GNU Scientific Library – Reference Manual

Edition 0.4after, for gsl Version 0.4after

Mark Galassi
Cygnus Solutions and Los Alamos National Laboratory
`rosalia@nis.lanl.gov`

Jim Davies
Space Data Systems Group, Los Alamos National Laboratory and
Department of Computer Science, Georgia Institute of Technology
`jimmyd@nis.lanl.gov`

James Theiler
Astrophysics and Radiation Measurements Group, Los Alamos National Laboratory
`jt@nis.lanl.gov`

Brian Gough
Theoretical Particle Physics Group, Los Alamos National Laboratory
`bjg@vvv.lanl.gov`

Reid Priedhorsky
Mathematical Modeling and Analysis Group, Los Alamos National Laboratory
`rp@lanl.gov`

Gerard Jungman
Theoretical Particle Physics Group, Los Alamos National Laboratory
`jungman@nnn.lanl.gov`

Michael Booth
Department of Physics and Astronomy, The Johns Hopkins University
`booth@planck.pha.jhu.edu` or `booth@debian.org`

Table of Contents

1 Preliminaries

The GNU Scientific Library (GSL) is a collection of routines for numerical analysis. The routines are written from scratch by the GSL team (see Appendix B [Contributors to GSL], page 143) in C, and are meant to present a modern Applications Programming Interface (API) for C programmers, while allowing wrappers to be written for very high level languages.

2 Using the library

This chapter describes how to compile programs that use GSL. The library is written in ANSI-C and is intended to conform to the standard. The library does not make use of any extensions in the interface it exports to the user, except where they can be implemented in a way compatible with pure ANSI C. Thus programs you write using GSL should be portable to any system with a working ANSI C compiler and system library, and simultaneously be able to take advantage of compiler extensions on those platforms which support them. When an ANSI C feature is known to be broken on a particular system the library will exclude any related functions so that it is impossible to link a program that would use them and give incorrect results.

2.1 Inline functions

The `inline` keyword is not part of ANSI C, only C++. Since the library header files conform to the ANSI standard this prevents the use `inline`. The library does not export any inline function definitions by default. However, for many frequently used functions there are equivalent inline definitions which can be turned on by defining the macro `HAVE_INLINE` when compiling an application. If you use `autoconf` this can be done automatically using the following test,

```
AC_C_INLINE

if test "$ac_cv_c_inline" != no ; then
  AC_DEFINE(HAVE_INLINE,1)
  AC_SUBST(HAVE_INLINE)
fi
```

and then including '`config.h`' before including any library headers. If you do not define the macro `HAVE_INLINE` then the slower non-inlined versions of the functions are used instead.

2.2 Long double

The extended numerical type `long double` is part of the ANSI C standard and should be available in every modern compiler. However, the `stdio.h` formatted input/output functions `printf` and `scanf` are not always implemented correctly for `long double` in some system libraries, perhaps because the `long double` type is not widely used outside numerical programming. In order to avoid undefined or incorrect results these functions are tested during the `configure` stage of library compilation and certain GSL functions which depend on them are eliminated if necessary,

```
checking whether printf/scanf works with long double... no
```

Consequently if `long double` formatted input/output does not work on a given system then it will not be possible to link a program which uses GSL functions relying on `long double` formatted i/o.

If it is necessary to work on a system which does not support formatted `long double` i/o then the options are to use binary i/o or to convert `long double` results into `double` for reading and writing.

3 Error handling in GSL

This chapter describes the way that GSL functions report and handle errors. By examining the status information returned by every GSL function you can determine whether it succeeded or failed, and if it failed you can find out what the precise cause of failure was. You can also define your own error handling functions to modify the default behavior of the library.

3.1 Error reporting

GSL follows the thread-safe error reporting conventions of the POSIX Threads library. Functions in GSL return a non-zero error code to indicate an error and `0` to indicate success.

```
int status = gsl_function(...)

if (status) { /* an error occurred */
  .....       /* the value of status specifies the type of error */
}
```

GSL routines report an error whenever they cannot perform the task requested of them. For example, a root-finding function would return a non-zero error code if could not converge to the requested accuracy, or exceeded a limit on the number of iterations. Situations like this are a normal occurrence when using any mathematical library and you should check the return status of the GSL functions that you call.

Whenever a GSL routine reports an error the return value specifies the type of error. The return value is analogous to the value of the variable `errno` in the C library. However, the C library's `errno` is a global variable, which is not thread-safe (There can be only one instance of a global variable per program. Different threads of execution may overwrite `errno` simultaneously). By returning the error number directly we can avoid this problem in a simple, portable way. The caller can examine the return code and decide what action to take, including ignoring the error if it is not considered serious.

The error code numbers are defined in the file '`gsl_errno.h`'. They all have the prefix `GSL_` and expand to non-zero constant integer values. Many of the error codes use the same base name as a corresponding error code in C library. Here are some of the most common error codes,

`int` **GSL_EDOM** Macro

Domain error; used by mathematical functions when an argument value does not fall into the domain over which the function is defined (like EDOM in the C library)

`int` **GSL_ERANGE** Macro

Range error; used by mathematical functions when the result value is not representable because of overflow or underflow (like ERANGE in the C library)

`int` **GSL_NOMEM** Macro

No memory available. The system cannot allocate more virtual memory because its capacity is full (like ENOMEM in the C library). This error is reported when a GSL routine encounters problems when trying to allocate memory with `malloc`.

`int` **GSL_EINVAL** Macro

Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function (like EINVAL in the C library).

Here is an example of some code which checks the return value of a function where an error might be reported,

```
int status = gsl_fft_complex_radix2_forward (data, length);

if (status) {
    if (status == GSL_EINVAL) {
       fprintf (stderr, "invalid argument, length=%d\n", length);
    } else {
       fprintf (stderr, "failed, gsl_errno=%d\n", status);
    }
    exit (-1);
}
```

The function `gsl_fft_complex_radix2` only accepts integer lengths which are a power of two. If the variable `length` is not a power of two then the call to the library function will return `GSL_EINVAL`, indicating that the length argument is invalid. The `else` clause catches any other possible errors.

3.2 Error handlers

In addition to reporting errors the library also provides a simple error handler. The error handler is called by library functions when they are about to report an error (for example, just before they return).

The default behavior of the error handler is to print a short message and call `abort()` whenever an error is reported by the library. If a library routine reports an error then the whole program will core-dump. This is a safe default for lazy programmers who do not check the return status of library routines (we don't encourage you to write programs this way). If you turn off the default error handler or provide your own error handler then it is your responsibility to check the return values of the GSL routines.

All GSL error handlers have the type `gsl_error_handler_t`, which is defined in '`gsl_errno.h`',

void gsl_error_handler_t Data Type

This is the type of GSL error handler functions. An error handler will be passed three arguments, specifying the reason for the error, the source file in which it occurred, and the line number in that file. The source file and line number are set at compile time using the `__FILE__` and `__LINE__` directives in the preprocessor. An error handler function returns type `void`. Error handler functions should be defined like this,

`void` *handler* `(const char * reason, const char * file, int line)`

To request the use of your own error handler you need to call the function `gsl_set_error_handler` which is also declared in '`gsl_errno.h`', `gsl_error_handler_t` **gsl_set_error_handler** (`gsl_error_handler_t` *new_handler*) Function

This functions sets a new error handler, *new_handler*, for the GSL library routines. The previous handler is returned (so that you can restore it later). Note that the pointer to a user defined error handler function is stored in a static variable, so there can only be one error handler per program.

```
old_handler = gsl_set_error_handler (&my_error_handler);

.....     /* code uses new handler */

gsl_set_error_handler (old_handler) ; /* restore old handler */
```

To use the default behavior (`abort` on error) set the error handler to `NULL`,

```
old_handler = gsl_set_error_handler (NULL);
```

Here is a skeleton outline of a program which defines its own error handler. Imagine that the program does interactive data analysis – there is a main loop which reads commands from the user and calls library routines with user-supplied arguments,

```
#include <setjmp.h>
#include <gsl_errno.h>

jmp_buf main_loop;
void my_error_handler (const char *reason, const char *file, int line);

main ()
{
   gsl_set_error_handler (&my_error_handler);

   while (1)
     {
       .... /* read command from user */

       if (setjmp (main_loop) == 0)
         {
            .... /* call GSL routines requested by user */
         }
       else
         {
            .... /* my_error_handler bailed out, GSL gave an error */
         }
     }
}

void
my_error_handler (const char *reason, const char *file, int line)
{
    fprintf (stderr, "GSL error: %s\n", reason);
    longjmp (main_loop);
}
```

Before entering the interactive loop the program uses `gsl_set_error_handler` to provide its own error handler `my_error_handler` for GSL error reports. After this point the function `my_error_handler` will be invoked whenever an error is reported by GSL. The new error handler prints the cause of the error (the string `reason`) and then does a non-local jump back to the main loop. This would allow the user to fix the command which caused the error and try again.

3.3 Error streams

GSL supports the concert of an error stream, which is a place where errors are logged as they occur. An error stream allows the library to report an error message directly to the user rather than to the calling program. This can sometimes be useful because it reduces the amount of error checking that the program needs to do.

For example, many mathematical functions compute floating point numbers or other numerical values. The standard versions of these functions accept a pointer for storing their numerical result, so that the status can be returned separately. For example, to compute the first-order Bessel function $J_1(x)$ for $x = 1.23$ and obtain the status we write,

```
double result;
int status = gsl_sf_bessel_J1_e (1.23, &result);
```

where `gsl_sf_bessel_J1_e` is the appropriate function from the special functions (`sf`) module. The suffix `_e` appended to the function name indicates that the return value gives the error status. This style of function is safe and avoids any confusion about what the return value means, but requires a lot of error checking.

For many numerical functions it would be more intuitive to write something like $y = f(x)$. The library provides functions with an alternative interface which allows this,

```
double result = gsl_sf_bessel_J1 (1.23)
```

However, in this case there is no way for the calling program to test for an error. Instead if there are any errors (such as underflow) they are logged to the error stream, and can be examined by the user at the end of the run. It is up to the programmer to decide which form is best suited to a given application. For a truly robust program the standard error checking versions of the functions should be used, since they don't rely on the user examining the error stream.

3.4 Manipulating the error stream

By default the error stream is sent to `stderr`, and you can redirect it to a file on the command line. There are also two ways to change this within your program. Firstly, the stream can be redirected to another file by providing a suitable file pointer. Alternatively you can set up an error stream handler, which is a function that accepts error message strings. By using an error stream handler function you have complete control over where the messages are stored.

`FILE *` **gsl_set_stream** (`FILE *` *new_stream*) Function

This function selects the stream used for GSL error messages. After calling `gsl_set_stream` any further messages sent to the default stream handler will be printed on *new_stream*. The previous stream is returned, so that you can close it or restore it later. Note that the stream is stored in a static variable, so there can only be one error stream per program.

void gsl_stream_handler_t Data Type

This is the type of GSL stream handler functions. A stream handler will be passed four arguments, specifying a label (such as ERROR or WARNING), the source file in which the error occurred, the line number in that file and a description of the error. The source file and line number are set at compile time using the `__FILE__` and `__LINE__` directives in the preprocessor. A stream handler function returns type `void`. Stream handler functions should be defined like this,

```
void handler (const char * label, const char * file,
                   int line, const char * reason)
```

To request the use of your own stream handler you need to call the function `gsl_set_stream_handler` which is also declared in '`gsl_errno.h`',

`gsl_stream_handler_t` **gsl_set_stream_handler** (`gsl_stream_handler_t` *new_handler*) Function

This functions sets a new stream handler, *new_handler*, for the GSL library routines. The previous handler is returned (so that you can restore it later). Note that the pointer to a user defined stream handler function is stored in a static variable, so there can only be one error handler per program.

```
old_handler = gsl_set_stream_handler (&my_error_stream);

.....     /* code uses new handler */

gsl_set_stream_handler (old_handler) ; /* restore old handler */
```

To use the default behavior (print the message to `stderr`) set the stream handler to `NULL`,

```
old_handler = gsl_set_stream_handler (NULL);
```

3.5 Using GSL error reporting in your own functions

If you are writing numerical functions in program which also uses GSL code you may find it convenient to adopt the same error reporting conventions as in the library.

To report an error you need to call the function `gsl_error` with a string describing the error and then return an appropriate error code from `gsl_errno.h`, or a special value, such as `NaN`. For convenience '`gsl_errno.h`' defines two macros to carry out these steps:

GSL_ERROR (*reason, gsl_errno*) Macro

This macro reports an error using the GSL conventions and returns a status value of `gsl_errno`. It expands to the following code fragment,

```
gsl_error (reason, __FILE__, __LINE__, gsl_errno) ;
return gsl_errno ;
```

The macro definition in '`gsl_errno.h`' actually wraps the code in a `do { ... } while (0)` block to prevent possible parsing problems.

Here is an example of how the macro could be used to report that a routine did not achieve a requested tolerance. To report the error the routine needs to return the error code `GSL_ETOL`.

```
if (residual > tolerance)
  {
    GSL_ERROR("residual exceeds specified tolerance", GSL_ETOL) ;
  }
```

GSL_ERROR_RETURN (*reason*, *gsl_errno*, *value*) Macro

This macro is the same as `GSL_ERROR` but returns a user-defined status value of *value* instead of an error code. It can be used for mathematical functions that return a floating point value.

Here is an example where a function needs to return a `NaN` because of a mathematical singularity,

```
if (x == 0)
  {
    GSL_ERROR_RETURN("argument lies on singularity", GSL_ERANGE, NAN) ;
  }
```

4 Random number generation

The *GNU Scientific Library* provides a large collection of random number generators which can be accessed through a uniform interface. Environment variables allow you to select different generators and seeds at runtime, so that you can easily switch between generators without needing to recompile your program. Each instance of a generator keeps track of its own state, allowing the generators to be used in multi-threaded programs. Additional functions are available for transforming uniform random numbers into samples from continuous or discrete probability distributions such as the gaussian, log-normal or poisson distributions.

4.1 General comments on random numbers

In 1988, Park and Miller wrote a paper entitled "Random number generators: good ones are hard to find." [Commun. ACM, 31, 1192–1201]. Fortunately, some excellent random number generators are available, though poor ones are still in common use. You may be happy with the system-supplied random number generator on your computer, but you should be aware that as computers get faster, requirements on random number generators increase. Nowadays, a simulation that calls a random number generator millions of times can often finish before you can make it down the hall to the coffee machine and back.

A very nice review of random number generators was written by Pierre L'Ecuyer, as Chapter 4 of the book: Handbook on Simulation, Jerry Banks, ed. (Wiley, 1997). The chapter is available in postscript from from L'Ecuyer's ftp site (see references). Knuth's volume on Seminumerical Algorithms (originally published in 1968) devotes 170 pages to random number generators, and has recently been updated in its 3rd edition (1997). It is brilliant, a classic. If you don't own it, you should stop reading right now, run to the nearest bookstore, and buy it.

A good random number generator will satisfy both theoretical and statistical properties. Theoretical properties are often hard to obtain (they require real math!), but one prefers a random number generator with a long period, low serial correlation, and a tendency *not* to "fall mainly on the planes." Statistical tests are performed with numerical simulations. Generally, a random number generator is used to estimate some quantity for which the theory of probability provides an exact answer. Comparison to this exact answer provides a measure of "randomness".

4.2 The Random Number Generator Interface

4.2.1 Random number generator initialization

It is important to remember that a random number generator is not a "real" function like sine or cosine. Unlike real functions, successive calls to a random number generator yield different return values. Of course that is just what you want for a random number generator, but to achieve this effect, the generator must keep track of some kind of "state" variable. Sometimes this state is just an integer (sometimes just the value of the previously generated random number), but often it is more complicated than that and may involve a whole array of numbers, possibly with some indices thrown in. To use the random number generators, you do not need to know the details of what comprises the state, and besides that varies from algorithm to algorithm.

