOpen() will be followed by a call to the FunColumnSelect() routine to select columns to be read from the input file and/or written to the output file:

```
typedef struct evstruct{
  double time;
  int time2;
} *Ev, EvRec;
FunColumnSelect(fun, sizeof(EvRec), NULL,
                "time",      "D",     "rw",  FUN_OFFSET(Ev, time),
                "time2",     "J",     "w",   FUN_OFFSET(Ev, time2),
                NULL);
```

Columns whose (third) mode argument contains an "r" are "readable", i.e., columns will be read from the input file and converted into the data type specified in the call's second argument. These columns values then are stored in the specified offset of the user record structure. Columns whose mode argument contains a "w" are "writable", i.e., these values will be written to the output file. The FunColumnSelect() routine also offers the option of automatically merging user columns with the original input columns when writing the output rows.

Once a set of columns has been specified, you can retrieve rows using FunTableRowGet(), and write the rows using FunTableRowPut():

```
Ev ebuf, ev;
/* get rows -- let routine allocate the array */
while( (ebuf = (Ev)FunTableRowGet(fun, NULL, MAXROW, NULL, &got)) ){
  /* process all rows */
  for(i=0; i<got; i++){
    /* point to the i'th row */
    ev = ebuf+i;
    /* time2 is generated here */
    ev->time2 = (int)(ev->time+.5);
    /* change the input time as well */
    ev->time = -(ev->time/10.0);
  }
  /* write out this batch of rows with the new column */
  FunTableRowPut(fun2, (char *)ebuf, got, 0, NULL);
  /* free row data */
  if( ebuf ) free(ebuf);
}
```

The input rows are retrieved into an array of user structs, which can be accessed serially as shown above. Once again, Funtools automatically takes care of book-keeping tasks such as reading and writing FITS headers (although you can, of course, write your own header or add your own parameters to a header).

When all processing is done, you can call FunClose() to close the file(s):

```
FunClose(fun2);
FunClose(fun);
```

These are the basics of processing FITS files (and arrays or raw event data) using Funtools. The routines in these examples are described in more detail below, along with a few other routines that support parameter access, data flushing, etc.

Compiling and Linking

To create a Funtools application, a software developer will include the funtools.h definitions file in Funtools code:

```
#include <funtools.h>
```

The program is linked against the libfuntools.a library, along with the math library (and the dynamic load library, if the latter is available on your system):

```
gcc -o foo foo.c -lfuntools -ldl -lm
```

If gcc is used, Funtools filtering can be performed using dynamically loaded shared objects that are built at run-time. Otherwise, filtering is performed using a slave process.

Funtools has been built on the following systems:

- Sun/Solaris 5.X
- Linux/RedHat Linux 5.X,6.X,7.X
- Dec Alpha/OSF1 V4.X
- WindowsNT/Cygwin 1.0
- SGI/IRIX64 6.5

A Short Digression on Subroutine Order

There is a natural order for all I/O access libraries. You would not think of reading a file without first opening it, or writing a file after closing it. A large part of the experiment in funtools is to use the idea of "natural order" as a means of making programming easier. We do this by maintaining the state of processing for a given funtools file, so that we can do things like write headers and flush extension padding at the right time, without you having to do it.

For example, if you open a new funtools file for writing using FunOpen(), then generate an array of image data and call FunImagePut(), funtools knows to write the image header automatically. There is no need to think about writing a standard header. Of course, you can add parameters to the file first by calling one of the FunParamPut() routines, and these parameters will automatically be added to the header when it is written out. There still is no need to write the header explicitly.

Maintaining state in this way means that there are certain rules of order which should be maintained in any funtools program. In particular, we strongly recommend the following ordering rules be adhered to:

- When specifying that input extensions be copied to an output file via a reference handle, open the output file **before** reading the input file. (Otherwise the initial copy will not occur).
- Always write parameters to an output file using one of the FunParamPut() calls **before** writing any data. (This is a good idea for all FITS libraries, to avoid having to recopy data is the FITS header needs to be extended by adding a single parameter.)
- If you retrieve an image, and need to know the data type, use the FUN_SECT_BITPIX option of FunInfoGet(), **after** calling FunImageGet(), since it is possible to change the value of BITPIX from the latter.
- When specifying that input extensions be copied to an output file via a reference handle, close the output file **before** closing input file, or else use FunFlush() explicitly on the output file **before** closing the input file. (Otherwise the final copy will not occur).

We believe that these are the natural rules that are implied in most FITS programming tasks. However, we recognize that making explicit use of "natural order" to decide what automatic action to take on behalf of the programmer is experimental. Therefore, if you find that your needs are not compatible with our preferred order, please let us know -- it will be most illuminating for us as we evaluate this experiment.

Funtools Programming Examples

The following complete coding examples are provided to illustrate the simplicity of Funtools applications. They can be found in the funtest subdirectory of the Funtools distribution. In many cases, you should be able to modify one of these programs to generate your own Funtools program:

- evread.c: read and write binary tables
- evcols.c: add column and rows to binary tables
- evmerge.c: merge new columns with existing columns
- evnext.c: manipulate raw data pointers
- imblank.c: blank out image values below a threshold
- asc2fits.c: convert a specific ASCII table to FITS binary table

The Funtools Programming Reference Manual

```
#include <funtools.h>

Fun FunOpen(char *name, char *mode, Fun ref)

void *FunImageGet(Fun fun, void *buf, char *plist)

int FunImagePut(Fun fun, void *buf, int dim1, int dim2, int bitpix, char *plist)

void * FunImageRowGet(Fun fun, void *buf, int rstart, int rstop, char *plist)

void * FunImageRowPut(Fun fun, void *buf, int rstart, int rstop, int dim1, int dim2, int bitpix, char *plist)

int FunColumnSelect(Fun fun, int size, char *plist, ...)

void FunColumnActivate(Fun fun, char *s, char *plist)

int FunColumnLookup(Fun fun, char *s, int which, char **name, int *type, int *mode, int *offset, int *n, int *width)
```

void *FunTableRowGet(Fun fun, void *rows, int maxrow, char *plist, int *nrow)

int FunTableRowPut(Fun fun, void *rows, int nev, int idx, char *plist)

int FunParamGetb(Fun fun, char *name, int n, int defval, int *got)

int FunParamGeti(Fun fun, char *name, int n, int defval, int *got)

double FunParamGetd(Fun fun, char *name, int n, double defval, int *got)

char *FunParamGets(Fun fun, char *name, int n, char *defval, int *got)

int FunParamPutb(Fun fun, char *name, int n, int value, char *comm, int append)

int FunParamPuti(Fun fun, char *name, int n, int value, char *comm, int append)

int FunParamPutd(Fun fun, char *name, int n, double value, int prec, char *comm, int append)

int FunParamPuts(Fun fun, char *name, int n, char *value, char *comm, int append)

int FunInfoGet(Fun fun, int type, ...)

int FunInfoPut(Fun fun, int type, ...)

void FunFlush(Fun fun, char *plist)

void FunClose(Fun fun)

FunOpen - open a Funtools data file

```
#include <funtools.h>

Fun FunOpen(char *name, char *mode, Fun ref);
```

The **FunOpen()** routine opens a Funtools data file for reading or appending, or creates a new FITS file for writing. The **name** argument specifies the name of the Funtools data file to open. You can use IRAF-style bracket notation to specify Funtools Files, Extensions, and Filters. A separate call should be made each time a different FITS extension is accessed:

```
Fun fun;
char *iname;
...
if( !(fun = FunOpen(iname, "r", NULL)) ){
  fprintf(stderr, "could not FunOpen input file: %s\n", iname);
  exit(1);
}
```

If **mode** is "r", the file is opened for reading, and processing is set up to begin at the specified extension. For reading, **name** can be **stdin**, in which case the standard input is read.

If **mode** is "w", the file is created if it does not exist, or opened and truncated for writing if it does exist. Processing starts at the beginning of the file. The **name** can be **stdout**, in which case the standard output is readied for processing.

If **mode** is "a", the file is created if it does not exist, or opened if it does exist. Processing starts at the end of the file. The **name** can be **stdout**, in which case the standard output is readied for processing.

When a Funtools file is opened for writing or appending, a previously opened Funtools reference handle can be specified as the third argument. This handle typically is associated with the input Funtools file that will be used to generate the data for the output data. When a reference file is specified in this way, the output file will inherit the (extension) header parameters from the input file:

```
Fun fun, fun2;
...
/* open input file */
if( !(fun = FunOpen(argv[1], "r", NULL)) )
  gerror(stderr, "could not FunOpen input file: %s\n", argv[1]);
/* open the output FITS image, inheriting params from input */
if( !(fun2 = FunOpen(argv[2], "w", fun)) )
  gerror(stderr, "could not FunOpen output file: %s\n", argv[2]);
```

Thus, in the above example, the output FITS binary table file will inherit all of the parameters associated with the input binary table extension.

A file opened for writing with a Funtools reference handle also inherits the selected columns (i.e. those columns chosen for processing using the FunColumnSelect() routine) from the reference file as its default columns. This makes it easy to open an output file in such a way that the columns written to the output file are the same as the columns read in the input file. Of course, column selection can easily be tailored using the FunColumnSelect() routine. In particular, it is easy to merge user-defined columns with the input columns to generate a new file. See the evmerge for a complete example.

In addition, when a Funtools reference handle is supplied in a FunOpen() call, it is possible also to specify that all other extensions from the reference file (other than the input extension being processed) should be copied from the reference file to the output file. This is useful, for example, in a case where you are processing a FITS binary table or image and you want to copy all of the other extensions to the output file as well. Copy of other extensions is controlled by adding a "C" or "c" to the mode string of the FunOpen() call **of the input reference file**. If "C" is specified, then other extensions are **always** copied (i.e., copy is forced by the application). If "c" is used, then other extensions are copied if the user requests copying by adding a plus sign "+" to the extension name in the bracket specification. For example, the **funtable** program utilizes "c" mode, giving users the option of copying all other extensions:

```
/* open input file -- allow user copy of other extensions */
if( !(fun = FunOpen(argv[1], "rc", NULL)) )
  gerror(stderr, "could not FunOpen input file: %s\n", argv[1]);
/* open the output FITS image, inheriting params from input */
if( !(fun2 = FunOpen(argv[2], "w", fun)) )
  gerror(stderr, "could not FunOpen output file: %s\n", argv[2]);
```

Thus, **funtable** supports either of these command lines:

```
# copy only the EVENTS extension
csh> funtable "test.ev[EVENTS,circle(512,512,10)]" foo.ev
# copy ALL extensions
csh> funtable "test.ev[EVENTS+,circle(512,512,10)]" foo.ev
```

Use of a Funtools reference handle implies that the input file is opened before the output file. However, it is important to note that if copy mode ("c" or "C") is specified for the input file, the actual input file open is delayed until just after the output file is opened, since the copy of prior extensions to the output file

takes place while Funtools is seeking to the specified input extension. This implies that the output file should be opened before any I/O is done on the input file or else the copy will fail. Note also that the copy of subsequent extension will be handled automatically by FunClose() if the output file is closed before the input file. Alternatively, it can be done explicitly by FunFlush(), but again, this assumes that the input file still is open.

Upon success FunOpen() returns a Fun handle that is used in subsequent Funtools calls. On error, NULL is returned.

FunImageGet - get an image or image section

```
#include <funtools.h>

void *FunImageGet(Fun fun, void *buf, char *plist)
```

The **FunImageGet()** routine returns an binned image array of the specified section of a Funtools data file. If the input data are already of type image, the array is generated by extracting the specified image section and then binning it according to the specified bin factor. If the input data are contained in a binary table or raw event file, the rows are binned on the columns specified by the **bincols=** keyword (using appropriate default columns as necessary), after which the image section and bin factors are applied. In both cases, the data is automatically converted from FITS to native format, if necessary.

The first argument is the Funtools handle returned by FunOpen(). The second **buf** argument is a pointer to a data buffer to fill. If NULL is specified, FunImageGet will allocate a buffer of the appropriate size. Generally speaking, you always want Funtools to allocate the buffer because the image dimensions will be determined by Funtools image sectioning on the command line.

The third **plist** (i.e., parameter list) argument is a string containing one or more comma-delimited **keyword=value** parameters. It can be used to specify the return data type using the **bitpix=** keyword. If no such keyword is specified in the plist string, the data type of the returned image is the same as the data type of the original input file, or is of type int for FITS binary tables.

If the **bitpix=** keyword is supplied in the plist string, the data type of the returned image will be one of the supported FITS image data types:

- 8 unsigned char
- 16 short
- 32 int
- -32 float
- -64 double

For example:

```
void *buf;
/* extract data section into an image buffer */
if( !(buf = FunImageGet(fun, NULL, NULL)) )
  gerror(stderr, "could not FunImageGet: %s\n", iname);
```

will allocate buf and retrieve the image in the file data format. In this case, you will have to determine the

data type (using the FUN_SECT_BITPIX value in the FunInfoGet() routine) and then use a switch statement to process each data type:

```
  int bitpix;
  void *buf;
  unsigned char *cbuf;
  short *sbuf;
  int *ibuf;
  ...
  buf = FunImageGet(fun, NULL, NULL);
  FunInfoGet(fun, FUN_SECT_BITPIX,  &bitpix, 0);
  /* set appropriate data type buffer to point to image buffer */
  switch(bitpix){
  case 8:
    cbuf = (unsigned char *)buf; break;
  case 16:
    sbuf = (short *)buf; break;
  case 32:
    ibuf = (int *)buf; break;
 ...
```

See the imblank example code for more details on how to process an image when the data type is not specified beforehand.

It often is easier to specify the data type directly:

```
  double *buf;
  /* extract data section into a double image buffer */
  if( !(buf = FunImageGet(fun, NULL, "bitpix=-64")) )
    gerror(stderr, "could not FunImageGet: %s\n", iname);
```

will extract the image while converting to type double.

On success, a pointer to the image buffer is returned. (This will be the same as the second argument, if NULL is not passed to the latter.) On error, NULL is returned.

In summary, to retrieve image or row data into a binned image, you simply call FunOpen() followed by FunImageGet(). Generally, you then will want to call FunInfoGet() to retrieve the axis dimensions (and data type) of the section you are processing (so as to take account of sectioning and blocking of the original data):

```
  double *buf;
  int i, j;
  int dim1, dim2;
  ... other declarations, etc.

  /* open the input FITS file */
  if( !(fun = FunOpen(argv[1], "rc", NULL)) )
    gerror(stderr, "could not FunOpen input file: %s\n", argv[1]);

  /* extract and bin the data section into a double float image buffer */
  if( !(buf = FunImageGet(fun, NULL, "bitpix=-64")) )
    gerror(stderr, "could not FunImageGet: %s\n", argv[1]);

  /* get dimension information from funtools structure */
```

```
FunInfoGet(fun, FUN_SECT_DIM1, &dim1, FUN_SECT_DIM2, &dim2, 0);

/* loop through pixels and reset values below limit to value */
for(i=0; i<dim1*dim2; i++){
  if( buf[i] <= blimit ) buf[i] = bvalue;
}
```

Another useful plist string value is "mask=all", which returns an image populated with regions id values. Image pixels within a region will contain the associated region id (region values start at 1), and otherwise will contain a 0 value. Thus, the returned image is a region mask which can be used to process the image data (which presumably is retrieved by a separate call to FunImageGet) pixel by pixel.

If a FITS binary table or a non-FITS raw event file is being binned into an image, it is necessary to specify the two columns that will be used in the 2D binning. This usually is done on the command line using the **bincols=(x,y)** keyword:

```
funcnts "foo.ev[EVENTS,bincols=(detx,dety)]"
```

The full form of the **bincols=** specifier is:

```
bincols=([xname[:tlmin[:tlmax:[binsiz]]]],[yname[:tlmin[:tlmax[:binsiz]]]])
```

where the tlmin, tlmax, and binsiz specifiers determine the image binning dimensions:

```
dim = (tlmax - tlmin)/binsiz     (floating point data)
dim = (tlmax - tlmin)/binsiz + 1 (integer data)
```

These tlmin, tlmax, and binsiz specifiers can be omitted if TLMIN, TLMAX, and TDBIN header parameters (respectively) are present in the FITS binary table header for the column in question. Note that if only one parameter is specified, it is assumed to be tlmax, and tlmin defaults to 1. If two parameters are specified, they are assumed to be tlmin and tlmax.

If **bincols** is not specified on the command line, Funtools tries to use appropriate defaults: it looks for the environment variable FITS_BINCOLS (or FITS_BINKEY). Then it looks for the Chandra parameters CPREF (or PREFX) in the FITS binary table header. Failing this, it looks for columns named "X" and "Y" and if these are not found, it looks for columns containing the characters "X" and "Y".

See Binning FITS Binary Tables and Non-FITS Event Files for more information.

FunImagePut - put an image to a Funtools file

```
#include <funtools.h>

int FunImagePut(Fun fun, void *buf, int dim1, int dim2, int bitpix,
                char *plist)
```

The **FunImagePut()** routine outputs an image array to a FITS file. The image is written either as a primary header/data unit or as an image extension, depending on whether other data have already been written to the file. That is, if the current file position is at the beginning of the file, a primary HDU is written. Otherwise, an image extension is written.

The first argument is the Funtools handle returned by FunOpen(). The second **buf** argument is a pointer to a data buffer to write. The **dim1**and **dim2** arguments that follow specify the dimensions of the image, where dim1 corresponds to naxis1 and dim2 corresponds to naxis2. The **bitpix** argument specifies the data type of the image and can have the following FITS-standard values:

- 8 unsigned char
- 16 short
- 32 int
- -32 float
- -64 double

When FunTableRowPut() is first called for a given image, Funtools checks to see if the primary header has already been written (by having previously written an image or a binary table.) If not, this image is written to the primary HDU. Otherwise, it is written to an image extension.

Thus, a simple program to generate a FITS image might look like this:

```
int i;
int dim1=512, dim2=512;
double *dbuf;
Fun fun;
dbuf = malloc(dim1*dim2*sizeof(double));
/* open the output FITS image, preparing to copy input params */
if( !(fun = FunOpen(argv[1], "w", NULL)) )
  gerror(stderr, "could not FunOpen output file: %s\n", argv[1]);
for(i=0; i<(dim1*dim2); i++){
  ... fill dbuf ...
}
/* put the image (header will be generated automatically */
if( !FunImagePut(fun, buf, dim1, dim2, -64, NULL) )
  gerror(stderr, "could not FunImagePut: %s\n", argv[1]);
FunClose(fun);
free(dbuf);
```

In addition, if a Funtools reference handle was specified when this table was opened, the parameters from this Funtools reference handle are merged into the new image header. Furthermore, if a reference image was specified during FunOpen(), the values of **dim1**, **dim2**, and **bitpix** in the calling sequence can all be set to 0. In this case, default values are taken from the reference image section. This is useful if you are reading an image section in its native data format, processing it, and then writing that section to a new FITS file. See the imblank example code.

The data are assumed to be in the native machine format and will automatically be swapped to FITS big-endian format if necessary. This behavior can be over-ridden with the **convert=[true|false]** keyword in the **plist** param list string.

When you are finished writing the image, you should call FunFlush() to write out the FITS image padding. However, this is not necessary if you subsequently call FunClose() without doing any other I/O to the FITS file.

FunImageRowGet - get row(s) of an image

```
#include <funtools.h>

void *FunImageRowGet(Fun fun, void *buf, int rstart, int rstop,
                     char *plist)
```

The **FunImageRowGet()** routine returns one or more image rows from the specified section of a Funtools data file. If the input data are of type image, the array is generated by extracting the specified image rows and then binning them according to the specified bin factor. If the input data are contained in a binary table or raw event file, the rows are binned on the columns specified by the **bincols=** keyword (using appropriate default columns as needed), after which the image section and bin factors are applied.

The first argument is the Funtools handle returned by FunOpen(). The second **buf** argument is a pointer to a data buffer to fill. If NULL is specified, FunImageGet() will allocate a buffer of the appropriate size.

The third and fourth arguments specify the first and last row to retrieve. Rows are counted starting from 1, up to the value of FUN_YMAX(fun). The final **plist** (i.e., parameter list) argument is a string containing one or more comma-delimited **keyword=value** parameters. It can be used to specify the return data type using the **bitpix=** keyword. If no such keyword is specified in the plist string, the data type of the image is the same as the data type of the original input file, or is of type int for FITS binary tables.

If the **bitpix=**value is supplied in the plist string, the data type of the returned image will be one of the supported FITS image data types:

- 8 unsigned char
- 16 short
- 32 int
- -32 float
- -64 double

For example:

```
double *drow;
Fun fun;
... open files ...
/* get section dimensions */
FunInfoGet(fun, FUN_SECT_DIM1, &dim1, FUN_SECT_DIM2, &dim2, 0);
/* allocate one line's worth */
drow = malloc(dim1*sizeof(double));
/* retrieve and process each input row (starting at 1) */
for(i=1; i <= dim2; i++){
  if( !FunImageRowGet(fun, drow, i, i, "bitpix=-64") )
    gerror(stderr, "can't FunImageRowGet: %d %s\n", i, iname);
    /* reverse the line */
    for(j=1; j<=dim1; j++){
      ... process drow[j-1] ...
    }
}
...
```

On success, a pointer to the image buffer is returned. (This will be the same as the second argument, if NULL is not passed to the latter.) On error, NULL is returned. Note that the considerations described above for specifying binning columns in FunImageGet() also apply to **FunImageRowGet()**.

FunImageRowPut - put row(s) of an image

```
#include <funtools.h>

void *FunImageRowPut(Fun fun, void *buf, int rstart, int rstop,
                     int dim1, int dim2, int bitpix, char *plist)
```

The **FunImageRowPut()** routine writes one or more image rows to the specified FITS image file. The first argument is the Funtools handle returned by FunOpen(). The second **buf** argument is a pointer to the row data buffer, while the third and fourth arguments specify the starting and ending rows to write. Valid rows values range from 1 to dim2, i.e., row is one-valued.

The **dim1**and **dim2** arguments that follow specify the dimensions, where dim1 corresponds to naxis1 and dim2 corresponds to naxis2. The **bitpix** argument data type of the image and can have the following FITS-standard values:

- 8 unsigned char
- 16 short
- 32 int
- -32 float
- -64 double

For example:

```
double *drow;
Fun fun, fun2;
... open files ...
/* get section dimensions */
FunInfoGet(fun, FUN_SECT_DIM1, &dim1, FUN_SECT_DIM2, &dim2, 0);
/* allocate one line's worth */
drow = malloc(dim1*sizeof(double));
/* retrieve and process each input row (starting at 1) */
for(i=1; i <= dim2; i++){
  if( !FunImageRowGet(fun, drow, i, i, "bitpix=-64") )
    gerror(stderr, "can't FunImageRowGet: %d %s\n", i, iname);
  ... process drow ...
  if( !FunImageRowPut(fun2, drow, i, i, 64, NULL) )
    gerror(stderr, "can't FunImageRowPut: %d %s\n", i, oname);
}
...
```

The data are assumed to be in the native machine format and will automatically be swapped to big-endian FITS format if necessary. This behavior can be over-ridden with the **convert=[true|false]** keyword in the **plist** param list string.

When you are finished writing the image, you should call FunFlush() to write out the FITS image padding. However, this is not necessary if you subsequently call FunClose() without doing any other I/O to the FITS file.

FunColumnSelect - select Funtools columns

```
#include <funtools.h>

int FunColumnSelect(Fun fun, int size, char *plist,
                    char *name1, char *type1, char *mode1, int offset1,
                    char *name2, char *type2, char *mode2, int offset2,
                    ...,
                    NULL)

int FunColumnSelectArr(Fun fun, int size, char *plist,
                       char **names, char **types, char **modes,
                       int *offsets, int nargs);
```

The **FunColumnSelect()** routine is used to select the columns from a Funtools binary table extension or raw event file for processing. This routine allows you to specify how columns in a file are to be read into a user record structure or written from a user record structure to an output FITS file.

The first argument is the Fun handle associated with this set of columns. The second argument specifies the size of the user record structure into which columns will be read. Typically, the sizeof() macro is used to specify the size of a record structure. The third argument allows you to specify keyword directives for the selection and is described in more detail below.

Following the first three required arguments is a variable length list of column specifications. Each column specification will consist of four arguments:

- **name**: the name of the column
- **type**: the data type of the column as it will be stored in the user record struct (not the data type of the input file). The following basic data types are recognized:
  - A: ASCII characters
  - B: unsigned 8-bit char
  - I: signed 16-bit int
  - U: unsigned 16-bit int (not standard FITS)
  - J: signed 32-bit int
  - V: unsigned 32-bit int (not standard FITS)
  - E: 32-bit float
  - D: 64-bit float

  The syntax used is similar to that which defines the TFORM parameter in FITS binary tables. That is, a numeric repeat value can precede the type character, so that "10I" means a vector of 10 short ints, "E" means a single precision float, etc. Note that the column value from the input file will be converted to the specified data type as the data is read by FunTableRowGet().

[A short digression regarding bit-fields: Special attention is required when reading or writing the FITS bit-field type ("X"). Bit-fields almost always have a numeric repeat character preceding the 'X' specification. Usually this value is a multiple of 8 so that bit-fields fit into an integral number of bytes. For all cases, the byte size of the bit-field B is (N+7)/8, where N is the numeric repeat character.

A bit-field is most easily declared in the user struct as an array of type char of size B as defined above. In this case, bytes are simply moved from the file to the user space. If, instead, a short or int scalar or array is used, then the algorithm for reading the bit-field into the user space depends on the size of the data type used along with the value of the repeat character. That is, if the user data size is equal to the byte size of the bit-field, then the data is simply moved (possibly with endian-based byte-swapping) from one to the other. If, on the other hand, the data storage is larger than the bit-field size, then a data type cast conversion is performed to move parts of the bit-field into elements of the array. Examples will help make this clear:

- If the file contains a 16X bit-field and user space specifies a 2B char array[2], then the bit-field is moved directly into the char array.
- If the file contains a 16X bit-field and user space specifies a 1I scalar short int, then the bit-field is moved directly into the short int.
- If the file contains a 16X bit-field and user space specifies a 1J scalar int, then the bit-field is type-cast to unsigned int before being moved (use of unsigned avoids possible sign extension).
- If the file contains a 16X bit-field and user space specifies a 2J int array[2], then the bit-field is handled as 2 chars, each of which are type-cast to unsigned int before being moved (use of unsigned avoids possible sign extension).
- If the file contains a 16X bit-field and user space specifies a 1B char, then the bit-field is treated as a char, i.e., truncation will occur.
- If the file contains a 16X bit-field and user space specifies a 4J int array[4], then the results are undetermined.

For all user data types larger than char, the bit-field is byte-swapped as necessary to convert to native format, so that bits in the resulting data in user space can be tested, masked, etc. in the same way regardless of platform.]

In addition to setting data type and size, the **type** specification allows a few ancillary parameters to be set, using the full syntax for **type**:

```
[@][n]<type>[[['B']poff]][:[tlmin[:tlmax[:binsiz]]]]
```

The special character "@" can be prepended to this specification to indicated that the data element is a pointer in the user record, rather than an array stored within the record.

The [n] value is an integer that specifies the number of elements that are in this column (default is 1). TLMIN, TLMAX, and BINSIZ values also can be specified for this column after the type, separated by colons. If only one such number is specified, it is assumed to be TLMAX, and TLMIN and BINSIZ are set to 1.

The [poff] value can be used to specify the offset into an array. By default, this offset value is set to zero and the data specified starts at the beginning of the array. The offset usually is specified in terms of the data type of the column. Thus an offset specification of [5] means a 20-byte offset if the data type is a 32-bit integer, and a 40-byte offset for a double. If you want to specify a byte offset instead

of an offset tied to the column data type, precede the offset value with 'B', e.g. [B6] means a 6-bye offset, regardless of the column data type. The [poff] is especially useful in conjunction with the pointer @ specification, since it allows the data element to anywhere stored anywhere in the allocated array. For example, a specification such as "@I[2]" specifies the third (i.e., starting from 0) element in the array pointed to by the pointer value. A value of "@2I[4]" specifies the fifth and sixth values in the array. For example, consider the following specification:

```
typedef struct EvStruct{
  short x[4], *atp;
} *Event, EventRec;
/* set up the (hardwired) columns */
FunColumnSelect( fun, sizeof(EventRec), NULL,
                 "2i",    "2I  ",    "w", FUN_OFFSET(Event, x),
                 "2i2",   "2I[2]",   "w", FUN_OFFSET(Event, x),
                 "at2p",  "@2I",     "w", FUN_OFFSET(Event, atp),
                 "at2p4", "@2I[4]",  "w", FUN_OFFSET(Event, atp),
                 "atp9",  "@I[9]",   "w", FUN_OFFSET(Event, atp),
                 "atb20", "@I[B20]", "w", FUN_OFFSET(Event, atb),
                 NULL);
```

Here we have specified the following columns:

- 2i: two short ints in an array which is stored as part the record
- 2i2: the 3rd and 4th elements of an array which is stored as part of the record
- an array of at least 10 elements, not stored in the record but allocated elsewhere, and used by three different columns:
  - at2p: 2 short ints which are the first 2 elements of the allocated array
  - at2p4: 2 short ints which are the 5th and 6th elements of the allocated array
  - atp9: a short int which is the 10th element of the allocated array
- atb20: a short int which is at byte offset 20 of another allocated array

In this way, several columns can be specified, all of which are in a single array. **NB**: it is the programmer's responsibility to ensure that specification of a positive value for poff does not point past the end of valid data.

- **read/write mode**: "r" means that the column is read from an input file into user space by FunTableRowGet(), "w" means that the column is written to an output file. Both can specified at the same time.
- **offset**: the offset into the user data to store this column. Typically, the macro FUN_OFFSET(recname, colname) is used to define the offset into a record structure.

When all column arguments have been specified, a final NULL argument must added to signal the column selection list.

As an alternative to the varargs FunColumnSelect() routine, a non-varargs routine called FunColumnSelectArr() also is available. The first three arguments (fun, size, plist) of this routine are the same as in FunColumnSelect(). Instead of a variable argument list, however, FunColumnSelectArr() takes 5 additional arguments. The first 4 arrays arguments contain the names, types, modes, and offsets, respectively, of the columns being selected. The final argument is the number of columns that are contained in these arrays. It is the user's responsibility to free string space allocated in these arrays.