The random number generator library uses two special structs, `gsl_rng_type` which holds static information about each type of generator and `gsl_rng` which describes an instance of a generator created from a given `gsl_rng_type`.

The functions described in this section are declared in the header file '`gsl_rng.h`'.

`gsl_rng *` **gsl_rng_alloc** (`gsl_rng_type *` *T*) Random

This function returns a pointer to a newly-created instance of a random number generator of type *T*. For example, the following code creates an instance of the Tausworthe generator,

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

If there is insufficient memory to create the generator then the function returns a null pointer and the error handler is invoked with an error code of `GSL_ENOMEM`.

The generator is automatically initialized with the default seed, `gsl_rng_default_seed`. This is zero by default but can be changed either directly or usubf the environment variable `GSL_RNG_SEED`, see Section 4.2.4 [Random number environment variables], page 12.

The defined generator types are,

```
gsl_rng_cmrg, gsl_rng_minstd, gsl_rng_mrg, gsl_rng_mt19937,
gsl_rng_r250, gsl_rng_ran0, gsl_rng_ran1, gsl_rng_ran2,
gsl_rng_ran3, gsl_rng_rand, gsl_rng_rand48,
gsl_rng_random_bsd, gsl_rng_random_glibc2,
gsl_rng_random_libc5, gsl_rng_randu, gsl_rng_ranf,
gsl_rng_ranlux, gsl_rng_ranlux389, gsl_rng_ranmar,
gsl_rng_slatec, gsl_rng_taus, gsl_rng_tds, gsl_rng_tt800,
gsl_rng_uni, gsl_rng_uni32, gsl_rng_vax, gsl_rng_zuf
```

The details of each generator are given later in this chapter.

`void` **gsl_rng_set** (`const gsl_rng *` *r*, `unsigned long int` *s*) Random

This function initializes (or 'seeds') the random number generator. If the generator is seeded with the same value of *s* on two different runs, the same stream of random numbers will be generated by successive calls to the routines below. If different values of *s* are supplied, then the generated streams of random numbers should be completely different. If the seed *s* is zero then the standard seed from the original implementation is used instead. For example, the original Fortran source code for the `ranlux` generator used a seed of 314159265, and so choosing *s* equal to zero reproduces this when using `gsl_rng_ranlux`.

`void` **gsl_rng_free** (`gsl_rng *` *r*) Random

This function frees all the memory associated with the generator *r*.

4.2.2 Sampling from a random number generator

The following functions return uniformly distributed random numbers, either as integers or double precision floating point numbers. To obtain non-uniform distributions see Section 4.5 [Random Number Distributions], page 22.

`unsigned long int` **gsl_rng_get** (`const gsl_rng *` *r*) Random

This function returns a random integer from the generator *r*. All integers in the range [*min*,*max*] are equally likely. The maximum and minimum values, *max* and *min*, depend on the algorithm used. They can determined using the auxilliary functions `gsl_rng_max (r)` and `gsl_rng_min (r)`.

`double` **gsl_rng_uniform** (`const gsl_rng *` *r*) Random

This function returns a double precision floating point number uniformly distributed in the range [0,1). The range includes 0.0 but excludes 1.0. The value is typically obtained by dividing the result of `gsl_rng_get(r)` by `gsl_rng_max(r) + 1.0` in double precision. Some generators compute this ratio internally so that they can provide floating point numbers with more than 32 bits of randomness (the maximum number of bits that can be portably represented in a single `unsigned long int`).

`double` **gsl_rng_uniform_pos** (`const gsl_rng *` *r*) Random

This function returns a positive double precision floating point number uniformly distributed in the range (0,1), excluding both 0.0 and 1.0. The number is obtained by sampling the generator with the algorithm of `gsl_rng_uniform` until a non-zero value is obtained. You can use this function if you need to avoid a singularity at 0.0.

`unsigned long int` **gsl_rng_uniform_int** (`const gsl_rng *` *r*, `unsigned long int` *n*) Random

This function returns a random integer from 0 to *n-1* inclusive. All integers in the range [0,*n-1*] are equally likely, regardless of the generator used. An offset correction is applied so that zero is always returned with the correct probability, for any minimum value of the underlying generator.

If *n* is larger than the range of the generator then the function calls the error handler with an error code of `GSL_EINVAL` and returns zero.

4.2.3 Auxiliary random number generator functions

The following functions provide information about an existing generator. You should use them in preference to hard-coding the generator parameters into your own code.

`const char *` **gsl_rng_name** (`const gsl_rng *` *r*) Random

This function returns a pointer to the name of the generator. For example,

```
printf("r is a '%s' generator\n", gsl_rng_name (r)) ;
```

would print something like `r is a 'taus' generator`

`unsigned long int` **gsl_rng_max** (`const gsl_rng *` *r*) Random

`gsl_rng_max` returns the largest value that `gsl_rng_get` can return.

`unsigned long int` **gsl_rng_min** (`const gsl_rng *` *r*) Random

`gsl_rng_min` returns the smallest value that `gsl_rng_get` can return. Usually this value is zero. There are some generators with algorithms that cannot return zero, and for these generators the minimum value is 1.

`void *` **gsl_rng_state** (`const gsl_rng *` *r*) Random

`size_t` **gsl_rng_size** (`const gsl_rng *` *r*) Random

These function return a pointer to the state of generator *r* and its size. You can use this information to access the state directly. For example, the following code will write the state of a generator to a stream,

```
void * state = gsl_rng_state (r);
size_t n = gsl_rng_size (r);
fwrite (state, n, 1, stream);
```

4.2.4 Random number environment variables

The library allows you to choose a default generator and seed from the environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED` and the function `gsl_rng_env_setup`. This makes it easy try out different generators and seeds without having to recompile your program.

`const gsl_rng_type *` **gsl_rng_env_setup** (`void`) Function

This function reads the environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED` and uses their values to set the corresponding library variables `gsl_rng_default` and `gsl_rng_default_seed`. These global variables are defined as follows,

```
extern const gsl_rng_type *gsl_rng_default
extern unsigned long int gsl_rng_default_seed
```

The environment variable `GSL_RNG_TYPE` should be the name of a generator, such as `taus` or `mt19937`. The environment variable `GSL_RNG_SEED` should contain the desired seed value. It is converted to an `unsigned long int` using the C library function `strtoul`.

If you don't specify a generator for `GSL_RNG_TYPE` then `gsl_rng_mt19937` is used as the default. The initial value of `gsl_rng_default_seed` is zero.

Here is a short program which shows how to create a global generator using the environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED`,

```
#include <stdio.h>
#include <gsl_rng.h>

gsl_rng * r ;  /* global generator */

int
main ()
{
  gsl_rng_env_setup() ;
```

```
    r = gsl_rng_alloc (gsl_rng_default);

    printf("generator type: %s\n", gsl_rng_name (r));
    printf("seed = %u\n", gsl_rng_default_seed);
    printf("first value = %u\n", gsl_rng_get (r)) ;
  }
```

Running the program without any environment variables uses the initial defaults, an `mt19937` generator with a seed of 0,

```
bjg|zeke> ./a.out
generator type: mt19937
seed = 0
first value = 3510405877
```

By setting the two variables on the command line we can change the default generator and the seed,

```
bjg|zeke> GSL_RNG_TYPE="taus" GSL_RNG_SEED=123 ./a.out
GSL_RNG_TYPE=taus
GSL_RNG_SEED=123
generator type: taus
seed = 123
first value = 2720986350
```

4.2.5 Saving and restoring random number generator state

The above methods ignore the random number 'state' which changes from call to call. It is often useful to be able to save and restore the state. To permit these practices, a few somewhat more advanced functions are supplied. These include:

`gsl_rng *` **gsl_rng_cpy** (`gsl_rng *` *dest*, `const gsl_rng *` *src*) Random

This function copies the random number generator *src* into the pre-exisiting generator *dest*, making *dest* into an exact copy of *src*. It returns *dest* if successful and a null pointer if sufficient memory could not be allocated.

`gsl_rng *` **gsl_rng_clone** (`const gsl_rng *` *r*) Random

This function returns a pointer to a newly created generator which is an exact copy of the generator *r*.

`void` **gsl_rng_print_state** (`const gsl_rng *` *r*) Random

This function prints a hex-dump of the state of the generator *r* to stdout. At the moment its only use is for debugging.

4.3 Available random number generator algorithms

The functions described above make no reference to the actual algorithm used. This is deliberate so that you can switch algorithms without having to change any of your application source code. The library provides a large number of generators of different types, including simulation quality generators, generators provided for compatibility with other libraries and historical generators from the past.

4.3.1 Simulation quality generators

The following generators are recommended for use in simulation. They have extremely long periods and pass most statistical tests.

gsl_rng_mt19937 Generator

The MT19937 generator of Makoto Matsumoto and Takuji Nishimura is a variant of the twisted generalized feedback shift-register algorithm, and is known as the "Mersenne Twister" generator. It has a Mersenne prime period of $2^{19937} - 1$ (about 10^{6000}) and is equi-distributed in 623 dimensions. It has passed the DIEHARD statistical tests. It uses 624 words of state per generator and is comparable in speed to the other generators. The original generator used a default seed of 4357 and choosing *s* equal to zero in `gsl_rng_set` reproduces this.

For more information see,

> Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-dimensionally equidistributerd uniform pseudorandom number generator". *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1 (Jan. 1998), Pages 3-30

gsl_rng_ranlux Generator
gsl_rng_ranlux389 Generator

The `ranlux` generator uses the lagged-fibonacci-with-skipping algorithm of Luscher to produce "luxury random numbers". It is a 24-bit generator, originally designed for single-precision IEEE floating point numbers. The period of the generator is about 10^{171}. The generator is slow, but the algorithm has mathematically proven properties. It can provide truly decorrelated numbers at a known level of randomness. The default level of decorrelation recommended by Luscher is provided by `gsl_rng_ranlux`, while `gsl_rng_ranlux389` gives the highest level of randomness, with all 24 bits decorrelated. Both types of generator use 24 words of state per generator.

For more information see,

> M. Luscher, "A portable high-quality random number generator for lattice field theory calculations", *Computer Physics Communications*, 79 (1994) 100-110.
>
> F. James, "RANLUX: A Fortran implementation of the high-quality pseudo-random number generator of Luscher", *Computer Physics Communications*, 79 (1994) 111-114

gsl_rng_cmrg Generator

This is a combined multiple recursive generator by L'Ecuyer. Its sequence is,

$$z_n = (x_n - y_n) \bmod m_1 \qquad (4.1)$$

where the two underlying generators x_n and y_n are,

$$x_n = (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \bmod m_1 \qquad (4.2)$$

and

$$y_n = (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \bmod m_2 \tag{4.3}$$

with coefficients $a_1 = 0$, $a_2 = 63308$, $a_3 = -183326$, $b_1 = 86098$, $b_2 = 0$, $b_3 = -539608$, and moduli $m_1 = 2^{31} - 1 = 2147483647$ and $m_2 = 2145483479$.

The period of this generator is 2^{205} (about 10^{61}). It uses 6 words of state per generator. For more information see,

> P. L'Ecuyer, "Combined Multiple Recursive Random Number Generators," *Operations Research*, 44, 5 (1996), 816–822.

gsl_rng_mrg Generator

This is a fifth-order multiple recursive generator by L'Ecuyer, Blouin and Coutre. Its sequence is,

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \bmod m \tag{4.4}$$

with $a_1 = 107374182$, $a_2 = a_3 = a_4 = 0$, $a_5 = 104480$ and $m = 2^{31} - 1$.

The period of this generator is about 10^{46}. It uses 5 words of state per generator. More information can be found in the following paper,

> P. L'Ecuyer, F. Blouin, and R. Coutre, "A search for good multiple recursive random number generators", *ACM Transactions on Modeling and Computer Simulation* 3, 87-98 (1993).

gsl_rng_taus Generator

This is a maximally equidistributed combined Tausworthe generator by L'Ecuyer. The sequence is,

$$x_n = (s^1_n \oplus s^2_n \oplus s^3_n) \tag{4.5}$$

where,

$$s^1_{n+1} = (((s^1_n \& \ 4294967294) << 12) \oplus (((s^1_n << 13) \oplus s^1_n) >> 19)) \tag{4.6}$$

$$s^2_{n+1} = (((s^2_n \& \ 4294967288) << 4) \oplus (((s^2_n << 2) \oplus s^2_n) >> 25)) \tag{4.7}$$

$$s^3_{n+1} = (((s^3_n \& \ 4294967280) << 17) \oplus (((s^3_n << 3) \oplus s^3_n) >> 11)) \tag{4.8}$$

computed modulo 2^{32}. In the formulas above $\oplus$ denotes "exclusive-or". Note that the algorithm relies on the properties of 32-bit unsigned integers and has been implemented using a bitmask of `0xFFFFFFFF` to make it work on 64 bit machines.

The period of this generator is 2^{88} (about 10^{26}). It uses 3 words of state per generator. For more information see,

> P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators", *Mathematics of Computation*, 65, 213 (1996), 203–213.

4.3.2 Generators provided for compatibility

The generators in this section are provided for compatibility with existing libraries. If you are converting an existing program to use GSL then you can select these generators to check your new implementation against the original one, using the same random number generator. After verifying that your new program reproduces the original results you can then switch to a higher-quality generator.

Note that most of the generators in this section are based on single linear congruence relations, which are the least sophisticated type of generator. In particular, linear congruences have poor properties when used with a non-prime modulus, as several of these routines do (e.g. with a power of two modulus, 2^{31} or 2^{32}). This leads to periodicity in the least significant bits of each number, with only the higher bits having any randomness. Thus if you want to produce a random bitstream it is best to avoid using the least significant bits.

4.3.2.1 Unix random number generators

The standard Unix random number generators `rand`, `random` and `rand48` are provided as part of GSL. Although these generators are widely available individually often they aren't all available on the same platform. This makes it difficult to write portable code using them and so we have included the complete set of Unix generators in GSL for convenience. Note that these generators don't produce high-quality randomness and aren't suitable for work requiring accurate statistics. However, if you won't be measuring statistical quantities and just want to introduce some variation into your program then these generators are quite acceptable.

gsl_rng_rand Generator

This is the BSD `rand()` generator. Its sequence is

$$x_{n+1} = (a x_n + c) \bmod m \tag{4.9}$$

with $a = 1103515245$, $c = 12345$ and $m = 2^{31}$. The seed specifies the initial value, x_1. The period of this generator is 2^{31}, and it uses 1 word of storage per generator.

gsl_rng_random_bsd Generator
gsl_rng_random_libc5 Generator
gsl_rng_random_glibc2 Generator

These generators implement the `random()` family of functions, a set of linear feedback shift register generators originally used in BSD Unix. There are several versions of `random()` in use today: the original BSD version (e.g. on SunOS4), a libc5 version (common on existing GNU/Linux systems) and a glibc2 version. Each version uses a different seeding procedure, and thus produces different sequences.

The original BSD routines accepted a variable length buffer for the generator state, with longer buffers providing higher-quality randomness. The `random()` function implemented algorithms for buffer lengths of 8, 32, 64, 128 and 256 bytes, and the algorithm with the largest length that would fit into the user-supplied buffer was used. To support these algorithms additional generators are available with the following names,

```
gsl_rng_random8_bsd
gsl_rng_random32_bsd
gsl_rng_random64_bsd
gsl_rng_random128_bsd
gsl_rng_random256_bsd
```

where the numeric suffix indicates the buffer length. The original BSD `random` function used a 128-byte default buffer and so `gsl_rng_random_bsd` has been made equivalent to `gsl_rng_random128_bsd`. Corresponding versions of the `libc5` and `glibc2` generators are also avaliable, with the names `gsl_rng_random8_libc5`, `gsl_rng_random8_glibc2`, etc.

gsl_rng_rand48 Generator

This is the Unix `rand48` generator. Its sequence is

$$x_{n+1} = (ax_n + c) \bmod m \qquad (4.10)$$

defined on 48-bit unsigned integers with $a = 25214903917$, $c = 11$ and $m = 2^{48}$. The seed specifies the upper 32 bits of the initial value, x_1, with the lower 16 bits set to `0x330E`. The function `gsl_rng_get` returns the upper 32 bits from each term of the sequence. This does not have a direct parallel in the original `rand48` functions, but forcing the result to type `long int` reproduces the output of `mrand48`. The function `gsl_rng_uniform` uses the full 48 bits of internal state to return the double precision number x_n/m, which is equivalent to the function `drand48`. Note that some versions of the GNU C Library contained a bug in `mrand48` function which caused it to produce different results (only the lower 16-bits of the return value were set).