Consider the following example:

```
typedef struct evstruct{
  int status;
  float pi, pha, *phas;
  double energy;
} *Ev, EvRec;

FunColumnSelect(fun, sizeof(EvRec), NULL,
  "status",  "J",     "r",   FUN_OFFSET(Ev, status),
  "pi",      "E",     "r",  FUN_OFFSET(Ev, pi),
  "pha",     "E",     "r",  FUN_OFFSET(Ev, pha),
  "phas",    "@9E",   "r",  FUN_OFFSET(Ev, phas),
  NULL);
```

Each time a row is read into the Ev struct, the "status" column is converted to an int data type (regardless of its data type in the file) and stored in the status value of the struct. Similarly, "pi" and "pha", and the phas vector are all stored as floats. Note that the "@" sign indicates that the "phas" vector is a pointer to a 9 element array, rather than an array allocated in the struct itself. The row record can then be processed as required:

```
/* get rows -- let routine allocate the row array */
while( (ebuf = (Ev)FunTableRowGet(fun, NULL, MAXROW, NULL, &got)) ){
  /* process all rows */
  for(i=0; i<got; i++){
    /* point to the i'th row */
    ev = ebuf+i;
    ev->pi = (ev->pi+.5);
    ev->pha = (ev->pi-.5);
  }
```

FunColumnSelect() can also be called to define "writable" columns in order to generate a FITS Binary Table, without reference to any input columns. For example, the following will generate a 4-column FITS binary table when FunTableRowPut() is used to write Ev records:

```
typedef struct evstruct{
  int status;
  float pi, pha
  double energy;
} *Ev, EvRec;

FunColumnSelect(fun, sizeof(EvRec), NULL,
  "status",  "J",     "w",   FUN_OFFSET(Ev, status),
  "pi",      "E",     "w",   FUN_OFFSET(Ev, pi),
  "pha",     "E",     "w",   FUN_OFFSET(Ev, pha),
  "energy",  "D",       "w",   FUN_OFFSET(Ev, energy),
  NULL);
```

All columns are declared to be write-only, so presumably the column data is being generated or read from some other source.

In addition, FunColumnSelect() can be called to define **both** "readable" and "writable" columns. In this case, the "read" columns are associated with an input file, while the "write" columns are associated with the output file. Of course, columns can be specified as both "readable" and "writable", in which case they

are read from input and (possibly modified data values are) written to the output. The FunColumnSelect() call itself is made by passing the input Funtools handle, and it is assumed that the output file has been opened using this input handle as its Funtools reference handle.

Consider the following example:

```
typedef struct evstruct{
  int status;
  float pi, pha, *phas;
  double energy;
} *Ev, EvRec;

FunColumnSelect(fun, sizeof(EvRec), NULL,
  "status",  "J",     "r",   FUN_OFFSET(Ev, status),
  "pi",      "E",     "rw",  FUN_OFFSET(Ev, pi),
  "pha",     "E",     "rw",  FUN_OFFSET(Ev, pha),
  "phas",    "@9E",   "rw",  FUN_OFFSET(Ev, phas),
  "energy",  "D",     "w",   FUN_OFFSET(Ev, energy),
  NULL);
```

As in the "read" example above, each time an row is read into the Ev struct, the "status" column is converted to an int data type (regardless of its data type in the file) and stored in the status value of the struct. Similarly, "pi" and "pha", and the phas vector are all stored as floats. Since the "pi", "pha", and "phas" variables are declared as "writable" as well as "readable", they also will be written to the output file. Note, however, that the "status" variable is declared as "readable" only, and hence it will not be written to an output file. Finally, the "energy" column is declared as "writable" only, meaning it will not be read from the input file. In this case, it can be assumed that "energy" will be calculated in the program before being output along with the other values.

In these simple cases, only the columns specified as "writable" will be output using FunTableRowPut(). However, it often is the case that you want to merge the user columns back in with the input columns, even in cases where not all of the input column names are explicitly read or even known. For this important case, the **merge=[type]** keyword is provided in the plist string.

The **merge=[type]** keyword tells Funtools to merge the columns from the input file with user columns on output. It is normally used when an input and output file are opened and the input file provides the Funtools reference handle for the output file. In this case, each time FunTableRowGet() is called, the raw input rows are saved in a special buffer. If FunTableRowPut() then is called (before another call to FunTableRowGet()), the contents of the raw input rows are merged with the user rows according to the value of **type** as follows:

- **update**: add new user columns, and update value of existing ones (maintaining the input data type)
- **replace**: add new user columns, and replace the data type and value of existing ones. (Note that if tlmin/tlmax values are not specified in the replacing column, but are specified in the original column being replaced, then the original tlmin/tlmax values are used in the replacing column.)
- **append**: only add new columns, do not "replace" or "update" existing ones

Consider the example above. If **merge=update** is specified in the plist string, then "energy" will be added to the input columns, and the values of "pi", "pha", and "phas" will be taken from the user space (i.e., the values will be updated from the original values, if they were changed by the program). The data type for

"pi", "pha", and "phas" will be the same as in the original file. If **merge=replace** is specified, both the data type and value of these three input columns will be changed to the data type and value in the user structure. If **merge=append** is specified, none of these three columns will be updated, and only the "energy" column will be added. Note that in all cases, "status" will be written from the input data, not from the user record, since it was specified as read-only.

Standard applications will call FunColumnSelect() to define user columns. However, if this routine is not called, the default behavior is to transfer all input columns into user space. For this purpose a default record structure is defined such that each data element is properly aligned on a valid data type boundary. This mechanism is used by programs such as fundisp and funtable to process columns without needing to know the specific names of those columns. It is not anticipated that users will need such capabilities (contact us if you do!)

By default, FunColumnSelect() reads/writes rows to/from an "array of structs", where each struct contains the column values for a single row of the table. This means that the returned values for a given column are not contiguous. You can set up the IO to return a "struct of arrays" so that each of the returned columns are contiguous by specifying **org=structofarrays** (abbreviation: **org=soa**) in the plist. (The default case is **org=arrayofstructs** or **org=aos**.)

For example, the default setup to retrieve rows from a table would be to define a record structure for a single event and then call FunColumnSelect() as follows:

```
typedef struct evstruct{
  short region;
  double x, y;
  int pi, pha;
  double time;
} *Ev, EvRec;

got = FunColumnSelect(fun, sizeof(EvRec), NULL,
                      "x",       "D:10:10", mode, FUN_OFFSET(Ev, x),
                      "y",       "D:10:10", mode, FUN_OFFSET(Ev, y),
                      "pi",      "J",       mode, FUN_OFFSET(Ev, pi),
                      "pha",     "J",       mode, FUN_OFFSET(Ev, pha),
                      "time",    "1D",      mode, FUN_OFFSET(Ev, time),
                      NULL);
```

Subsequently, each call to FunTableRowGet() will return an array of structs, one for each returned row. If instead you wanted to read columns into contiguous arrays, you specify **org=soa**:

```
typedef struct aevstruct{
  short region[MAXROW];
  double x[MAXROW], y[MAXROW];
  int pi[MAXROW], pha[MAXROW];
  double time[MAXROW];
} *AEv, AEvRec;

got = FunColumnSelect(fun, sizeof(AEvRec), "org=soa",
                    "x",       "D:10:10", mode, FUN_OFFSET(AEv, x),
                    "y",       "D:10:10", mode, FUN_OFFSET(AEv, y),
```

```
                    "pi",       "J",        mode, FUN_OFFSET(AEv, pi),
                    "pha",      "J",        mode, FUN_OFFSET(AEv, pha),
                    "time",     "1D",       mode, FUN_OFFSET(AEv, time),
                    NULL);
```

Note that the only modification to the call is in the plist string.

Of course, instead of using staticly allocated arrays, you also can specify dynamically allocated pointers:

```
/* pointers to arrays of columns (used in struct of arrays) */
typedef struct pevstruct{
  short *region;
  double *x, *y;
  int *pi, *pha;
  double *time;
} *PEv, PEvRec;

got = FunColumnSelect(fun, sizeof(PEvRec), "org=structofarrays",
                    "$region", "@I",        mode, FUN_OFFSET(PEv, region),
                    "x",       "@D:10:10",  mode, FUN_OFFSET(PEv, x),
                    "y",       "@D:10:10",  mode, FUN_OFFSET(PEv, y),
                    "pi",      "@J",        mode, FUN_OFFSET(PEv, pi),
                    "pha",     "@J",        mode, FUN_OFFSET(PEv, pha),
                    "time",    "@1D",       mode, FUN_OFFSET(PEv, time),
                    NULL);
```

Here, the actual storage space is either allocated by the user or by the FunColumnSelect() call).

In all of the above cases, the same call is made to retrieve rows, e.g.:

```
  buf = (void *)FunTableRowGet(fun, NULL, MAXROW, NULL, &got);
```

However, the individual data elements are accessed differently. For the default case of an "array of structs", the individual row records are accessed using:

```
for(i=0; i<got; i++){
  ev = (Ev)buf+i;
  fprintf(stdout, "%.2f\t%.2f\t%d\t%d\t%.4f\t%.4f\t%21.8f\n",
          ev->x, ev->y, ev->pi, ev->pha, ev->dx, ev->dy, ev->time);
}
```

For a struct of arrays or a struct of array pointers, we have a single struct through which we access individual columns and rows using:

```
aev = (AEv)buf;
for(i=0; i<got; i++){
  fprintf(stdout, "%.2f\t%.2f\t%d\t%d\t%.4f\t%.4f\t%21.8f\n",
          aev->x[i], aev->y[i], aev->pi[i], aev->pha[i],
          aev->dx[i], aev->dy[i], aev->time[i]);
}
```

Support for struct of arrays in the FunTableRowPut() call is handled analogously.

See the evread example code and evmerge example code for working examples of how FunColumnSelect() is used.

FunColumnActivate - activate Funtools columns

```
#include <funtools.h>

void FunColumnActivate(Fun fun, char *s, char *plist)
```

The **FunColumnActivate()** routine determines which columns (set up by FunColumnSelect()) ultimately will be read and/or written. By default, all columns that are selected using FunColumnSelect() are activated. The FunColumnActivate() routine can be used to turn off/off activation of specific columns.

The first argument is the Fun handle associated with this set of columns. The second argument is a space-delimited list of columns to activate or de-activate. Columns preceded by "+" are activated and columns preceded by a "-" are de-activated. If a column is named without "+" or "-", it is activated. The reserved strings "$region" and '$n' are used to activate a special columns containing the filter region value and row value, respectively, associated with this row. For example, if a filter containing two circular regions is specified as part of the Funtools file name, this column will contain a value of 1 or 2, depending on which region that row was in. The reserved strings "$x" and "$y" are used to activate the current binning columns. Thus, if the columns DX and DY are specified as binning columns:

```
[sh $] fundisp foo.fits[bincols=(DX,DY)]
```

then "$x" and "$y" will refer to these columns in a call to FunColumnActivate().

In addition, if the activation string contains only columns to be activated, then the routine will de-activate all other columns. Similarly, if the activation string contains only columns to de-activate, then the routine will activate all other columns before activating the list. This makes it simple to change the activation state of all columns without having to know all of the column names. For example:

- **"pi pha time"** # only these three columns will be active
- **"-pi -pha -time"** # all but these columns will be active
- **"pi -pha"** # only pi is active, pha is not, others are not
- **"+pi -pha"** # same as above
- **"pi -pha -time"** # only pi is active, all others are not
- **"pi pha"** # pha and pi are active, all others are not
- **"pi pha -x -y"** # pha and pi are active, all others are not

You can use the column activation list to reorder columns, since columns are output in the order specified. For example:

```
# default output order
fundisp snr.ev'[cir 512 512 .1]'
       X        Y      PHA       PI                   TIME       DX       DY
-------- -------- -------- -------- -------------------- -------- --------
     512      512        6        7     79493997.45854475      578      574
     512      512        8        9     79494575.58943175      579      573
     512      512        5        6     79493631.03866175      578      575
     512      512        5        5     79493290.86521725      578      575
```

```
     512      512        8        9     79493432.00990875      579      573

# re-order the output by specifying explicit order
fundisp snr.ev'[cir 512 512 .1]' "time x y dy dx pi pha"
                  TIME        X        Y       DY       DX       PI      PHA
--------------------- -------- -------- -------- -------- -------- --------
    79493997.45854475      512      512      574      578        7        6
    79494575.58943175      512      512      573      579        9        8
    79493631.03866175      512      512      575      578        6        5
    79493290.86521725      512      512      575      578        5        5
    79493432.00990875      512      512      573      579        9        8
```

A "+" sign by itself means to activate all columns, so that you can reorder just a few columns without specifying all of them:

```
# reorder 3 columns and then output the rest
fundisp snr.ev'[cir 512 512 .1]' "time pi pha +"
                  TIME       PI      PHA        Y        X       DX       DY
--------------------- -------- -------- -------- -------- -------- --------
    79493997.45854475        7        6      512      512      578      574
    79494575.58943175        9        8      512      512      579      573
    79493631.03866175        6        5      512      512      578      575
    79493290.86521725        5        5      512      512      578      575
    79493432.00990875        9        8      512      512      579      573
```

The column activation/deactivation is performed in the order of the specified column arguments. This means you can mix "+", "-" (which de-activates all columns) and specific column names to reorder and select columns in one command. For example, consider the following:

```
# reorder and de-activate
fundisp snr.ev'[cir 512 512 .1]' "time pi pha + -x -y"
                  TIME       PI      PHA       DX       DY
--------------------- -------- -------- -------- --------
    79493997.45854475        7        6      578      574
    79494575.58943175        9        8      579      573
    79493631.03866175        6        5      578      575
    79493290.86521725        5        5      578      575
    79493432.00990875        9        8      579      573
```

We first activate "time", "pi", and "pha" so that they are output first. We then activate all of the other columns, and then de-activate "x" and "y". Note that this is different from:

```
# probably not what you want ...
fundisp snr.ev'[cir 512 512 .1]' "time pi pha -x -y +"
                  TIME       PI      PHA        Y        X       DX       DY
--------------------- -------- -------- -------- -------- -------- --------
    79493997.45854475        7        6      512      512      578      574
    79494575.58943175        9        8      512      512      579      573
    79493631.03866175        6        5      512      512      578      575
    79493290.86521725        5        5      512      512      578      575
    79493432.00990875        9        8      512      512      579      573
```

Here, "x" and "y" are de-activated, but then all columns including "x" and "y" are again re-activated.

Typically, FunColumnActivate() uses a list of columns that are passed into the program from the command line. For example, the code for funtable contains the following:

```
char *cols=NULL;

/* open the input FITS file */
if( !(fun = FunOpen(argv[1], "rc", NULL)) )
  gerror(stderr, "could not FunOpen input file: %s\n", argv[1]);

/* set active flag for specified columns */
if( argc >= 4 ) cols = argv[3];
FunColumnActivate(fun, cols, NULL);
```

The FunOpen() call sets the default columns to be all columns in the input file. The FunColumnActivate() call then allows the user to control which columns ultimately will be activated (i.e., in this case, written to the new file). For example:

```
funtable test.ev foo.ev "pi pha time"
```

will process only the three columns mentioned, while:

```
funtable test.ev foo.ev "-time"
```

will process all columns except "time".

If FunColumnActivate() is called with a null string, then the environment variable **FUN_COLUMNS** will be used to provide a global value, if present. This is the reason why we call the routine even if no columns are specified on the command line (see example above), instead of calling it this way:

```
/* set active flag for specified columns */
if( argc >= 4 ){
  FunColumnActivate(fun, argv[3], NULL);
}
```

FunColumnLookup - lookup a Funtools column

```
#include <funtools.h>

int FunColumnLookup(Fun fun, char *s, int which,
                    char **name, int *type, int *mode,
                    int *offset, int *n, int *width)
```

The **FunColumnLookup()** routine returns information about a named (or indexed) column. The first argument is the Fun handle associated with this set of columns. The second argument is the name of the column to look up. If the name argument is NULL, the argument that follows is the zero-based index into the column array of the column for which information should be returned. The next argument is a pointer to a char *, which will contain the name of the column. The arguments that follow are the addresses of int values into which the following information will be returned:

- **type**: data type of column:
  - A: ASCII characters
  - B: unsigned 8-bit char

    - I: signed 16-bit int
    - U: unsigned 16-bit int (not standard FITS)
    - J: signed 32-bit int
    - V: unsigned 32-bit int (not standard FITS)
    - E: 32-bit float
    - D: 64-bit float
- **mode**: bit flag status of column, including:
    - COL_ACTIVE 1 is column activated?
    - COL_IBUF 2 is column in the raw input data?
    - COL_PTR 4 is column a pointer to an array?
    - COL_READ 010 is read mode selected?
    - COL_WRITE 020 is write mode selected?
    - COL_REPLACEME 040 is this column being replaced by user data?
- **offset**: byte offset in struct
- **n**: number of elements (i.e. size of vector) in this column
- **width**: size in bytes of this column

If the named column exists, the routine returns a positive integer, otherwise zero is returned. (The positive integer is the index+1 into the column array where this column was located.) If NULL is passed as the return address of one (or more) of these values, no data is passed back for that information. For example:

```
if( !FunColumnLookup(fun, "phas", 0, NULL NULL, NULL, NULL, &npha, NULL) )
  gerror(stderr, "can't find phas column\n");
```

only returns information about the size of the phas vector.

FunTableRowGet - get Funtools rows

```
#include <funtools.h>

void *FunTableRowGet(Fun fun, void *rows, int maxrow, char *plist,
                     int *nrow)
```

The **FunTableRowGet()** routine retrieves rows from a Funtools binary table or raw event file, and places the values of columns selected by FunColumnSelect() into an array of user structs. Selected column values are automatically converted to the specified user data type (and to native data format) as necessary.

The first argument is the Fun handle associated with this row data. The second **rows** argument is the array of user structs into which the selected columns will be stored. If NULL is passed, the routine will automatically allocate space for this array. (This includes proper allocation of pointers within each struct, if the "@" pointer type is used in the selection of columns. Note that if you pass NULL in the second argument, you should free this space using the standard free() system call when you are finished with the array of rows.) The third **maxrow** argument specifies the maximum number of rows to be returned. Thus, if **rows** is allocated by the user, it should be at least of size maxrow*sizeof(evstruct).

The fourth **plist** argument is a param list string. Currently, the keyword/value pair "mask=transparent" is supported in the plist argument. If this string is passed in the call's plist argument, then all rows are passed back to the user (instead of just rows passing the filter). This is only useful when FunColumnSelect() also is used to specify "$region" as a column to return for each row. In such a case, rows found within a region have a returned region value greater than 0 (corresponding to the region id of the region in which they are located), rows passing the filter but not in a region have region value of -1, and rows not passing any filter have region value of 0. Thus, using "mask=transparent" and the returned region value, a program can process all rows and decide on an action based on whether a given row passed the filter or not.

The final argument is a pointer to an int variable that will return the actual number of rows returned. The routine returns a pointer to the array of stored rows, or NULL if there was an error. (This pointer will be the same as the second argument, if the latter is non-NULL).

```
/* get rows -- let routine allocate the row array */
while( (buf = (Ev)FunTableRowGet(fun, NULL, MAXROW, NULL, &got)) ){
  /* process all rows */
  for(i=0; i<got; i++){
    /* point to the i'th row */
    ev = buf+i;
    /* rearrange some values. etc. */
    ev->energy = (ev->pi+ev->pha)/2.0;
    ev->pha = -ev->pha;
    ev->pi = -ev->pi;
  }
  /* write out this batch of rows */
  FunTableRowPut(fun2, buf, got, 0, NULL);
  /* free row data */
  if( buf ) free(buf);
}
```

As shown above, successive calls to FunTableRowGet() will return the next set of rows from the input file until all rows have been read, i.e., the routine behaves like sequential Unix I/O calls such as fread(). See evmerge example code for a more complete example.

Note that FunTableRowGet() also can be called as FunEventsGet(), for backward compatibility.

FunTableRowPut - put Funtools rows

`int FunTableRowPut(Fun fun, void *rows, int nev, int idx, char *plist)`

The **FunTableRowPut()** routine writes rows to a FITS binary table, taking its input from an array of user structs that contain column values selected by a previous call to FunColumnSelect(). Selected column values are automatically converted from native data format to FITS data format as necessary.

The first argument is the Fun handle associated with this row data. The second **rows** argument is the array of user structs to output. The third **nrow** argument specifies the number number of rows to write. The routine will write **nrow** records, starting from the location specified by **rows**.

The fourth **idx** argument is the index of the first raw input row to write, in the case where rows from the user buffer are being merged with their raw input row counterparts (see below). Note that this **idx** value is has nothing to do with the row buffer specified in argument 1. It merely matches the row being written

with its corresponding (hidden) raw row. Thus, if you read a number of rows, process them, and then write them out all at once starting from the first user row, the value of **idx** should be 0:

```
Ev ebuf, ev;
/* get rows -- let routine allocate the row array */
while( (ebuf = (Ev)FunTableRowGet(fun, NULL, MAXROW, NULL, &got)) ){
  /* process all rows */
  for(i=0; i<got; i++){
    /* point to the i'th row */
    ev = ebuf+i;
    ...
  }
  /* write out this batch of rows, starting with the first */
  FunTableRowPut(fun2, (char *)ebuf, got, 0, NULL);
  /* free row data */
  if( ebuf ) free(ebuf);
}
```

On the other hand, if you write out the rows one at a time (possibly skipping rows), then, when writing the i'th row from the input array of rows, set **idx** to the value of i:

```
Ev ebuf, ev;
/* get rows -- let routine allocate the row array */
while( (ebuf = (Ev)FunTableRowGet(fun, NULL, MAXROW, NULL, &got)) ){
  /* process all rows */
  for(i=0; i<got; i++){
    /* point to the i'th row */
    ev = ebuf+i;
    ...
    /* write out the current (i.e., i'th) row */
    FunTableRowPut(fun2, (char *)ev, 1, i, NULL);
  }
  /* free row data */
  if( ebuf ) free(ebuf);
}
```

The final argument is a param list string that is not currently used. The routine returns the number of rows output. This should be equal to the value passed in the third **nrowFunParamGet - get a Funtools param value**

```
#include <funtools.h>

int FunParamGetb(Fun fun, char *name, int n, int defval, int *got)

int FunParamGeti(Fun fun, char *name, int n, int defval, int *got)

double FunParamGetd(Fun fun, char *name, int n, double defval, int *got)

char *FunParamGets(Fun fun, char *name, int n, char *defval, int *got)
```

The four routines **FunParamGetb()**, **FunParamGeti()**, **FunParamGetd()**, and **FunParamGets()**, return the value of a FITS header parameter as a boolean, int, double, and string, respectively. The string returned by **FunParamGets()** is a malloc'ed copy of the header value and should be freed when no longer needed.

The first argument is the Fun handle associated with the FITS header being accessed. Normally, the header is associated with the FITS extension that you opened with **FunOpen()**. However, you can use FunInfoPut() to specify access of the primary header. In particular, if you set the FUN_PRIMARYHEADER parameter to 1, then the primary header is used for all parameter access until the value is reset to 0. For example:

```
int val;
FunParamGeti(fun, "NAXIS", 1, 0, &got);              # current header
val=1;
FunInfoPut(fun, FUN_PRIMARYHEADER, &val, 0);         # switch to ...
FunParamGeti(fun, "NAXIS", 1, 0, &got);              # ... primary header
FunParamGeti(fun, "NAXIS", 2, 0, &got);              # ... primary header
val=0;
FunInfoPut(fun, FUN_PRIMARYHEADER, &val, 0);         # switch back to ...
FunParamGeti(fun, "NAXIS", 2, 0, &got);              # current header
```

Alternatively, you can use the FUN_PRIMARY macro to access parameters from the primary header on a per-parameter basis:

```
FunParamGeti(fun, "NAXIS1", 0, 0, &got);              # current header
FunParamGeti(FUN_PRIMARY(fun), "NAXIS1", 0, 0, &got); # primary header
```

NB: FUN_PRIMARY is deprecated. It makes use of a global parameter and therefore will not not appropriate for threaded applications, when we make funtools thread-safe. We recommend use of FunInfoPut() to switch between the extension header and the primary header.

For output data, access to the primary header is only possible until the header is written out, which usually takes place when the first data are written.

The second argument is the name of the parameter to access. The third **n** argument, if non-zero, is an integer that will be added as a suffix to the parameter name. This makes it easy to use a simple loop to process parameters having the same root name. For example, to gather up all values of TLMIN and TLMAX for each column in a binary table, you can use:

```
for(i=0, got=1; got; i++){
  fun->cols[i]->tlmin = (int)FunParamGeti(fun, "TLMIN", i+1, 0.0, &got);
  fun->cols[i]->tlmax = (int)FunParamGeti(fun, "TLMAX", i+1, 0.0, &got);
}
```

The fourth **defval** argument is the default value to return if the parameter does not exist. Note that the data type of this parameter is different for each specific FunParamGet() call. The final **got** argument will be 0 if no param was found. Otherwise the data type of the parameter is returned as follows: FUN_PAR_UNKNOWN ('u'), FUN_PAR_COMMENT ('c'), FUN_PAR_LOGICAL ('l'), FUN_PAR_INTEGER ('i'), FUN_PAR_STRING ('s'), FUN_PAR_REAL ('r'), FUN_PAR_COMPLEX ('x').

These routines return the value of the header parameter, or the specified default value if the header parameter does not exist. The returned value is a malloc'ed string and should be freed when no longer needed.

By default, **FunParamGets()** returns the string value of the named parameter. However, you can use FunInfoPut() to retrieve the raw 80-character FITS card instead. In particular, if you set the FUN_RAWPARAM parameter to 1, then card images will be returned by FunParamGets() until the value is reset to 0.

Alternatively, if the FUN_RAW macro is applied to the name, then the 80-character raw FITS card is returned instead. **NB: FUN_RAW is deprecated.** It makes use of a global parameter and therefore will not not appropriate for threaded applications, when we make funtools thread-safe. We recommend use of FunInfoPut() to switch between the extension header and the primary header.

Note that in addition to the behaviors described above, the routine **FunParamGets()** will return the 80 raw characters of the **nth** FITS card (including the comment) if **name** is specified as NULL and **n** is positive. For example, to loop through all FITS header cards in a given extension and print out the raw card, use:

```
for(i=1; ;i++){
  if( (s = FunParamGets(fun, NULL, i, NULL, &got)) ){
    fprintf(stdout, "%.80s\n", s);
    free(s);
  }
  else{
    break;
  }
}
```

FunParamPut - put a Funtools param value

```
#include <funtools.h>

int FunParamPutb(Fun fun, char *name, int n, int value, char *comm,
                 int append)

int FunParamPuti(Fun fun, char *name, int n, int value, char *comm,
                 int append)

int FunParamPutd(Fun fun, char *name, int n, double value, int prec,
                 char *comm, int append)

int FunParamPuts(Fun fun, char *name, int n, char *value, char *comm,
                 int append)
```

The four routines **FunParamPutb()**, **FunParamPuti()**, **FunParamPutd()**, and **FunParamPuts()**, will set the value of a FITS header parameter as a boolean, int, double, and string, respectively.

The first argument is the Fun handle associated with the FITS header being accessed. Normally, the header is associated with the FITS extension that you opened with **FunOpen()**. However, you can use FunInfoPut() to specify that use of the primary header. In particular, if you set the FUN_PRIMARYHEADER parameter to 1, then the primary header is used for all parameter access until the value is reset to 0. For example:

```
int val;
FunParamPuti(fun, "NAXIS1", 0, 10, NULL, 1);       # current header
val=1;
FunInfoPut(fun, FUN_PRIMARYHEADER, &val, 0);       # switch to ...
FunParamPuti(fun, "NAXIS1", 0, 10, NULL, 1);       # primary header
```

(You also can use the deprecated FUN_PRIMARY macro, to access parameters from the primary header.)

The second argument is the **name** of the parameter. (In accordance with FITS standards, the special names **COMMENT** and **HISTORY**, as well as blank names, are output without the "= " value indicator in columns 9 and 10.

The third **n** argument, if non-zero, is an integer that will be added as a suffix to the parameter name. This makes it easy to use a simple loop to process parameters having the same root name. For example, to set the values of TLMIN and TLMAX for each column in a binary table, you can use:

```
for(i=0; i<got; i++){
  FunParamPutd(fun, "TLMIN", i+1, tlmin[i], 7, "min column val", 1);
  FunParamPutd(fun, "TLMAX", i+1, tlmax[i], 7, "max column val", 1);
}
```

The fourth **defval** argument is the value to set. Note that the data type of this argument is different for each specific FunParamPut() call. The **comm** argument is the comment string to add to this header parameter. Its value can be NULL. The final **append** argument determines whether the parameter is added to the header if it does not exist. If set to a non-zero value, the header parameter will be appended to the header if it does not exist. If set to 0, the value will only be used to change an existing parameter.

Note that the double precision routine FunParamPutd() supports an extra **prec** argument after the **value** argument, in order to specify the precision when converting the double value to ASCII. In general a 20.[prec] format is used (since 20 characters are alloted to a floating point number in FITS) as follows: if the double value being put to the header is less than 0.1 or greater than or equal to 10**(20-2-[prec]), then %20.[prec]e format is used (i.e., scientific notation); otherwise %20.[prec]f format is used (i.e., numeric notation).

As a rule, parameters should be set before writing the table or image. It is, however, possible to update the value of an **existing** parameter after writing an image or table (but not to add a new one). Such updating only works if the parameter already exists and if the output file is seekable, i.e. if it is a disk file or is stdout being redirected to a disk file.

It is possible to add a new parameter to a header after the data has been written, but only if space has previously been reserved. To reserve space, add a blank parameter whose value is the name of the parameter you eventually will update. Then, when writing the new parameter, specify a value of 2 for the append flag. The parameter writing routine will first look to update an existing parameter, as usual. If an existing parameter is not found, an appropriately-valued blank parameter will be searched for and replaced. For example:

```
/* add blank card to be used as a place holder for IPAR1 update */
FunParamPuts(fun, NULL, 0, "IPAR1", "INTEGER Param", 0);
...
/* write header and data */
FunTableRowPut(fun, events, got, 0, NULL);
...
/* update param in file after writing data -- note append = 2 here */
FunParamPuti(fun, "IPAR", 1, 400, "INTEGER Param", 2);
```

The parameter routines return a 1 if the routine was successful and a 0 on failure. In general, the major reason for failure is that you did not set the append argument to a non-zero value and the parameter did not already exist in the file.

FunInfoGet - get information from Funtools struct

```
#include <funtools.h>

int FunInfoGet(Fun fun, int type, char *addr, ...)
```

The **FunInfoGet()** routine returns information culled from the Funtools structure. The first argument is the Fun handle from which information is to be retrieved. This first required argument is followed by a variable length list of pairs of arguments. Each pair consists of an integer representing the type of information to retrieve and the address where the information is to be stored. The list is terminated by a 0. The routine returns the number of get actions performed.