4.3.2.2 Numerical Recipes generators

The following generators are provided for compatibility with *Numerical Recipes*. Note that the original Numerical Recipes functions used single precision while we use double precision. This will lead to minor discrepancies, but only at the level of single-precision rounding error. If necessary you can force the returned values to single precision by storing them in a `volatile float`, which prevents the value being held in a register with double or extended precision. Apart from this difference the underlying algorithms for the integer part of the generators are the same.

gsl_rng_ran0 Generator

Numerical recipes `ran0` implements Park and Miller's MINSTD algorithm with a modified seeding procedure.

gsl_rng_ran1 Generator

Numerical recipes `ran1` implements Park and Miller's MINSTD algorithm with a 32-element Bayes-Durham shuffle box.

gsl_rng_ran2 Generator

Numerical recipes `ran2` implements a L'Ecuyer combined recursive generator with a 32-element Bayes-Durham shuffle-box.

gsl_rng_ran3 Generator

Numerical recipes `ran3` implements Knuth's portable subtractive generator.

4.3.2.3 Other random number generators

The following generator is provided for compatibility with the CRAY MATHLIB routine RANF. It produces double precision floating point numbers which should be identical to those from the original RANF.

gsl_rng_ranf Generator

This is the CRAY random number generator `RANF`. Its sequence is

$$x_{n+1} = (a x_n) \bmod m \tag{4.11}$$

defined on 48-bit unsigned integers with $a = 44485709377909$ and $m = 2^{48}$. The seed specifies the lower 32 bits of the initial value, x_1, with the lowest bit set to prevent the seed taking an even value. The upper 16 bits of x_1 are set to 0. A consequence of this procedure is that the pairs of seeds 2 and 3, 4 and 5, etc produce the same sequences.

There is a subtlety in the implementation of the seeding. The initial state is reversed through one step, by multiplying by the modular inverse of a mod m. This is done for compatibility with the original CRAY implementation.

Note that you can only seed the generator with integers up to 2^{32}, while the original CRAY implementation uses non-portable wide integers which can cover all 2^{48} states of the generator.

The function `gsl_rng_get` returns the upper 32 bits from each term of the sequence. The function `gsl_rng_uniform` uses the full 48 bits to return the double precision number x_n/m.

The period of this generator is 2^{46}.

gsl_rng_ranmar Generator

This is the RANMAR lagged-fibonacci generator of Marsaglia, Zaman and Tsang. It is a 24-bit generator, originally designed for single-precision IEEE floating point numbers. It was included in the CERNLIB high-energy physics library.

gsl_rng_r250 Generator

This is the shift-register generator of Kirkpatrick and Stoll. The sequence is

$$x_n = x_{n-103} \oplus x_{n-250} \tag{4.12}$$

where $\oplus$ denote “exclusive-or”, defined on 32-bit words. The period of this generator is about 2^{250} and it uses 250 words of state per generator.

For more information see,

> S. Kirkpatrick and E. Stoll, "A very fast shift-register sequence random number generator", *Journal of Computational Physics*, 40, 517-526 (1981)

gsl_rng_tt800 Generator

This is an earlier version of the twisted generalized feedback shift-register generator, and has been superseded by the development of MT19937. However, it is still an acceptable generator in its own right. It has a period of 2^{800} and uses 33 words of storage per generator.

For more information see,

> From: Makoto Matsumoto and Yoshiharu Kurita, "Twisted GFSR Generators II", *ACM Transactions on Modelling and Computer Simulation*, Vol. 4, No. 3, 1994, pages 254-266.

gsl_rng_vax Generator

This is the VAX generator `MTH$RANDOM`. Its sequence is,

$$x_{n+1} = (ax_n + c) \bmod m \tag{4.13}$$

with $a = 69069$, $c = 1$ and $m = 2^{32}$. The seed specifies the initial value, x_1. The period of this generator is 2^{32} and it uses 1 word of storage per generator.

gsl_rng_transputer Generator

This is the random number generator from the INMOS Transputer Development system. Its sequence is,

$$x_{n+1} = (ax_n) \bmod m \tag{4.14}$$

with $a = 1664525$ and $m = 2^{32}$. The seed specifies the initial value, x_1.

gsl_rng_randu Generator

This is the IBM `RANDU` generator. Its sequence is

$$x_{n+1} = (ax_n) \bmod m \tag{4.15}$$

with $a = 65539$ and $m = 2^{31}$. The seed specifies the initial value, x_1. The period of this generator was only 2^{29}. It has become a textbook example of a poor generator.

gsl_rng_minstd Generator

This is Park and Miller's "minimal standard" MINSTD generator, a simple linear congruence which takes care to avoid the major pitfalls of such algorithms. Its sequence is,

$$x_{n+1} = (ax_n) \bmod m \tag{4.16}$$

with $a = 16807$ and $m = 2^{31} - 1 = 2147483647$. The seed specifies the initial value, x_1. The period of this generator is about 2^{31}.

This generator is used in the IMSL Library (subroutine RNUN) and in MATLAB (the RAND function). It is also sometimes known by the acronym "GGL" (I'm not sure what that stands for).

For more information see,

> Park and Miller, "Random Number Generators: Good ones are hard to find", *Communications of the ACM*, October 1988, Volume 31, No 10, pages 1192-1201.

gsl_rng_uni Generator
gsl_rng_uni32 Generator

This is a reimplementation of the 16-bit SLATEC random number generator RUNIF. A generalisation of the generator to 32 bits is provided by `gsl_rng_uni32`. The original source code is available from NETLIB.

gsl_rng_slatec Generator

This is the SLATEC random number generator RAND. It is ancient. The original source code is available from NETLIB.

gsl_rng_zuf Generator

This is the ZUFALL lagged Fibonacci series generator of Peterson. Its sequence is,

$$t = u_{n-273} + u_{n-607} \tag{4.17}$$

$$u_n = t - floor(t) \tag{4.18}$$

The original source code is available from NETLIB. For more information see,

> W. Petersen, "Lagged Fibonacci Random Number Generators for the NEC SX-3", *International Journal of High Speed Computing* (1994).

4.4 Performance

```
 ranlux389 --
    ranlux ---
      cmrg ----
       mrg --------
   mt19937 ------------
     tt800 ----------------
      taus ------------------

      ran0 ----------
      ran1 -----------
      ran2 -----
      ran3 -----------------

      ranf --------
    rand48 ---------

    ranmar --------------
       zuf --------------
    slatec ---------------
      r250 --------------------
    random --------------------

    minstd ------------
       uni -----------------
     uni32 ------------------
       vax -----------------------
transputer -----------------------
      rand -------------------------
   random8 -------------------------
     randu ---------------------------
```

```
|---------|---------|---------|
0         1         2         3

Millions of random numbers per second
```

4.5 Random Number Distributions

Distributions of random numbers can be obtained from any of the generators using the functions described in this section. In the simplest cases a non-uniform distribution can be obtained analytically from the uniform distribution with an appropriate transformation. This method uses one call to the random number generator.

More complicated distributions are created by the *acceptance-rejection* method, which compares the desired distribution against a distribution which is similar and known analytically. This usually requires several samples from the generator.

The functions described in this section are declared in '`gsl_randist.h`'.

4.5.1 The Gaussian Distribution

`double` **gsl_ran_gaussian** (`const gsl_rng *` *r*, `double` *sigma*) Random

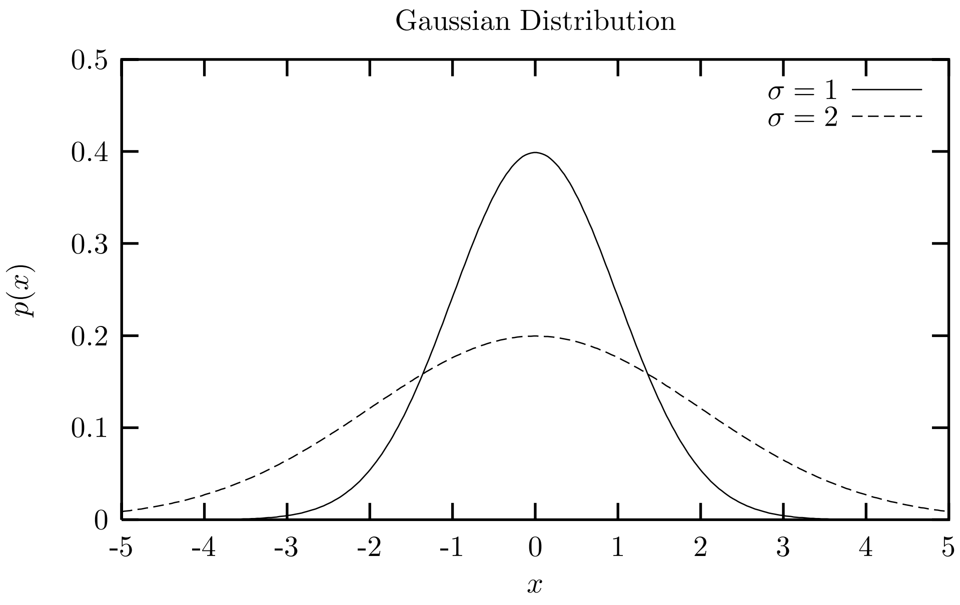
This function returns a gaussian random number, with mean zero and standard deviation *sigma*. The probability distribution for gaussian random numbers is,

$$p(z)dz = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-(z-\mu)^2/2\sigma^2)dz \tag{4.19}$$

for x in the range $-\infty$ to $+\infty$. Use the transformation $z = \mu + x$ on the numbers returned by `gsl_ran_gaussian` to obtain a gaussian distribution with mean μ.

`double` **gsl_ran_gaussian_pdf** (`double` *x*, `double` *sigma*) Function

This function computes the probability density $p(x)$ at x for a gaussian distribution with standard deviation *sigma*, using the formula given above.



Properties

Mean $= \mu$, Variance $= \sigma^2$, Skewness $= 0$, Excess $= 0$

Cumulants, $\kappa_1 = \mu$, $\kappa_2 = \sigma^2$, $\kappa_n = 0$ for $n > 2$

Characteristic function, $\phi(t) = \exp(-i\mu t - \sigma^2 t^2/2)$

Cumulative distribution function, $CDF(x) = \int_{-\infty}^{x} p(x')dx' = \frac{1}{2}(1 - \mathrm{erf}((x-m)/\sqrt{2\sigma^2}))$

Confidence limits:

$p(|x-\mu| < \sigma) = 0.683$, $p(|x-\mu| < 2\sigma) = 0.954$, $p(|x-\mu| < 3\sigma) = 0.9973$

Useful integral:

$$\int_{-\infty}^{+\infty} x^{2n} p(x)dx = 2^n \sigma^{2n+1} \Gamma(n + \frac{1}{2})/\sqrt{\pi} \tag{4.20}$$

4.5.2 The Bivariate Gaussian Distribution

`void` **gsl_ran_bivariate_gaussian** (`const gsl_rng *` *r*, `double` *sigma_x*, `double` *sigma_y*, `double` *rho*, `double *` *x*, `double *` *y*) Random

This function generates a pair of correlated gaussian variates, with mean zero, correlation coefficient *rho* and standard deviations *sigma_x* and *sigma_y* in the x and y directions. The probability distribution for bivariate gaussian random numbers is,

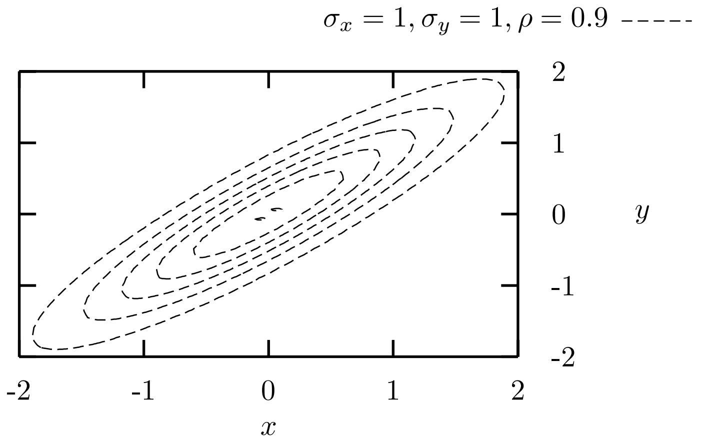
$$p(x,y)dxdy = \frac{1}{2\pi\sigma_x\sigma_y\sqrt{1-\rho^2}}\exp(-(x^2+y^2-2\rho xy)/2\sigma_x^2\sigma_y^2(1-\rho^2))dz \qquad (4.21)$$

for x, y in the range $-\infty$ to $+\infty$. The correlation coefficient *rho* should lie between 1 and -1.

`double` **gsl_ran_bivariate_gaussian_pdf** (`double` *x*, `double` *y*, `double` *sigma_x*, `double` *sigma_y*, `double` *rho*) Function

This function computes the probability density $p(x,y)$ at (*x*,*y*) for a bivariate gaussian distribution with standard deviations *sigma_x*, *sigma_y* and correlation coefficient *rho*, using the formula given above.

Bivariate Gaussian Distribution



4.5.3 The Exponential Distribution

`double` **gsl_ran_exponential** (`const gsl_rng *` *r*, `double` *mu*) Random

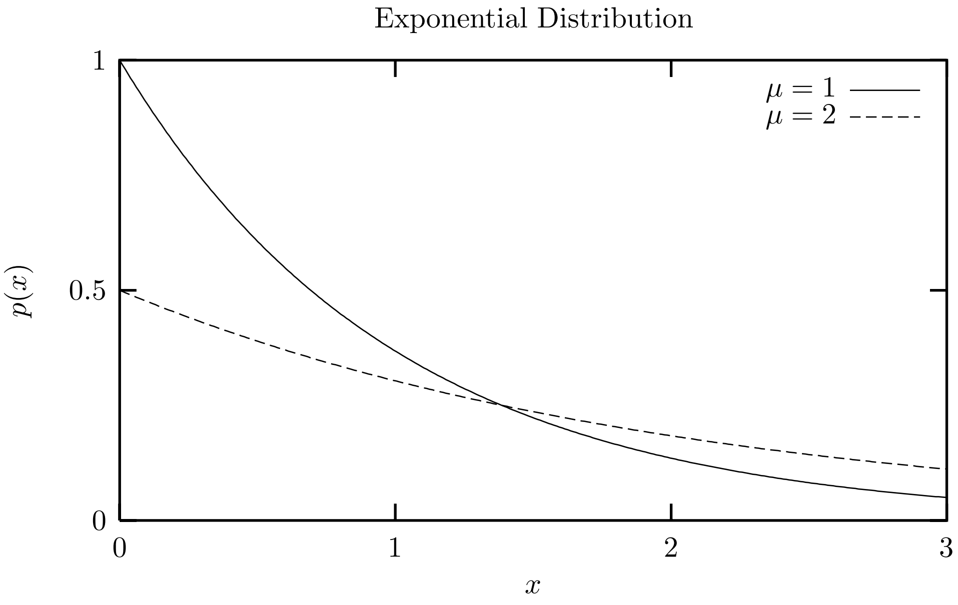
This function returns a random number from the exponential distribution with mean *mu*.

$$p(x)dx = \frac{1}{\mu}\exp(-x/\mu)dx \tag{4.22}$$

for $x \geq 0$.