The full list of available information is described below. Please note that only a few of these will be useful to most application developers. For imaging applications, the most important types are:

```
FUN_SECT_DIM1   int  /* dim1 for section */
FUN_SECT_DIM2   int  /* dim2 for section */
FUN_SECT_BITPIX int  /* bitpix for section */
```

These would be used to determine the dimensions and data type of image data retrieved using the FunImageGet() routine. For example:

```
/* extract and bin the data section into an image buffer */
buf = FunImageGet(fun, NULL, NULL);
/* get required information from funtools structure.
   this should come after the FunImageGet() call, in case the call
   changed sect_bitpix */
FunInfoGet(fun,
           FUN_SECT_BITPIX,  &bitpix,
           FUN_SECT_DIM1,    &dim1,
           FUN_SECT_DIM2,    &dim2,
           0);
/* loop through pixels and reset values below limit to value */
for(i=0; i<dim1*dim2; i++){
  switch(bitpix){
  case 8:
    if( cbuf[i] <= blimit ) cbuf[i] = bvalue;
  ...
}
```

It is important to bear in mind that the call to FunImageGet() can change the value of FUN_SECT_BITPIX (e.g. if "bitpix=n" is passed in the param list). Therefore, a call to FunInfoGet() should be made **after** the call to FunImageGet(), in order to retrieve the updated bitpix value. See the imblank example code for more details.

It also can be useful to retrieve the World Coordinate System information from the Funtools structure. Funtools uses the the WCS Library developed by Doug Mink at SAO, which is available here. (More information about the WCSTools project in general can be found here.) The FunOpen() routine initializes two WCS structures that can be used with this WCS Library. Applications can retrieve either of these two WCS structures using **FunInfoGet()**:

```
FUN_WCS  struct WorldCoor * /* wcs structure, for image coordinates*/
FUN_WCS0 struct WorldCoor * /* wcs structure, for physical coordinates */
```

The structure retrieved by FUN_WCS is a WCS library handle containing parameters suitable for use with image coordinates, regardless of whether the data are images or tables. For this structure, the WCS reference point (CRPIX) has been converted to image coordinates if the underlying file is a table (and therefore in physical coordinates). You therefore must ensure that the positions being passed to a routine like pix2wcs are in image coordinates. The FUN_WCS0 structure has not had its WCS reference point converted to image coordinates. It therefore is useful when passing processing physical coordinates from a table.

Once a WCS structure has been retrieved, it can be used as the first argument to the WCS library routines. (If the structure is NULL, no WCS information was contained in the file.) The two important WCS routines that Funtools uses are:

```
#include <wcs.h>
void pix2wcs (wcs,xpix,ypix,xpos,ypos)
  struct WorldCoor *wcs; /* World coordinate system structure */
  double xpix,ypix;      /* x and y coordinates in pixels */
  double *xpos,*ypos;    /* RA and Dec in degrees (returned) */
```

which converts pixel coordinates to sky coordinates, and:

```
void wcs2pix (wcs, xpos, ypos, xpix, ypix, offscl)
  struct WorldCoor *wcs; /* World coordinate system structure */
  double xpos,ypos;      /* World coordinates in degrees */
  double *xpix,*ypix;    /* coordinates in pixels */
  int *offscl;           /* 0 if within bounds, else off scale */
```

which converts sky coordinates to pixel coordinates. Again, please note that the wcs structure returned by FUN_WCS assumes that image coordinates are passed to the pix2wcs routine, while FUN_WCS0 assumes that physical coordinates are passed.

Note that funtools.h file automatically includes wcs.h. An example program that utilizes these WCS structure to call WCS Library routines is twcs.c.

The following is the complete list of information that can be returned:

```
name            type            comment
---------       --------        -------------------------------------------
FUN_FNAME     char *            /* file name */
FUN_GIO       GIO               /* gio handle */
FUN_HEADER    FITSHead          /* fitsy header struct */
FUN_TYPE      int               /* TY_TABLE,TY_IMAGE,TY_EVENTS,TY_ARRAY */
FUN_BITPIX    int               /* bits/pixel in file */
FUN_MIN1      int               /* tlmin of axis1 -- tables */
FUN_MAX1      int               /* tlmax of axis1 -- tables */
FUN_MIN2      int               /* tlmin of axis2 -- tables */
FUN_MAX2      int               /* tlmax of axis2 -- tables */
FUN_DIM1      int               /* dimension of axis1 */
FUN_DIM2      int               /* dimension of axis2 */
FUN_ENDIAN    int               /* 0=little, 1=big endian */
FUN_FILTER    char *            /* supplied filter */
FUN_IFUN      FITSHead          /* pointer to reference header */
FUN_IFUN0     FITSHead          /* same as above, but no reset performed */
/* image information */
FUN_DTYPE     int               /* data type for images */
FUN_DLEN      int               /* length of image in bytes */
FUN_DPAD      int               /* padding to end of extension */
FUN_DOBLANK   int               /* was blank keyword defined? */
FUN_BLANK     int               /* value for blank */
FUN_SCALED    int               /* was bscale/bzero defined? */
FUN_BSCALE    double            /* bscale value */
FUN_BZERO     double            /* bzero value */
/* table information */
FUN_NROWS     int               /* number of rows in file (naxis2) */
FUN_ROWSIZE   int               /* size of user row struct */
FUN_BINCOLS   char *            /* specified binning columns */
FUN_OVERFLOW  int               /* overflow detected during binning? */
/* array information */
FUN_SKIP      int               /* bytes to skip in array header */
/* section information */
FUN_SECT_X0   int               /* low dim1 value of section */
FUN_SECT_X1   int               /* hi dim1 value of section */
FUN_SECT_Y0   int               /* low dim2 value of section */
FUN_SECT_Y1   int               /* hi dim2 value of section */
FUN_SECT_BLOCK int              /* section block factor */
FUN_SECT_BTYPE int              /* 's' (sum), 'a' (average) for binning */
FUN_SECT_DIM1 int               /* dim1 for section */
FUN_SECT_DIM2 int               /* dim2 for section */
FUN_SECT_BITPIX int             /* bitpix for section */
FUN_SECT_DTYPE int              /* data type for section */
FUN_RAWBUF    char *            /* pointer to raw row buffer */
FUN_RAWSIZE   int               /* byte size of raw row records */
/* column  information */
FUN_NCOL      int               /* number of row columns defined */
FUN_COLS      FunCol            /* array of row columns */
/* WCS information */
FUN_WCS       struct WorldCoor * /* wcs structure, converted for images*/
FUN_WCS0      struct WorldCoor * /* wcs structure, not converted */
```

Row applications would not normally need any of this information. An example of how these values can be used in more complex programs is the evnext example code. In this program, the time value for each row is changed to be the value of the succeeding row. The program thus reads the time values for a batch

of rows, changes the time values to be the value for the succeeding row, and then merges these changed time values back with the other columns to the output file. It then reads the next batch, etc.

This does not work for the last row read in each batch, since there is no succeeding row until the next batch is read. Therefore, the program saves that last row until it has read the next batch, then processes the former before starting on the new batch. In order to merge the last row successfully, the code uses FUN_RAWBUF to save and restore the raw input data associated with each batch of rows. Clearly, this requires some information about how funtools works internally. We are happy to help you write such programs as the need arises.

FunInfoPut - put information into a Funtools struct

```
#include <funtools.h>

int FunInfoPut(Fun fun, int type, char *addr, ...)
```

The **FunInfoPut()** routine puts information into a Funtools structure. The first argument is the Fun handle from which information is to be retrieved. After this first required argument comes a variable length list of pairs of arguments. Each pair consists of an integer representing the type of information to store and the address of the new information to store in the struct. The variable list is terminated by a 0. The routine returns the number of put actions performed.

The full list of available information is described above with the FunInfoPut() routine. Although use of this routine is expected to be uncommon, there is one important situation in which it plays an essential part: writing multiple extensions to a single output file.

For input, multiple extensions are handled by calling FunOpen() for each extension to be processed. When opening multiple inputs, it sometimes is the case that you will want to process them and then write them (including their header parameters) to a single output file. To accomplish this, you open successive input extensions using FunOpen() and then call **FunInfoPut()** to set the Funtools reference handle of the output file to that of the newly opened input extension:

```
/* open a new input extension */
ifun=FunOpen(tbuf, "r", NULL)) )
/* make the new extension the reference handle for the output file */
FunInfoPut(ofun, FUN_IFUN, &ifun, 0);
```

Resetting FUN_IFUN has same effect as when a funtools handle is passed as the final argument to FunOpen(). The state of the output file is reset so that a new extension is ready to be written. Thus, the next I/O call on the output extension will output the header, as expected.

For example, in a binary table, after resetting FUN_IFUN you can then call FunColumnSelect() to select the columns for output. When you then call FunImagePut() or FunTableRowPut(), a new extension will be written that contains the header parameters from the reference extension. Remember to call FunFlush() to complete output of a given extension.

A complete example of this capability is given in the evcol example code. The central algorithm is:

- open the output file without a reference handle
- loop: open each input extension in turn
  - set the reference handle for output to the newly opened input extension
  - read the input rows or image and perform processing
  - write new rows or image to the output file
  - flush the output
  - close input extension
- close output file

Note that FunFlush() is called after processing each input extension in order to ensure that the proper padding is written to the output file. A call to FunFlush() also ensures that the extension header is written to the output file in the case where there are no rows to output.

If you wish to output a new extension without using a Funtools reference handle, you can call FunInfoPut() to reset the FUN_OPS value directly. For a binary table, you would then call FunColumnSelect() to set up the columns for this new extension.

```
/* reset the operations performed on this handle */
int ops=0;
FunInfoPut(ofun, FUN_OPS, &ops, 0);
FunColumnSelect(fun, sizeof(EvRec), NULL,
                "MYCOL", "J", "w", FUN_OFFSET(Ev, mycol),
                NULL);
```

Once the FUN_OPS variable has been reset, the next I/O call on the output extension will output the header, as expected.

FunFlush - flush data to output file

```
#include <funtools.h>

void FunFlush(Fun fun, char *plist)
```

The **FunFlush** routine will flush data to a FITS output file. In particular, it can be called after all rows have been written (using the FunTableRowPut() routine) in order to add the null padding that is required to complete a FITS block. It also should be called after completely writing an image using FunImagePut() or after writing the final row of an image using FunTableRowPut().

The **plist** (i.e., parameter list) argument is a string containing one or more comma-delimited **keyword=value** parameters. If the plist string contains the parameter "copy=remainder" and the file was opened with a reference file, which, in turn, was opened for extension copying (i.e. the input FunOpen() mode also was "c" or "C"), then FunFlush also will copy the remainder of the FITS extensions from the input reference file to the output file. This normally would be done only at the end of processing.

Note that FunFlush() is called with "copy=remainder" in the mode string by FunClose(). This means that if you close the output file before the reference input file, it is not necessary to call FunFlush() explicitly, unless you are writing more than one extension. See the evmerge example code. However, it is safe to call FunFlush() more than once without fear of re-writing either the padding or the copied extensions.

In addition, if FunFlush() is called on an output file with the plist set to "copy=reference" and if the file was opened with a reference file, the reference extension is written to the output file. This mechanism provides a simple way to copy input extensions to an output file without processing the former. For example, in the code fragment below, an input extension is set to be the reference file for a newly opened output extension. If that reference extension is not a binary table, it is written to the output file:

```
/* process each input extension in turn */
for(ext=0; ;ext++){
  /* get new extension name */
  sprintf(tbuf, "%s[%d]", argv[1], ext);
  /* open input extension -- if we cannot open it, we are done */
  if( !(ifun=FunOpen(tbuf, "r", NULL)) )
    break;
  /* make the new extension the reference handle for the output file */
  FunInfoPut(ofun, FUN_IFUN, &ifun, 0);
  /* if its not a binary table, just write it out */
  if( !(s=FunParamGets(ifun, "XTENSION", 0, NULL, &got)) ||
    strcmp(s, "BINTABLE")){
    if( s ) free(s);
    FunFlush(ofun, "copy=reference");
    FunClose(ifun);
    continue;
  }
  else{
    /* process binary table */
    ....
  }
}
```

FunClose - close a Funtools data file

```
#include <funtools.h>

void FunClose(Fun fun)
```

The **FunClose()** routine closes a previously-opened Funtools data file, freeing control structures. If a Funtools reference handle was passed to the FunOpen() call for this file, and if copy mode also was specified for that file, then FunClose() also will copy the remaining extensions from the input file to the output file (if the input file still is open). Thus, we recommend always closing the output Funtools file **before** the input file. (Alternatively, you can call FunFlush() explicitly).

FunRef: the Funtools Reference Handle

Summary

A description of how to use a Funtools reference handle to connect a Funtools input file to an output file.

Description

The Funtools reference handle connects a Funtools input file to a Funtools output file so that parameters (or even whole extensions) can be copied from the one to the other. To make the connection, the Funtools handle of the input file is passed to the final argument of the FunOpen() call for the output file:

```
if( !(ifun = FunOpen(argv[1], "r", NULL)) )
  gerror(stderr, "could not FunOpen input file: %s\n", argv[1]);
if( !(ofun = FunOpen(argv[2], "w", ifun)) )
  gerror(stderr, "could not FunOpen output file: %s\n", argv[2]);
```

It does not matter what type of input or output file (or extension) is opened, or whether they are the same type. When the output image or binary table is written using FunImagePut() or FunTableRowPut() an appropriate header will be written first, with parameters copied from the input extension. Of course, invalid parameters will be removed first, e.g., if the input is a binary table and the output is an image, then binary table parameters such as TFORM, TUNIT, etc. parameters will not be copied to the output.

Use of a reference handle also allows default values to be passed to FunImagePut() in order to write out an output image with the same dimensions and data type as the input image. To use the defaults from the input, a value of 0 is entered for dim1, dim2, and bitpix. For example:

```
fun = FunOpen(argv[1], "r", NULL);
fun2 = FunOpen(argv[2], "w", fun);
buf = FunImageGet(fun, NULL, NULL);
... process image data ...
FunImagePut(fun2, buf, 0, 0, 0, NULL);
```

Of course, you often want to get information about the data type and dimensions of the image for processing. The above code is equivalent to the following:

```
fun = FunOpen(argv[1], "r", NULL);
fun2 = FunOpen(argv[2], "w", fun);
buf = FunImageGet(fun, NULL, NULL);
FunInfoGet(fun, FUN_SECT_DIM1, &dim1, FUN_SECT_DIM2, &dim2,
           FUN_SECT_BITPIX, &bitpix, 0);
... process image data ...
FunImagePut(fun2, buf, dim1, dim2, bitpix, NULL);
```

It is possible to change the reference handle for a given output Funtools handle using the FunInfoPut() routine:

```
/* make the new extension the reference handle for the output file */
FunInfoPut(fun2, FUN_IFUN, &fun, 0);
```

When this is done, Funtools specially resets the output file to start a new output extension, which is connected to the new input reference handle. You can use this mechanism to process multiple input extensions into a single output file, by successively opening the former and setting the reference handle for the latter. For example:

```
/* open a new output FITS file */
if( !(fun2 = FunOpen(argv[2], "w", NULL)) )
  gerror(stderr, "could not FunOpen output file: %s\n", argv[2]);
/* process each input extension in turn */
for(ext=0; ;ext++){
  /* get new extension name */
  sprintf(tbuf, "%s[%d]", argv[1], ext);
  /* open it -- if we cannot open it, we are done */
  if( !(fun=FunOpen(tbuf, "r", NULL)) )
    break;
  /* make the new extension the reference handle for the output file */
  FunInfoPut(fun2, FUN_IFUN, &fun, 0);
  ... process ...
  /* flush output extension (write padding, etc.) */
  FunFlush(fun2, NULL);
  /* close the input extension */
  FunClose(fun);
}
```

In this example, the output file is opened first. Then each successive input extension is opened, and the output reference handle is set to the newly opened input handle. After data processing is performed, the output extension is flushed and the input extension is closed, in preparation for the next input extension.

Finally, a reference handle can be used to copy other extensions from the input file to the output file. Copy of other extensions is controlled by adding a "C" or "c" to the mode string of the FunOpen() call **of the input reference file**. If "C" is specified, then other extensions are **always** copied (i.e., copy is forced by the application). If "c" is used, then other extensions are copied if the user requests copying by adding a plus sign "+" to the extension name in the bracket specification. For example, the **funtable** program utilizes user-specified "c" mode so that the second example below will copy all extensions:

```
# copy only the EVENTS extension
csh> funtable "test.ev[EVENTS,circle(512,512,10)]" foo.ev
# copy ALL extensions
csh> funtable "test.ev[EVENTS+,circle(512,512,10)]" foo.ev
```

When extension copy is specified in the input file, the call to FunOpen() on the input file delays the actual file open until the output file also is opened (or until I/O is performed on the input file, which ever happens first). Then, when the output file is opened, the input file is also opened and input extensions are copied to the output file, up to the specific extension being opened. Processing of input and output extensions then proceed.

When extension processing is complete, the remaining extensions need to be copied from input to output. This can be done explicitly, using the FunFlush() call with the "copy=remaining" plist:

```
FunFlush(fun, "copy=remaining");
```

Alternatively, this will happen automatically, if the output file is closed **before** the input file:

```
/* we could explicitly flush remaining extensions that need copying */
/* FunFlush(fun2, "copy=remaining"); */
/* but if we close output before input, end flush is done automatically  */
FunClose(fun2);
FunClose(fun);
```

Go to Funtools Help Index

Last updated: December 1, 2005

FunFiles: Funtools Data Files

Summary

This document describes the data file formats (FITS, array, raw events) as well as the file types (gzip, socket, etc.) supported by Funtools.

Description

Funtools supports FITS images and binary tables, and binary files containing array (homogeneous) data or event (heterogeneous) data. IRAF-style brackets are appended to the filename to specify various kinds of information needed to characterize these data:

```
file[ext|ind|ARRAY()|EVENTS(),section][filters]
or
file[ext|ind|ARRAY()|EVENTS(),section,filters]
```

where:

- **file** is the Funtools file name
- **ext** is the FITS extension name
- **ind** is the FITS extension number
- **ARRAY()** is an array specification
- **EVENTS()** is an event specification
- **section** is the image section specification
- **filters** are spatial region and table (row) filters

Supported Data Formats

Funtools programs (and the underlying libraries) support the following data file formats:

- FITS images (and image extensions)
- FITS binary tables
- binary files containing an array of homogeneous data
- binary files containing events, i.e. records of heterogeneous data
- column-based text files, which are documented here
- non-disk files and lists of files

Information needed to identify and characterize the event or image data can be specified on the command line using IRAF-style bracket notation appended to the filename:

```
foo.fits                              # open FITS default extension
image.fits[3]                         # open FITS extension #3
events.fits[EVENTS]                   # open EVENTS extension
array.file[ARRAY(s1024)]              # open 1024x1024 short array
events.file[EVENTS(x:1024,y:1024...)] # open non-FITS event list
```

Note that in many Unix shells (e.g., csh and tcsh), filenames must be enclosed in quotes to protect the brackets from shell processing.

FITS Images and Binary Tables

When FunOpen() opens a FITS file without a bracket specifier, the default behavior is to look for a valid image in the primary HDU. In the absence of a primary image, Funtools will try to open an extension named either **EVENTS** or **STDEVT**, if one of these exists. This default behavior supports both FITS image processing and standard X-ray event list processing (which, after all, is what we at SAO/HEAD do).

In order to open a FITS binary table or image extension explicitly, it is necessary to specify either the extension name or the extension number in brackets:

```
foo.fits[1]                    # open extension #1: the primary HDU
foo.fits[3]                    # open extension #3 of a FITS file
foo.fits[GTI]                  # open GTI extension of a FITS file
```

The ext argument specifies the name of the FITS extension (i.e. the value of the EXTENSION header parameter in a FITS extension), while the index specifies the value of the FITS EXTVER header parameter. Following FITS conventions, extension numbers start at 1.

When a FITS data file is opened for reading using FunOpen(), the specified extension is automatically located and is used to initialize the Funtools internal data structures.

Non-FITS Raw Event Files

In addition to FITS tables, Funtools programs and libraries can operate on non-FITS files containing heterogeneous event records. To specify such an event file, use:

- file[EVENTS(event-spec)]
- file[EVENTS()]

where **event-spec** is a string that specified the names, data types, and optional image dimensions for each element of the event record:

- [name]:[n][type]:[(lodim:)hidim]

Data types follow standard conventions for FITS binary tables, but include two extra unsigned types ('U' and 'V'):

- **B** -- unsigned 8-bit char
- **I** -- signed 16-bit int
- **J** -- signed 32-bit int
- **K** -- signed 64-bit int
- **E** -- 32-bit float
- **D** -- 64-bit float
- **U** -- unsigned 16-bit int
- **V** -- unsigned 32-bit int

An optional integer value **n** can be prefixed to the type to indicate that the element is an array of n values. For example:

```
foo.fits[EVENTS(x:I,y:I,status:4J)]
```

defines x and y as 16-bit ints and status as an array of 4 32-bit ints.

Furthermore, image dimensions can be attached to the event specification in order to tell Funtools how to bin the events into an image. They follow the conventions for the FITS TLMIN/TLMAX keywords. If the low image dimension is not specified, it defaults to 1. Thus:

- RAWX:J:1:100
- RAWX:J:100

both specify that the dimension of this column runs from 1 to 100.

NB: it is required that all padding be specified in the record definition. Thus, when writing out whole C structs instead of individual record elements, great care must be taken to include the compiler-added padding in the event definition.

For example, suppose a FITS binary table has the following set of column definitions:

```
TTYPE1  = 'X                 ' / Label for field
TFORM1  = '1I                ' / Data type for field
TLMIN1  =                    1 / Min. axis value
TLMAX1  =                   10 / Max. axis value
TTYPE2  = 'Y                 ' / Label for field
TFORM2  = '1I                ' / Data type for field
TLMIN2  =                    2 / Min. axis value
TLMAX2  =                   11 / Max. axis value
TTYPE3  = 'PHA               ' / Label for field
TFORM3  = '1I                ' / Data type for field
TTYPE4  = 'PI                ' / Label for field
TFORM4  = '1J                ' / Data type for field
TTYPE5  = 'TIME              ' / Label for field
TFORM5  = '1D                ' / Data type for field
TTYPE6  = 'DX                ' / Label for field
TFORM6  = '1E                ' / Data type for field
TLMIN6  =                    1 / Min. axis value
TLMAX6  =                   10 / Max. axis value
TTYPE7  = 'DY                ' / Label for field
TFORM7  = '1E                ' / Data type for field
TLMIN7  =                    3 / Min. axis value
TLMAX7  =                   12 / Max. axis value
```

An raw event file containing these same data would have the event specification:

```
EVENTS(X:I:10,Y:I:2:11,PHA:I,PI:J,TIME:D,DX:E:10,DY:E:3:12)
```

If no event specification string is included within the EVENTS() operator, then the event specification is taken from the **EVENTS** environment variable:

```
  setenv EVENTS "X:I:10,Y:I:10,PHA:I,PI:J,TIME:D,DX:E:10,DY:E:10"
```

In addition to knowing the data structure, it is necessary to know the *endian* ordering of the data, i.e., whether or not the data is in *bigendian* format, so that we can convert to the native format for this platform. This issue does not arise for FITS Binary Tables because all FITS files use big-endian ordering, regardless of platform. But for non-FITS data, big-endian data produced on a Sun workstation but read on a Linux PC needs to be byte-swapped, since PCs use little-endian ordering. To specify an ordering, use the *bigendian*= or *endian*= keywords on the command-line or the EVENTS_BIGENDIAN or EVENTS_ENDIAN environment variables. The value of the *bigendian* variables should be "true" or "false", while the value of the *endian* variables should be "little" or "big".

For example, a PC can access data produced by a Sun using:

```
  hrc.nepr[EVENTS(),bigendian=true]
or
  hrc.nepr[EVENTS(),endian=big]
or
  setenv EVENTS_BIGENDIAN true
or
  setenv EVENTS_ENDIAN big
```

If none of these are specified, the data are assumed to follow the format for that platform and no byte-swapping is performed.

Non-FITS Array Files

In addition to FITS images, Funtools programs and libraries can operate on non-FITS files containing arrays of homogeneous data. To specify an array file, use:

- file[ARRAY(array-spec)]
- file[ARRAY()]

where array-spec is of the form:

- [type][dim1][.dim2][:skip][endian]

and where [type] is:

- b (8-bit unsigned char)
- s (16-bit short int)
- u (16-bit unsigned short int)
- i (32-bit int)
- r,f (32-bit float)
- d (64-bit float)

The dim1 specification is required, but dim2 is optional and defaults to dim1. The skip specification is optional and defaults to 0. The optional endian specification can be ’l’ or ’b’ and defaults to the endian type for the current machine.

If no array specification is included within the ARRAY() operator, then the array specification is taken from the **ARRAY** environment variable. For example:

```
foo.arr[ARRAY(r512)]           # bitpix=-32 dim1=512 dim2=512
foo.arr[ARRAY(r512.400)]       # bitpix=-32 dim1=512 dim2=400
foo.arr[ARRAY(r512.400])       # bitpix=-32 dim1=512 dim2=400
foo.arr[ARRAY(r512.400:2880)] # bitpix=-32 dim1=512 dim2=400 skip=2880
foo.arr[ARRAY(r512l)]          # bitpix=-32 dim1=512 dim2=512 endian=little
setenv ARRAY "r512.400:2880"
foo.arr[ARRAY()]               # bitpix=-32 dim1=512 dim2=400 skip=2880
```

Specifying Image Sections

Once a data file (and possibly, a FITS extension) has been specified, the next (optional) part of a bracket specification can be used to select image **section** information, i.e., to specify the x,y limits of an image section, as well as the blocking factor to apply to that section. This information can be added to any file specification but only is used by Funtools image processing routines.

The format of the image section specification is one of the following:

- file[xy0:xy1,block]
- file[x0:x1,y0:y1,block]
- file[x0:x1,*,block]
- file[*,y0:y1,block]
- file[*,block]

where the limit values can be ints or "*" for default. A single "*" can be used instead of val:val, as shown. Note that blocking is applied to the section after it is extracted.

In addition to image sections specified by the lo and hi x,y limits, image sections using center positions can be specified:

- file[dim1@xcen,dim2@ycen]
- file[xdim2@xcen@ycen]
- file[dim1@xcen,dim2@ycen,block]
- file[dim@xcen@ycen,block]

Note that the (float) values for dim, dim1, dim2, xcen, ycen must be specified or else the expression does not make sense!

In all cases, block is optional and defaults to 1. An ’s’ or ’a’ can be appended to signify "sum" or "average" blocking (default is "sum"). Section specifications are given in image coordinates by default. If you wish to specify physical coordinates, add a ’p’ as the last character of the section specification, before the closing bracket. For example:

- `file[-8:-7,-8:-7p]`

- `file[-8:-7,-8:-7,2p]`

  A section can be specified in any Funtools file name. If the operation to be applied to that file is an imaging operation, then the specification will be utilized. If the operation is purely a table operation, then the section specification is ignored.

  Do not be confused by:

  ```
  foo.fits[2]
  foo.fits[*,2]
  ```

  The former specifies opening the second extension of the FITS file. The latter specifies application of block 2 to the image section.

  Note that the section specification must come after any of FITS **ext** name or **ind** number, but all sensible defaults are supported:

  - file[ext]
  - file[ext,index]
  - file[index]
  - file[ext,section]
  - file[ext,index,section]
  - file[index,section]
  - file[section]

Binning FITS Binary Tables and Non-FITS Event Files

If a FITS binary table or a non-FITS raw event file is to be binned into a 2D image (e.g., using the funimage program), it is necessary to specify the two columns to be used for the binning, as well as the dimensions of the image. Funtools first looks for a specifier of the form:

```
bincols=([xnam[:tlmin[:tlmax:[binsiz]]]],[ynam[:tlmin[:tlmax[:binsiz]]]])
```

in bracket syntax, and uses the column names thus specified. The tlmin, tlmax, and binsiz specifiers determine the image binning dimensions using:

```
dim = (tlmax - tlmin)/binsiz     (floating point data)
dim = (tlmax - tlmin)/binsiz + 1 (integer data)
```

These tlmin, tlmax, and binsiz specifiers can be omitted if TLMIN, TLMAX, and TDBIN header parameters are present in the FITS binary table header, respectively. If only one parameter is specified, it is assumed to be tlmax, and tlmin defaults to 1. If two parameters are specified, they are assumed to be tlmin and tlmax. For example, to bin an HRC event list columns "VPOS" and "UPOS", use:

```
hrc.nepr[bincols=(VPOS,UPOS)]
```

or

```
hrc.nepr[bincols=(VPOS:49152,UPOS:4096)]
```

Note that you can optionally specify the dimensions of these columns to cover cases where neither TLMAX keywords are defined in the header. If either dimension is specified, then both must be specified.

You can set the FITS_BINCOLS or EVENTS_BINCOLS environment variable as an alternative to adding the "bincols=" specifier to each file name for FITS binary tables and raw event files, respectively. If no binning keywords or environment variables are specified, or if the specified columns are not in the binary table, the Chandra parameters CPREF (or PREFX) are searched for in the FITS binary table header. Failing this, columns named "X" and "Y" are sought. If these are not found, the code looks for columns containing the characters "X" and "Y". Thus, you can bin on "DETX" and "DETX" columns without specifying them, if these are the only column names containing the "X" and "Y" characters.

Ordinarily, each event or row contributes one count to an image pixel during the 2D binning process. Thus, if five events all have the same (x,y) position, the image pixel value for that position will have a value of five. It is possible to specify a variable contribution for each event by using the vcol=[colname] filter spec:

```
vcol=[colname]
```

The vcol colname is a column containing a numeric value in each event row that will be used as the contribution of the given event to its image pixel. For example, consider an event file that has the following content:

```
x:e:4    y:e:4    v:e
------   ------   ----
1        1        1.0
2        2        2.0
3        3        3.0
4        4        0.0
1        1        1.0
2        2        2.0
3        3        3.0
4        4        4.0
```

There are two events with x,y value of (1,1) so ordinarily a 2D image will have a value of 2 in the (1,1) pixel. If the v column is specified as the value column:

```
foo.fits'[vcol=v]'
```

then each pixel will contain the additive sum of the associated (x,y) column values from the v column. For example, image pixel (1,1) will contain 1. + 1. = 2, image pixel (2,2) will contain (2 + 2) = 4, etc.

An important variation on the use of a value column to specify the contribution an event makes to an image pixel is when the value column contains the reciprocal of the event contribution. For this case, the column name should be prefixed with a / (divide sign) thus:

```
foo.fits'[vcol=/v]'
```

Each image pixel value will then be the sum of the reciprocals of the value column. A zero in the value column results in NaN (not a number). Thus, in the above example, image pixel (1.1) will contain 1/1 + 1/1 = 2, image pixel (2,2) will contain (1/2 + 1/2) = 1, etc. Image pixel (4,4) will contain (1/0 + 1/4) =

NaN.

You can set the FITS_VCOL or EVENTS_VCOL environment variable as an alternative to adding the "vcol=" specifier to each file name for FITS binary tables and raw event files, respectively.

Finally, when binning events, the data type of the resulting 2D image must be specified. This can be done with the "bitpix=[n]" keyword in the bracket specification. For example:

```
events.fits[bincols=(VPOS,UPOS),bitpix=-32]
```

will create a floating point image binned on columns VPOS and UPOS. If no bitpix keyword is specified, bitpix=32 is assumed. As with bincols values, you also can use the FITS_BITPIX and EVENTS_BITPIX environment variables to set this value for FITS binary tables and raw event files, respectively.

The **funimage** program also allows you to create a 1D image projection along any column of a table by using the **bincols=[column]** filter specification and specifying a single column. For example, the following command projects a 1D image along the chipx column of a table:

```
funimage ev.fits'[bincols=chipx]' im.fits
```

See funimage for more information about creating 1D and 2D images.

Finally, please note that Funtools supports most FITS standards. We will add missing support as required by the community. In general, however, we do not support non-standard extensions. For example, we sense the presence of the binary table 'variable length array' proposed extension and we pass it along when copying and filtering files, but we do not process it. We will add support for new standards as they become official.

Table and Spatial Region Filters

Note that, in addition extensions and image sections, Funtools bracket notation can be used to specify table and spatial region filters. These filters are always placed after the image section information. They can be specified in the same bracket or in a separate bracket immediately following:

- file[ext|ind|ARRAY()|EVENTS(),section][filters]
- file[ext|ind|ARRAY()|EVENTS(),section,filters]

where:

- **file** is the Funtools file name
- **ARRAY()** is an array specification
- **EVENTS()** is an event list specification
- **ext** is the FITS extension name
- **ind** is the FITS extension number
- **section** is the image section to extract
- **filters** are spatial region and table (row) filters to apply

The topics of table and region filtering are covered in detail in:

- Table Filtering
- Spatial Region Filtering

Disk Files and Other Supported File Types

The specified **file** usually is an ordinary disk file. In addition, gzip'ed files are supported in Funtools: gzip'ed input files are automatically uncompressed as they are read, and gzip'ed output files are compressed as they are written. NB: if a FITS binary table is written in gzip format, the number of rows in the table will be set to -1. Such a file will work with Funtools programs but will not work with other FITS programs such as ds9.

The special keywords "stdin" and "stdout" designate Unix standard input and standard output, respectively. The string "-" (hyphen) will be taken to mean "stdin" if the file is opened for reading and "stdout" if the file is opened for writing.

A file also can be an INET socket on the same or another machine using the syntax:

```
machine:port
```

Thus, for example:

```
karapet:1428
```

specifies that I/O should be performed to/from port 1428 on the machine karapet. If no machine name is specified, the default is to use the current machine:

```
:1428
```

This means to open port 1428 on the current machine. Socket support allows you to generate a distributed pipe:

```
on karapet:       funtask1 in.fits bynars:1428
on bynars:        funtask2 :1428 out.fits
```

The socket mechanism thus supports simple parallel processing using **process decomposition**. Note that parallel processing using **data decomposition** is supported via the **section** specifier (see below), and the **row#** specifier, which is part of Table Filtering.

A file also can be a pointer to shared memory using the syntax:

```
shm:[id|@key][:size]
```

A shared memory segment is specified with a **shm:** prefix, followed by either the shared memory id or the shared memory key (where the latter is prefixed by the '@' character). The size (in bytes) of the shared memory segment can then be appended (preceded by the ':' character). If the size specification is absent, the code will attempt to determine the length automatically. If the open mode contains the string "w+", then the memory segment will be created if it does not exist. (It also will be released and deleted when the file is closed.) In the case where a memory segment is being created, the length of the segment is required.

A file also can be Unix piped command (i.e. a program to run) using the syntax:

```
"pipe: command arg1 ... argn"
```

The output from the command must be a valid FITS file. It is important to use quotes to protect spaces so that command arguments are passed correctly. A silly example is:

```
fundisp "pipe: funtable 'foo.fits[cir 512 512 .1]' stdout"
```

This seemed like a good idea at the time ...

Lists of Files

Funtools also will process a list of files as a single file using the syntax:

```
"list: file1 file2 ... filen"
```

The files in the list are separated by whitespace. Any of the above file types can be used. For example, if two files, foo1.fits and foo2.fits, are part of the same observation, they can be processed as a single file (using their own filters):

```
fundisp "list: foo1.fits[cir(512,512,10)] foo2.fits[cir(511,511,10)]"
       X        Y      PHA       PI                  TIME       DX       DY
-------- -------- -------- -------- -------------------- -------- --------
     512      512        6        7     79493997.45854475      578      574
     512      512        8        9     79494575.58943175      579      573
     512      512        5        6     79493631.03866175      578      575
     512      512        5        5     79493290.86521725      578      575
     512      512        8        9     79493432.00990875      579      573
     511      511        5        5     79488631.09462625      580      575
     511      511       10       11     79488780.60006675      580      573
     511      511        4        4     79494562.35474326      580      575
     511      511        6        6     79488203.01561825      580      575
     511      511        6        6     79488017.99730176      580      575
     511      511        4        4     79494332.45355175      580      575
     511      511        9       10     79492685.94014275      581      574
     511      511        5        5     79487708.71298325      580      575
     511      511        8        9     79493719.00160225      581      573
```

Again, note that it is important to avoid spaces in the filters because the list separator also is whitespace. To protect whitespace in a filter, enclose the file specification in quotes:

```
fundisp "list: 'foo1.fits[cir 512 512 .1]' foo2.fits[cir(511,511,.1)]"
```

Go to Funtools Help Index

Last updated: February 15, 2006

Funtext: Support for Column-based Text Files

Summary

This document contains a summary of the options for processing column-based text files.

Description

Funtools will automatically sense and process "standard" column-based text files as if they were FITS binary tables without any change in Funtools syntax. In particular, you can filter text files using the same syntax as FITS binary tables:

```
fundisp foo.txt'[cir 512 512 .1]'
fundisp -T foo.txt > foo.rdb
funtable foo.txt'[pha=1:10,cir 512 512 10]' foo.fits
```

The first example displays a filtered selection of a text file. The second example converts a text file to an RDB file. The third example converts a filtered selection of a text file to a FITS binary table.

Text files can also be used in Funtools image programs. In this case, you must provide binning parameters (as with raw event files), using the bincols keyword specifier:

```
bincols=([xname[:tlmin[:tlmax:[binsiz]]]],[yname[:tlmin[:tlmax[:binsiz]]]
```

For example:

```
funcnts foo'[bincols=(x:1024,y:1024)]' "ann 512 512 0 10 n=10"
```

Standard Text Files

Standard text files have the following characteristics:

- Optional comment lines start with #
- Optional blank lines are considered comments
- An optional table header consists of the following (in order):
  - a single line of alpha-numeric column names
  - an optional line of unit strings containing the same number of cols
  - an optional line of dashes containing the same number of cols
- Data lines follow the optional header and (for the present) consist of the same number of columns as the header.
- Standard delimiters such as space, tab, comma, semi-colon, and bar.

Examples:

```
# rdb file
foo1  foo2    foo3    foos
----  ----    ----    ----
1     2.2     3       xxxx
10    20.2    30      yyyy

# multiple consecutive whitespace and dashes
foo1   foo2    foo3 foos
---    ----    ---- ----
   1    2.2    3    xxxx
  10   20.2    30   yyyy

# comma delims and blank lines
foo1,foo2,foo3,foos

1,2.2,3,xxxx
10,20.2,30,yyyy

# bar delims with null values
foo1|foo2|foo3|foos
1||3|xxxx
10|20.2||yyyy

# header-less data
1     2.2   3 xxxx
10    20.2 30 yyyy
```

The default set of token delimiters consists of spaces, tabs, commas, semi-colons, and vertical bars. Several parsers are used simultaneously to analyze a line of text in different ways. One way of analyzing a line is to allow a combination of spaces, tabs, and commas to be squashed into a single delimiter (no null values between consecutive delimiters). Another way is to allow tab, semi-colon, and vertical bar delimiters to support null values, i.e. two consecutive delimiters implies a null value (e.g. RDB file). A successful parser is one which returns a consistent number of columns for all rows, with each column having a consistent data type. More than one parser can be successful. For now, it is assumed that successful parsers all return the same tokens for a given line. (Theoretically, there are pathological cases, which will be taken care of as needed). Bad parsers are discarded on the fly.

If the header does not exist, then names "col1", "col2", etc. are assigned to the columns to allow filtering. Furthermore, data types for each column are determined by the data types found in the columns of the first data line, and can be one of the following: string, int, and double. Thus, all of the above examples return the following display:

```
fundisp foo'[foo1>5]'
      FOO1                  FOO2       FOO3         FOOS
---------- -------------------- ---------- ------------
        10           20.20000000         30         yyyy
```

Comments Convert to Header Params

Comments which precede data rows are converted into header parameters and will be written out as such using funimage or funhead. Two styles of comments are recognized:

1. FITS-style comments have an equal sign "=" between the keyword and value and an optional slash "/" to signify a comment. The strict FITS rules on column positions are not enforced. In addition, strings only need to be quoted if they contain whitespace. For example, the following are valid FITS-style comments:

```
# fits0 = 100
# fits1 = /usr/local/bin
# fits2 = "/usr/local/bin /opt/local/bin"
# fits3c = /usr/local/bin /opt/local/bin /usr/bin
# fits4c = "/usr/local/bin /opt/local/bin" / path dir
```

Note that the fits3c comment is not quoted and therefore its value is the single token "/usr/local/bin" and the comment is "opt/local/bin /usr/bin". This is different from the quoted comment in fits4c.

2. Free-form comments can have an optional colon separator between the keyword and value. In the absence of quote, all tokens after the keyword are part of the value, i.e. no comment is allowed. If a string is quoted, then slash "/" after the string will signify a comment. For example:

```
# com1 /usr/local/bin
# com2 "/usr/local/bin /opt/local/bin"
# com3 /usr/local/bin /opt/local/bin /usr/bin
# com4c "/usr/local/bin /opt/local/bin" / path dir

# com11: /usr/local/bin
# com12: "/usr/local/bin /opt/local/bin"
# com13: /usr/local/bin /opt/local/bin /usr/bin
# com14c: "/usr/local/bin /opt/local/bin" / path dir
```

Note that com3 and com13 are not quoted, so the whole string is part of the value, while comz4c and com14c are quoted and have comments following the values.

Some text files have column name and data type information in the header. You can specify the format of column information contained in the header using the "hcolfmt=" specification. See below for a detailed description.

Multiple Tables in a Single File

Multiple tables are supported in a single file. If an RDB-style file is sensed, then a ^L (vertical tab) will signify end of table. Otherwise, an end of table is sensed when a new header (i.e., all alphanumeric columns) is found. (Note that this heuristic does not work for single column tables where the column type is ASCII and the table that follows also has only one column.) You also can specify characters that signal an end of table condition using the **eot=** keyword. See below for details.

You can access the nth table (starting from 1) in a multi-table file by enclosing the table number in brackets, as with a FITS extension:

```
fundisp foo'[2]'
```

The above example will display the second table in the file. (Index values start at 1 in oder to maintain logical compatibility with FITS files, where extension numbers also start at 1).

TEXT() Specifier

As with ARRAY() and EVENTS() specifiers for raw image arrays and raw event lists respectively, you can use TEXT() on text files to pass key=value options to the parsers. An empty set of keywords is equivalent to not having TEXT() at all, that is:

```
fundisp foo
fundisp foo'[TEXT()]'
```

are equivalent. A multi-table index number is placed before the TEXT() specifier as the first token, when indexing into a multi-table: fundisp foo'[2,TEXT(...)]'

The filter specification is placed after the TEXT() specifier, separated by a comma, or in an entirely separate bracket:

```
fundisp foo'[TEXT(...),circle 512 512 .1]'
fundisp foo'[2,TEXT(...)][circle 512 512 .1]'
```

Text() Keyword Options

The following is a list of keywords that can be used within the TEXT() specifier (the first three are the most important):

delims="[delims]"
: Specify token delimiters for this file. Only a single parser having these delimiters will be used to process the file.

```
fundisp foo.fits'[TEXT(delims="!")]'
fundisp foo.fits'[TEXT(delims="\t%")]'
```

comchars="[comchars]"
: Specify comment characters. You must include "\n" to allow blank lines. These comment characters will be used for all standard parsers (unless delims are also specified).

```
fundisp foo.fits'[TEXT(comchars="!\n")]'
```

cols="[name1:type1 ...]"
: Specify names and data type of columns. This overrides header names and/or data types in the first data row or default names and data types for header-less tables.

```
fundisp foo.fits'[TEXT(cols="x:I,y:I,pha:I,pi:I,time:D,dx:E,dy:e")]'
```

: If the column specifier is the only keyword, then the cols= is not required (in analogy with EVENTS()):

```
fundisp foo.fits'[TEXT(x:I,y:I,pha:I,pi:I,time:D,dx:E,dy:e)]'
```

: Of course, an index is allowed in this case:

```
fundisp foo.fits'[2,TEXT(x:I,y:I,pha:I,pi:I,time:D,dx:E,dy:e)]'
```

eot="[eot delim]"
: Specify end of table string specifier for multi-table files. RDB files support ^L. The end of table specifier is a string and the whole string must be found alone on a line to signify EOT. For example:

```
fundisp foo.fits'[TEXT(eot="END")]'
```

will end the table when a line contains "END" is found. Multiple lines are supported, so that:

```
fundisp foo.fits'[TEXT(eot="END\nGAME")]'
```

will end the table when a line contains "END" followed by a line containing "GAME".

In the absence of an EOT delimiter, a new table will be sensed when a new header (all alphanumeric columns) is found.

null1="[datatype]"
: Specify data type of a single null value in row 1. Since column data types are determined by the first row, a null value in that row will result in an error and a request to specify names and data types using cols=. If you only have a one null in row 1, you don't need to specify all names and columns. Instead, use null1="type" to specify its data type.

alen=[n]
: Specify size in bytes for ASCII type columns. FITS binary tables only support fixed length ASCII columns, so a size value must be specified. The default is 16 bytes.

nullvalues=["true"|"false"]
: Specify whether to expect null values. Give the parsers a hint as to whether null values should be allowed. The default is to try to determine this from the data.

whitespace=["true"|"false"]
: Specify whether surrounding white space should be kept as part of string tokens. By default surrounding white space is removed from tokens.

header=["true"|"false"]
: Specify whether to require a header. This is needed by tables containing all string columns (and with no row containing dashes), in order to be able to tell whether the first row is a header or part of the data. The default is false, meaning that the first row will be data. If a row dashes are present, the previous row is considered the column name row.

units=["true"|"false"]
: Specify whether to require a units line. Give the parsers a hint as to whether a row specifying units should be allowed. The default is to try to determine this from the data.

i2f=["true"|"false"]
: Specify whether to allow int to float conversions. If a column in row 1 contains an integer value, the data type for that column will be set to int. If a subsequent row contains a float in that same column, an error will be signaled. This flag specifies that, instead of an error, the float should be silently truncated to int. Usually, you will want an error to be signaled, so that you can specify the data type

using cols= (or by changing the value of the column in row 1).

comeot=["true"|"false"|0|1|2]
: Specify whether comment signifies end of table. If comeot is 0 or false, then comments do not signify end of table and can be interspersed with data rows. If the value is true or 1 (the default for standard parsers), then non-blank lines (e.g. lines beginning with '#') signify end of table but blanks are allowed between rows. If the value is 2, then all comments, including blank lines, signify end of table.

lazyeot=["true"|"false"]
: Specify whether "lazy" end of table should be permitted (default is true for standard formats, except rdb format where explicit ^L is required between tables). A lazy EOT can occur when a new table starts directly after an old one, with no special EOT delimiter. A check for this EOT condition is begun when a given row contains all string tokens. If, in addition, there is a mismatch between the number of tokens in the previous row and this row, or a mismatch between the number of string tokens in the prev row and this row, a new table is assumed to have been started. For example:

```
ival1 sval3
----- -----
1     two
3     four

jval1 jval2 tval3
----- ----- ------
10    20    thirty
40    50    sixty
```

Here the line "jval1 ..." contains all string tokens. In addition, the number of tokens in this line (3) differs from the number of tokens in the previous line (2). Therefore a new table is assumed to have started. Similarly:

```
ival1 ival2 sval3
----- ----- -----
1     2     three
4     5     six

jval1 jval2 tval3
----- ----- ------
10    20    thirty
40    50    sixty
```

Again, the line "jval1 ..." contains all string tokens. The number of string tokens in the previous row (1) differs from the number of tokens in the current row(3). We therefore assume a new table as been started. This lazy EOT test is not performed if lazyeot is explicitly set to false.

hcolfmt=[header column format]
: Some text files have column name and data type information in the header. For example, VizieR catalogs have headers containing both column names and data types:

```
#Column e_Kmag  (F6.3)  ?(k_msigcom) K total magnitude uncertainty (4)  [ucd=ERROR]
#Column Rflg    (A3)    (rd_flg) Source of JHK default mag (6)  [ucd=REFER_CODE]
#Column Xflg    (I1)    [0,2] (gal_contam) Extended source contamination (10) [ucd=CODE_MISC]
```

while Sextractor files have headers containing column names alone:

```
#   1 X_IMAGE         Object position along x                         [pixel]
#   2 Y_IMAGE         Object position along y                         [pixel]
#   3 ALPHA_J2000     Right ascension of barycenter (J2000)           [deg]
#   4 DELTA_J2000     Declination of barycenter (J2000)               [deg]
```

The hcolfmt specification allows you to describe which header lines contain column name and data type information. It consists of a string defining the format of the column line, using "$col" (or "$name") to specify placement of the column name, "$fmt" to specify placement of the data format, and "$skip" to specify tokens to ignore. You also can specify tokens explicitly (or, for those users familiar with how sscanf works, you can specify scanf skip specifiers using "%*"). For example, the VizieR hcolfmt above might be specified in several ways:

```
Column $col ($fmt)    # explicit specification of "Column" string
$skip  $col ($fmt)    # skip one token
%*s $col  ($fmt)      # skip one string (using scanf format)
```

while the Sextractor format might be specified using:

```
$skip $col            # skip one token
%*d $col              # skip one int (using scanf format)
```

You must ensure that the hcolfmt statement only senses actual column definitions, with no false positives or negatives. For example, the first Sextractor specification, "$skip $col", will consider any header line containing two tokens to be a column name specifier, while the second one, "%*d $col", requires an integer to be the first token. In general, it is preferable to specify formats as explicitly as possible.

Note that the VizieR-style header info is sensed automatically by the funtools standard VizieR-like parser, using the hcolfmt "Column $col ($fmt)". There is no need for explicit use of hcolfmt in this case.

debug=["true"|"false"]
: Display debugging information during parsing.

Environment Variables

Environment variables are defined to allow many of these TEXT() values to be set without having to include them in TEXT() every time a file is processed:

```
keyword       environment variable
-------       --------------------
delims        TEXT_DELIMS
comchars      TEXT_COMCHARS
cols          TEXT_COLUMNS
eot           TEXT_EOT
null1         TEXT_NULL1
alen          TEXT_ALEN
bincols       TEXT_BINCOLS
hcolfmt       TEXT_HCOLFMT
```

Restrictions and Problems

As with raw event files, the '+' (copy extensions) specifier is not supported for programs such as funtable.

String to int and int to string data conversions are allowed by the text parsers. This is done more by force of circumstance than by conviction: these transitions often happens with VizieR catalogs, which we want to support fully. One consequence of allowing these transitions is that the text parsers can get confused by columns which contain a valid integer in the first row and then switch to a string. Consider the following table:

```
xxx   yyy     zzz
----  ----    ----
111   aaa     bbb
ccc   222     ddd
```

The xxx column has an integer value in row one a string in row two, while the yyy column has the reverse. The parser will erroneously treat the first column as having data type int:

```
fundisp foo.tab
       XXX           YYY           ZZZ
---------- ------------ ------------
       111         'aaa'         'bbb'
1667457792         '222'         'ddd'
```

while the second column is processed correctly. This situation can be avoided in any number of ways, all of which force the data type of the first column to be a string. For example, you can edit the file and explicitly quote the first row of the column:

```
xxx   yyy     zzz
----  ----    ----
"111" aaa     bbb
ccc   222     ddd

[sh] fundisp foo.tab
         XXX           YYY           ZZZ
------------ ------------ ------------
       '111'         'aaa'         'bbb'
       'ccc'         '222'         'ddd'
```

You can edit the file and explicitly set the data type of the first column:

```
xxx:3A   yyy  zzz
------   ---- ----
111      aaa  bbb
ccc      222  ddd

[sh] fundisp foo.tab
         XXX          YYY          ZZZ
------------ ------------ ------------
       '111'        'aaa'        'bbb'
       'ccc'        '222'        'ddd'
```

You also can explicitly set the column names and data types of all columns, without editing the file:

```
[sh] fundisp foo.tab'[TEXT(xxx:3A,yyy:3A,zzz:3a)]'
         XXX          YYY          ZZZ
------------ ------------ ------------
       '111'        'aaa'        'bbb'
       'ccc'        '222'        'ddd'
```

The issue of data type transitions (which to allow and which to disallow) is still under discussion.

Go to Funtools Help Index

Last updated: August 3, 2007

Funview: Database View Support for Tables

Summary

This document contains a summary of the options for utilizing database-inspired Views of tables.

Description

Database Views

In database parlance, a **View** defines a "virtual table", i.e., a description of row and/or column selection filters (but with no permanent storage space allocated). When used in place of a table, a View selects the specified rows and/or columns from one or more real tables. Views enable you to see complicated data tables in a more convenient format. They also can be used as a security mechanism, by restricting user access to specific columns and/or rows. [See:

`http://www.cs.unibo.it/~ciaccia/COURSES/RESOURCES/SQLTutorial/sqlch5.htm`

for a good discussion of SQL Views.]

Funtools supports an expanded notion of Views for all tabular data (FITS tables, raw binary tables, and ASCII column files). Funtools Views allow you to pre-set values for the filter specification, the columns to activate, and display format (though the latter is for fundisp only). Setting the filter and column activation values provides functionality equivalent to that of a classical database View, while the ability to set the format is similar to classical report writing capabilities.

Funtools View Attributes

A Funtools View is a text file containing one or more of the following columns:

```
column          description
------          ----------------------------
view            name of view
file            data file name or template
filter          filter specification
columns         columns to activate
format          fundisp format specification
```

All of the attribute columns are optional, including the **view** name itself. This means that a View can be named or unnamed. Unnamed Views can refer to a specific file or a template of files (obviously if neither the view or the file column is specified, the input View specification will never be used). You can specify any combination of filter, column, and format parameters. (It also is possible to apply file-specific View to other files; see the discussion on **View Lists** below). Each column has a size limit of 1024 characters.

For example, consider the following View file:

```
view    file                    format  columns       filter
----    ----------------------  ------  ------------  -------
x3      ${HOME}/data/snr.ev     I=%4d   x y pi pha    cir 512 512 .1
x2      ${HOME}/data/snr.ev             x y pi pha    cir 512 512 .1
x1      ${HOME}/data/snr.ev                           cir 512 512 .1
x1a     ${HOME}/data/snr.ev             x y pi pha
x0      ${HOME}/data/snr.ev
xf                              I=%4d
xc                                      x y pi pha
xr                                                    cir 512 512 .1
        *.ev                            x y pi pha
        *.fit                           x y dx dy     cir 400 400  3
        *.fits                  I=%3d   x y dx dy     cir 400 400  3
```

This database example is in rdb format, i.e. using tab delimiters and permitting null values. Any valid ASCII table format is acceptable, but if you use a format that does not permit null values, it will be necessary to quote the null strings.

The first five entries (x3, x2, x1, x1a, x0) are named entries defining default values specifically for the snr.ev data file. Typically, you would use these Views by specifying View name, and the corresponding file, filter, column, and format values would be used. Note that the x0 View is essentially an alias for the pathname of this file.

The next three entries define defaults that can be applied to any file. You typically would use these View names in conjunction with a specific file name (see **View Lists** below) so that the associated parameter(s) were applied to that file.

The last three entry in the database define unnamed Views that pertains to all files ending with the specified templates. In these cases, any View that specifies a file name matching the file template would be processed with the associated parameter attributes.

Invoking a Funtools View (in Place of an Input File)

To use a Funtools View, you simply pre-pend the "v:" prefix to a View name or a file name where an input file name usually is specified. For example:

```
fundisp v:x3
```

specifies that the View named x3 (with its file name and associated parameters) is processed as the input file to fundisp. Using the example database, above, this is equivalent to:

```
fundisp  -f "I=%4d" ${HOME}/data/snr.ev'[cir 512 512 .1]'  "x y pi pha"
```

That is, the format is used with fundisp's -f (format) switch, while the filename and extension are composed of the x3 View's filename and region filter.

Similarly, executing a command such as:

```
fundisp v:foo.fit
```

will match the unnamed View associated with the template "*.fit". This is equivalent to executing:

```
fundisp foo.fit'[cir 400 400 3]' "x y dx dy"
```

Of course, if you omit the "v:" prefix, then no View processing takes place:

```
fundisp foo.fit    # process foo.fit without any View parameters
fundisp x3         # error (assuming there is no file named x3)
```

Basic View Matching Rules

When a "v:" prefix is recognized, Funtools searches for a View database file in the following order:

```
location             description
------------         ------------------------------------
FUN_VIEWFILE         environment variable (any file name)
./.funtools.vu       hidden file, default name
$HOME/.funtools.vu   hidden file, default name
```

The first View database file located is used to construct a new filename, as well as an activation column specification and a format specification. The following rules are used:

1. An attempt is made to match the input name (i.e., the part of the input View after the "v:" prefix) against the **view** column value (if present) of each row in the database. If a match is found, the values of all non-blank columns are saved for later use. Also note that the first match terminates the search: i.e., **the order of the database rows matters**.

2. If no **view** match is made, an attempt is made to match the input name against the **file** column value (if present). Matching is performed on the full pathname of both the input name and the database file name, and on the non-directory (root) part of these files. This means that the root specification:

```
fundisp v:snr.ev
```

will match a row in the database that has a full pathname in the file, allowing you to use a **file**-matched View without having to specify the full pathname. In this example, the "v:snr.ev" View specification will match the first row (v:x3) in the database:

```
x3   ${HOME}/data/snr.ev     I=%4d   x y pi pha    cir 512 512 .1
```

even though the row contains a fully qualified pathname as the file value. Once again, values of all non-blank columns are saved, and the first match terminates the search.

3. If neither a **view** or a **view** match has been found, then a simple template match is attempted against the **view** values. Template matching supports a simplified version of file globbing (not a regular expression), with support for a single "*" (all characters), "?" (single character), or "[...]" (range) specification.

4. If no template match was found on the **view** column, then a simple template match is attempted against the **file** columns.

5. If no match is found, then the filename (minus the "v:" prefix) is returned.

More on View Matching Rules: Single vs. Multiple Matches

The matching rules described above stop after the first match, regardless of whether that match provides values for all three parameters (filter, columns, and format). In cases where a **view** or **file** match does not provide all three values, it is possible that a template match might do so. With regard to the example View database above, the x1 View provides only a filter, while omitting both the format and columns values. But note that the final rows in the database could provide the values via a template match on the filename. This sort of multiple matching is especially valuable in order to provide "global" values to several Views.

Obviously, multiple matching might not be wanted in every case. Therefore, we support both multiple matching and single matching according to the value of the FUN_VIEWMATCH environment variable. If the FUN_VIEWMATCH environment variable exists and if its value begins with "s", then a single match is used and missing parameters are not filled in with subsequent template matches on the file name. That is, matching rules above are followed exactly as explained above. If the value of this environment variable begins with "m" (or does not exist), then multiple matches are used to try to fill in missing parameters. In this case, template matching always takes place and missing values are taken from these template matches.

Thus, in the example above, the View specification:

```
fundisp v:x1
```

will take the file name and filter value from the x1 View:

```
x1      ${HOME}/data/snr.ev                          cir 512 512 .1
```

The column value then will be taken from the "*.ev" file template match against the x1 file name:

```
        *.ev                            x y pi pha
```

Note once again that order is important: missing values are taken in the order in which the template matches are processed.

View Lists: Applying a View to Any File

It is possible to apply a named View, or even several Views, to any data file by appending a **viewlist** immediately after the standard "v:" prefix. A viewlist takes the form:

```
:v1,v2,...vn:
```

where v1, v2, etc. are named Views. The two ":" colon characters surrounding the list are required. Thus, the syntax for applying a viewlist to a file is:

```
v::view1,view2,...viewn:filename
```

Note that the name after the last ":" is assumed to be a file; it is not permissible (or sensible) to use a View name.

For example, the View specification:

```
fundisp v::x2:foo
```

applies the x2 View to the file foo (even if there is a View named foo) and (in using our example database) is equivalent to:

```
./fundisp foo'[cir 512 512 .1] "x y pi pha"
```

The same command can be effected using a list of Views:

```
fundisp v::x1,x1a:foo
```

What happens if a viewlist is used and the file also matches a template? Consider, for example, this View specification:

```
fundisp v::x2:foo.fit
```

Here, the x2 View will supply filter and column values, while the template *.fit can also supply (different) filter and column values. In this case, the explicitly specified Views of the viewlist trump the matched view values.

On the other hand, if a file template match can supply a View value that is not supplied by the viewlist, then that value will be taken from the file template match. For example:

```
fundisp v::x2:foo.fits
```

does not explicitly supply a format value, but the file match on *.fits can and does. You can avoid supplying missing values using file template matching by replacing the first ":" with a "-" in a viewlist specification:

```
fundisp v:-x2:foo.fits
```

The use of ":+" to explicitly allow file template matching is also supported, but is the same as the default case. Note that the nuances of viewlist support are subject to change as our experience and understanding grow.

Overriding Values Associated with a View

To override values associated with a View, simply supply the override values in the correct place on the command line. Thus, given the example database described above, the command:

```
fundisp v:x3
```

specifies that the View named x3, along with its file name and associated parameters, be processed as the input file to fundisp in this way:

```
fundisp  -f "I=%4d" ${HOME}/data/snr.ev'[cir 512 512 .1]'  "x y pi pha"
```

To override one or more of these values, simply specify a new value for the format, filter, or columns. For example, if your input View file contains a filter, then the View will use that filter as an override of the

View filter:

```
fundisp v:x3'[cir 400 400 3]'
```

will use the columns and format of the x3 View but not the x3 filter. Further examples are:

```
fundisp v:x3 "x y dx dy"    # activate a different set of columns
fundisp -f "I=%3d" v:x3     # use a different format statement
```

Note that extension names, extension index values, and other non-filter specifications **do not** override the View filter. Thus:

```
fundisp v:foo.fit[3]
```

will still use the filter associated with the .fit template (see above), since the "3" is an extension index, not a filter.

Environment Variables

The following environment variables are used by Funtools Views:

FUN_VIEWNAME
: The **FUN_VIEWNAME** environment variable specifies the name and location of the View database file. If not present, the files ./.funtools.vu and $HOME/.funtools.vu are searched for, in that order.

FUN_VIEWMATCH
: The **FUN_VIEWMATCH** environment variable specifies whether a single match or multiple match algorithm is used to locate parameter values. If the value of this environment variable begins with "s", then a single match is used and missing parameters are not filled in with subsequent template matches on the file name. If the value begins with "m", then multiple matches are used to try to fill in missing parameters. The default is to use multiple matches.