`double` **gsl_ran_exponential_pdf** (`double` *x*, `double` *mu*) Function

This function computes the probability density $p(x)$ at x for an exponential distribution with mean *mu*, using the formula given above.



Properties

Mean $= \mu$, Variance $= \mu^2$, Skewness $= 2$, Excess $= 6$

Cumulants, $\kappa_1 = \mu$, $\kappa_n = \mu^n \Gamma(n)$, for $n > 1$

Characteristic function, $\phi(t) = 1/(1 - i\mu t)$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = 1 - \exp(-x/\mu)$

Confidence limits:

$p(x < \mu) = 0.632$, $p(x < 2\mu) = 0.865$, $p(x < 3\mu) = 0.950$

Useful integral:

$$\int_0^\infty x^n p(x)dx = n!\mu^{n+1} \tag{4.23}$$

4.5.4 The Laplace Distribution

`double` **gsl_ran_laplace** (`const gsl_rng *` *r*, `double` *mu*) Random

This function returns a random number from the the Laplace distribution with width *mu*. The distribution is,

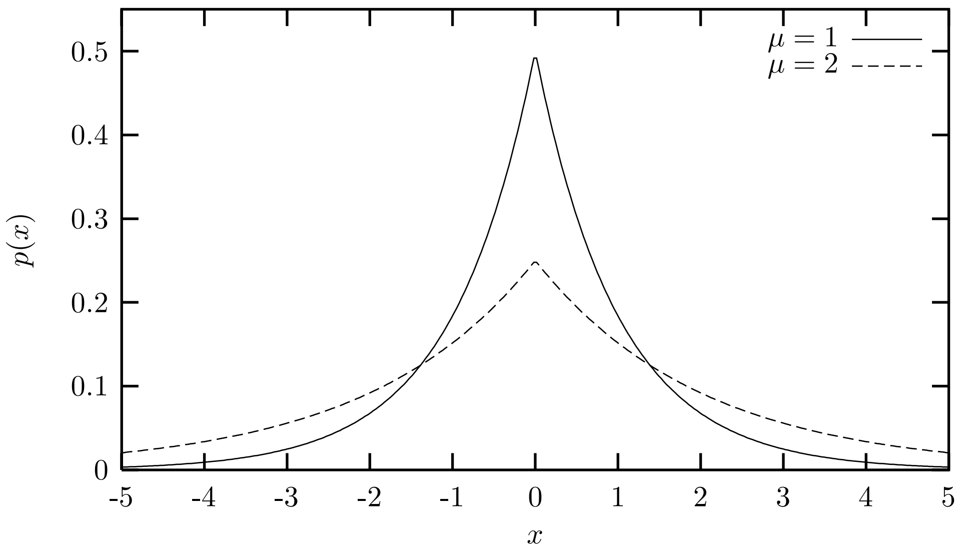
$$p(x)dx = \frac{1}{2\mu}\exp(-|x/\mu|)dx \qquad (4.24)$$

for $-\infty < x < \infty$.

`double` **gsl_ran_laplace_pdf** (`double` *x*, `double` *mu*) Function

This function computes the probability density $p(x)$ at x for a Laplace distribution with mean *mu*, using the formula given above.

Laplace Distribution (Two-sided Exponential)



Properties

Mean = 0, Variance = $2\mu^2$, Skewness = 0, Excess = 6

Cumulants, $\kappa_1 = 0$, $\kappa_2 = 2\mu^2$, $\kappa_{2n+1} = 0$, $\kappa_{2n} = (2n)!\mu^{2n}/n$

Characteristic function, $\phi(t) = 1/(1+\mu^2 t^2$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = \frac{1}{2}\exp(-|x|)$ for $x < 0$ and $1 - \frac{1}{2}\exp(-|x|)$ for $x > 0$.

Confidence limits:

Useful integral:

4.5.5 The Exponential Power Distribution

`double` **gsl_ran_exppow** (`const gsl_rng *` *r*, `double` *mu*, `double` *a*) Random

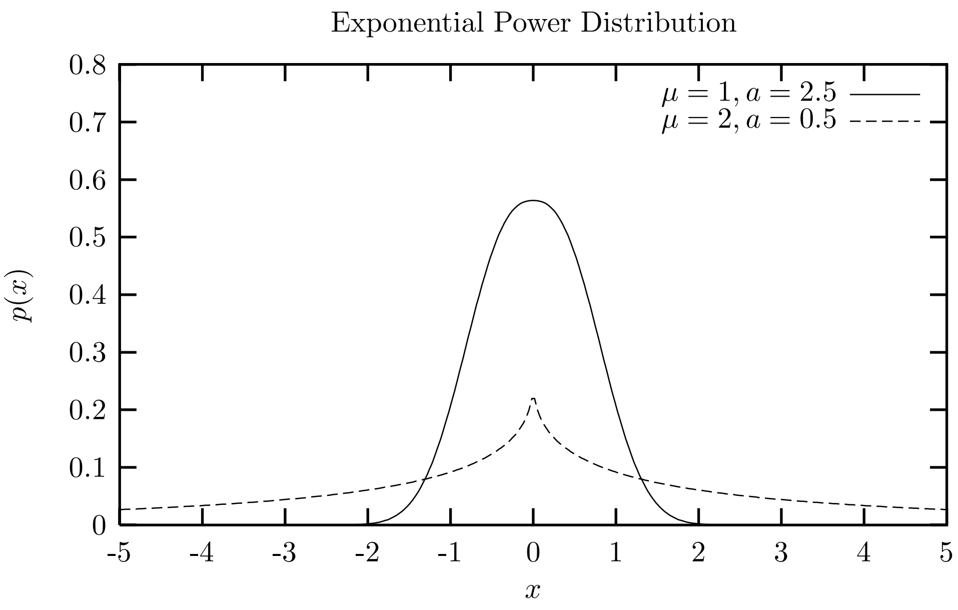
This function returns a random number from the exponential power distribution with scale parameter *mu* and exponent *a*.

$$p(x)dx = \frac{1}{2\mu\Gamma(1+1/a)} \exp(-|x/\mu|^a) dx \tag{4.25}$$

for $x \geq 0$. For $a = 1$ this reduces to the laplace distribution. For $a = 2$ it has the same form as a gaussian distribution, but with $\mu = \sqrt{2}\sigma$.

`double` **gsl_ran_exppow_pdf** (`double` *x*, `double` *mu*, `double` *a*) Function

This function computes the probability density $p(x)$ at x for an exponential power distribution with scale parameter *mu* and exponent *a*, using the formula given above.



Properties

Mean = 0, Variance = $\mu^2\Gamma(3/a)/\Gamma(1/a)$, Skewness = 0, Excess = $\Gamma(5/a)\Gamma(1/a)/\Gamma(3/a)^2$

Cumulants, $\kappa_1 = FIXME$, $\kappa_n = FIXME$, for $n > 1$

Characteristic function, $\phi(t) = FIXME$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = FIXME$

4.5.6 The Cauchy Distribution

`double` **gsl_ran_cauchy** (`const gsl_rng *` *r*, `double` *mu*) Random

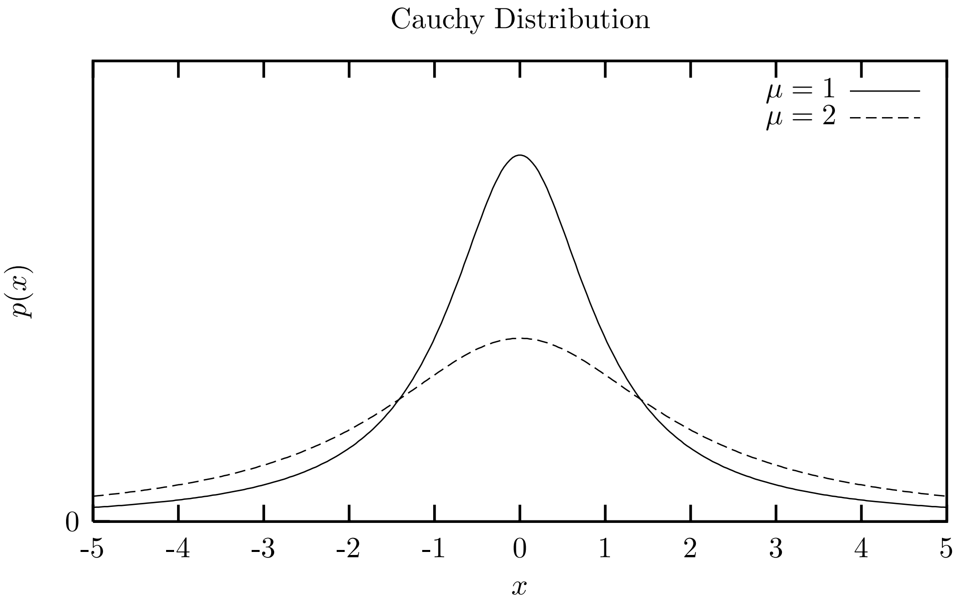
This function returns a random number from the Cauchy distribution with scale parameter *mu*. The probability distribution for Cauchy random numbers is,

$$p(x)dx = \frac{1}{\mu\pi(1 + (x/\mu)^2)}dx \tag{4.26}$$

for x in the range $-\infty$ to $+\infty$. The Cauchy distribution is also known as the Lorentz distribution.

`double` **gsl_ran_cauchy_pdf** (`double` *x*, `double` *mu*) Function

This function computes the probability density $p(x)$ at x for an Cauchy distribution with scale parameter *mu*, using the formula given above.



Properties

The Cauchy distribution decreases as $1/|x|^2$ for large x. This makes the variance infinite. Other higher moments are either infinite or undefined.

Characteristic function, $\phi(t) = \exp(-i\alpha t - \beta|t|)$

Cumulative distribution function, $CDF(x) = \int_{-\infty}^{x} p(x')dx' = \frac{1}{2} + \frac{1}{\pi}\arctan(x/\mu)$

Confidence limits:

$p(|x| < \mu) = 0.5$, $p(|x| < 2\mu) = 0.705$, $p(|x| < 3\mu) = 0.795$

4.5.7 The Rayleigh Distribution

`double` **gsl_ran_rayleigh** (`const gsl_rng *` *r*, `double` *sigma*) Random

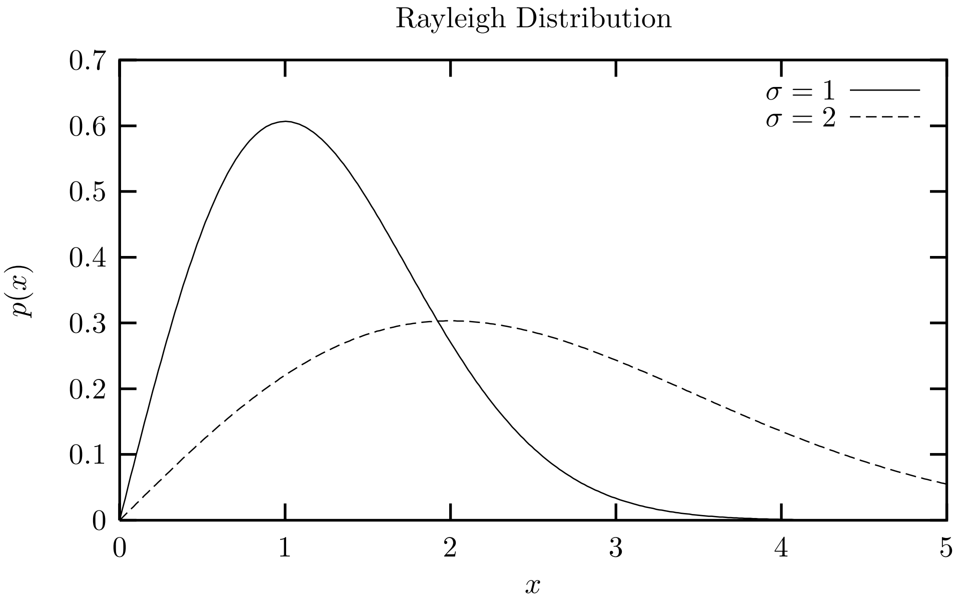
This function returns a random number from the Rayleigh distribution with scale parameter *sigma*. The probability distribution for Rayleigh random numbers is,

$$p(x)dx = \frac{x}{\sigma^2} \exp(-x^2/(2\sigma^2))dx \tag{4.27}$$

for $x > 0$.

`double` **gsl_ran_rayleigh_pdf** (`double` *x*, `double` *sigma*) Function

This function computes the probability density $p(x)$ at x for an Rayleigh distribution with scale parameter *sigma*, using the formula given above.



Properties

Cumulative distribution function, $CDF(x) = \int_{-\infty}^{x} p(x')dx' = 1 - \exp(-x^2/(2\sigma^2))$

4.5.8 The Rayleigh Tail Distribution

`double` **gsl_ran_rayleigh_tail** (`const gsl_rng *` *r*, `double` *a* `double` *sigma*) Random

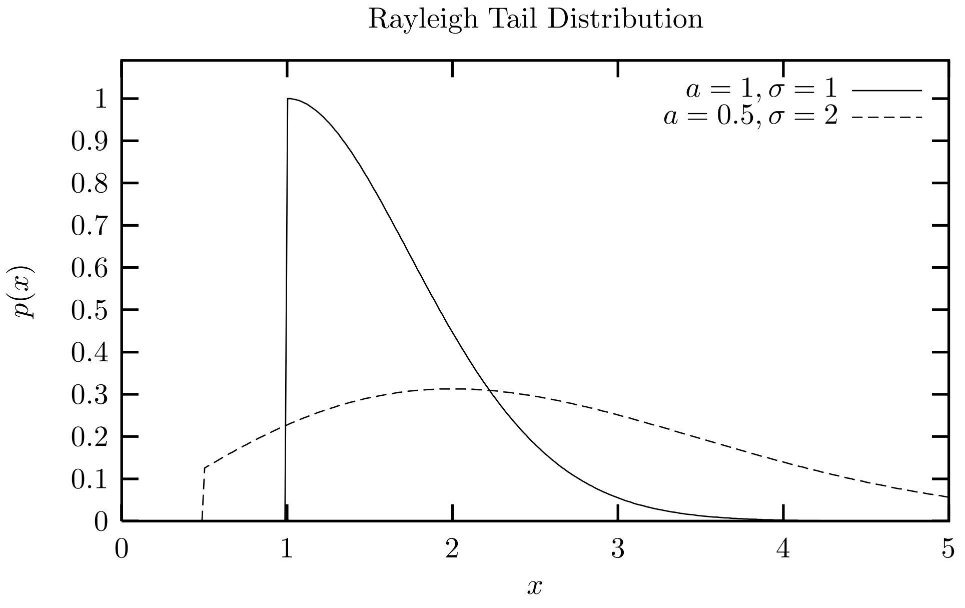
This function returns a random number from the tail of the Rayleigh distribution with scale parameter *sigma* and a lower limit of *a*. The probability distribution for Rayleigh tail random numbers is,

$$p(x)dx = \frac{x}{\sigma^2} \exp((a^2 - x^2)/(2\sigma^2)) dx \tag{4.28}$$

for $x > a$.

`double` **gsl_ran_rayleigh_tail_pdf** (`double` *x*, `double` *a*, `double` *sigma*) Function

This function computes the probability density $p(x)$ at x for an Rayleigh tail distribution with scale parameter *sigma* and lower limit *a*, using the formula given above.