Restrictions and Problems

Support for overriding a filter (while not overriding extension names, extension indexes, etc.) requires that we can sense the presence of a filter in a bracket specification. It is unclear yet whether our algorithm is perfect.

Go to Funtools Help Index

Last updated: August 3, 2007

Funfilters: Filtering Rows in a Table

Summary

This document contains a summary of the user interface for filtering rows in binary tables.

Description

Table filtering allows a program to select rows from an table (e.g., X-ray event list) by checking each row against one or more expressions involving the columns in the table. When a table is filtered, only valid rows satisfying these expressions are passed through for processing.

A filter expression is specified using bracket notation appended to the filename of the data being processed:

```
foo.fits[pha==1&π==2]
```

It is also possible to put region specification inside a file and then pass the filename in bracket notation:

```
foo.fits[@my.reg]
```

Filters must be placed after the extension and image section information, when such information is present. The correct order is:

- file[fileinfo,sectioninfo][filters]
- file[fileinfo,sectioninfo,filters]

where:

- **file** is the Funtools file name
- **fileinfo** is an ARRAY, EVENT, FITS extension, or FITS index
- **sectioninfo** is the image section to extract
- **filters** are spatial region and table (row) filters to apply

See Funtools Files for more information on file and image section specifications.

Filter Expressions

Table filtering can be performed on columns of data in a FITS binary table or a raw event file. Table filtering is accomplished by means of **table filter specifications**. An table filter specification consists of one or more **filter expressions** Filter specifications also can contain comments and local/global processing directives.

More specifically, a filter specification consist of one or more lines containing:

```
# comment until end of line
# include the following file in the table descriptor
@file
# each row expression can contain filters separated by operators
[filter_expression] BOOLOP [filter_expression2], ...
# each row expression can contain filters separated by the comma operator
[filter_expression1], [filter_expression2], ...
# the special row# keyword allows a range of rows to be processed
row#=m:n
# or a single row
row#=m
# regions are supported -- but are described elsewhere
[spatial_region_expression]
```

A single filter expression consists of an arithmetic, logical, or other operations involving one or more column values from a table. Columns can be compared to other columns, to header values, or to numeric constants. Standard math functions can be applied to columns. Separate filter expressions can be combined using boolean operators. Standard C semantics can be used when constructing expressions, with the usual precedence and associativity rules holding sway:

```
Operator                                Associativity
--------                                -------------
()                                      left to right
!! (logical not)                        right to left
!  (bitwise not) - (unary minus)        right to left
*  /                                    left to right
+  -                                    left to right
< <= > >=                               left to right
== !=                                   left to right
&  (bitwise and)                        left to right
^  (bitwise exclusive or)               left to right
|  (bitwise inclusive or)               left to right
&& (logical and)                        left to right
|| (logical or)                         left to right
=                                       right to left
```

For example, if energy and pha are columns in a table, then the following are valid expressions:

```
pha>1
energy == pha
(pha>1) && (energy<=2)
max(pha,energy)>=2.5
```

Comparison values can be integers or floats. Integer comparison values can be specified in decimal, octal (using '0' as prefix), hex (using '0x' as prefix) or binary (using '0b' as prefix). Thus, the following all specify the same comparison test of a status mask:

```
(status & 15) == 8           # decimal
(status & 017) == 010        # octal
(status & 0xf) == 0x8        # hex
(status & 0b1111) == 0b1000  # binary
```

The special keyword row# allows you to process a range of rows. When row# is specified, the filter code skips to the designated row and only processes the specified number of rows. The "*" character can be utilized as the high limit value to denote processing of the remaining rows. Thus:

```
row#=100:109
```

processes 10 rows, starting with row 100 (counting from 1), while:

```
row#=100:*
```

specifies that all but the first 99 rows are to be processed.

Spatial region filtering allows a program to select regions of an image or rows of a table (e.g., X-ray events) using simple geometric shapes and boolean combinations of shapes. For a complete description of regions, see Spatial Region Filtering.

Separators Also Are Operators

As mentioned previously, multiple filter expressions can be specified in a filter descriptor, separated by commas or new-lines. When such a comma or new-line separator is used, the boolean AND operator is automatically generated in its place. Thus and expression such as:

```
pha==1,pi=2:4
```

is equivalent to:

```
(pha==1) && (pi>=2&π<=4)
```

[Note that the behavior of separators is different for filter expressions and spatial region expressions. The former uses AND as the operator, while the latter user OR. See Combining Region and Table Filters for more information about these conventions and how they are treated when combined.]

Range Lists

Aside from the standard C syntax, filter expressions can make use of IRAF-style **range lists** which specify a range of values. The syntax requires that the column name be followed by an '=' sign, which is followed by one or more comma-delimited range expressions of the form:

```
col = vv              # col == vv in range
col = :vv             # col <= vv in range
col = vv:             # col >= vv in range
col = vv1:vv2         # vv1 <= col <= vv2 in range
```

The vv's above must be numeric constants; the right hand side of a range list cannot contain a column name or header value.

Note that, unlike an ordinary comma separator, the comma separator used between two or more range expressions denotes OR. Thus, when two or more range expressions are combined with a comma separator, the resulting expression is a shortcut for more complicated boolean logic. For example:

```
col = :3,6:8,10:
```

is equivalent to:

```
(col<=3) || (col>=6 && col <=8) || (col >=10)
```

Note also that the single-valued rangelist:

```
col = val
```

is equivalent to the C-based filter expression:

```
col == val
```

assuming, of course, that val is a numeric constant.

Math Operations and Functions

It is permissible to specify C math functions as part of the filter syntax. When the filter parser recognizes a function call, it automatically includes the math.h and links in the C math library. Thus, it is possible to filter rows by expressions such as these:

- (pi+pha)>(2+log(pi)-pha)
- min(pi,pha)*14>x
- max(pi,pha)==(pi+1)
- feq(pi,pha)
- div(pi,pha)>0

The function feq(a,b) returns true (1) if the difference between a and b (taken as double precision values) is less than approximately 10E-15. The function div(a,b) divides a by b, but returns NaN (not a number) if b is 0. It is a safe way to avoid floating point errors when dividing one column by another.

Include Files

The special **@filename** directive specifies an include file containing filter expressions. This file is processed as part of the overall filter descriptor:

```
foo.fits[pha==1,@foo]
```

Header Parameters

The filter syntax supports comparison between a column value and a header parameter value of a FITS binary tables (raw event files have no such header). The header parameters can be taken from the binary table header or the primary header. For example, assuming there is a header value MEAN_PHA in one of these headers, you can select photons having exactly this value using:

- pha==MEAN_PHA

Examples

Table filtering is more easily described by means of examples. Consider data containing the following table structure:

- double TIME
- int X
- int Y
- short PI
- short PHA
- int DX
- int DY

Tables can be filtered on these columns using IRAF/QPOE range syntax or any valid C syntax. The following examples illustrate the possibilities:

pha=10
pha==10
: select rows whose pha value is exactly 10

pha=10:50
: select rows whose pha value is in the range of 10 to 50

pha=10:50,100
: select rows whose pha value is in the range of 10 to 50 or is equal to 100

pha>=10 && pha<=50
: select rows whose pha value is in the range of 10 to 50

pi=1,2&&pha>3
: select rows whose pha value is 1 or 2 and whose pi value is 3

pi=1,2 || pha>3
: select rows whose pha value is 1 or 2 or whose pi value is 3

pha==pi+1
: select rows whose pha value is 1 less than the pi value

(pha==pi+1) && (time>50000.0)
: select rows whose pha value is 1 less than the pi value and whose time value is greater than 50000

(pi+pha)>20
: select rows in which the sum of the pi and pha values is greater than 20

pi%2==1
: select rows in which the pi value is odd

Currently, integer range list limits cannot be specified in binary notation (use decimal, hex, or octal instead). Please contact us if this is a problem.

Go to Funtools Help Index

Last updated: November 17, 2005

Funidx: Using Indexes to Filter Rows in a Table

Summary

This document contains a summary of the user interface for filtering rows in binary tables with indexes.

Description

Funtools Table Filtering allows rows in a table to be selected based on the values of one or more columns in the row. Because the actual filter code is compiled on the fly, it is very efficient. However, for very large files (hundreds of Mb or larger), evaluating the filter expression on each row can take a long time. Therefore, funtools supports index files for columns, which are used automatically during filtering to reduce dramatically the number of row evaluations performed. The speed increase for indexed filtering can be an order of magnitude or more, depending on the size of the file.

The funindex program creates an index on one or more columns in a binary table. For example, to create an index for the column pi in the file huge.fits, use:

```
funindex huge.fits pi
```

This will create an index named huge_pi.idx.

When a filter expression is initialized for row evaluation, funtools looks for an index file for each column in the filter expression. If found, and if the file modification date of the index file is later than that of the data file, then the index will be used to reduce the number of rows that are evaluated in the filter. When Spatial Region Filtering is part of the expression, the columns associated with the region are checked for index files.

If an index file is not available for a given column, then in general, all rows must be checked when that column is part of a filter expression. This is not true, however, when a non-indexed column is part of an AND expression. In this case, only the rows that pass the other part of the AND expression need to be checked. Thus, in some cases, filtering speed can increase significantly even if all columns are not indexed.

Also note that certain types of filter expression syntax cannot make use of indices. For example, calling functions with column names as arguments implies that all rows must be checked against the function value. Once again, however, if this function is part of an AND expression, then a significant improvement in speed still is possible if the other part of the AND expression is indexed.

For example, note below the dramatic speedup in searching a 1 Gb file using an AND filter, even when one of the columns (pha) has no index:

```
time fundisp \
huge.fits'[idx_activate=0,idx_debug=1,pha=2348&&cir 4000 4000 1]' \
"x y pha"
         x            y         pha
---------- ------------ ----------
   3999.48      4000.47       2348
```

```
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   42.36u 13.07s 6:42.89 13.7%

 time fundisp \
 huge.fits'[idx_activate=1,idx_debug=1,pha=2348&&cir 4000 4000 1]' \
 "x y pha"
         x           y         pha
---------- ----------- ----------
idxeq: [INDEF]
idxand sort: x[ROW 8037025:8070128] y[ROW 5757665:5792352]
idxand(1): INDEF [IDX_OR_SORT]
idxall(1): [IDX_OR_SORT]
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   3999.48     4000.47       2348
   1.55u 0.37s 1:19.80 2.4%
```

When all columns are indexed, the increase in speed can be even more dramatic:

```
 time fundisp \
 huge.fits'[idx_activate=0,idx_debug=1,pi=770&&cir 4000 4000 1]' \
 "x y pi"
         x           y          pi
---------- ----------- ----------
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
```

```
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   42.60u 12.63s 7:28.63 12.3%

 time fundisp \
 huge.fits'[idx_activate=1,idx_debug=1,pi=770&&cir 4000 4000 1]' \
 "x y pi"
         x           y          pi
---------- ----------- ----------
idxeq: pi start=9473025,stop=9492240 => pi[ROW 9473025:9492240]
idxand sort: x[ROW 8037025:8070128] y[ROW 5757665:5792352]
idxor sort/merge: pi[ROW 9473025:9492240] [IDX_OR_SORT]
idxmerge(5): [IDX_OR_SORT] pi[ROW]
idxall(1): [IDX_OR_SORT]
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   3999.48     4000.47        770
   1.67u 0.30s 0:24.76 7.9%
```

The miracle of indexed filtering (and indeed, of any indexing) is the speed of the binary search on the index, which is of order log2(n) instead of n. (The funtools binary search method is taken from http://www.tbray.org/ongoing/When/200x/2003/03/22/Binary, to whom grateful acknowledgement is made.) This means that the larger the file, the better the performance. Conversely, it also means that for small files, using an index (and the overhead involved) can slow filtering down somewhat. Our tests indicate that on a file containing a few tens of thousands of rows, indexed filtering can be 10 to 20 percent slower than non-indexed filtering. Of course, your mileage will vary with conditions (disk access speed, amount of available memory, process load, etc.)

Any problem encountered during index processing will result in indexing being turned off, and replaced by filtering all rows. You can turn filtering off manually by setting the idx_activate variable to 0 (in a filter expression) or the FILTER_IDX_ACTIVATE environment variable to 0 (in the global environment). Debugging output showing how the indexes are being processed can be displayed to stderr by setting the idx_debug variable to 1 (in a filter expression) or the FILTER_IDX_DEBUG environment variable to 1

(in the global environment).

Currently, indexed filtering only works with FITS binary tables and raw event files. It does not work with text files. This restriction might be removed in a future release.

Go to Funtools Help Index

Last updated: August 3, 2007

Regions: Spatial Region Filtering

Summary

This document contains a summary of the user interface for spatial region filtering images and tables.

Description

Spatial region filtering allows a program to select regions of an image or rows of a table (e.g., X-ray events) to process using simple geometric shapes and boolean combinations of shapes. When an image is filtered, only pixels found within these shapes are processed. When a table is filtered, only rows found within these shapes are processed.

Spatial region filtering for images and tables is accomplished by means of **region specifications**. A region specification consists of one or more **region expressions**, which are geometric shapes,combined according to the rules of boolean algebra. Region specifications also can contain comments and local/global processing directives.

Typically, region specifications are specified using bracket notation appended to the filename of the data being processed:

```
foo.fits[circle(512,512,100)]
```

It is also possible to put region specification inside a file and then pass the filename in bracket notation:

```
foo.fits[@my.reg]
```

When region filters are passed in bracket notation in this manner, the filtering is set up automatically when the file is opened and all processing occurs through the filter. Programs also can use the filter library API to open filters explicitly.

Region Expressions

More specifically, region specifications consist of one or more lines containing:

```
# comment until end of line
global   keyword=value keyword=value  ... # set global value(s)
# include the following file in the region descriptor
@file
# use the FITS image as a mask (cannot be used with other regions)
@fitsimage
# each region expression contains shapes separated by operators
[region_expression1], [region_expression2], ...
[region_expression], [region_expression], ...
```

A single region expression consists of:

```
  # parens and commas are optional, as is the + sign
  [+-]shape(num , num , ...) OP1 shape num num num OP2 shape ...

e.g.:

  ([+-]shape(num , num , ...) && shape num  num || shape(num, num)
  # a comment can come after a region -- reserved for local properties
  [+-]shape(num , num , ...)  # local properties go here, e.g. color=red
```

Thus, a region descriptor consists of one or more **region expressions** or **regions**, separated by comas, new-lines, or semi-colons. Each **region** consists of one or more **geometric shapes** combined using standard boolean operation. Several types of shapes are supported, including:

```
  shape:        arguments:
  -----         ----------------------------------------
  ANNULUS       xcenter ycenter inner_radius outer_radius
  BOX           xcenter ycenter xwidth yheight (angle)
  CIRCLE        xcenter ycenter radius
  ELLIPSE       xcenter ycenter xwidth yheight (angle)
  FIELD         none
  LINE          x1 y1 x2 y2
  PIE           xcenter ycenter angle1 angle2
  POINT         x1 y1
  POLYGON       x1 y1 x2 y2 ... xn yn
```

In addition, the following regions accept **accelerator** syntax:

```
  shape      arguments
  -----      ----------------------------------------
  ANNULUS    xcenter ycenter radius1 radius2 ... radiusn
  ANNULUS    xcenter ycenter inner_radius outer_radius n=[number]
  BOX        xcenter ycenter xw1 yh1 xw2 yh2 ... xwn yhn (angle)
  BOX        xcenter ycenter xwlo yhlo xwhi yhhi n=[number] (angle)
  CIRCLE     xcenter ycenter r1 r2 ... rn              # same as annulus
  CIRCLE     xcenter ycenter rinner router n=[number]  # same as annulus
  ELLIPSE    xcenter ycenter xw1 yh1 xw2 yh2 ... xwn yhn (angle)
  ELLIPSE    xcenter ycenter xwlo yhlo xwhi yhhi n=[number] (angle)
  PIE        xcenter ycenter angle1 angle2 (angle3) (angle4) (angle5) ...
  PIE        xcenter ycenter angle1 angle2 (n=[number])
  POINT      x1 y1 x2 y2 ... xn yn
```

Note that the circle accelerators are simply aliases for the annulus accelerators. See region geometry for more information about accelerators.

Finally, the following are combinations of pie with different shapes (called "panda" for "Pie AND Annulus") allow for easy specification of radial sections:

```
  shape:  arguments:
  -----   ---------
  PANDA   xcen ycen ang1 ang2 nang irad orad nrad   # circular
  CPANDA  xcen ycen ang1 ang2 nang irad orad nrad   # circular
  BPANDA  xcen ycen ang1 ang2 nang xwlo yhlo xwhi yhhi nrad (ang) # box
  EPANDA  xcen ycen ang1 ang2 nang xwlo yhlo xwhi yhhi nrad (ang) # ellipse
```

The panda and cpanda specify combinations of annulus and circle with pie, respectively and give identical

results. The bpanda combines box and pie, while epanda combines ellipse and pie. See region geometry for more information about pandas.

The following "shapes" are ignored by funtools (generated by ds9):

```
shape:        arguments:
-----         ---------
PROJECTION    x1 y1 x2 y2 width    # NB: ignored by funtools
RULER         x1 y1 x2 y2          # NB: ignored by funtools
TEXT          x y                  # NB: ignored by funtools
GRID                               # NB: ignored by funtools
TILE                               # NB: ignored by funtools
COMPASS                            # NB: ignored by funtools
```

All arguments to regions are real values; integer values are automatically converted to real where necessary. All angles are in degrees and run from the positive image x-axis to the positive image y-axis. If a rotation angle is part of the associated WCS header, that angle is added implicitly as well.

Note that 3-letter abbreviations are supported for all shapes, so that you can specify "circle" or "cir".

Columns Used in Region Filtering

By default, the x,y values in a region expression refer to the two "image binning" columns, i.e. the columns that would be used to bin the data into an image. For images, these are just the 2 dimensions of the image. For tables, these usually default to x and y but can be changed as required. For example, in Funtools, new binning columns are specified using a bincols=(col1,col2) statement within the bracket string on the command line.

Alternate columns for region filtering can be specified by the syntax:

```
(col1,col2)=region(...)
```

e.g.:

```
(X,Y)=annulus(x,y,ri,ro)
(PHA,PI)=circle(x,y,r)
(DX,DY)=ellipse(x,y,a,b[,angle])
```

Region Algebra

(See also Region Algebra for more complete information.)

Region shapes can be combined together using Boolean operators:

```
Symbol        Operation       Use
--------      ---------       -----------------------------------
!             not             Exclude this shape from this region
& or &&       and             Include only the overlap of these shapes
| or ||       inclusive or    Include all of both shapes
^             exclusive or    Include both shapes except their overlap
```

Note that the !region syntax must be combined with another region in order that we be able to assign a region id properly. That is,

```
!circle(512,512,10)
```

is not a legal region because there is no valid region id to work with. To get the full field without a circle, combine the above with field(), as in:

```
field() && !circle(512,512,10)
```

Region Separators Also Are Operators

As mentioned previously, multiple region expressions can be specified in a region descriptor, separated by commas, new-lines, or semi-colons. When such a separator is used, the boolean OR operator is automatically generated in its place but, unlike explicit use of the OR operator, the region ID is incremented (starting from 1).

For example, the two shapes specified in this example are given the same region value:

```
foo.fits[circle(512,512,10)||circle(400,400,20)]
```

On the other hand, the two shapes defined in the following example are given different region values:

```
foo.fits[circle(512,512,10),circle(400,400,20)]
```

Of course these two examples will both mask the same table rows or pixels. However, in programs that distinguish region id's (such as funcnts), they will act differently. The explicit OR operator will result in one region expression consisting of two shapes having the same region id and funcnts will report a single region. The comma operator will cause funcnts to report two region expressions, each with one shape, in its output.

In general, commas are used to separate region expressions entered in bracket notation on the command line:

```
# regions are added to the filename in bracket notation
foo.fits[circle(512,512,100),circle(400,400,20)]
```

New-lines are used to separate region expressions in a file:

```
# regions usually are separated by new-lines in a file
# use @filename to include this file on the command line
circle(512,512,100)
circle(400,400,20)
```

Semi-colons are provided for backward compatibility with the original IRAF/PROS implementation and can be used in either case.

If a pixel is covered by two different regions expressions, it is given the mask value of the **first** region that contains that pixel. That is, successive regions **do not** overwrite previous regions in the mask, as was the case with the original PROS regions. In this way, an individual pixel is covered by one and only one region. This means that one must sometimes be careful about the order in which regions are defined. If

region N is fully contained within region M, then N should be defined **before** M, or else it will be "covered up" by the latter.

Region Exclusion

Shapes also can be globally excluded from all the region specifiers in a region descriptor by using a minus sign before a region:

```
operator      arguments:
--------      -----------
-             Globally exclude the region expression following '-' sign
              from ALL regions specified in this file
```

The global exclude region can be used by itself; in such a case, field() is implied.

A global exclude differs from the local exclude (i.e. a shape prefixed by the logical not "!" symbol) in that global excludes are logically performed last, so that no region will contain pixels from a globally excluded shape. A local exclude is used in a boolean expression with an include shape, and only excludes pixels from that include shape. Global excludes cannot be used in boolean expressions.

Include Files

The **@filename** directive specifies an include file containing region expressions. This file is processed as part of the overall region descriptor:

```
foo.fits[circle(512,512,10),@foo]
```

A filter include file simply includes text without changing the state of the filter. It therefore can be used in expression. That is, if the file foo1 contains "pi==1" and foo2 contains "pha==2" then the following expressions are equivalent:

```
"[@foo1&&@foo2]"   is equivalent to   "[pi==1&&pha==2]"
"[pha==1||@foo2]"  is equivalent to   "[pi==1||pha==2]"
"[@foo1,@foo2]"    is equivalent to   "[pi==1,pha==2]"
```

Be careful that you specify evaluation order properly using parenthesis, especially if the include file contains multiple filter statements. For example, consider a file containing two regions such as:

```
circle 512 512 10
circle 520 520 10
```

If you want to include only events (or pixels) that are in these regions and have a pi value of 4, then the correct syntax is:

```
  pi==4&&(@foo)
```

since this is equivalent to:

```
pi==4 && (circle 512 512 10 || circle 520 520 10)
```

If you leave out the parenthesis, you are filtering this statement:

```
pi==4 && circle 512 512 10 || circle 520 520 10)
```

which is equivalent to:

```
(pi==4 && circle 512 512 10) || circle 520 520 10)
```

The latter syntax only applies the pi test to the first region.

For image-style filtering, the **@filename** can specify an 8-bit or 16-bit FITS image. In this case, the pixel values in the mask image are used as the region mask. The valid pixels in the mask must have positive values. Zero values are excluded from the mask and negative values are not allowed. Moreover, the region id value is taken as the image pixel value and the total number of regions is taken to be the highest pixel value. The dimensions of the image mask must be less than or equal to the image dimensions of the data. The mask will be replicated as needed to match the size of the image. (Thus, best results are obtained when the data dimensions are an even multiple of the mask dimensions.)

An image mask can be used in any image filtering operation, regardless of whether the data is of type image or table. For example, the funcnts) program performs image filtering on images or tables, and so FITS image masks are valid input for either type of data in this program.. An image mask cannot be used in a program such as fundisp) when the input data is a table, because fundisp displays rows of a table and processes these rows using event-style filtering.

Global and Local Properties of Regions

The ds9 image display program describes a host of properties such as color, font, fix/free state, etc. Such properties can be specified globally (for all regions) or locally (for an individual region). The **global** keyword specifies properties and qualifiers for all regions, while local properties are specified in comments on the same line as the region:

```
global color=red
circle(10,10,2)
circle(20,20,3) # color=blue
circle(30,30,4)
```

The first and third circles will be red, which the second circle will be blue. Note that funtools currently ignores region properties, as they are used in display only.

Coordinate Systems

For each region, it is important to specify the coordinate system used to interpret the region, i.e., to set the context in which position and size values are interpreted. For this purpose, the following keywords are recognized:

```
name                  description
----                  -----------------------------------------
PHYSICAL              pixel coords of original file using LTM/LTV
IMAGE                 pixel coords of current file
FK4, B1950            sky coordinate systems
FK5, J2000            sky coordinate systems
GALACTIC              sky coordinate systems
ECLIPTIC              sky coordinate systems
ICRS                  currently same as J2000
LINEAR                linear wcs as defined in file
AMPLIFIER             mosaic coords of original file using ATM/ATV
DETECTOR              mosaic coords of original file using DTM/DTV
```

Specifying Positions, Sizes, and Angles

The arguments to region shapes can be floats or integers describing positions and sizes. They can be specified as pure numbers or using explicit formatting directives:

```
position arguments    description
------------------    ------------------------------
[num]                 context-dependent (see below)
[num]d                degrees
[num]r                radians
[num]p                physical pixels
[num]i                image pixels
[num]:[num]:[num]     hms for 'odd' position arguments
[num]:[num]:[num]     dms for 'even' position arguments
[num]h[num]m[num]s    explicit hms
[num]d[num]m[num]s    explicit dms

size arguments        description
--------------        -----------
[num]                 context-dependent (see below)
[num]"                arc seconds
[num]'                arc minutes
[num]d                degrees
[num]r                radians
[num]p                physical pixels
[num]i                image pixels
```

When a "pure number" (i.e. one without a format directive such as 'd' for 'degrees') is specified, its interpretation depends on the context defined by the 'coordsys' keyword. In general, the rule is:

All pure numbers have implied units corresponding to the current coordinate system.

If no such system is explicitly specified, the default system is implicitly assumed to be PHYSICAL.

In practice this means that for IMAGE and PHYSICAL systems, pure numbers are pixels. Otherwise, for all systems other than linear, pure numbers are degrees. For LINEAR systems, pure numbers are in the units of the linear system. This rule covers both positions and sizes.

The input values to each shape can be specified in several coordinate systems including:

```
name                  description
----                  ---------------------------
IMAGE                 pixel coords of current file
LINEAR                linear wcs as defined in file
FK4, B1950            various sky coordinate systems
FK5, J2000
GALACTIC
ECLIPTIC
ICRS
PHYSICAL              pixel coords of original file using LTM/LTV
AMPLIFIER             mosaic coords of original file using ATM/ATV
DETECTOR              mosaic coords of original file using DTM/DTV
```

If no coordinate system is specified, PHYSICAL is assumed. PHYSICAL or a World Coordinate System such as J2000 is preferred and most general. The coordinate system specifier should appear at the beginning of the region description, on a separate line (in a file), or followed by a new-line or semicolon; e.g.,

```
global coordsys physical
circle 6500 9320 200
```

The use of celestial input units automatically implies WORLD coordinates of the reference image. Thus, if the world coordinate system of the reference image is J2000, then

```
circle 10:10:0 20:22:0 3’
```

is equivalent to:

```
circle 10:10:0 20:22:0 3’ # j2000
```

Note that by using units as described above, you may mix coordinate systems within a region specifier; e.g.,

```
circle 6500 9320 3’ # physical
```

Note that, for regions which accept a rotation angle:

```
ellipse (x, y, r1, r2, angle)
box(x, y, w, h, angle)
```

the angle is relative to the specified coordinate system. In particular, if the region is specified in WCS coordinates, the angle is related to the WCS system, not x/y image coordinate axis. For WCS systems with no rotation, this obviously is not an issue. However, some images do define an implicit rotation (e.g., by using a non-zero CROTA value in the WCS parameters) and for these images, the angle will be relative to the WCS axes. In such case, a region specification such as:

```
fk4;ellipse(22:59:43.985, +58:45:26.92,320", 160", 30)
```

will not, in general, be the same region specified as:

```
physical;ellipse(465, 578, 40, 20, 30)
```

even when positions and sizes match. The angle is relative to WCS axes in the first case, and relative to physical x,y axes in the second.

More detailed descriptions are available for: Region Geometry, Region Algebra, Region Coordinates, and Region Boundaries.

Go to Funtools Help Index

Last updated: November 17, 2005

RegGeometry: Geometric Shapes in Spatial Region Filtering

Summary

This document describes the geometry of regions available for spatial filtering in IRAF/PROS analysis.

Geometric shapes

Several geometric shapes are used to describe regions. The valid shapes are:

```
shape:        arguments:
-----         ---------------------------------------
ANNULUS       xcenter ycenter inner_radius outer_radius
BOX           xcenter ycenter xwidth yheight (angle)
CIRCLE        xcenter ycenter radius
ELLIPSE       xcenter ycenter xwidth yheight (angle)
FIELD         none
LINE          x1 y1 x2 y2
PIE           xcenter ycenter angle1 angle2
POINT         x1 y1
POLYGON       x1 y1 x2 y2 ... xn yn
```

All arguments are real values; integer values are automatically converted to real where necessary. All angles are in degrees and specify angles that run counter-clockwise from the positive y-axis.

Shapes can be specified using "command" syntax:

```
[shape] arg1 arg2 ...
```

or using "routine" syntax:

```
[shape](arg1, arg2, ...)
```

or by any combination of the these. (Of course, the parentheses must balance and there cannot be more commas than necessary.) The shape keywords are case-insensitive. Furthermore, any shape can be specified by a three-character unique abbreviation. For example, one can specify three circular regions as:

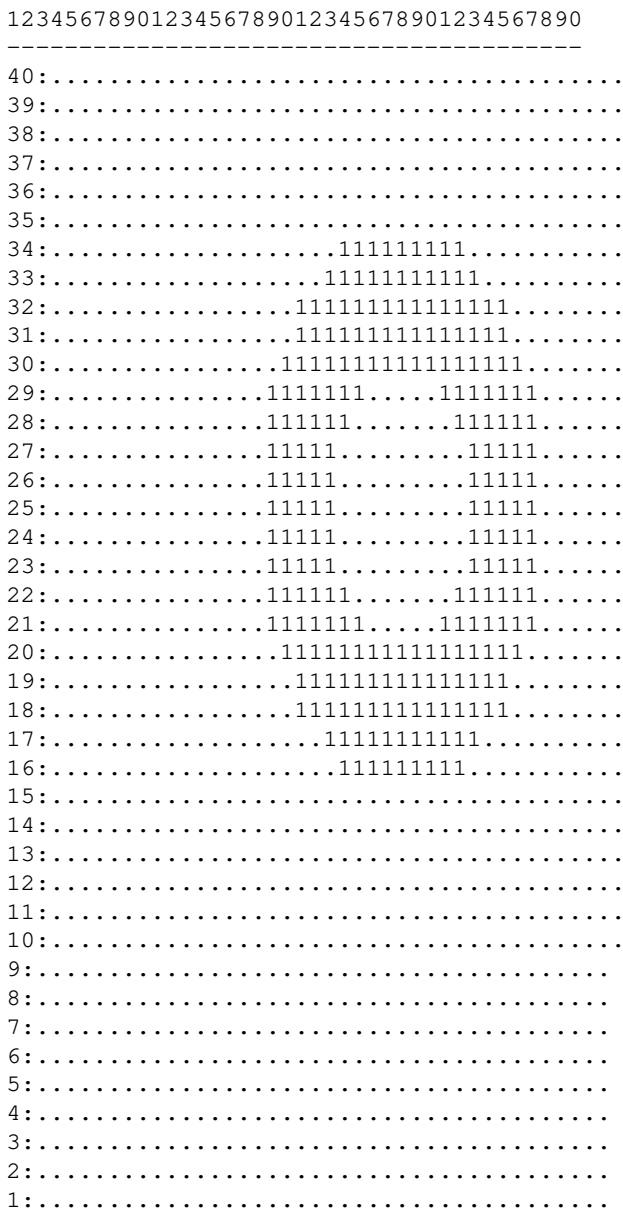
```
"foo.fits[CIRCLE 512 512 50;CIR(128 128, 10);cir(650,650,20)]"
```

(Quotes generally are required to protect the region descriptor from being processed by the Unix shell.)

The **annulus** shape specifies annuli, centered at xcenter, ycenter, with inner and outer radii (r1, r2). For example,

```
ANNULUS 25 25 5 10
```

specifies an annulus centered at 25.0 25.0 with an inner radius of 5.0 and an outer radius of 10. Assuming (as will be done for all examples in this document, unless otherwise noted) this shape is used in a mask of size 40x40, it will look like this:



The **box** shape specifies an orthogonally oriented box, centered at xcenter, ycenter, of size xwidth, yheight. It requires four arguments and accepts an optional fifth argument to specify a rotation angle. When the rotation angle is specified (in degrees), the box is rotated by an angle that runs counter-clockwise from the positive y-axis.

The **box** shape specifies a rotated box, centered at xcenter, ycenter, of size xwidth, yheight. The box is rotated by an angle specified in degrees that runs counter-clockwise from the positive y-axis. If the angle argument is omitted, it defaults to 0.

The **circle** shape specifies a circle, centered at xcenter, ycenter, of radius r. It requires three arguments.

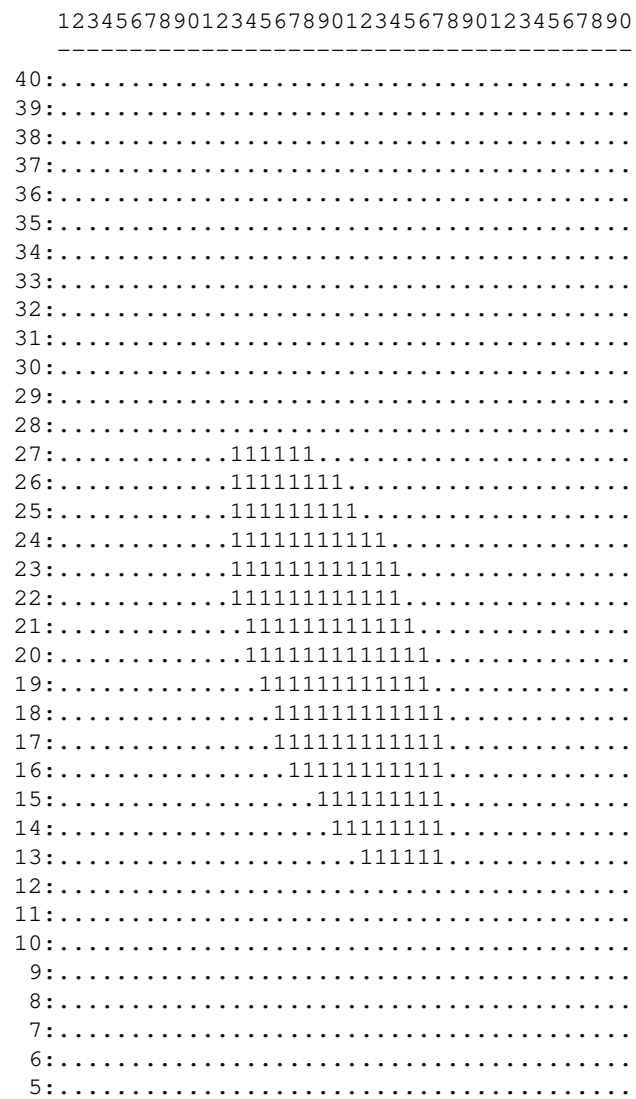
The **ellipse** shape specifies an ellipse, centered at xcenter, ycenter, with y-axis width a and the y-axis length b defined such that:

```
x**2/a**2 + y**2/b**2 = 1
```

Note that a can be less than, equal to, or greater than b. The ellipse is rotated the specified number of degrees. The rotation is done according to astronomical convention, counter-clockwise from the positive y-axis. An ellipse defined by:

```
ELLIPSE 20 20 5 10 45
```

will look like this:



```
4:....................................
3:....................................
2:....................................
1:....................................
```

The **field** shape specifies the entire field as a region. It is not usually specified explicitly, but is used implicitly in the case where no regions are specified, that is, in cases where either a null string or some abbreviation of the string "none" is input. **Field** takes no arguments.

The **pie** shape specifies an angular wedge of the entire field, centered at xcenter, ycenter. The wedge runs between the two specified angles. The angles are given in degrees, running counter-clockwise from the positive x-axis. For example,

```
PIE 20 20 90 180
```

defines a region from 90 degrees to 180 degrees, i.e., quadrant 2 of the Cartesian plane. The display of such a region looks like this:

```
12345678901234567890123456789012345678901
-----------------------------------------
40:1111111111111111111.....................
39:1111111111111111111.....................
38:1111111111111111111.....................
37:1111111111111111111.....................
36:1111111111111111111.....................
35:1111111111111111111.....................
34:1111111111111111111.....................
33:1111111111111111111.....................
32:1111111111111111111.....................
31:1111111111111111111.....................
30:1111111111111111111.....................
29:1111111111111111111.....................
28:1111111111111111111.....................
27:1111111111111111111.....................
26:1111111111111111111.....................
25:1111111111111111111.....................
24:1111111111111111111.....................
23:1111111111111111111.....................
22:1111111111111111111.....................
21:1111111111111111111.....................
20:.....................................
19:.....................................
18:.....................................
17:.....................................
16:.....................................
15:.....................................
14:.....................................
13:.....................................
12:.....................................
11:.....................................
10:.....................................
9:......................................
8:......................................
```

```
7:....................................
6:....................................
5:....................................
4:....................................
3:....................................
2:....................................
1:....................................
```

The pie slice specified is always a counter-clockwise sweep between the angles, starting at the first angle and ending at the second. Thus:

```
PIE 10 15 30 60
```

describes a 30 degree sweep from 2 o'clock to 1 o'clock, while:

```
PIE 10 15 60 30
```

describes a 330 degree counter-clockwise sweep from 1 o'clock to 2 o'clock passing through 12 o'clock (0 degrees). Note in both of these examples that the center of the slice can be anywhere on the plane. The second mask looks like this:

```
12345678901234567890123456789012345678901
-----------------------------------------
40:11111111111111111111111.................
39:11111111111111111111111.................
38:11111111111111111111111.................
37:11111111111111111111111.................
36:11111111111111111111111.................
35:11111111111111111111111.................
34:11111111111111111111111.................
33:11111111111111111111111.................
32:11111111111111111111111.................1
31:11111111111111111111111.................111
30:11111111111111111111111.................11111
29:11111111111111111111111.................111111
28:11111111111111111111111.................11111111
27:11111111111111111111111.................1111111111
26:11111111111111111111111.................11111111111
25:11111111111111111111111.................1111111111111
24:11111111111111111111111.................11111111111111
23:11111111111111111111111.................1111111111111111
22:11111111111111111111111.................1111111111111111
21:11111111111111111111111.................1111111111111111
20:11111111111111111111111.................1111111111111111
19:11111111111111111111111.................1111111111111111
18:11111111111111111111111.................1111111111111111
17:11111111111111111111111.................1111111111111111
16:11111111111111111111111.................1111111111111111
15:11111111111111111111111.................1111111111111111
14:11111111111111111111111.................1111111111111111
13:11111111111111111111111.................1111111111111111
12:11111111111111111111111.................1111111111111111
11:11111111111111111111111.................1111111111111111
10:11111111111111111111111.................1111111111111111
9:11111111111111111111111.................1111111111111111
8:11111111111111111111111.................1111111111111111
```

```
7:11111111111111111111111111111111111111
6:11111111111111111111111111111111111111
5:11111111111111111111111111111111111111
4:11111111111111111111111111111111111111
3:11111111111111111111111111111111111111
2:11111111111111111111111111111111111111
1:11111111111111111111111111111111111111
```

The pie slice goes to the edge of the field. To limit its scope, pie usually is is combined with other shapes, such as circles and annuli, using boolean operations. (See below and in "help regalgebra").

Pie Performance Notes:

Pie region processing time is proportional to the size of the image, and not the size of the region. This is because the pie shape is the only infinite length shape, and we essentially must check all y rows for inclusion (unlike other regions, where the y limits can be calculated beforehand). Thus, pie can run very slowly on large images. In particular, it will run MUCH more slowly than the panda shape in image-based region operations (such as funcnts). We recommend use of panda over pie where ever possible.

If you must use pie, always try to put it last in a boolean && expression. The reason for this is that the filter code is optimized to exit as soon as the result is know. Since pie is the slowest region, it is better to avoid executing it if another region can decide the result. Consider, for example, the difference in time required to process a Chandra ACIS file when a pie and circle are combined in two different orders:

```
  time ./funcnts nacis.fits "circle 4096 4096 100 && pie 4096 4096 10 78"
2.87u 0.38s 0:35.08 9.2%

  time ./funcnts nacis.fits "pie 4096 4096 10 78 && circle 4096 4096 100 "
89.73u 0.36s 1:03.50 141.8%
```

Black-magic performance note:

Panda region processing uses a **quick test** pie region instead of the normal pie region when combining its annulus and pie shapes. This **qtpie** shape differs from the normal pie in that it utilizes the y limits from the previous region with which it is combined. In a panda shape, which is a series of annuli combined with pies, the processing time is thus reduced to that of the annuli.

You can use the qtpie shape instead of pie in cases where you are combining pie with another shape using the && operator. This will cause the pie limits to be set using limits from the other shape, and will speed up the processing considerably. For example, the above execution of funcnts can be improved considerably using this technique:

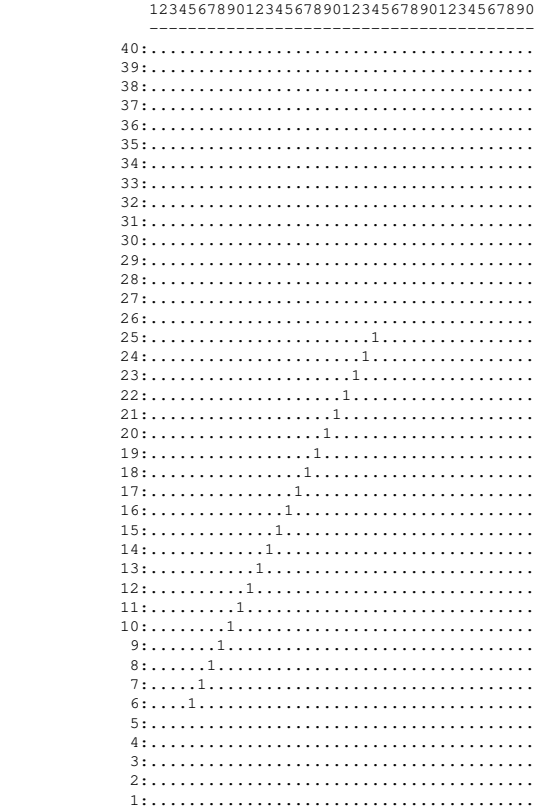
```
  time ./funcnts nacis.fits "circle 4096 4096 100 && qtpie 4096 4096 10 78"
4.66u 0.33s 0:05.87 85.0%
```

We emphasize that this is a quasi-documented feature and might change in the future. The qtpie shape is not recognized by ds9 or other programs.

The **line** shape allows single pixels in a line between (x1,y1) and (x2,y2) to be included or excluded. For example:

```
  LINE (5,6, 24,25)
```

displays as:

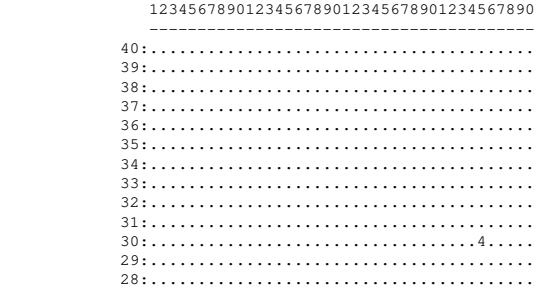


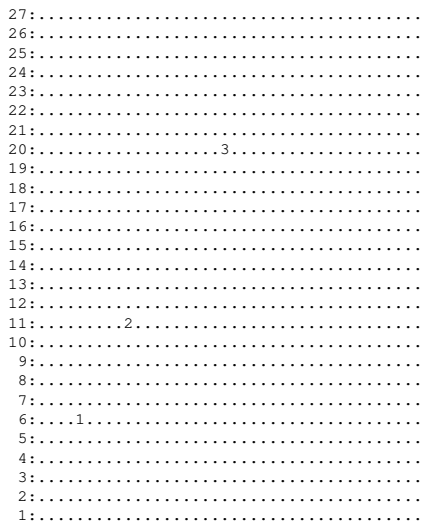
The **point** shape allows single pixels to be included or excluded. Although the (x,y) values are real numbers, they are truncated to integer and the corresponding pixel is included or excluded, as specified.

Several points can be put in one region declaration; unlike the original IRAF implementation, each now is given a different region mask value. This makes it easier, for example, for funcnts to determine the number of photons in the individual pixels. For example,

```
  POINT (5,6,  10,11,  20,20,  35,30)
```

will give the different region mask values to all four points, as shown below:

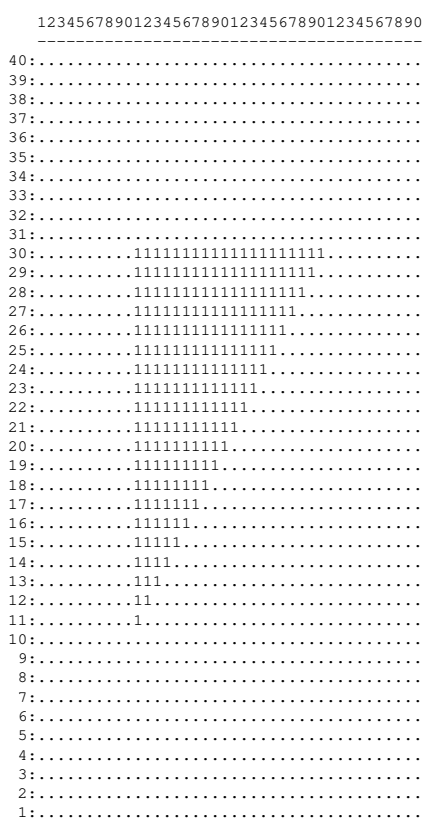




The **polygon** shape specifies a polygon with vertices (x1, y1) ... (xn, yn). The polygon is closed automatically: one should not specify the last vertex to be the same as the first. Any number of vertices are allowed. For example, the following polygon defines a right triangle as shown below:

```
POLYGON (10,10,  10,30,  30,30)
```

looks like this:



Note that polygons can get twisted upon themselves if edge lines cross. Thus:

```
POL (10,10,  20,20,  20,10,  10,20)
```

will produce an area which is two triangles, like butterfly wings, as shown below:

```
        12345678901234567890123456789012345678901234567890
        ----------------------------------------
      40:........................................
      39:........................................
      38:........................................
      37:........................................
      36:........................................
      35:........................................
      34:........................................
      33:........................................
      32:........................................
      31:........................................
      30:........................................
      29:........................................
      28:........................................
      27:........................................
      26:........................................
      25:........................................
      24:........................................
      23:........................................
      22:........................................
      21:........................................
      20:........................................
      19:..........1........1.....................
      18:..........11......11.....................
      17:..........111....111.....................
      16:..........1111..1111.....................
      15:..........1111111111.....................
      14:..........1111..1111.....................
      13:..........111....111.....................
      12:..........11......11.....................
      11:..........1........1.....................
      10:........................................
       9:........................................
       8:........................................
       7:........................................
       6:........................................
       5:........................................
       4:........................................
       3:........................................
       2:........................................
       1:........................................
```

The following are combinations of pie with different shapes (called "panda" for "Pie AND Annulus") allow for easy specification of radial sections:

```
shape:   arguments:
-----    ---------
PANDA    xcen ycen ang1 ang2 nang irad orad nrad   # circular
CPANDA   xcen ycen ang1 ang2 nang irad orad nrad   # circular
BPANDA   xcen ycen ang1 ang2 nang xwlo yhlo xwhi yhhi nrad (ang) # box
EPANDA   xcen ycen ang1 ang2 nang xwlo yhlo xwhi yhhi nrad (ang) # ellipse
```

The **panda** (**P**ies **AND A**nnuli) shape can be used to create combinations of pie and annuli markers. It is analogous to a Cartesian product on those shapes, i.e., the result is several shapes generated by performing a boolean AND between pies and annuli. Thus, the panda and cpanda specify combinations of annulus and circle with pie, respectively and give identical results. The bpanda combines box and pie, while epanda combines ellipse and pie.

Consider the example shown below:

```
PANDA(20,20, 0,360,3, 0,15,4)
```

Here, 3 pie slices centered at 20, 20 are combined with 4 annuli, also centered at 20, 20. The result is a mask with 12 regions (displayed in base 16 to save characters):

```
        12345678901234567890123456789012345678901234567890
        ----------------------------------------
        40:........................................
        39:........................................
        38:........................................
        37:........................................
        36:........................................
        35:........................................
```

```
34:..............44444444444...............
33:............444444444444444.............
32:...........88444444444444444............
31:.........888844443333344444444..........
30:........88888833333333333444444.........
29:........88888733333333333344444.........
28:.......888887773333333333344444........
27:......888887777332222233333344444.......
26:......888877777622222222333334444.......
25:.....8888777776662222222233333444.......
24:.....8888777766662222222233334444......
23:.....8888777766665111122223333444.......
22:.....8887777666655111122223333444......
21:.....8887777666655511112222333344.......
20:.....888777766665559999aaaabbbbccc......
19:.....888777766665559999aaaabbbbccc......
18:.....888777766665599999aaaabbbbccc......
17:.....88887777666659999aaaabbbbcccc......
16:.....888877776666aaaaaaaaabbbbcccc......
15:.....888877777666aaaaaaaabbbbbcccc......
14:......8888777776aaaaaaaabbbbbcccc.......
13:......888887777bbaaaaabbbbbbccccc.......
12:.......88888777bbbbbbbbbbbbccccc........
11:........888887bbbbbbbbbbbbccccc.........
10:........888888bbbbbbbbbbbcccccc.........
9:.........8888ccccbbbbbcccccccc..........
8:...........88cccccccccccccc............
7:............ccccccccccccccc.............
6:..............cccccccccc...............
5:.........................................
4:.........................................
3:.........................................
2:.........................................
1:.........................................
```

Several regions with different mask values can be combined in the same mask. This supports comparing data from the different regions. (For information on how to combine different shapes into a single region, see "help regalgebra".) For example, consider the following set of regions:

```
ANNULUS 25 25 5 10
ELLIPSE 20 20 5 10 315
BOX 15 15 5 10
```

The resulting mask will look as follows:

```
        1234567890123456789012345678901234567890
        ----------------------------------------
     40:.........................................
     39:.........................................
     38:.........................................
     37:.........................................
     36:.........................................
     35:.........................................
     34:...................111111111...........
     33:..................11111111111..........
     32:................111111111111111........
     31:................111111111111111........
     30:...............11111111111111111.......
     29:...............1111111.....1111111......
     28:...............111111.......111111......
     27:...............11111.222222..11111......
     26:...............111112222222..11111......
     25:...............111112222222..11111......
     24:...............111112222222..11111......
     23:...............111112222222..11111......
     22:...............111111222222.111111......
     21:..............211111112222.1111111......
     20:............322211111111111111111.......
     19:............322221111111111111111........
     18:............222221111111111111111........
     17:............222222211111111111..........
     16:............22222222111111111...........
     15:............222222222...................
     14:............22222222....................
     13:............222222......................
     12:............33333.......................
     11:............33333.......................
     10:.........................................
      9:.........................................
      8:.........................................
      7:.........................................
```

```
6:....................................
5:....................................
4:....................................
3:....................................
2:....................................
1:....................................
```

Note that when a pixel is in 2 or more regions, it is arbitrarily assigned to a one of the regions in question (often based on how a give C compiler optimizes boolean expressions).

Region accelerators

Two types of \fBaccelerators, to simplify region specification, are provided as natural extensions to the ways shapes are described. These are: extended lists of parameters, specifying multiple regions, valid for annulus, box, circle, ellipse, pie, and points; and **n=**, valid for annulus, box, circle, ellipse, and pie (not point). In both cases, one specification is used to define several different regions, that is, to define shapes with different mask values in the region mask.

The following regions accept **accelerator** syntax:

```
shape      arguments
-----      ----------------------------------------
ANNULUS    xcenter ycenter radius1 radius2 ... radiusn
ANNULUS    xcenter ycenter inner_radius outer_radius n=[number]
BOX        xcenter ycenter xw1 yh1 xw2 yh2 ... xwn yhn (angle)
BOX        xcenter ycenter xwlo yhlo xwhi yhhi n=[number] (angle)
CIRCLE     xcenter ycenter r1 r2 ... rn              # same as annulus
CIRCLE     xcenter ycenter rinner router n=[number]  # same as annulus
ELLIPSE    xcenter ycenter xw1 yh1 xw2 yh2 ... xwn yhn (angle)
ELLIPSE    xcenter ycenter xwlo yhlo xwhi yhhi n=[number] (angle)
PIE        xcenter ycenter angle1 angle2 (angle3) (angle4) (angle5) ...
PIE        xcenter ycenter angle1 angle2 (n=[number])
POINT      x1 y1 x2 y2 ... xn yn
```

Note that the circle accelerators are simply aliases for the annulus accelerators.

For example, several annuli at the same center can be specified in one region expression by specifying more than two radii. If **N** radii are specified, then **N**-1 annuli result, with the outer radius of each preceding annulus being the inner radius of the succeeding annulus. Each annulus is considered a separate region, and is given a separate mask value. For example,

```
ANNULUS 20 20 0 2 5 10 15 20
```

specifies five different annuli centered at 20 20, and is equivalent to:

```
ANNULUS 20.0 20.0  0  2
ANNULUS 20.0 20.0  2  5
ANNULUS 20.0 20.0  5 10
ANNULUS 20.0 20.0 10 15
ANNULUS 20.0 20.0 15 20
```

The mask is shown below:

```
        12345678901234567890123456789012345678901
        -----------------------------------------
     40:.........................................
     39:.............5555555555555...............
     38:...........555555555555555.5.............
     37:.........5555555555555555555555..........
     36:........55555555555555555555555.........
     35:......555555555555555555555555555.......
     34:.....55555555544444444444555555555......
```

```
33:....55555555444444444444455555555.....
32:....55555554444444444444445555555.....
31:...555555444444444444444444555555....
30:..5555554444444444444444444444555555...
29:..5555554444444333333333444444455555...
28:.55555544444443333333333344444444555555..
27:.5555544444443333333333333344444455555..
26:555555444444333333333333333444444555555.
25:555554444443333333333333333444444455555.
24:555554444433333332222233333334444455555.
23:555554444433333322222223333334444455555.
22:555554444433333222222222333334444455555.
21:555554444433333222111222333334444455555.
20:555554444433333222111222333334444455555.
19:555554444433333222111222333334444455555.
18:555554444433333222222222333334444455555.
17:555554444433333322222223333334444455555.
16:555554444433333332222233333334444455555.
15:555554444443333333333333333344444455555.
14:555555444444333333333333333444444555555.
13:.55555444444333333333333333444444455555..
12:.55555544444443333333333334444444555555..
11:..5555554444444333333333444444455555...
10:..555555444444444444444444444555555...
 9:...55555544444444444444444444555555....
 8:....5555555444444444444444455555555.....
 7:....5555555544444444444444455555555.....
 6:.....555555555444444444445555555555.....
 5:......55555555555555555555555555.......
 4:........555555555555555555555.........
 3:.........5555555555555555555..........
 2:...........55555555555555555...........
 1:.............555555555555..............
```

For boxes and ellipses, if an odd number of arguments is specified, then the last argument is assumed to be an angle. Otherwise, the angle is assumed to be zero. For example:

```
ellipse 20 20 3 5 6 10 9 15 12 20 45
```

specifies an 3 ellipses at a 45 degree angle:

```
        12345678901234567890123456789012345678901234567890
        ----------------------------------------
        40:........................................
        39:........................................
        38:........................................
        37:........................................
        36:........33333333........................
        35:......333333333333......................
        34:.....33333333333333333..................
        33:....333333333333333333..................
        32:....3333333222233333333.................
        31:...33333322222222233333.................
        30:...333332222222222233333................
        29:...3333322222222222223333...............
        28:...333322222221111222223333.............
        27:...3333222221111111222223333............
        26:...33332222211111111122223333...........
        25:...333322221111111111122223333..........
        24:....3332222111111..1111122223333........
        23:....333322211111.....11112222333........
        22:....33332222111.......11112223333.......
        21:.....33322221111.......11122223333......
        20:.....33332221111.......11112223333......
        19:.....33332222111.......11112222333......
        18:......33332221111.......11122223333.....
        17:.......33322221111.....111112223333.....
```

```
16:.......3333222211111..1111112222333.....
15:........333322221111111111122223333....
14:.........33332222111111111222223333....
13:..........3333222221111111222223333....
12:...........333322222211112222222333....
11:............33332222222222222233333....
10:.............3333322222222222233333....
9:..............333333222222222333333....
8:...............33333333222223333333.....
7:.................333333333333333333.....
6:..................3333333333333333......
5:.....................333333333333.......
4:.......................33333333.........
3:.......................................
2:.......................................
1:.......................................
```

Note in the above example that the lower limit is not part of the region for boxes, circles, and ellipses. This makes circles and annuli equivalent, i.e.:

```
circle  20 20 5 10 15 20
annulus 20 20 5 10 15 20
```

both give the following region mask:

```
12345678901234567890123456789012345678 90
----------------------------------------
40:.......................................
39:.............3333333333333..............
38:...........33333333333333333............
37:.........333333333333333333333..........
36:........3333333333333333333333.........
35:......333333333333333333333333333.......
34:.....33333333322222222222333333333......
33:....3333333322222222222222233333333.....
32:....3333333222222222222222223333333.....
31:...333333222222222222222222222333333....
30:..33333322222222222222222222222333333...
29:..33333322222221111111112222222333333...
28:.3333332222222111111111112222222333333..
27:.333332222221111111111111112222222333333..
26:333333222222111111111111111122222233333.
25:333332222221111111111111111122222233333.
24:33333222221111111.....11111112222233333.
23:3333322222111111.......1111112222233333.
22:333332222211111.........111112222233333.
21:333332222211111.........111112222233333.
20:333332222211111.........111112222233333.
19:333332222211111.........111112222233333.
18:333332222211111.........111112222233333.
17:3333322222111111.......1111112222233333.
16:33333222221111111.....11111112222233333.
15:333332222221111111111111111122222233333.
14:333333222222111111111111111122222233333.
13:.333332222221111111111111112222222333333..
12:.333333222222211111111112222222333333..
11:..33333322222221111111112222222333333...
10:..333333222222222222222222222222333333...
9:...333333222222222222222222222333333....
8:....333333322222222222222222333333.....
7:....333333332222222222222233333333.....
6:.....33333333322222222222333333333......
5:......333333333333333333333333333.......
4:........33333333333333333333333.........
3:.........333333333333333333333..........
2:...........33333333333333333............
1:.............3333333333333..............
```

As a final example, specifying several angles in one pie slice expression is equivalent to specifying several separate slices with the same center. As with the annulus, if **N** angles are specified, then **N**-1 slices result, with the ending angle of each preceding slice being the starting angle of the succeeding slice. Each slice is considered a separate region, and is given a separate mask value. For example,

```
PIE 12 12 315 45 115 270
```

specifies three regions as shown below:

```
       12345678901234567890123456789012345678901234567890
       ----------------------------------------
       40:22222222222222222222222222222222222222222222222
       39:22222222222222222222222222222222222222222222221
       38:22222222222222222222222222222222222222222222211
       37:22222222222222222222222222222222222222222222111
       36:22222222222222222222222222222222222222222221111
       35:22222222222222222222222222222222222222222211111
       34:22222222222222222222222222222222222222222111111
       33:22222222222222222222222222222222222222221111111
       32:22222222222222222222222222222222222222211111111
       31:22222222222222222222222222222222222222111111111
       30:22222222222222222222222222222222222221111111111
       29:22222222222222222222222222222222222211111111111
       28:22222222222222222222222222222222222111111111111
       27:22222222222222222222222222222222222211111111111
       26:22222222222222222222222222222222222111111111111
       25:22222222222222222222222222222222222111111111111
       24:22222222222222222222222222222222221111111111111
       23:22222222222222222222222222222222221111111111111
       22:22222222222222222222222222222222221111111111111
       21:22222222222222222222222222222222221111111111111
       20:22222222222222222222222222222222221111111111111
       19:22222222222222222222222222222222221111111111111
       18:22222222222222222222222222222222221111111111111
       17:22222222222222222222222222222222221111111111111
       16:22222222222222222222222222222222221111111111111
       15:22222222222222222222222222222222221111111111111
       14:22222222222222222222222222222222221111111111111
       13:22222222222222222222222222222222221111111111111
       12:22222222222.22222222222222222222222222222222222
       11:22222222222222222222222222222222222222222222222
       10:22222222222.22222222222222222222222222222222222
       9:222222222222..2222222222222222222222222222222222
       8:222222222222...22222222222222222222222222222222
       7:222222222222....22222222222222222222222222222222
       6:222222222222.....22222222222222222222222222222222
       5:222222222222......22222222222222222222222222222222
       4:222222222222.......22222222222222222222222222222222
       3:222222222222........22222222222222222222222222222222
       2:222222222222.........22222222222222222222222222222222
       1:222222222222..........22222222222222222222222222222222
```

The annulus, box, circle, ellipse, and pie shapes also accept an **n=[int]** syntax for specifying multiple regions. The **n=[int]**syntax interprets the previous (shape-dependent) arguments as lower and upper limits for the region and creates n shapes with evenly spaced boundaries. For example, if **n=[int]** is specified in an annulus, the two immediately preceding radii (**rn** and **rm**) are divided into **int** annuli, such that the inner radius of the first is **rn** and the outer radius of the last is **rm**. For example,

```
ANNULUS 20 20 5 20 n=3
```

is equivalent to:

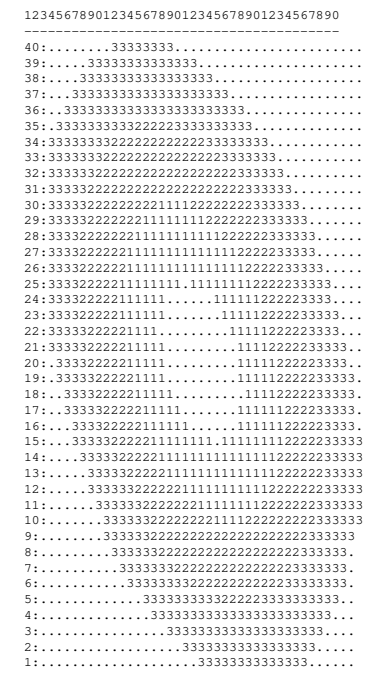
```
ANNULUS 20 20 5 10 15 20
```

If this syntax is used with an ellipse or box, then the two preceding pairs of values are taken to be lower and upper limits for a set of ellipses or boxes. A circle uses the two preceding arguments for upper and lower radii. For pie, the two preceding angles are divided into n wedges such that the starting angle of the first is the lower bound and the ending angle of the last is the upper bound. In all cases, the **n=[int]** syntax allows any single alphabetic character before the "=", i.e, i=3, z=3, etc. are all equivalent.