Properties

Cumulative distribution function, $CDF(x) = \int_{-\infty}^{x} p(x')dx' = 1 - \exp((a^2 - x^2)/(2\sigma^2))$

4.5.9 The Symmetric Levy Distribution

`double` **gsl_ran_levy** (`const gsl_rng *` *r*, `double` *mu*, `double` *a*) Random

This function returns a random number from the symmetric Levy distribution with scale *mu* and exponent *a*. The symmetric Levy probability distribution is defined by a fourier transform,

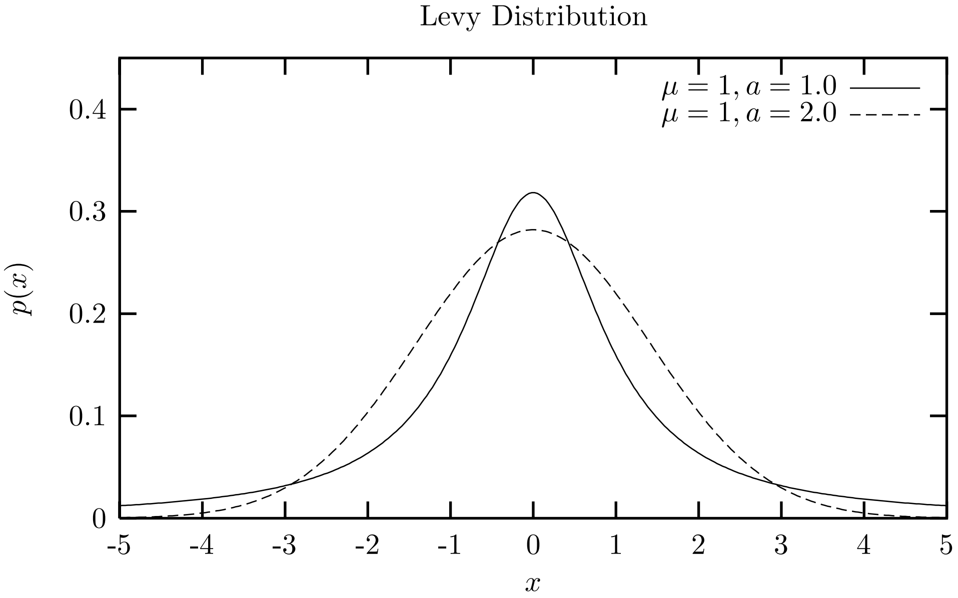
$$p(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} dt \exp(it(\mu - x) - |t|^a) \tag{4.29}$$

There is no explicit solution for the form of $p(x)$. For $a = 1$ the distribution reduces to the Cauchy distribution. For $a = 2$ it is a Gaussian distribution with $\sigma = \sqrt{2}\mu$. For $a < 1$ the tails of the distribution become extremely wide.

The algorithm only works for $0 < a \leq 2$.

`double` **gsl_ran_levy_pdf** (`double` *x*, `double` *mu*) Function

This function computes the probability density $p(x)$ at x for a symmetric Levy distribution with scale parameter *mu* and exponent *a*, using the formula given above.



Properties

Mean = 0, Variance = $FIXME$, Skewness = 0, Excess = FIXME

Cumulants, $\kappa_1 = FIXME$, $\kappa_n = FIXME$, for $n > 1$

Characteristic function, $\phi(t) = FIXME$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = FIXME$

4.5.10 The Gamma Distribution

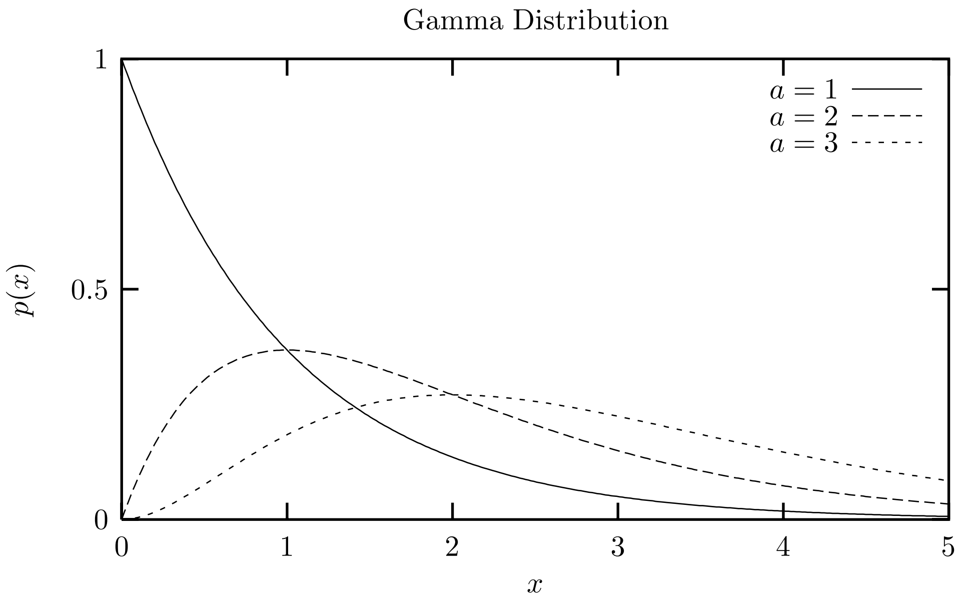
`double` **gsl_ran_gamma** (`const gsl_rng *` *r*, `double` *a*, `double` *b*) Random

This function returns a random number from the gamma distribution. The distribution function is

$$p(x)dx = \frac{1}{\Gamma(a)b^a} x^{a-1} e^{-x/b} dx \tag{4.30}$$

`double` **gsl_ran_gamma_pdf** (`double` *x*, `double` *a*, `double` *b*) Function

This function computes the probability density $p(x)$ at x for a gamma distribution with parameters a and b, using the formula given above.



Properties

Mean $= ab$, Variance $= ab^2$, Skewness $= 2/\sqrt{a}$, Excess $= 6/a$

Cumulants, $\kappa_1 = ab$, $\kappa_n = a\Gamma(n)b^n$, for $n > 1$

Characteristic function, $\phi(t) = 1 - ibt$

4.5.11 The Flat (Uniform) Distribution

`double` **gsl_ran_flat** (`const gsl_rng *` *r*, `double` *a*, `double` *b*) Random

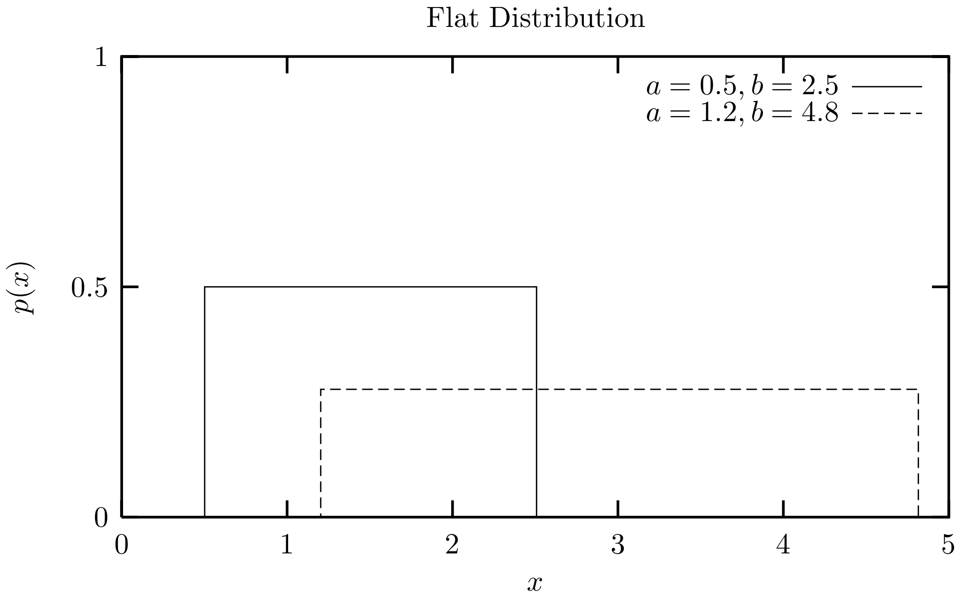
This function returns a random number from the flat (uniform) distribution from a to b.

$$p(x)dx = \frac{1}{(b-a)} dx \qquad (4.31)$$

if $a \le x < b$ and 0 otherwise.

`double` **gsl_ran_flat_pdf** (`double` *x*, `double` *a*, `double` *b*) Function

This function computes the probability density $p(x)$ at x for a uniform distribution from a to b, using the formula given above.



Properties

Mean $= (a+b)/2$, Variance $= (b-a)^2/12$, Skewness $= 0$, Excess $= -6/5$

Cumulants, $\kappa_1 = (a+b)/2$, $\kappa_{2n+1} = 0$, $\kappa_{2n} = (b-a)^{2n} B_{2n}/(2n)$,

Characteristic function, $\phi(t) = (2\sin(ht/2)/(ht))\exp(it(b+a)/2)$ where $h = (b-a)$.

4.5.12 The Lognormal Distribution

`double` **gsl_ran_lognormal** (`const gsl_rng *` *r*, `double` *zeta*, `double` *sigma*) Random

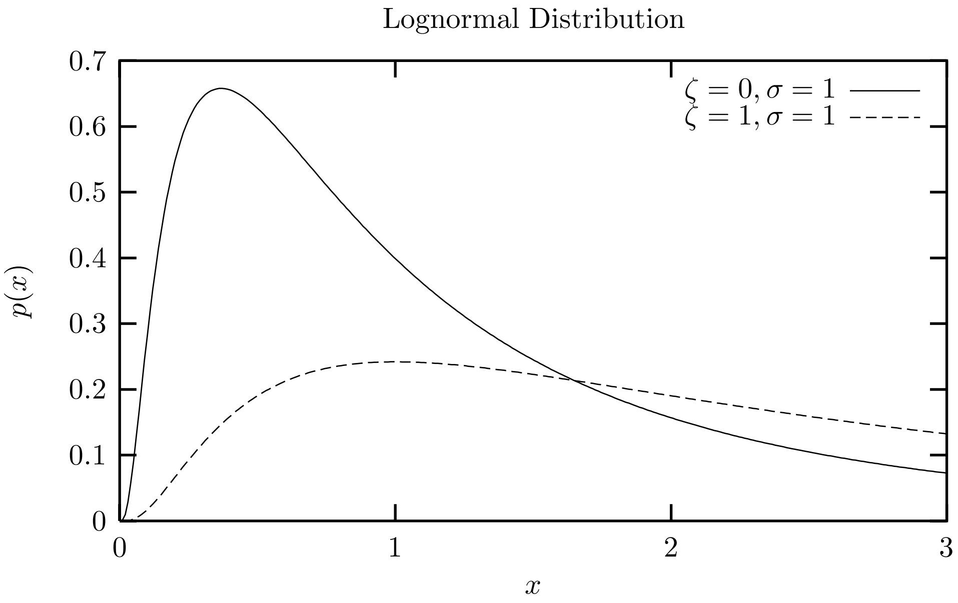
This function returns a random number from the lognormal distribution. The distribution function is

$$p(x)dx = \frac{1}{x\sqrt{2\pi\sigma^2}} \exp(-(\ln(x) - \zeta)^2/2\sigma^2)dx \qquad (4.32)$$

for $x > 0$

`double` **gsl_ran_lognormal_pdf** (`double` *x*, `double` *zeta*, `double` *sigma*) Function

This function computes the probability density $p(x)$ at x for a lognormal distribution with parameters *zeta* and *sigma*, using the formula given above.



Properties

Mean $= st$, Variance $= s(s-1)t^2$, Skewness $= (s+2)\sqrt{s-1}$, Excess $= (s-1)(s^3 + 3s^2 + 6s + 6)$

where $s = \exp(\sigma^2)$ and $t = \exp(\zeta)$.

The mode (maximum) occurs at $x = \exp(\zeta - \sigma^2)$.

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = \frac{1}{2}erfc((\zeta - \ln(x))/(\sqrt{2}\sigma))$

4.5.13 The Chi-squared Distribution

The chi-squared distribution arises in statistics If Y_i are n independent gaussian random numbers with unit variance then the sum-of-squares,

$$X_i = \sum_i Y_i^2 \tag{4.33}$$

has a chi-squared distribution with n degrees of freedom.

`double` **gsl_ran_chisq** (`const gsl_rng *` *r*, `double` *nu*) Random

This function returns a random number from the chi-squared distribution with *nu* degrees of freedom.

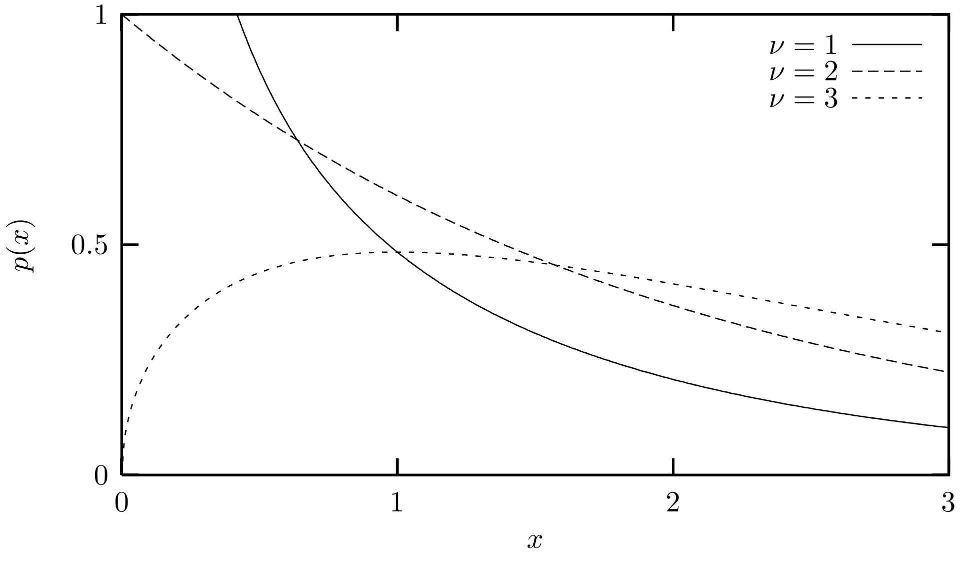
$$p(x)dx = \frac{1}{\Gamma(\nu/2)}(x/2)^{\nu/2-1}\exp(-x/2)dx \tag{4.34}$$

for $x \geq 0$.

`double` **gsl_ran_chisq_pdf** (`double` *x*, `double` *nu*) Function

This function computes the probability density $p(x)$ at x for a chi-squared distribution with *nu* degrees of freedom, using the formula given above.

Chi-squared Distribution



Properties

Mean = ν, Variance = 2ν, Skewness = $2\sqrt{2/\nu}$, Excess = $12/\nu$

Cumulants, $\kappa_1 = FIXME$, $\kappa_n = FIXME$, for $n > 1$

Characteristic function, $\phi(t) = FIXME$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = FIXME$

Confidence limits:

Useful integral:

4.5.14 The F-distribution

The F-distribution arises in statistics. If Y_1 and Y_2 are chi-squared deviates with ν_1 and ν_2 degrees of freedom then the ratio,

$$X = \frac{(Y_1/\nu_1)}{(Y_2/\nu_2)} \tag{4.35}$$

has an F-distribution $F(x; \nu_1, \nu_2)$.

`double` **gsl_ran_fdist** (`const gsl_rng *` *r*, `double` *nu1*, `double` *nu2*) Random

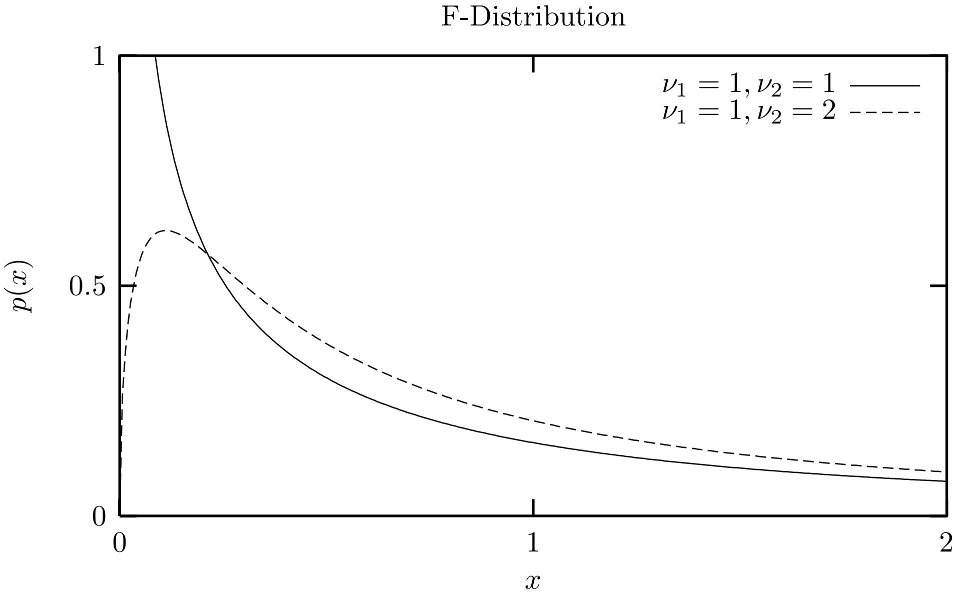
This function returns a random number from the F-distribution with degrees of freedom *nu1* and *nu2*.

$$p(x)dx = \frac{\Gamma((\nu_1 + \nu_2)/2)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \nu_1^{\nu_1/2} \nu_2^{\nu_2/2} x^{\nu_1/2-1} (\nu_2 + \nu_1 x)^{-\nu_1/2-\nu_2/2} \tag{4.36}$$

for $x \geq 0$.

`double` **gsl_ran_fdist_pdf** (`double` *x*, `double` *nu1*, `double` *nu2*) Function

This function computes the probability density $p(x)$ at x for an F-distribution with *nu1* and *nu2* degrees of freedom, using the formula given above.



Properties

Mean = $\nu_2/(\nu_2 - 2)$ (for $\nu_2 > 2$), Variance = $2\nu_2^2(\nu_1 + \nu_2 - 2)/(\nu_1(\nu_2 - 2)^2(\nu_2 - 4))$ (for $\nu_2 > 4$)

4.5.15 The t-distribution

The t-distribution arises in statistics. If Y_1 has a normal distribution and Y_2 has a chi-squared distribution with ν degrees of freedom then the ratio

$$X = \frac{Y_1}{\sqrt{Y_2/\nu}} \tag{4.37}$$

has a t-distribution $t(x;\nu)$ with ν degrees of freedom.

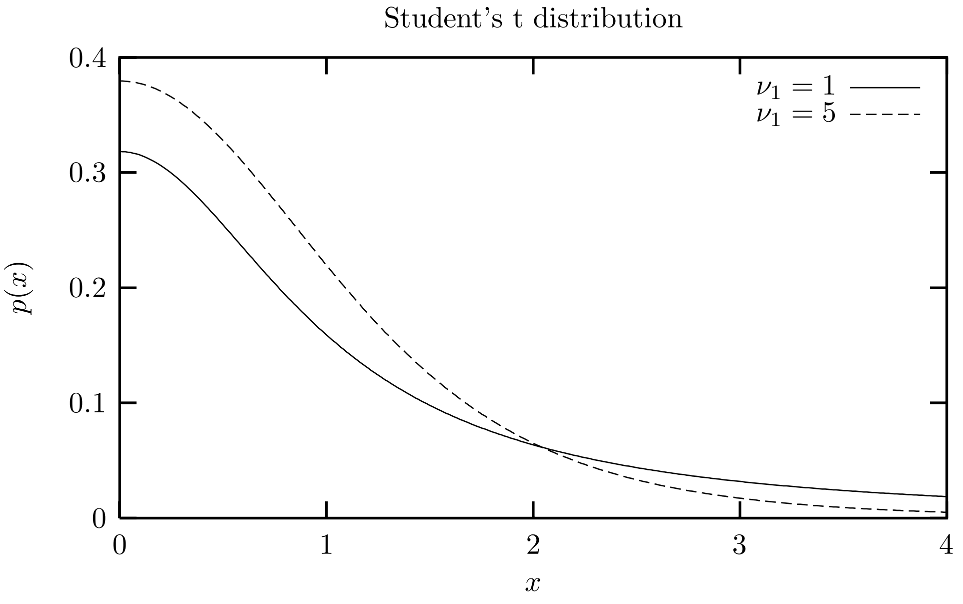
`double` **gsl_ran_tdist** (`const gsl_rng *` *r*, `double` *nu*) Random

This function returns a random number from the t-distribution. The distribution is,

$$p(x)dx = \frac{\Gamma((\nu+1)/2)}{\sqrt{\pi\nu}\Gamma(\nu/2)}(1+x^2/\nu)^{-(\nu+1)/2}dx \tag{4.38}$$

`double` **gsl_ran_tdist_pdf** (`double` *x*, `double` *nu*) Function

This function computes the probability density $p(x)$ at x for a t-distribution with *nu* degrees of freedom, using the formula given above.



4.5.16 The Beta Distribution

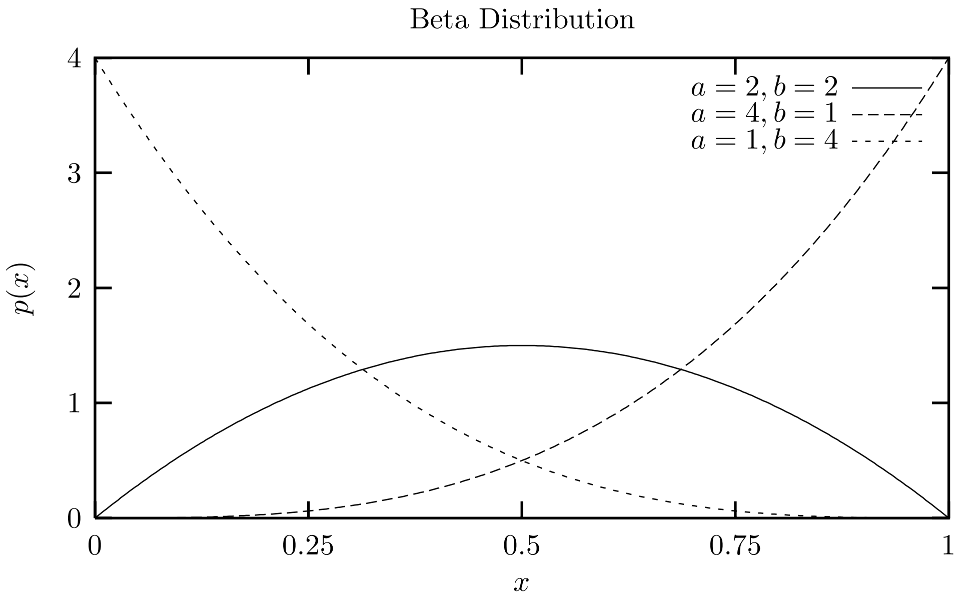
`double` **gsl_ran_beta** (`const gsl_rng *` *r*, `double` *a*, `double` *b*) Random

This function returns a random number from the beta distribution. The distribution function is

$$p(x)dx = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} x^{a-1}(1-x)^{b-1}dx \tag{4.39}$$

`double` **gsl_ran_beta_pdf** (`double` *x*, `double` *a*, `double` *b*) Function

This function computes the probability density $p(x)$ at x for a beta distribution with parameters a and b, using the formula given above.



Properties

Mean $= a/(a+b)$, Variance $= ab/((a+b)^2(a+b+1))$, Skewness $= 2(a-b)/(a+b+2)$,
Excess $= \sqrt{\frac{(a+b+1)}{ab}}\left(\frac{3(a+b+1)(2(a+b)^2+ab(a+b-6))}{ab(a+b+2)(a+b+3)} - 3\right)$
In the symmetric case $a = b$ these results simplify to,
Mean $= 1/2$, Variance $= 1/(4(2a+1))$, Skewness $= 0$, Excess $= -6\sqrt{2a+1}/(a(2a+3))$

4.5.17 The Logistic Distribution

`double` **gsl_ran_logistic** (`const gsl_rng *` *r*, `double` *mu*) Random

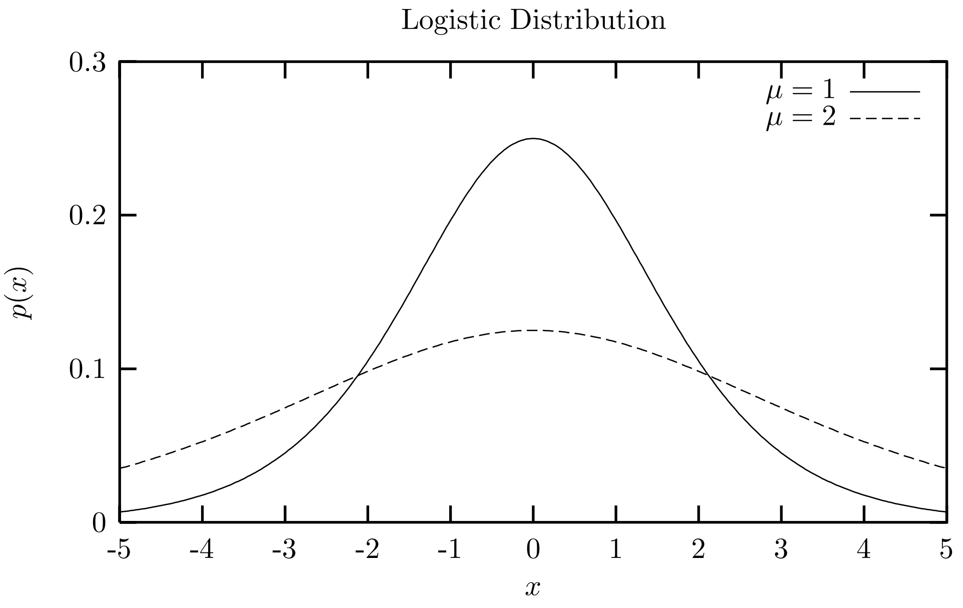
This function returns a random number from the logistic distribution. The distribution function is

$$p(x)dx = \frac{\exp(-x/\mu)}{\mu(1+\exp(-x/\mu))^2}dx \tag{4.40}$$

for $x > 0$

`double` **gsl_ran_logistic_pdf** (`double` *x*, `double` *mu*) Function

This function computes the probability density $p(x)$ at x for a logistic distribution with scale parameter *mu*, using the formula given above.



Properties

Mean = 0, Variance = $\pi^2\mu^2/3$, Skewness = 0, Excess = 6/5

Cumulative distribution function, $CDF(x) = \int_{-\infty}^{x} p(x')dx' = 1/(1+\exp(-x/\mu))$

Confidence limits:

Useful integral:

4.5.18 The Pareto Distribution

`double` **gsl_ran_pareto** (`const gsl_rng *` *r*, `double` *a*, `double` *b*) Random

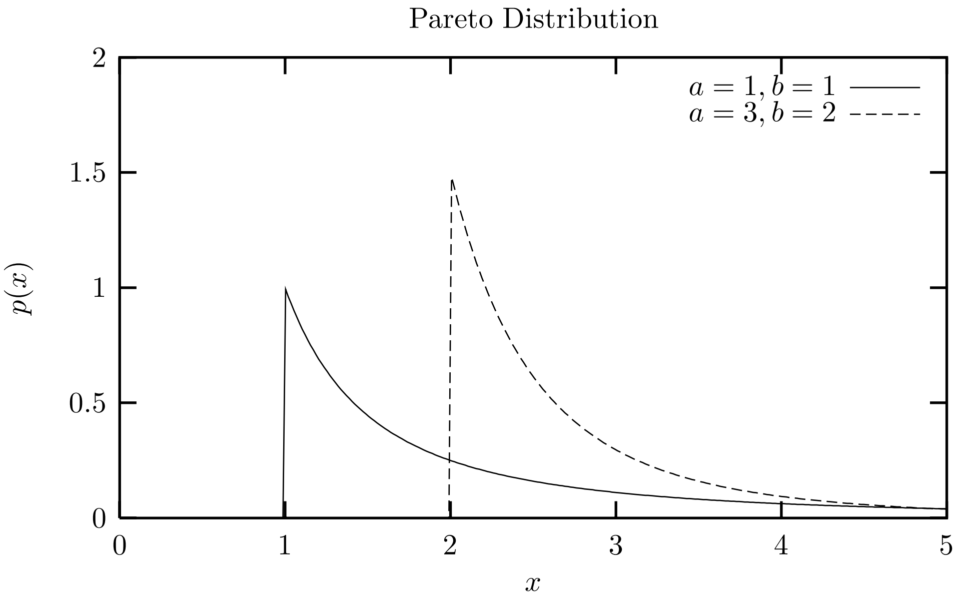
This function returns a random number from the Pareto distribution of order a. The distribution function is,

$$p(x)dx = ab^a/x^{a+1}dx \tag{4.41}$$

for $x \geq b$

`double` **gsl_ran_pareto_pdf** (`double` *x*, `double` *a*, `double` *b*) Function

This function computes the probability density $p(x)$ at x for a Pareto distribution with exponent a and scale b, using the formula given above.



Properties

Mean $= ab/(a-1)$ (for $a > 1$), Variance $= ab^2/((a-2)(a-1)^2)$ (for $a > 2$, Skewness $= 2(a+1)\sqrt{(a-2)/a}/(a-3)$ (for $a > 3$), Excess $= 6(a(a^2+a-6)-2)/(a(a-3)(a-4))$ (for $a > 4$), Mode $= b$

Cumulants, $\kappa_1 = FIXME$, $\kappa_n = FIXME$, for $n > 1$

Characteristic function, $\phi(t) = FIXME$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = 1 - (b/x)^a$

Confidence limits:

Useful integral:

4.5.19 The Spherical Distribution (2D & 3D)

The spherical distributions generate random vectors, located on a spherical surface. They can be used as random directions, for example in the steps of a random walk.

`void` **gsl_ran_dir_2d** (`const gsl_rng *` *r*, `double *`*x*, `double *`*y*) Random

This function returns a random direction vector $v = (x,y)$ in two dimenions. The vector is normalized such that $|v|^2 = x^2 + y^2 = 1$.

`void` **gsl_ran_dir_3d** (`const gsl_rng *` *r*, `double *`*x*, `double *`*y*, `double *` *z*) Random

This function returns a random direction vector $v = (x,y,z)$ in three dimenions. The vector is normalized such that $|v|^2 = x^2 + y^2 + z^2 = 1$.

The follwing program generates a random walk in two dimensions.

```
#include <stdio.h>
#include <gsl_rng.h>
#include <gsl_randist.h>

main ()
{
  gsl_rng * r = gsl_rng_alloc (gsl_rng_env_setup()) ;
  int i ;
  double x = 0, y = 0, dx, dy;

  printf("%g %g\n", x, y) ;

  for (i = 0; i < 10; i++)
    {
      gsl_ran_dir_2d (r, &dx, &dy) ;
      x += dx ; y += dy;
      printf("%g %g\n", x, y) ;
    }
}
```

Example output from the program, three 10-step random walks from the origin.