Also note that for boxes and ellipses, the optional angle argument is always specified after the **n=[int]** syntax. For example:

```
ellipse 20 20 4 6 16 24 n=3 45
```

specifies 3 elliptical regions at an angle of 45 degrees:



```
1234567890123456789012345678901234567890
----------------------------------------
40:........33333333.......................
39:.....33333333333333....................
38:....33333333333333333..................
37:...333333333333333333.................
36:..3333333333333333333333...............
35:.33333333332222233333333333.............
34:333333332222222222233333333............
33:3333333222222222222223333333...........
32:33333322222222222222222333333..........
31:333332222222222222222222333333.........
30:3333322222222211112222222233333........
29:333332222222111111112222222333333.......
28:33332222222111111111112222223333333......
27:333322222211111111111111222223333333.....
26:3333222222111111111111111222223333333....
25:333322222111111111.1111111122222333333...
24:333322222111111.....111111222223333....
23:333322222111111.......111112222233333...
22:33333222221111.........11111222223333...
21:333332222211111.........11112222233333..
20:.33332222211111.........11111222223333..
19:.33333222221111.........111112222233333.
18:..33332222211111.........11112222233333.
17:..333332222211111.......111111222233333.
16:...333322222111111......111111222223333.
15:...333332222211111111.1111111122222333333
14:....3333322222111111111111111122222233333
13:.....33333222221111111111111122222233333
12:.....333333222222111111111122222222333333
11:......3333332222222111111112222222333333
10:.......33333322222222111122222222333333
9:........333333222222222222222222233333
8:.........33333322222222222222222333333.
7:..........3333333222222222222222333333.
6:...........333333332222222222222333333.
5:.............33333333332222233333333..
4:..............333333333333333333333...
3:................3333333333333333333....
2:..................33333333333333333.....
1:...................33333333333333......
```

Both the variable argument syntax and the **n=[int]** syntax must occur alone in a region descriptor (aside from the optional angle for boxes and ellipses). They cannot be combined. Thus, it is not valid to precede or follow an **n=[int]** accelerator with more angles or radii, as in this example:

```
# INVALID -- one too many angles before a=5 ...
# and no angles are allowed after a=5
PIE 12 12 10 25 50 a=5 85 135
```

Instead, use three separate specifications, such as:

```
PIE 12 12 10 25
PIE 12 12 25 50 a=5
PIE 12 12 85 135
```

The original (IRAF) implementation of region filtering permitted this looser syntax, but we found it caused more confusion than it was worth and therefore removed it.

NB: Accelerators may be combined with other shapes in a boolean expression in any order. (This is a change starting with funtools v1.1.1. Prior to this release, the accelerator shape had to be specified last). The actual region mask id values returned depend on the order in which the shapes are specified, although the total number of pixels or rows that pass the filter will be consistent. For this reason, use of accelerators in boolean expressions is discouraged in programs such as funcnts, where region mask id values are used to count events or image pixels.

[All region masks displayed in this document were generated using the **fundisp** routine and the undocumented "mask=all" argument (with spaced removed using sed):

```
fundisp "funtools/funtest/test40.fits[ANNULUS 25 25 5 10]" mask=all |\
sed 's/ //g'
```

Note that you must supply an image of the appropriate size -- in this case, a FITS image of dimension 40x40 is used.]

Go to Funtools Help Index

Last updated: March 12, 2007

RegAlgebra: Boolean Algebra on Spatial Regions

Summary

This document describes the boolean arithmetic defined for region expressions.

Description

When defining a region, several shapes can be combined using boolean operations. The boolean operators are (in order of precedence):

```
Symbol        Operator               Associativity
------        --------               -------------
!             not                    right to left
&             and                    left to right
^             exclusive or           left to right
|             inclusive or           left to right
```

For example, to create a mask consisting of a large circle with a smaller box removed, one can use the **and** and **not** operators:

```
CIRCLE(11,11,15) & !BOX(11,11,3,6)
```

and the resulting mask is:

```
        12345678901234567890123456789012345678901
        -----------------------------------------
      1:11111111111111111111.....................
      2:11111111111111111111.....................
      3:11111111111111111111.....................
      4:11111111111111111111.....................
      5:11111111111111111111.....................
      6:11111111111111111111.....................
      7:11111111111111111111.....................
      8:11111111111111111111.....................
      9:111111111...1111111111111................
     10:111111111...1111111111111................
     11:111111111...1111111111111................
     12:111111111...1111111111111................
     13:111111111...1111111111111................
     14:111111111...1111111111111................
     15:11111111111111111111111111...............
     16:11111111111111111111111111...............
     17:11111111111111111111111111...............
     18:11111111111111111111111111...............
     19:11111111111111111111111111...............
     20:11111111111111111111.....................
     21:11111111111111111111.....................
     22:11111111111111111111.....................
     23:..1111111111111111.......................
     24:...11111111111111........................
     25:.....11111111111.........................
     26:.........................................
```

```
27:....................................
28:....................................
29:....................................
30:....................................
31:....................................
32:....................................
33:....................................
34:....................................
35:....................................
36:....................................
37:....................................
38:....................................
39:....................................
40:....................................
```

A three-quarter circle can be defined as:

```
CIRCLE(20,20,10) & !PIE(20,20,270,360)
```

and looks as follows:

```
   12345678901234567890123456789012345678890
   ----------------------------------------
 1:........................................
 2:........................................
 3:........................................
 4:........................................
 5:........................................
 6:........................................
 7:........................................
 8:........................................
 9:........................................
10:........................................
11:..............111111111.................
12:............11111111111.................
13:...........111111111111111..............
14:...........111111111111111..............
15:..........1111111111111111..............
16:.........11111111111111111..............
17:.........11111111111111111..............
18:.........11111111111111111..............
19:.........11111111111111111..............
20:.........11111111111111111..............
21:.........1111111111.....................
22:.........1111111111.....................
23:.........1111111111.....................
24:.........1111111111.....................
25:...........111111111....................
26:...........11111111.....................
27:...........11111111.....................
28:.............111111.....................
29:.............11111.....................
30:........................................
31:........................................
32:........................................
33:........................................
34:........................................
```

```
35:....................................
36:....................................
37:....................................
38:....................................
39:....................................
40:....................................
```

Two non-intersecting ellipses can be made into the same region:

```
ELL(20,20,10,20,90) | ELL(1,1,20,10,0)
```

and looks as follows:

```
   12345678901234567890123456789012345678901234567890
   ----------------------------------------
 1:11111111111111111111..................
 2:11111111111111111111..................
 3:11111111111111111111..................
 4:11111111111111111111..................
 5:1111111111111111111...................
 6:1111111111111111111...................
 7:111111111111111.......................
 8:111111111111111.......................
 9:11111111111...........................
10:111111111.............................
11:...........1111111111111111...........
12:........11111111111111111111111........
13:.....1111111111111111111111111111.....
14:....111111111111111111111111111111....
15:..11111111111111111111111111111111...
16:.111111111111111111111111111111111..
17:1111111111111111111111111111111111111.
18:1111111111111111111111111111111111111.
19:1111111111111111111111111111111111111.
20:1111111111111111111111111111111111111.
21:1111111111111111111111111111111111111.
22:1111111111111111111111111111111111111.
23:1111111111111111111111111111111111111.
24:.111111111111111111111111111111111111..
25:..1111111111111111111111111111111111...
26:...11111111111111111111111111111111.....
27:.....1111111111111111111111111111......
28:.......11111111111111111111111.........
29:...........1111111111111111............
30:....................................
31:....................................
32:....................................
33:....................................
34:....................................
35:....................................
36:....................................
37:....................................
38:....................................
39:....................................
40:....................................
```

You can use several boolean operations in a single region expression, to create arbitrarily complex

regions. With the important exception below, you can apply the operators in any order, using parentheses if necessary to override the natural precedences of the operators.

NB: Using a panda shape is always much more efficient than explicitly specifying "pie & annulus", due to the ability of panda to place a limit on the number of pixels checked in the pie shape. If you are going to specify the intersection of pie and annulus, use panda instead.

As described in "help regreometry", the **PIE** slice goes to the edge of the field. To limit its scope, **PIE** usually is is combined with other shapes, such as circles and annuli, using boolean operations. In this context, it is worth noting that that there is a difference between **-PIE** and **&!PIE**. The former is a global exclude of all pixels in the **PIE** slice, while the latter is a local excludes of pixels affecting only the region(s) with which the **PIE** is combined. For example, the following region uses **&!PIE** as a local exclude of a single circle. Two other circles are also defined and are unaffected by the local exclude:

```
CIRCLE(1,8,1)
CIRCLE(8,8,7)&!PIE(8,8,60,120)&!PIE(8,8,240,300)
CIRCLE(15,8,2)
```

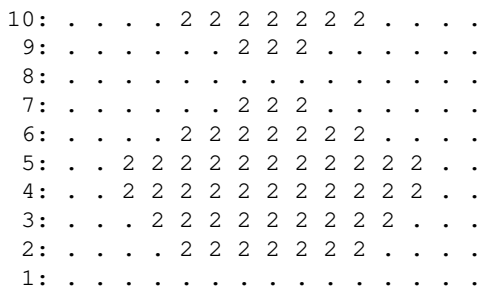
```
   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   - - - - - - - - - - - - - - -
15: . . . . . . . . . . . . . . .
14: . . . . 2 2 2 2 2 2 2 . . . .
13: . . . 2 2 2 2 2 2 2 2 2 . . .
12: . . 2 2 2 2 2 2 2 2 2 2 2 . .
11: . . 2 2 2 2 2 2 2 2 2 2 2 . .
10: . . . . 2 2 2 2 2 2 2 . . . .
 9: . . . . . . . 2 2 2 . . . . 3 3
 8: 1 . . . . . . . . . . . . . 3 3
 7: . . . . . . . 2 2 2 . . . . 3 3
 6: . . . . 2 2 2 2 2 2 2 . . . .
 5: . . 2 2 2 2 2 2 2 2 2 2 2 . .
 4: . . 2 2 2 2 2 2 2 2 2 2 2 . .
 3: . . . 2 2 2 2 2 2 2 2 2 . . .
 2: . . . . 2 2 2 2 2 2 2 . . . .
 1: . . . . . . . . . . . . . . .
```

Note that the two other regions are not affected by the **&!PIE**, which only affects the circle with which it is combined.

On the other hand, a **-PIE** is an global exclude that does affect other regions with which it overlaps:

```
CIRCLE(1,8,1)
CIRCLE(8,8,7)
-PIE(8,8,60,120)
-PIE(8,8,240,300)
CIRCLE(15,8,2)
```

```
   1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
   - - - - - - - - - - - - - - -
15: . . . . . . . . . . . . . . .
14: . . . . 2 2 2 2 2 2 2 . . . .
13: . . . 2 2 2 2 2 2 2 2 2 . . .
12: . . 2 2 2 2 2 2 2 2 2 2 2 . .
11: . . 2 2 2 2 2 2 2 2 2 2 2 . .
```



The two smaller circles are entirely contained within the two exclude **PIE** slices and therefore are excluded from the region.

Go to Funtools Help Index

Last updated: November 17, 2005

RegCoords: Spatial Region Coordinates

Summary

This document describes the specification of coordinate systems, and the interpretation of coordinate values, for spatial region filtering.

Pixel coordinate systems

The default coordinate system for regions is PHYSICAL, which means that region position and size values are taken from the original data. (Note that this is a change from the original IRAF/PROS implementation, in which the IMAGE coordinate system was the default.) PHYSICAL coordinates always refer to pixel positions on the original image (using IRAF LTM and LTV keywords). With PHYSICAL coordinates, if a set of coordinates specifies the position of an object in an original FITS file, the same coordinates will specify the same object in any FITS derived from the original. Physical coordinates are invariant with blocking of FITS files or taking sections of images, even when a blocked section is written to a new file.

Thus, although a value in pixels refers, by default, to the PHYSICAL coordinate system, you may specify that position values refer to the image coordinate system using the **global** or **local** properties commands:

```
global coordsys image
circle 512 512 100
```

The **global** command changes the coordinate system for all regions that follow, while the **local** command changes the coordinate system only for the region immediately following:

```
local coordsys image
circle 512 512 100
circle 1024 1024 200
```

This changes the coordinate system only for the region that follows. In the above example, the second region uses the global coordinate system (PHYSICAL by default).

World Coordinate Systems

If World Coordinate System information is contained in the data file being filtered, it also is possible to define regions using a sky coordinate system. Supported systems include:

```
name                 description
----                 -----------
PHYSICAL             pixel coords of original file using LTM/LTV
IMAGE                pixel coords of current file
FK4, B1950           sky coordinate systems
FK5, J2000           sky coordinate systems
GALACTIC             sky coordinate systems
ECLIPTIC             sky coordinate systems
ICRS                 currently same as J2000
LINEAR               linear wcs as defined in file
```

In addition, two mosaic coordinate systems have been defined that utilize the (evolving) IRAF mosaic keywords:

```
name                  description
----                  -----------
AMPLIFIER             mosaic coords of original file using ATM/ATV
DETECTOR              mosaic coords of original file using DTM/DTV
```

Again, to use one of these coordinate systems, the **global** or **local** properties commands are used:

```
global coordsys galactic
```

WCS Positions and Sizes

In addition to pixels, positional values in a WCS-enabled region can be specified using sexagesimal or degrees format:

```
position arguments    description
------------------    -----------
[num]                 context-dependent (see below)
[num]d                degrees
[num]r                radians
[num]p                physical pixels
[num]i                image pixels
[num]:[num]:[num]     hms for 'odd' position arguments
[num]:[num]:[num]     dms for 'even' position arguments
[num]h[num]m[num]s    explicit hms
[num]d[num]m[num]s    explicit dms
```

If ':' is used as sexagesimal separator, the value is considered to be specifying hours/minutes/seconds if it is the first argument of a positional pair, and degrees/minutes/seconds for the second argument of a pair (except for galactic coordinates, which always use degrees):

```
argument      description
-----------   -----------
10:20:30.0    10 hours, 20 minutes, 30 seconds for 1st positional argument
              10 degrees, 20 minutes, 30 seconds for 2nd positional argument
10h20m30.0    10 hours, 20 minutes, 30 seconds
10d20m30.0    10 degrees, 20 minutes, 30 seconds
10.20d        10.2 degrees
```

Similarly, the units of size values are defined by the formating character(s) attached to a number:

```
size arguments        description
--------------        -----------
[num]                 context-dependent (see below)
[num]"                arc seconds
[num]'                arc minutes
[num]d                degrees
[num]r                radians
[num]p                physical pixels
[num]i                image pixels
```

For example:

```
argument      description
-----------   -----------
10            ten pixels
10’           ten minutes of arc
10"           ten seconds of arc
10d           ten degrees
10p           ten pixels
0.5r          half of a radian
```

An example of using sky coordinate systems follows:

```
global coordsys B1950
-box 175.54d 20.01156d 10’ 10’
local coordsys J2000
pie 179.57d 22.4d 0 360 n=4 && annulus 179.57d 22.4d 3’ 24’ n=5
```

At the FK4 1950 coordinates 175.54d RA, 20.01156d DEC exclude a 10 minute by 10 minute box. Then at the FK5 2000 coordinates 179.57d RA 22.4d DEC draw a radial profile regions pattern with 4 quadrants and 5 annuli ranging from 3 minutes to 24 minutes in diameter. In this example, the default coordinate system is overridden by the commands in the regions spec.

NB: The Meaning of Pure Numbers Are Context Sensitive

When a "pure number" (i.e. one without a format directive such as ’d’ for ’degrees’) is specified as a position or size, its interpretation depends on the context defined by the ’coordsys’ keyword. In general, the rule is:

All pure numbers have implied units corresponding to the current coordinate system.

If no coordinate system is explicitly specified, the default system is implicitly assumed to be PHYSICAL. In practice this means that for IMAGE and PHYSICAL systems, pure numbers are pixels. Otherwise, for all systems other than LINEAR, pure numbers are degrees. For LINEAR systems, pure numbers are in the units of the linear system. This rule covers both positions and sizes.

As a corollary, when a sky-formatted number is used with the IMAGE or PHYSICAL coordinate system (which includes the default case of no coordsys being specified), the formatted number is assumed to be in the units of the WCS contained in the current file. If no sky WCS is specified, an error results.

Examples:

```
circle(512,512,10)
ellipse 202.44382d 47.181656d 0.01d 0.02d
```

In the absence of a specified coordinate system, the circle uses the default PHYSICAL units of pixels, while the ellipse explicitly uses degrees, presumably to go with the WCS in the current file.

```
global coordsys=fk5
global color=green font="system 10 normal"
circle 202.44382 47.181656 0.01
circle 202.44382 47.181656 10p
ellipse(512p,512p,10p,15p,20)
```

Here, the circles use the FK5 units of degrees (except for the explicit use of pixels in the second radius), while the ellipse explicitly specifies pixels. The ellipse angle is in degrees.

Note that Chandra data format appears to use "coordsys=physical" implicitly. Therefore, for most Chandra applications, valid regions can be generated safely by asking ds9 to save/display regions in pixels using the PHYSICAL coordsys.

Go to Funtools Help Index

Last updated: November 17, 2005

RegBounds: Region Boundaries

Summary

Describes how spatial region boundaries are handled.

Description

The golden rule for spatial region filtering was first enunciated by Leon VanSpeybroeck in 1986:

Each photon will be counted once, and no photon will be counted more than once.

This means that we must be careful about boundary conditions. For example, if a circle is contained in an annulus such that the inner radius of the annulus is the same as the radius of the circle, then photons on that boundary must always be assigned to one or the other region. That is, the number of photons in both regions must equal the sum of the number of photons in each region taken separately. With this in mind, the rules for determining whether a boundary image pixel or table row are assigned to a region are defined below.

Image boundaries : radially-symmetric shapes (circle, annuli, ellipse)

For image filtering, pixels whose center is inside the boundary are included. This also applies non-radially-symmetric shapes. When a pixel center is exactly on the boundary, the pixel assignment rule is:

- the outer boundary of a symmetric shape does not include such pixels
- the inner boundary of a symmetric shape (annulus) includes such pixels

In this way, an annulus with radius from 0 to 1, centered exactly on a pixel, includes the pixel on which it is centered, but none of its neighbors. These rules ensure that when defining concentric shapes, no pixels are omitted between concentric regions and no pixels are claimed by two regions. When applied to small symmetric shapes, the shape is less likely to be skewed, as would happen with non-radially-symmetric rules. These rules differ from the rules for box-like shapes, which are more likely to be positioned adjacent to one another.

Image Boundaries: non-radially symmetric shapes (polygons, boxes)

For image filtering, pixels whose center is inside the boundary are included. This also applies radially-symmetric shapes. When a pixel center is exactly on the boundary of a non-radially symmetric region, the pixel is included in the right or upper region, but not the left or lower region. This ensures that geometrically adjoining regions touch but don’t overlap.

Row Boundaries are Analytic

When filtering table rows, the boundary rules are the same as for images, except that the calculation is not done on the center of a pixel, (since table rows, especially X-ray events rows, often have discrete, floating point positions) but are calculated exactly. That is, an row is inside the boundary without regard to its integerized pixel value. For rows that are exactly on a region boundary, the above rules are applied to ensure that all rows are counted once and no row is counted more than once.

Because row boundaries are calculated differently from image boundaries, certain programs will give different results when filtering the same region file. In particular, fundisp/funtable (which utilize analytic row filtering) perform differently from funcnts (which performs image filtering, even on tables).

Image Boundaries vs. Row Boundaries: Practical Considerations

You will sometimes notice a discrepancy between running funcnts on an binary table file and running fundisp on the same file with the same filter. For example, consider the following:

```
fundisp test1.fits"[box(4219,3887,6,6,0)]" | wc
8893  320148 3752846
```

Since fundisp has a 2-line header, there are actually 8891 photons that pass the filter. But then run funtable and select only the rows that pass this filter, placing them in a new file:

```
./funtable test1.fits"[box(4219,3887,6,6,0)]" test2.fits
```

Now run funcnts using the original filter on the derived file:

```
./funcnts test2.fits "physical; box(4219,3887,6,6,0)"

[... lot of processed output ...]

# the following source and background components were used:
source region(s)
----------------
physical; box(4219,3887,6,6,0)

 reg       counts    pixels
---- ------------ ---------
   1     7847.000        36
```

There are 1044 rows (events) that pass the row filter in fundisp (or funtable) but fail to make it through funcnts. Why?

The reason can be traced to how analytic row filtering (fundisp, funtable) differs from integerized pixel filtering(funcnts, funimage). Consider the region:

```
box(4219,3887,6,6,0)
```

Analytically (i.e., using row filtering), positions will pass this filter successfully if:

```
4216 <= x <= 4222
3884 <= y <= 3890
```

For example, photons with position values of x=4216.4 or y=3884.08 will pass.

Integerized image filtering is different in that the pixels that will pass this filter have centers at:

```
x = 4217, 4218, 4219, 4220, 4221, 4222
y = 3885, 3886, 3887, 3888, 3889, 3890
```

Note that there are 6 pixels in each direction, as specified by the region. That means that positions will pass the filter successfully if:

```
4217 <= (int)x <= 4222
3885 <= (int)y <= 3890
```

Photons with position values of x=4216.4 or y=3884.08 will NOT pass.

Note that the position values are integerized, in effect, binned into image values. This means that x=4222.4 will pass this filter, but not the analytic filter above. We do this to maintain the design goal that either all counts in a pixel are included in an integerized filter, or else none are included.

[It could be argued that the correct photon limits for floating point row data really should be:

```
4216.5 <= x <= 4222.5
3884.5 <= y <= 3890.5
```

since each pixel extends for .5 on either side of the center. We chose to the maintain integerized algorithm for all image-style filtering so that funcnts would give the exact same results regardless of whether a table or a derived non-blocked binned image is used.]

Go to Funtools Help Index

Last updated: November 16, 2005

RegDiff:Differences Between Funtools and IRAF Regions

Summary

Describes the differences between Funtools/ds9 regions and the old IRAF/PROS regions.

Description

We have tried to make Funtools regions compatible with their predecessor, IRAF/PROS regions. For simple regions and simple boolean algebra between regions, there should be no difference between the two implementations. The following is a list of differences and incompatibilities between the two:

- If a pixel is covered by two different regions expressions, Funtools assigns the mask value of the **first** region that contains that pixel. That is, successive regions **do not** overwrite previous regions in the mask, as was the case with the original PROS regions. This means that one must define overlapping regions in the reverse order in which they were defined in PROS. If region N is fully contained within region M, then N should be defined **before** M, or else it will be "covered up" by the latter. This change is necessitated by the use of optimized filter compilation, i.e., Funtools only tests individual regions until a proper match is made.

- The **PANDA** region has replaced the old PROS syntax in which a **PIE** accelerator was combined with an **ANNULUS** accelerator using **AND**. That is,

  ```
  ANNULUS(20,20,0,15,n=4) & PIE(20,20,0,360,n=3)
  ```

  has been replaced by:

  ```
  PANDA(20,20,0,360,3,0,15,4)
  ```

  The PROS syntax was inconsistent with the meaning of the **AND** operator.

- The meaning of pure numbers (i.e., without format specifiers) in regions has been clarified, as has the syntax for specifying coordinate systems. See the general discussion on Spatial Region Filtering for more information.

Go to Funtools Help Index

Last updated: November 16, 2005

FunCombine: Combining Region and Table Filters

Summary

This document discusses the conventions for combining region and table filters, especially with regards to the comma operator.

Comma Conventions

Filter specifications consist of a series of boolean expressions, separated by commas. These expressions can be table filters, spatial region filters, or combinations thereof. Unfortunately, common usage requires that the comma operator must act differently in different situations. Therefore, while its use is intuitive in most cases, commas can be a source of confusion.

According to long-standing usage in IRAF, when a comma separates two table filters, it takes on the meaning of a boolean **and**. Thus:

```
foo.fits[pha==1,pi==2]
```

is equivalent to:

```
foo.fits[pha==1 && pi==2]
```

When a comma separates two spatial region filters, however, it has traditionally taken on the meaning of a boolean **or**. Thus:

```
foo.fits[circle(10,10,3),ellipse(20,20,8,5)]
```

is equivalent to:

```
foo.fits[circle(10,10,3) || ellipse(20,20,8,5)]
```

(except that in the former case, each region is given a unique id in programs such as funcnts).

Region and table filters can be combined:

```
foo.fits[circle(10,10,3),pi=1:5]
```

or even:

```
foo.fits[pha==1&&circle(10,10,3),pi==2&&ellipse(20,20,8,5)]
```

In these cases, it is not obvious whether the command should utilize an **or** or **and** operator. We therefore arbitrarily chose to implement the following rule:

- if both expressions contain a region, the operator used is **or**.
- if one (or both) expression(s) does not contain a region, the operator used is **and**.

This rule handles the cases of pure regions and pure column filters properly. It unambiguously assigns the boolean **and** to all mixed cases. Thus:

```
foo.fits[circle(10,10,3),pi=1:5]
```

and

```
foo.fits[pi=1:5,circle(10,10,3)]
```

both are equivalent to:

```
foo.fits[circle(10,10,3) && pi=1:5]
```

[NB: This arbitrary rule **replaces the previous arbitrary rule** (pre-funtools 1.2.3) which stated:

- if the 2nd expression contains a region, the operator used is **or**.
- if the 2nd expression does not contain a region, the operator used is **and**.

In that scenario, the **or** operator was implied by:

```
pha==4,circle 5 5 1
```

while the **and** operator was implied by

```
circle 5 5 1,pha==4
```

Experience showed that this non-commutative treatment of the comma operator was confusing and led to unexpected results.]

The comma rule must be considered provisional: comments and complaints are welcome to help clarify the matter. Better still, we recommend that the comma operator be avoided in such cases in favor of an explicit boolean operator.

Go to Funtools Help Index

Last updated: November 16, 2005

FunEnv: Funtools Environment Variables

Summary

Describes the environment variables which can be used to tailor the overall Funtools environment.

Description

The following environment variables are supported by Funtools:

FITS_EXTNAME
The **FITS_EXTNAME** environment variable specifies the default FITS extension name when FunOpen() is called on a file lacking a primary image. Thus,

```
setenv FITS_EXTNAME "NEWEV"
```

will allow you to call FunOpen() on files without specifying NEWEV in the Funtools bracket specification. If no FITS_EXTNAME variable is defined and the extension name also is not passed in the bracket specification, then the default will be to look for standard X-ray event table extension names "EVENTS" or "STDEVT" (we are, after all, and X-ray astronomy group at heart!).

FITS_EXTNUM
The **FITS_EXTNUM** environment variable specifies the default FITS extension number when FunOpen() is called on a file lacking a primary image. Thus,

```
setenv FITS_EXTNUM 7
```

will allow you to call FunOpen() on files to open the seventh extension without specifying the number in the Funtools bracket specification.

FITS_BINCOLS and **EVENTS_BINCOLS**
These environment variable specifies the default binning key for FITS binary tables and raw event files, respectively. They can be over-ridden using the **bincols=[naxis1,naxis2]** keyword in a Funtools bracket specification. The value of each environment variable is a pair of comma-delimited columns, enclosed in parentheses, to use for binning. For example, if you want to bin on detx and dety by default, then use:

```
setenv FITS_BINCOLS "(detx,dety)"
```

in preference to adding a bincols specification to each filename:

```
foo.fits[bincols=(detx,dety)]
```

FITS_BITPIX and **EVENTS_BITPIX**
These environment variable specifies the default bitpix value for binning FITS binary tables and raw event files, respectively. They can be over-ridden using the **bitpix=[value]** keyword in a Funtools bracket specification. The value of each environment variable is one of the standard FITS bitpix values (8,16,32,-32,-64). For example, if you want binning routines to create a floating array, then

use:

```
setenv FITS_BITPIX -32
```

in preference to adding a bitpix specification to each filename:

```
foo.fits[bitpix=-32]
```

ARRAY

The **ARRAY** environment variable specifies the default definition of an array file for Funtools. It is used if there is no array specification passed in the **ARRAY()** directive in a Non-FITS Array specification. The value of the environment variable is a valid array specification such as:

```
setenv ARRAY "s100.150"
foo.arr[ARRAY()]
```

This can be defined in preference to adding the specification to each filename:

```
foo.arr[ARRAY(s100.150)]
```

EVENTS

The **EVENTS** environment variable specifies the default definition of an raw event file for Funtools. It is used if there is no EVENTS specification passed in the **EVENTS()** directive in a Non-FITS EVENTS specification. The value of the environment variable is a valid EVENTS specification such as:

```
setenv EVENTS "x:J:1024,y:J:1024,pi:I,pha:I,time:D,dx:E:1024,dx:E:1024"
foo.ev[EVENTS()]
```

This can be defined in preference to adding the specification to each filename:

```
foo.ev[EVENTS(x:J:1024,y:J:1024,pi:I,pha:I,time:D,dx:E:1024,dx:E:1024)]
```

The following filter-related environment variables are supported by Funtools:

FILTER_PTYPE

The **FILTER_PTYPE** environment variable specifies how to build a filter. There are three possible methods:

process or p
: The filter is compiled and linked against the funtools library (which must therefore be accessible in the original install directory) to produce a slave program. This program is fed events or image data and returns filter results.

dynamic or d (gcc only)
: The filter is compiled and linked against the funtools library (which must therefore be accessible in the original install directory) to produce a dynamic shared object, which is loaded into the funtools program and executed as a subroutine. (Extensive testing has shown that, contrary to expectations, this method is no faster than using a slave process.)

contained or c
: The filter and all supporting region code is compiled and linked without reference to the funtools library to produce a slave program (which is fed events or image data and returns filter results). This method is slower than the other two, because of the time it takes to compile the region

filtering code. It is used by stand-alone programs such as ds9, which do not have access to the funtools library.