4.5.20 The Weibull Distribution

`double` **gsl_ran_weibull** (`const gsl_rng *` *r*, `double` *mu*, `double` *a*) Random

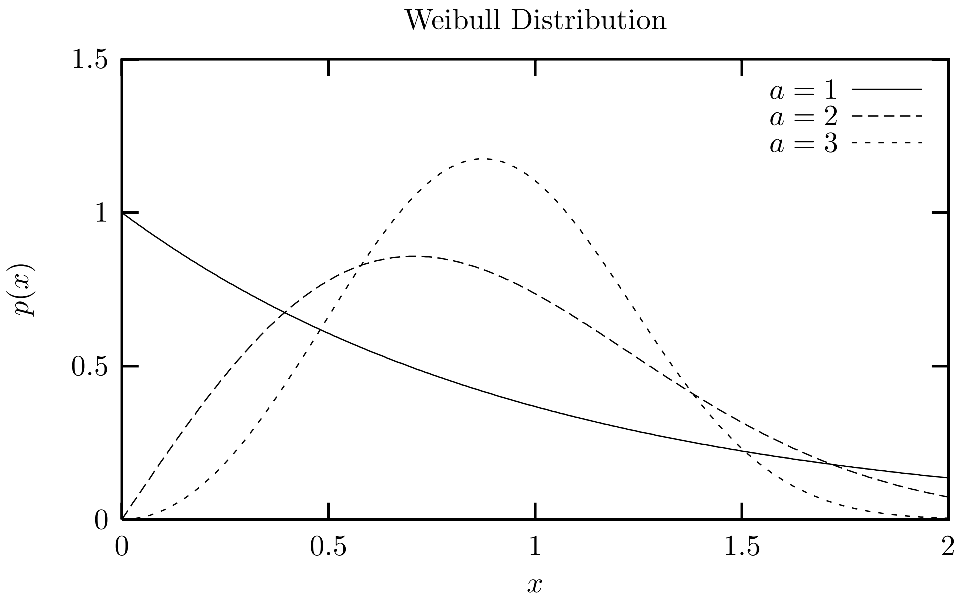
This function returns a random number from the Weibull distribution. The distribution is,

$$p(x)dx = \frac{a}{\mu^a} x^{a-1} \exp(-(x/\mu)^a) dx \tag{4.42}$$

for $-\infty < x < \infty$.

`double` **gsl_ran_weibull_pdf** (`double` *x*, `double` *mu*, `double` *a*) Function

This function computes the probability density $p(x)$ at x for a Weibull distribution with scale *mu* and exponent *a*, using the formula given above.



Properties

Mean = $\Gamma(1 + 1/a)\mu$, Variance = $(\Gamma(1 + 2/a) - \Gamma(1 + 1/a))\mu^2$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = 1 - \exp(-(x/\mu)^a)$

4.5.21 The Gumbel Distribution

`double` **gsl_ran_gumbel1** (`const gsl_rng *` *r*, `double` *a*, `double` *b*) Random

`double` **gsl_ran_gumbel2** (`const gsl_rng *` *r*, `double` *a*, `double` *b*) Random

These functions return random numbers from the Type-1 and Type-2 Gumbel distributions. The Type-1 Gumbel distribution is,

$$p(x)dx = ab \exp(-(b \exp(-ax) + ax))dx \qquad (4.43)$$

for $-\infty < x < \infty$. The Type-2 Gumbel distribution is,

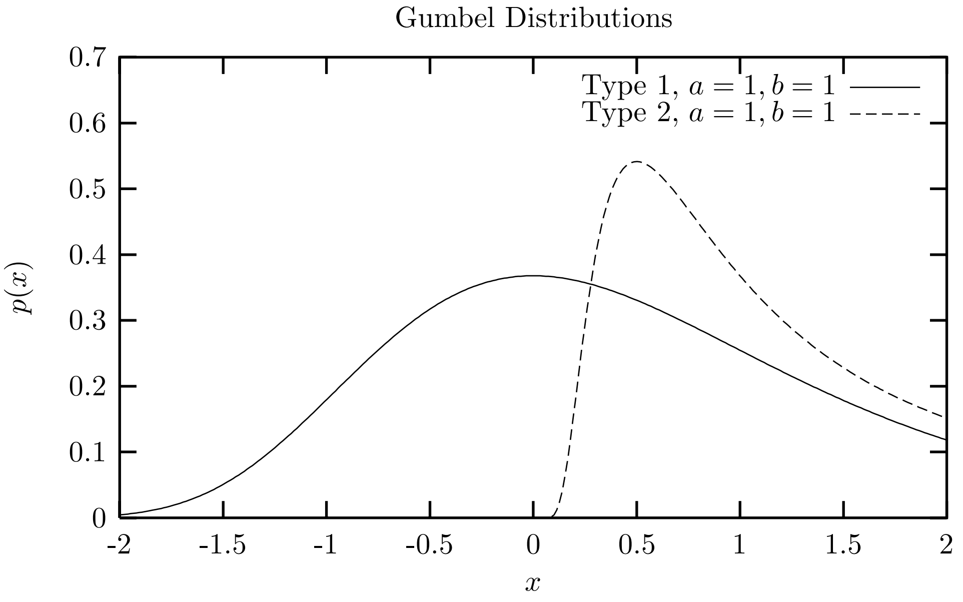
$$p(x)dx = abx^{-a-1} \exp(-bx^{-a})dx \qquad (4.44)$$

for $0 < x < \infty$.

`double` **gsl_ran_gumbel1_pdf** (`double` *x*, `double` *a*, `double` *b*) Function

`double` **gsl_ran_gumbel2_pdf** (`double` *x*, `double` *a*, `double` *b*) Function

These function computes the probability density $p(x)$ at x for a Type-1 or Type-2 Gumbel distribution with parameters a and b, using the formulas given above.



Properties

Mean = $\Gamma(1 + 1/a)\mu$, Variance = $(\Gamma(1 + 2/a) - \Gamma(1 + 1/a))\mu^2$

Cumulative distribution function, $CDF(x) = \int_0^x p(x')dx' = 1 - \exp(-(x/\mu)^a)$

4.6 Discrete distributions

4.6.1 The Poisson Distribution

`unsigned int` **gsl_ran_poisson** (`const gsl_rng *` *r*, `double` *mu*) Random

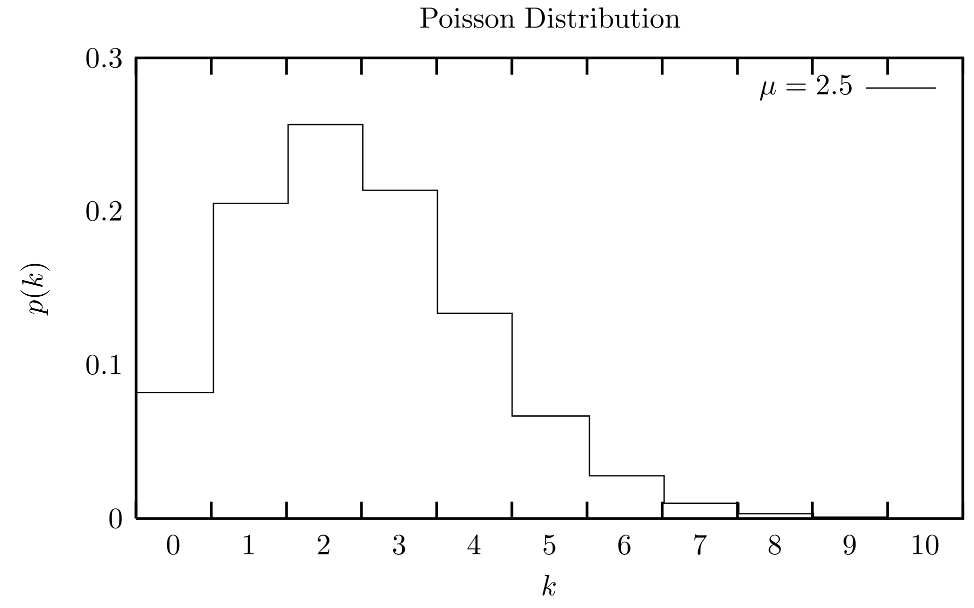
This function returns a random integer from the Poisson distribution with mean *mu*. The probability distribution for Poisson random numbers is,

$$p(k) = \frac{\mu^k}{k!}\exp(-\mu) \tag{4.45}$$

for $k \geq 0$.

`double` **gsl_ran_poisson_pdf** (`unsigned int` *k*, `double` *mu*) Function

This function computes the probability $p(k)$ of obtaining k from a Poisson distribution with mean *mu*, using the formula given above.



Properties

Mean = μ, Variance = μ^2, Skewness = $1/\sqrt{\mu}$, Excess = $1/\mu$

Cumulants, $\kappa_n = m$ for all n

Characteristic function, $\phi(t) = \exp(m(e^{it} - 1))$

4.6.2 The Bernoulli Distribution

`unsigned int` **gsl_ran_bernoulli** (`const gsl_rng *` *r*, `double` *p*) Random

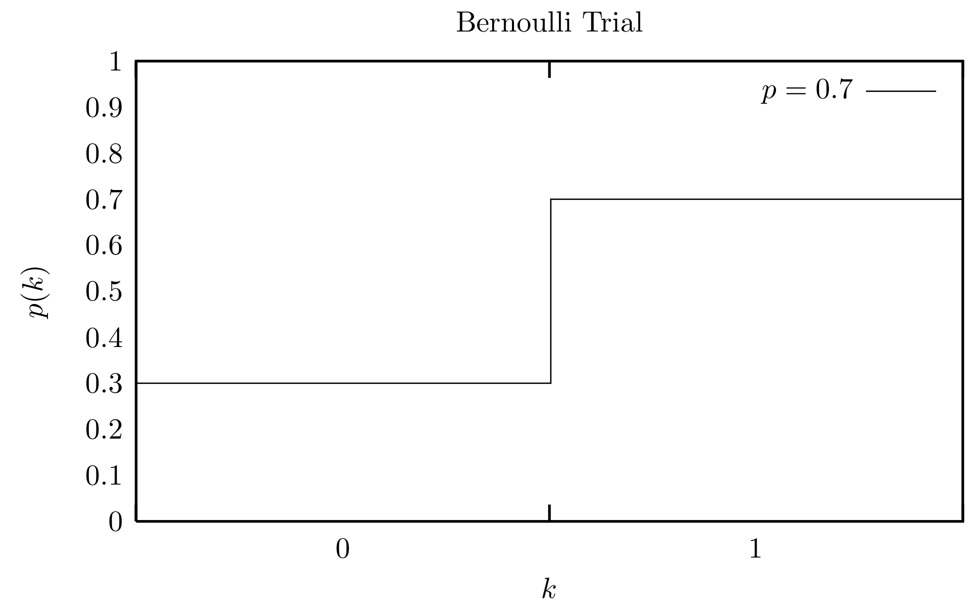
This function returns the result either 0 or 1, the result of a Bernoulli trial with probability p. The probability distribution for a Bernoulli trial is,

$$p(0) = 1 - p \tag{4.46}$$

$$p(1) = p \tag{4.47}$$

`double` **gsl_ran_bernoulli_pdf** (`unsigned int` *k*, `double` *p*) Function

This function computes the probability $p(k)$ of obtaining k from a Bernoulli distribution with probability parameter p, using the formula given above.



Properties

Mean $= p$, Variance $= p(1-p)$, Skewness $= (1-2p)/\sqrt{p(1-p)}$, Excess $= 1 - 6p(1-p))/(p(1-p))$

Cumulants, $\kappa_n = FIXME$ for all n

Characteristic function, $\phi(t) = FIXME$

4.6.3 The Binomial Distribution

`unsigned int` **gsl_ran_binomial** (`const gsl_rng *` *r*, `double` *p*, `unsigned int` *n*) Random

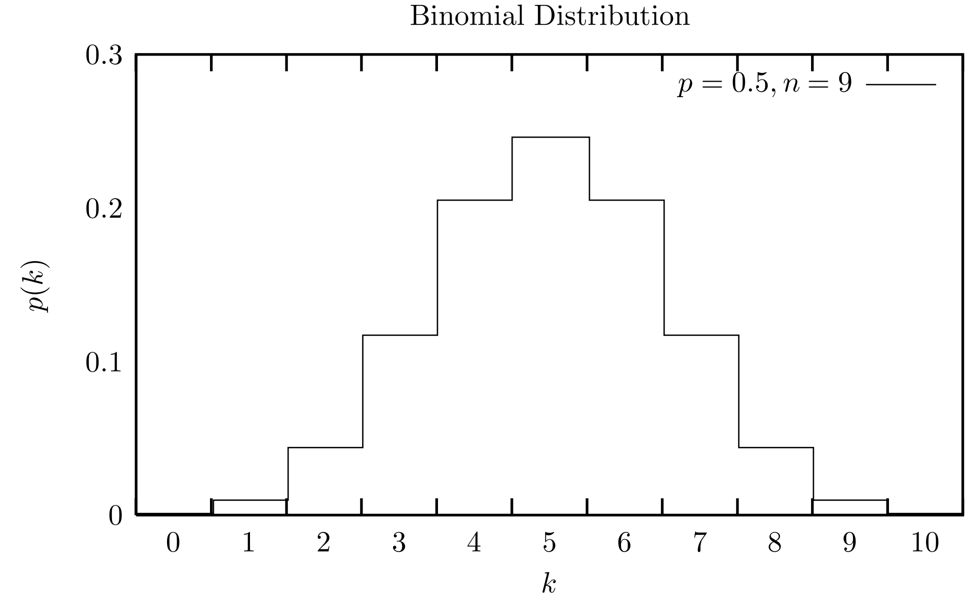
This function returns a random integer from the binomial distribution, the number of successes in n independent trials with probability p. The probability distribution for binomial random numbers is,

$$p(k) = \frac{n!}{k!(n-k)!} p^k (1-p)^{n-k} \tag{4.48}$$

for $0 \le k \le n$.

`double` **gsl_ran_binomial_pdf** (`unsigned int` *k*, `double` *p*, `unsigned int` *n*) Function

This function computes the probability $p(k)$ of obtaining k from a binomial distribution with parameters p and n, using the formula given above.



Properties

Mean $= np$, Variance $= np(1-p)$, Skewness $= (1-2p)/\sqrt{np(1-p}$, Excess $= (1-6p(1-p))/(np(1-p))$

Cumulants, $\kappa_1 = np$, $\kappa_{r+1} = p(1-p)(d\kappa_r/dp)$, for $n > 1$

Characteristic function, $\phi(t) = (1 - p + p\exp(it))^n$

4.6.4 The Negative Binomial Distribution

`unsigned int` **gsl_ran_negative_binomial** (`const gsl_rng *` *r*, `double` *p*, `double` *n*) Random

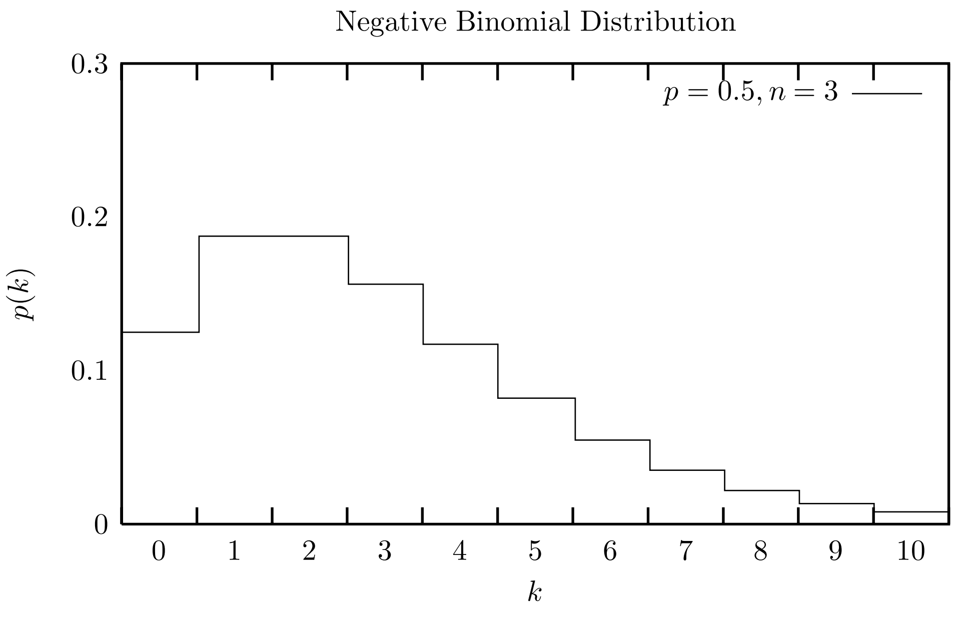
This function returns a random integer from the negative binomial distribution, the number of failures occurring before n successes in independent trials with probability p of success. The probability distribution for negative binomial random numbers is,

$$p(k) = \frac{\Gamma(n+k)}{\Gamma(k+1)\Gamma(n)} p^n (1-p)^k \tag{4.49}$$

Note that k is not required to be an integer.