By default, **dynamic** is generally used for gcc compilers and **process** for other compilers. However the filter building algorithm will check for required external files and will use **contained** is these are missing.

FUN_MAXROW

The **FUN_MAXROW** environment variable is used by core row-processing Funtools programs (funtable, fundisp, funcnts, funhist, funmerge, and funcalc) to set the maximum number of rows read at once (i.e. it sets the third argument to the FunTableRowGet() call). The default is 8192. Note that this variable is a convention only: it will not be a part of a non-core Funtools program unless code is explicitly added, since each call to FunTableRowGet() specifies its own maximum number of rows to read. NB: if you make this value very large, you probably will need to increase **FUN_MAXBUFSIZE** (see below) as well.

FUN_MAXBUFSIZE

The **FUN_MAXBUFSIZE** environment variable is used to limit the max buffer size that will be allocated to hold table row data. This buffer size is calculated to be the row size of the table multiplied by the maximum number of rows read at once (see above). Since the row size is unlimited (and we have examples of it being larger than 5 Mb), it is possible that the total buffer size will exceed the machine capabilities. We therefore set a default value of 5Mb for the max buffer size, and adjust maxrow so that the total size calculated is less than this max buffer size. (If the row size is greater than this max buffer size, then maxrow is set to 1.) This environment variable will change the max buffer size allowed.

FILTER_CC

The **FILTER_CC** environment variable specifies the compiler to use for compiling a filter specification. You also can use the **CC** environment variable. If neither has been set, then gcc will be used if available. Otherwise cc is used if available.

FILTER_EXTRA

The **FILTER_EXTRA** environment variable specifies extra options to add to a filter compile command line. In principle, you can add libraries, include files, and compiler switches. This variable should be used with care.

FILTER_TMPDIR

The **FILTER_TMPDIR** environment variable specifies the temporary directory for filter compilation intermediate files. You also can use the **TMPDIR** and **TMP** variables. By default, /tmp is used as the temporary directory.

FILTER_KEEP

The **FILTER_KEEP** environment variable specifies whether the intermediate filter files (i.e. C source file and compile log file) should be saved after a filter is built. The default is "false", so that these intermediate files are deleted. This variable is useful for debugging, but care should be taken to reset its value to false when debugging is complete.

Go to Funtools Help Index

Last updated: November 16, 2005

Funtools ChangeLog

This ChangeLog covers both the Funtools library and the suite of applications. It will be updated as we continue to develop and improve Funtools. The up-to-date version can be found here. [The changelog for the initial development of Funtools, covering the beta releases, can be found here.]

Patch Release 1.4.5 (internal ds9 release)

- Removed permission checking from Find() on cygwin systems. This was broken by Windows 7.
- Removed addition of -no-cpp-precomp flag from gcc 4.2 and beyond.

Patch Release 1.4.4 (internal ds9 release)

- Fixed -J funcone, which was not properly outputting all rows.
- Filter: when an image is flipped, the rotation angle must be reversed.
- Filter: add support for windows-based ipc communication when using tcc compiler.

Patch Release 1.4.3 (internal ds9 release)

- Filter: improve checks for existence of compiler, even if CC is set.
- Change launch.h to xlaunch.h to avoid conflict with OS X.
- handle flipped images in filtering

Patch Release 1.4.2 (internal ds9 release)

- Port to gcc 4.2.
- Fix 1-byte filters on intel machines (missing SW1 no-op).
- Remove ambiguity from filt.l (and calc.l) using [A-z] in a case-insensitive lexer.
- In funsky, the default unit for RA was changed from hours to degrees.
- Fixed bug in funtable in which TCRVL header values were output as strings.
- Added support for running funtools filters in Rosetta (i.e. running PPC executables on an Intel Mac) by sensing and handling data swap requirements. Only works with FILTER_PTYPE set to 'c' (can't link against wrong architecture libraries).

- Fixed bug in FITS library to allow "-" in extension names.
- Code and documentation now agree that the copy extension specifier ('+') comes after the extension name.

Patch Release 1.4.1 (internal ds9 release)

- Modified internal Launch() routine to use posix_spawn(), if necessary. This is required for OS X 10.5 (leopard), which frowns upon use of fork() and exec(). Also modified zprocess routines to use Launch().

Public Release 1.4.0 (15 August 2007)

- Public release of production-quality code, incorporating changes and improvements from previous beta releases, including:
  - Support for access to ASCII text column files.
  - Support for fast indexed access of binary tables.
  - Support for database views of tables, i.e. pre-set values for the filter specification, the columns to activate, and display format.
  - New programs include funcone (cone search), funindex (create index files), and funcen (calculate centroids within regions).

Release 1.3.0b[n] (mainly internal SAO beta releases)

- Added -F[c] switch to change the column delimiter to the specified character.
- Extended fundisp's format switch (-f) so that it can now handle complex formats such as 'x=sometext%3d- y=othertest%3d.ext'.
- Added support for creating and processing 1D FITS images.
- Added vcol=colname and vcol=/colname to filter specifications to support use of a third value column when binning 2D images.
- Added switches to funcone to write out data rows are not within any cone (-J, -X) and centers which have no close data rows (-L).
- In funjoin, added ability to specify a numeric tolerance for when joining two files.
- shared memory support in gio now can create a shared segment if w+ is specified as the open mode.
- Changed reggeometry man page so that examples correctly show angles going counter-clockwise from the x-axis instead of from the y-axis.

- Added checks to funmerge to ensure that all files have the same columns.
- Fixed bug in text support that prevented header-less files from being processed properly.
- Added support for 64-bit images (bitpix=64) and table columns (TFORM=K).
- Filter code was not applying bscale/bzero to columns.
- Fixed funimage bug that caused a .5/block error in WCS CRPIX values generated from binary tables.
- Added feq(a,b) and div(a,b) macros to funcalc.
- Added support for single-line #define to funcalc.
- Updated wcs library to 3.6.6
- Fix bug in funcen in which ra,dec was not being calculated correctly if physical and image coords did not match up.
- The filter syntax "col1 = col2" now explicitly generates an error (you really want to do "col1 == col2").
- Added -o switch to include offset from the nominal target position.
- Fundisp now displays multi-dimensional vector columns properly.
- Documented support for lists of files processed as a single file using "list: file1 ... filen" syntax.
- Fixed bugs in support for pipe file type (i.e. ability to pass commands as a filename using "pipe: cmd arg1 ... argn" syntax).
- Fixed bug in funhist processing of image-based pixel histograms (i.e using "xy" for columns) where a region was specified. All pixels outside the region were erroneously being added to the bin containing the 0 value.
- Disabled multi-file processing in funds9, which was breaking support for pathnames containing spaces and is not used by ds9 anyway.
- Added support for Views of tables, i.e. pre-set values for the filter specification, the columns to activate, and display format (though the latter is for fundisp only).
- Added -l switch to funimage to read x, y, val columns from a list.
- Removed useless and meaningless section syntax foo'[*]' because it breaks pointer de-referencing on string columns (i.e. foo'[*xxx=='a']'). Use foo'[*,*]' instead, as documented.
- String variables were not always being terminated properly in the filter code because FITS 'A' data is not necessarily null-terminated.

- Added funtools version number to all usage() displays.
- Added explanation of switch arguments to many usage() displays.
- The filter keyword row# now supports single row selection as well as range selection, i.e., "row#=100" along with previous "row#=100:200".
- fundisp now outputs "0x" before hex values.
- Fixed bug in filter parser which processed rangelists incorrectly if spaces were put into the rangelist (i.e. "pha= 1 : 3" instead of pha=1:3).
- Fixed a bug in funindex which created a wrongly named index file if more than one "." was in the input file name.
- Added support to funcone to take ra, dec, radius from a list (i.e. columns in a FITS file or a text file).
- Fixed a bug in FunColumnActivate so that if some columns are explicitly activated while others are de-activated, only the explicitly activated columns are activated (code was activating all columns in this case).
- Fixed a bug in funindex which prevented indexing tables containing a column named N.
- fundisp now encloses ASCII column values in single quotes (unless -T is specified to output RDB format).
- If a filter specification only involves indexed columns, then the compiled filter is not used.
- Funmerge can now be given a list of files to merge using @list syntax. Also removed the restriction on how many files can be merged (was limited to the max number of open files).
- Added ability to edit (add, delete, modify) header parameters in funhead by specifying an output file (editing acts as a filter) and an edit command file (which can be stdin).
- Funtools now contains preliminary code to support (fast) indexed access of binary tables. See idx.html or "man funidx" for more details.
- Funtools now contains preliminary code supporting access to ASCII column files. See text.html or "man funtext" for more details.
- Fixed bug in funcalc in which columns used in an expression were always being replaced by new columns, with all associated parameters (e.g. WCS) were being deleted. Now this only happens if the column explicitly changes its data type.
- Fixed bug in funcalc in which the raw data and user data became out of sync for one row after every 8192 (FUN_MAXROW) rows.

- Fixed bug in gio in which gseek returned 0 instead of the current byte offset for disk files.
- Added funcone program to perform cone search on RA, Dec columns in a FITS binary table.
- Fixed bug in polygon, pie and rotated box region filtering for tables (nearby rows exactly in line between two non-vertical or non-horizontal vertices were being accepted incorrectly).
- Fixed pie and panda regions so that the angles now start from positive x axis == 0 degrees and run counter-clockwise, as documented. They were going from positive y. NB: a similar change was made to ds9 release 4.0b3. You must be using ds9 4.0b3 or later in order to have the correct behavior when generating regions in ds9 and using them in funtools.
- Added -p [prog] switch to funcalc to save the generated program. instead of executing (and deleting) it.
- Upgraded zlib to 1.2.3.

Patch Release 1.2.4 (internal SAO and beta release only)

- In funcalc, added support for user-specified arguments via the -a [argstr] switch. These arguments are accessed in the compiled program using the supplied ARGC and ARGV(n) macros.
- Added -n (no header display) to fundisp to skip outputting header.
- Added checks for various types of blank filters.
- Added macros NROW (current row number) and WRITE_ROW (write current row to disk) to funcalc.
- funcalc no longer requires that at least one data column be specified in the compiled expression.
- Added FUN_NROWS to FunInfoGet() to return the total number of rows in an input table (i.e. value of NAXIS2).
- The compiled funcalc program now includes stdlib.h and unistd.h.
- The util/NaN.h header file is now modified at configure time to contain endian status for the target architecture. References to specific platforms have been removed.
- Added -m switch to funtable to output multiple files, one for each input region (and a separate file for events that pass the filters but are not in any region).
- Added ability to add new parameters (FunParamPutx) after writing data if space is previously reserved in the form of a blank parameter whose value is the name of the param to be updated. (Also requires the append argument of FunParamPutx be set to 2).

- Added ability to build shared libraries. With --enable-shared=yes, shared library is built but not used. With --enable-shared=link, shared library is linked against (requires proper installation and/or use of LD_LIBRARY_PATH).

- Added -v [column] support to funcnts so that counts in a table can be accumulated using values from a specified column (instead of the default case where an integral count is accumulated for each event in a region).

- Added funcen program to calculate centroids within regions (binary tables only). Also added support for a funcen-based centroid tool to funtools.ds9.

- Fixed bug which prevented successful filtering of columns containing arrays.

- Added filter check to ensure that a column is not incorrectly used as an array.

- Fundisp now displays column arrays indexed from 0, not 1.

- Added -i [interval] support to funcnts so that multiple intervals can be processed in a single pass through the data. For example, specifying -i "pha=1:5;pha=6:10;pha=11:15" will generate results in each of 3 pha bands.

- Fixed calculation of LTV quantities when binning floating point column data (value was off by 0.5).

- Added support for 'D' in floating point header values.

- Added -a switch to funimage and funtable to append output image or table to an existing FITS file (as an IMAGE or BINTABLE extension).

- Added support for column scaling (TSCAL and TZERO) on input columns. Note that the default column type is changed to accommodate scaling (e.g. a column of type 'I' is changed to 'J', 'J' is changed to 'D') so that the scaled values can be handled properly by programs such as fundisp (which utilize default types).

- Added support to FunColumnSelect() for handling structs of arrays (i.e. where returned columns are contiguous) instead of the default array of structs (returned row are contiguous). This is done by specifying "org=structofarrays" in the plist and passing a single struct containing the arrays.

- When writing an rdb/starbase file, fundisp now outputs the full column name, regardless of the width of the column (which ordinarily is truncated to match).

- Fixed support for large files by changing all file positions variables from "long" declarations to "off_t.

- Fixed bug in funcalc incorrectly processed multiple array references (e.g. cur->foo[0]=cur->x;cur->foo[1]=cur->y;) within a single line of code.

- Added FILTER_CFLAGS environment variable for all filtering. Also added --with-filter-cc and --with-filter-cflags options on configure to allow specification of a default C compiler and associated CFLAGS for filtering. All of this is necessary in order to support 64-bit libraries under Solaris.

- Added the funtbl script to extract a table from Funtools ASCII output.
- Added code to funimage to update IRAF DATASEC keyword.
- Added checks to ensure that image dimensions are positive.
- Fixed a bug in funimage where int data was being scaled using BSCALE and BZERO but these keywords also were being retained in the output image header. Now the data are not scaled unless the output data type is float (in which case the scaling parameters are removed).
- Fixed a bug in funmerge which prevented merging of files unless one of the -f, -w, or -x switches were used.
- Fixed a bug in funtable and fundisp which caused the special '$n' column to be output incorrectly.
- Fixed sort option in funtable, which previously worked only if the record size was an even divisor of 8192 (and returned garbage otherwise).
- Fixed bug in filters involving FITS data type 'X' (bitfield).
- Fixed bug in funcnts in which the output angles and radii were being displayed incorrectly when multiple panda shapes were specified.
- Fixed bug in pandas and pies using n= syntax when first angle specified was greater than second. The resulting mask was of the correct shape but contained only a single region.
- Table row access routines will now decrease maxrows if memory cannot be allocated for maxrows*sizeof(row), i.e. if the size of a row is so large that space for maxrows cannot be allocated.
- The FUN_MAXBUFSIZE environment variable was added to limit the max buffer size that will be allocated to hold table row data. The default is 5Mb.
- Generated PostScript and PDF versions of the help pages.
- Moved OPTIONS section before (often-lengthy) DESCRIPTION section in man pages.
- All memory allocation now does error checking on the result (except wcs library, which is external code).
- Removed some compiler warnings that surfaced when using gcc -O2.
- Updated wcs library to 3.5.5.
- Upgraded zlib to 1.2.1.

Patch Release 1.2.3 (12 January 2004)

- Generated man pages from the html pages. These are installed automatically at build time.
- Changed instances of sprintf() to snprintf() to protect against buffer overflow.
- Fixed a number of compiler warnings in non-ANSI compilers.
- Increased SZ_LINE parameter value from 1024 to 4096.

Patch Release 1.2.3b1 (19 August 2003)

- The rule for using comma to separate a table filter expression and a region expression has been changed. The rule now states:
  - if both expressions contain a region, the operator used is **or**.
  - if one (or both) expression(s) does not contain a region, the operator used is **and**.

  This rule handles the cases of pure regions and pure column filters properly. It unambiguously assigns the boolean **and** to all mixed cases. Thus:

  ```
  foo.fits[circle(10,10,3),pi=1:5]
  ```

  and

  ```
  foo.fits[pi=1:5,circle(10,10,3)]
  ```

  both are equivalent to:

  ```
  foo.fits[circle(10,10,3) && pi=1:5]
  ```

- When include files are used in filters, they now have implied parentheses surrounding them. Thus, if a region file foo.reg contains two regions (e.g. circle 1 2 3 and circle 4 5 6), the syntax:

  ```
  pha=4:5&&@foo.reg
  ```

  is equivalent to:

  ```
  pha=4:5 && (circle 1 2 3 || cir 4 5 6)
  ```

  instead of:

  ```
  pha=4:5 && circle 1 2 3 || cir 4 5 6
  ```

  and the pha filter is applied to both regions.

- Filters and comments now can be terminated with the string literal "\n" as well as ";" and the new-line character. This means that a region can have comments embedded in it:

  ```
  funcnts foo.fits "circle 512 512 10 # color=red\n circle 512 512 20"
  ```

- Added capability to update the value of an existing parameter after writing the table or image (assuming the output image is a disk file or is being redirected into a file).
- Improved handling of parentheses in filter expressions.
- Fixed a bug in image (not event) regions in which circles and annuli with radius of 1 pixel were not being processed. No counts and no area would be found in such regions.
- Fixed a bug in funcnts in which the radii column values for out of sync if multiple annuli were specified (instead of a single varargs or accel annulus).
- By default, fundisp will display integer image data as floats if the BSCALE and BZERO header parameters are present.
- Added -L switch to funhead to output starbase list format.
- Changed the name of the routine _FunColumnSelect to FunColumnSelectArr, in order to emphasize that it is not a private routine.
- Funcalc now checks to ensure that a column was specified as part of the expression.
- Funcalc local variables in the compiled program now use a "__" prefix to avoid conflicts with user-defined variables.
- Unofficial unsigned short (bitpix=-16) image data now is scaled correctly using BSCALE and BZERO header parameters.
- Ported to Intel icc and gcc 3.3 compilers.
- Updated wcs library to 3.5.1.
- Changed license from public domain to GNU GPL.

Patch Release 1.2.2 (18 May 2003)

- Fixed funcalc so that it now actually compiles an expression and runs it, instead of getting a "filter compilation error". Oops!
- Fixed bug in FunOpen in which the bracket specification was being removed from the filename if a disk file was opened for "w" or "a".
- Fixed bug in FunFlush which prevented two successive calls to FunImagePut from writing the second extension header properly.
- All filter routines now use gerror(stderr, ...) call instead of fprintf(stderr, ...) so that output to stderr can be turned off (via setgerror(level) or GERROR environment variable).

- All standard Funtools programs check for GERROR environment variable before setting gerror flag.
- Some error messages about invalid region arguments were not being printed.
- FITS parameters/headers now conform more closely to FITS standard:
  - Blank keywords are treated in the same way as COMMENTS and HISTORY cards
  - XTENSION keywords are now exactly 8 characters long
  - 'E' is output instead of 'e' in floating point param values
  - PCOUNT and GCOUNT are output correctly for image extensions
  - EXTEND=T is output in primary header
  - COMMENTS and HISTORY start in column 9

Patch Release 1.2.1 (24 April 2003)

- Varargs ellipse and box annular regions were being processed incorrectly when the following conditions all were met:
  - the region was specified in physical or wcs coordinates
  - the data file contained LTM/LTV keywords, i.e., it was blocked with respect to the original data file
  - the program being run was an image program (e.g. funcnts, funimage)

  Varargs ellipse and boxes are regions of the form:

  ```
  ellipse x y a1 b1 a2 b2 ... an bn [angle]
  box x y l1 w1 l2 w2 ... ln wn [angle]
  ```

  where at least 2 sets of axis (length) values were specified to form an annulus (i.e. simple ellipses and boxes worked properly). With all of the above conditions met, a region in physical coordinates saw its second length argument converted incorrectly from physical coordinates to image coordinates. In simple terms, this means that funcnts did not process elliptical or box regions in physical coords on blocked images properly. Note that blocking on the command line (e.g. foo.fits[*,*,2]) did work when no LTM/LTV keywords existed in the file.

- The fundisp -f switch now supports specification of column-specific display formats as well as a more convenient way to specify datatype-specific display formats. Both use keyword=value specifiers. For columns, use:

  ```
  fundisp -f "colname1=format1 colname2=format2 ..." ...
  ```

  e.g.

  ```
  fundisp -f "time=%13.2f pha=%3d" ...
  ```

  You also can specify display formats for individual datatypes using the FITS binary table TFORM variables as the keywords:

  ```
  fundisp -f "D=double_format E=float_format J=int_format etc."
  ```

  e.g.

```
fundisp -f "D=%13.2f I=%3d" ...
```

The old position-dependent syntax is deprecated.

- Fundisp will now print out a single 16-bit (or 32-bit) unsigned int for a column whose data format is 16X (or 32X), instead of printing 2 (or 4) unsigned chars.
- Fixed bug in which fundisp was not able to display bitfield data for raw event lists.
- Previously, when binning columns used implicitly in a region and explicitly in a filter could suffer from a case sensitivity problem. This has been fixed.
- Fixed internal mask=all switch on fundisp.
- Filter include files now simply include text without changing the state of the filter. They therefore can be used in expression. That is, if foo1 contains "pi==1" and foo2 contains "pha==2" then the following expressions are equivalent:

```
"[@foo1&&@foo2]" is equivalent to "[pi==1&&pha==2]"
"[pha==1||@foo2]"  is equivalent to "[pi==1||pha==2]"
"[@foo1,@foo2]" is equivalent to "[pi==1,pha==2]"
```

- Fixed bug in filter specification which caused a SEGV if a varargs-style region was enclosed in parens.
- Updated wcs library to 3.3.2.

Public Release 1.2.0 (24 March 2003)

- BSCALE and BZERO are now always applied to int pixel data, instead of only being applied if the desired output is floating point.

Beta Release 1.2.b3 (4 February 2003)

- In FunColumnSelect, added the ability to specify an offset into an array in the type specification, using the extended syntax:

```
[@][n]<type>[[poff]][:[tlmin[:tlmax[:binsiz]]]]
```

The [poff] string specifies the offset. For example, a type specification such as "@I[2]" specifies the third (i.e., starting from 0) element in the array pointed to by the pointer value. A value of "@2I[4]" specifies the fifth and sixth values in the array.

- Added a non-varargs version of FunColumnSelect called _FunColumnSelect:

```
int _FunColumnSelect(Fun fun, int size, char *plist,
                     char **names, char **types, char **modes, int *offsets,
                     int nargs);
```

- Added support for sorting binary tables by column name using: funtable -s "col1 col2 ... coln" ...
- Added the FUN_RAW macro which, when applied to the "name" parameter of FunParamGets(), returns the 80-character raw FITS card instead of only the value.
- Added support for comparing column values with binary masks of the form 0b[01]+, e.g.:

  ```
  (status&0b111)==0b001
  ```

  Previously, such masks had to be specified in decimal, octal, or hex.
- Completed support for type 'L' (logical) in fundisp and in filtering of binary tables.
- Fixed bug in funhist that was improperly setting the number of bins when the data was of type float.
- Fixed bug in filter/Makefile where the filter OBJPATH #define was being passed to the wrong module.

Beta Release 1.2.b2 (7 October 2002)

- Updated wcs library to 3.1.3.
- Added support for reading gzip'ed files via stdin.

Beta Release 1.2.b1 (24 September 2002)

- Added the following accelerators to region filtering:

  ```
  shape:      arguments:
  -----       ---------
  BOX         xcenter ycenter xw1 yh1 xw2 yh2 ... xwn yhn (angle)
  BOX         xcenter ycenter xwlo yhin xwout yhhi n=[number] (angle)
  CIRCLE      xcenter ycenter r1 r2 ... rn              # same as annulus
  CIRCLE      xcenter ycenter rinner router n=[number]  # same as annulus
  ELLIPSE     xcenter ycenter xw1 yh1 xw2 yh2 ... xwn yhn (angle)
  ELLIPSE     xcenter ycenter xwlo yhin xwout yhhi n=[number] (angle)
  ```

- Added the following new pandas (Pie AND Annulus) to region filtering:

  ```
  shape:    arguments:
  -----     ---------
  CPANDA    xcen ycen ang1 ang2 nang irad orad nrad  # same as panda
  BPANDA    xcen ycen ang1 ang2 nang ixlo iylo ixhi iyhi nrad (ang) # box
  EPANDA    xcen ycen ang1 ang2 nang ixlo iylo ixhi iyhi nrad (ang) # ellipse
  ```

- Added support for filtering images using simple FITS image masks, i.e. 8-bit or 16-bit FITS images where the value of a pixel is the region id number for that pixel (and therefore must be greater than 0). The image section being filtered must either be the same size as the mask dimensions or else be an even multiple of the mask. This works with image-style filtering, i.e., funcnts can utilize a mask on both images and binary tables.

- Added '$n' to fundisp column specification to allow display of ordinal value of each row passing the filter.
- Added code to support region filtering on image sections.
- Fixed bugs which prevented filtering more than one ASCII region file.
- Fixed bug occasionally causing filter slave processes to become zombies.
- Fixed bugs in event filtering: annulus with inner radius of 0 (i.e., a circle) was rejecting events with coordinates xcen, ycen. Also, pie with angles of 0 and 360 was rejecting some events. Image filtering (e.g. funcnts) did not have these problems.
- Filters now accept global exclude regions without an include region. In such a case, the field region is implied. That is, "-circle(x,y,r)" is equivalent to "field; -circle(x,y,r)", etc.
- Fixed panda so that it can be used as a global exclude.
- Allow empty ds9 region file (comments and globals only) to be a valid filter. Totally ignore zero length region or include file.
- Fixed funcnts bug that was displaying 0 value as inner radius of a circle, instead of just one radius value.

Public Release 1.1.0 (22 April 2002)

New features include:

- Funtools programs now accept gzip'ed files as valid input.
- Improved security via replacement of system() function.
- fundisp, funcnts, funhist can output starbase/rdb format (tabs between columns, form-feeds between tables).
- Improved support for Windows platform, as well as new support for Mac OSX.

Pre-Release 1.1.0e (10 April 2002)

- Added enough support to skip over variable length arrays in BINTABLES. We will add full support if this non-standard construct becomes more widely used.
- Fixed bug in underlying fitsy _gread() routine that was returning an arbitrary bytes-read value if the input fd was invalid.

Pre-Release 1.1.0e (19 March 2002)

- Added additional check for Windows/PC to filter/Nan.h.
- Upgraded zlib library to 1.1.4 (fix double free security hole).

Pre-Release 1.1.0e (27 February 2002)

- Changed filter/process.[ch] to filter/zprocess.[ch] to avoid name collision with Cygwin include file.
- Added -a switch to funhead to display all headers in a FITS file.

Pre-Release 1.1.0e (11 February 2002)

- Fixed filter parser so that it ignores ds9 "ruler" and "text" markers only up to the first \n or ; (was ignoring to last \n).
- The NBLOCK parameter in fitsy/headdata.c was too large for Mac OS X (max size of a declared char buf seems to be about .5 Mb).

Beta Release 1.0.1b5 (31 January 2002)

- Fixed bug introduced in calculated IRAF LTM values in 1.0.1b3.
- Fixed bug in filter parser giving wrong answers when two range lists were combined with and explicit boolean operator:

  ```
  $ fundisp $S"[x=512&&y=511,512]"
  ```

  incorrectly acted like:

  ```
  fundisp $S"[(x=512&&y=511)||(y=512)]"
  ```

  instead of:

  ```
  fundisp $S"[x=512&&(y=511||y=512)]"
  ```

  In general, we recommend use of explicit parentheses.
- Fixed filter/NaN.h to recognize Compaq Alpha again (broken by their last change to cc).
- Removed redundant varargs definitions that conflicted with Alpha compiler definitions.
- Added blank line to inc.sed to work around Apple Mac OS X bug in which the "i" (insert) command was treating final \\ as continuation \ in the text.

- Added include of mkrtemp.h to mkrtemp.c to get conditional compilation for Mac OSX.
- Added support for --with-zlib to fitsy so that ds9 could use its own copy of zlib (and not build the copy in fitsy).
- Removed config.cache and Makefile files from distribution tar file.

Beta Release 1.0.1b4 (26 January 2002)

- Make explicit that column filters are not permitted in an image expression (such as the funcnts region arguments).
- Fix bug in region parser in which a region (without parens), followed immediately by an operator:

  ```
  circle 512 512 .5&π==1
  ```

  was not processing the final argument of the region correctly.
- Ignore new "tile" directive in filters (used by ds9).

Beta Release 1.0.1b3 (4 January 2002)

- Made modifications to Makefile.in to make releases easier.
- Added instructions Makefile.in so that funtools.h will always have correct #defines for FUN_VERSION, FUN_MAJOR_VERSION, FUN_MINOR_VERSION, and FUN_PATCH_LEVEL.
- Allow #include statements in funcalc program files.
- funimage now updates all 4 CDX_Y values by the block factor.
- Minor changes to make funtools work under darwin (Mac OS X).

Beta Release 1.0.1b2 (14 November 2001)

- Fixed FunOpen() bug (introduced in b1) in which filenames without extensions SEGV'ed on open. Yikes!
- Funmerge now extends the tlmin/tlmax values of the output binning columns so that merged events from widely separated files are valid in the output table.
- In funhist, added -w switch to specify bin width (lo:hi:width) instead of number of bins (lo:hi:num). Added support for this new width option in funtools.ds9.

- If a tdbin value was set using bincols=(name:tlmin:tlmax:tdbin, ...), the WCS parameters were not being updated properly.
- Cleaned up build support for zlib.

Beta Release 1.0.1b1 (6 November 2001)

- Added support for gzip'ed files to the underlying fitsy/gio library. This means that all funtools programs now accept gzip'ed files as valid input:

  ```
  funcnts foo.fits.gz "circle 504 512 10"
  ```

  It is no longer necessary to run gunzip and pipe the results to stdin of a funtools program.
- Funtools tasks are now placed in a sub-menu in the DS9 Analysis menu, instead of at the top level.
- Fixed a bug in funcnts in which the bottom-most pixel of a small circle or annulus region could be missed when the region is only one pixel wide for that value of y.
- Added -n switch to funhist so that table histograms could be normalized by the width of the bin (val/(hi_edge-lo_edge)).
- Added -T switch to fundisp, funcnts, funhist to output in starbase/rdb format (uses tabs instead of spaces between columns, form-feeds between tables, etc.)
- Fixed a bug in which the field() region was not being properly processed in combination with an image section. This could affect funcnts processing of image data where an image section was specified (though it usually resulted in a funcnts error).
- Fixed bug in display of binary table header for vector columns.
- Filters now recognize hex constants (starting with 0x) and long constants (ending with L).
- Filenames containing a ':' are now only treated as sockets if they actually are in the form of a valid ip:port.
- Replaced funtools.ds9 with a new version that calls a new funds9 script, instead of calling funcnts or funhist directly. The new script supports gzip'ed files and bracket specifications on filenames at the same time, which the direct call could not. Also the new script has better error reporting.
- Replaced system() call used to compile filter and funcalc expression with a special launch() call, which performs execvp() directly without going through sh. (launch() works under DOS and has fewer security problems.)
- Fixed image filter code in which the field() region was being ignored if it was combined with one or more exclude regions (and no other include regions), resulting in no valid pixels.

- Changed use of getdtable() to FD_SETSIZE in calls to select().
- Added code to guard against FITS binary tables without proper TFORMx parameters.
- Added support to FunParamGets so that it returns the raw FITS card if the specified input name is NULL and the input n value is positive.
- Fixed bug in underlying fitsy code that set the comment in a header parameter.

Public Release 1.0.0 (31 July 2001)

- "a new day with no mistakes ... yet"

Index to the Funtools Help Pages

Last updated: 22 April 2002