`double` **gsl_ran_nbinomial_pdf** (`unsigned int` *k*, `double` *p*, `double` *n*) Function

This function computes the probability $p(k)$ of obtaining k from a negative binomial distribution with parameters p and n, using the formula given above.



`unsigned int` **gsl_ran_pascal** (`const gsl_rng *` *r*, `double` *p*, `unsigned int` *k*) Random

This function returns a random integer from the Pascal distribution. The Pascal distribution is simply a negative binomial distribution with an integer value of n.

$$p(k) = \frac{(n+k-1)!}{k!(n-1)!} p^n (1-p)^k \tag{4.50}$$

`double` **gsl_ran_pascal_pdf** (`unsigned int` *k*, `double` *p*, `unsigned int` *n*) Function

This function computes the probability $p(k)$ of obtaining k from a Pascal distribution with parameters p and n, using the formula given above.

Properties

Mean $= n(1-p)/p$, Variance $= n(1-p)/p^2$, Skewness $= (2-p)/\sqrt{n(1-p)}$, Excess $=$ $(1/n)(6+\frac{p^2}{1-p})$

4.6.5 The Geometric Distribution

`unsigned int` **gsl_ran_geometric** (`const gsl_rng *` *r*, `double` *p*) Random

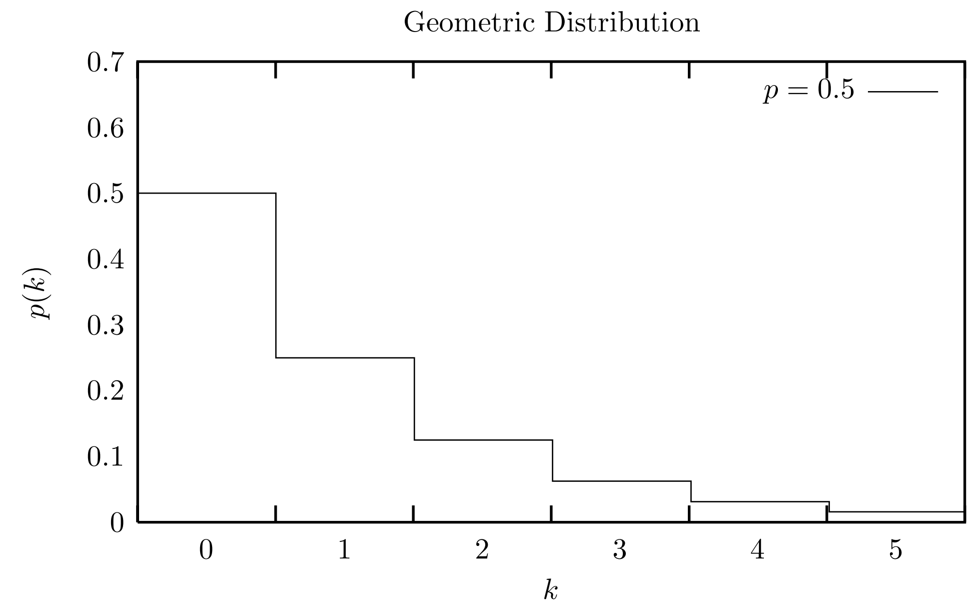
This function returns a random integer from the geometric distribution, the number of independent trials with probability p until the first success. The probability distribution for geometric random numbers is,

$$p(k) = p(1-p)^k \tag{4.51}$$

for $k \geq 1$.

`double` **gsl_ran_geometric_pdf** (`unsigned int` *k*, `double` *p*) Function

This function computes the probability $p(k)$ of obtaining k from a geometric distribution with probability parameter p, using the formula given above.



Properties

Mean $= (1-p)/p$, Variance $= (1-p)/p^2$, Skewness $= (2-p)/\sqrt{1-p}$, Excess $= 6 + (p^2/(1-p))$

Cumulants, $\kappa_1 = (1-p)/p$, $\kappa_{r+1} = -(1-p)(d\kappa_r/dp)$, for $r > 1$

Characteristic function, $\phi(t) = p/(1-(1-p)\exp(it)$

4.6.6 The Hypergeometric Distribution

`unsigned int` **gsl_ran_hypergeometric** (`const gsl_rng *` *r*, `unsigned int` *n1*, `unsigned int` *n2*, `unsigned int` *t*) Random

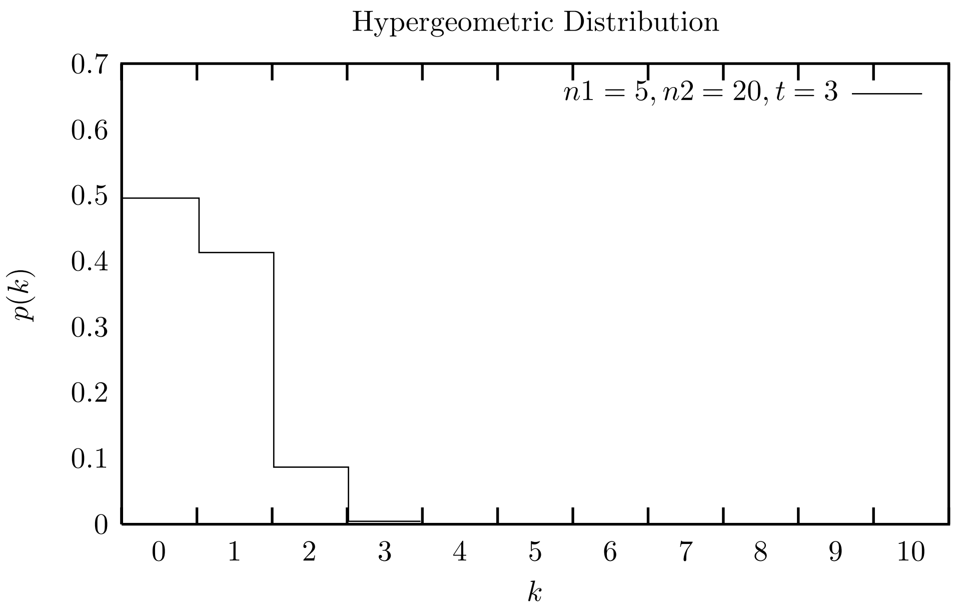
This function returns a random integer from the hypergeometric distribution. The probability distribution for hypergeometric random numbers is,

$$p(k) = C(n1, k)C(n2, t - k)/C(n1 + n2, k) \tag{4.52}$$

where $C(a, b) = a!/(b!(a-b)!)$. The domain of k is $max(0, t-n_2), ..., max(t, n_1)$.

`double` **gsl_ran_hypergeometric_pdf** (`unsigned int` *k*, `unsigned int` *n1*, `unsigned int` *n2*, `unsigned int` *t*) Function

This function computes the probability $p(k)$ of obtaining k from a hypergeometric distribution with parameters *n1*, *n2*, *n3*, using the formula given above.



Properties

Mean = tn_1/N, Variance = $tn_1n_2(N-t)/(N^2(N-1))$, Skewness = $(n_2-n_1)(N-2t)\sqrt{(N-1)/(tn_1n_2(N-t))}$, where $N = n_1 + n_2$.

4.6.7 The Logarithmic Distribution

`unsigned int` **gsl_ran_logarithmic** (`const gsl_rng *` *r*, `double` *p*) Random

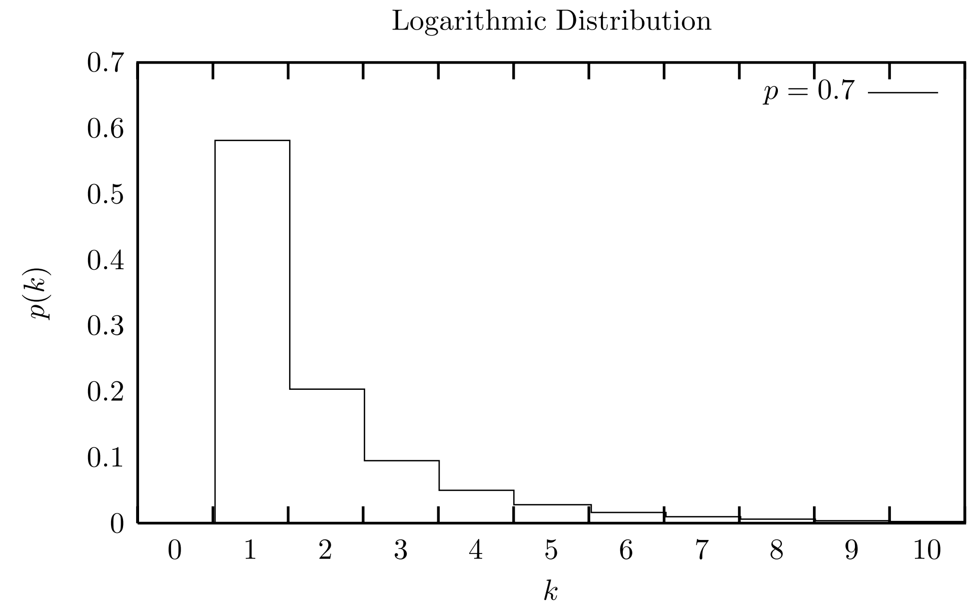
This function returns a random integer from the logarithmic distribution. The probability distribution for logarithmic random numbers is,

$$p(k) = \frac{-1}{\log(1-p)} \left(\frac{p^k}{k} \right) \tag{4.53}$$

for $n \geq 1$.

`double` **gsl_ran_logarithmic_pdf** (`unsigned int` *k*, `double` *p*) Function

This function computes the probability $p(k)$ of obtaining k from a logarithmic distribution with probability parameter p, using the formula given above.



Properties

Mean $= ap/(1-p)$, Variance $= ap(1-ap)/(1-p)^2$ where $a = -1/\log(1-p)$

4.7 Shuffling and Sampling

`void` **gsl_ran_shuffle** (`const gsl_rng *` *r*, `void *` *base*, `size_t` *n*, `size_t` *size*) Random

This function randomly shuffles the order of *n* objects, each of size *size*, stored in the array *base*[0..*n*-1].

The following code shows how to shuffle the numbers from 0 to 51,

```
int a[52];

for (i = 0; i < 52; i++)
  {
    a[i] = i ;
  }

gsl_ran_shuffle (r, a, 52, sizeof (int));
```

`void *` **gsl_ran_choose** (`const gsl_rng *` *r*, `void *` *dest*, `size_t` *k*, `void *` *src*, `size_t` *n*, `size_t` *size*) Random

This function fills the array *dest[k]* with *k* objects taken randomly from the *n* elements of the array *src[n]*. The objects are each of size *size*. The objects are sampled without replacement, thus each object can only appear once in *dest[k]*. It is required that *k* be less than or equal to `n`. The objects in *dest* will be in the same relative order as those in *src*. You will need to call `gsl_ran_shuffle(r, dest, n, size)` if you want to randomize the order.

The following code shows how to select a random sample of three unique numbers from the set 0 to 99,

```
double a[3], b[100];

for (i = 0; i < 100; i++)
  {
    b[i] = (double) i ;
  }

gsl_ran_choose (r, a, 3, b, 100, sizeof (double));
```

4.8 Random Number References and Further Reading

`ftp://ftp.iro.umontreal.ca/pub/simulation/lecuyer/papers/handsim.ps`.

Donald E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms* (Vol 2, 3rd Ed, 1997), Addison-Wesley, ISBN 0201896842.

Luc Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, ISBN 0-387-96305-7.

See the pLab home page (http://random.mat.sbg.ac.at/) for a lot of information on the state-of-the-art in random number generation, and for numerous links to various "random" WWW sites.

For physicists the Particle Data Group provides a useful short review of techniques for generating distributions of random numbers in the "Monte Carlo" section of its Annual Review of Particle Physics.

Review of Particle Properties R.M. Barnett et al., Physical Review D54, 1 (1996) `http://pdg.lbl.gov/`.

The Review of Particle Physics is available online in postscript and pdf format.

The source code for the DIEHARD random number generator tests is also available online.

DIEHARD source code G. Marsaglia `http://stat.fsu.edu/pub/diehard/`

4.9 Random Number Acknowledgements

Thanks to Makoto Matsumoto, Takuji Nishimura and Yoshiharu Kurita for making the source code to their generators (MT19937, MM&TN; TT800, MM&YK) available under the GNU General Public License.

5 Statistics

This chapter describes the statistical functions in the library. The basic statistical functions include routines to compute the mean, variance and standard deviation. More advanced functions allow you to calculate absolute deviations, skewness, and kurtosis as well as the median and arbitrary percentiles. Statistical tests for comparing different datasets, such as the t-test, are also included.

All the functions are available in versions for floating-point datasets and integer datasets. The versions for floating-point data have the prefix `gsl_stats` and the versions for integer data have the prefix `gsl_stats_int`. All the algorithms use a recurrence relation to compute average quantities in a stable way, without large intermediate values that might overflow.

5.1 Statistical Concepts

A probability density function tells us the frequency with which a random variable takes a particular value or lies in a given range,

$$\text{probability}(a < x < b) = \int_a^b dx\, p(x) \tag{5.1}$$

The ensemble of all possible values generated with the appropriate frequency by the probability density function is called the *population*. We can define parameters describing the probability density function, such as the population mean μ and population variance σ^2,

$$\mu = \int_{-\infty}^{+\infty} dx\, x p(x) \tag{5.2}$$

$$\sigma^2 = \int_{-\infty}^{+\infty} dx\, (x-\mu)^2 p(x) \tag{5.3}$$

The parameters cannot be observed directly. In statistical calculations we try to estimate these parameters from a finite set of observations.

An observed datapoint drawn from a population is called a *sample*. A *statistic* is a function of a set of samples, such as their mean, median or maximum value. An *estimator* is a statistic that we use to extract the underlying parameters of the probability density function.

For example the arithmetic mean of an observed dataset, also known as the sample mean, can be shown to be a good estimator for the population mean μ. We denote estimators with a "hat", so the sample mean is written $\hat{\mu}$.

There can be many possible estimators for a parameter, but some will be better than others. Using probability theory and the probability density function we can calculate the behavior of any estimator. An estimator is called *unbiased* if its expectation value in repeated observations equals the underlying parameter of the probability density function. An unbiased estimator is desirable. There are also other measures of an estimator properties, such as *precision* and *efficiency* (consult the references for details).

5.2 Mean, Standard Deviation and Variance

`double` **gsl_stats_mean** (`const double` *data*[], `size_t` *n*) Statistics
`double` **gsl_stats_int_mean** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_mean` returns the arithmetic mean of *data*, a double-precision array of length *n*. The function `gsl_stats_int_mean` returns the sample mean of an integer array.

The sample mean is an unbiased estimator of the population mean and is denoted by $\hat{\mu}$.

$$\hat{\mu} = \frac{1}{N} \sum x_i \tag{5.4}$$

where x_i are the elements of the array *data*.

For a gaussian distribution the variance of $\hat{\mu}$ is σ^2/N, so the one standard-deviation error on the mean is $\sigma/\sqrt{N}$.

If the underlying distribution has long tails then the mean may be subject to large fluctations.

`double` **gsl_stats_est_variance** (`const double` *data*[], `size_t` *n*) Statistics
`double` **gsl_stats_int_est_variance** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_est_variance` returns the estimated variance of *data*, a double-precision array of length *n*. The function `gsl_stats_int_est_variance` returns the estimated variance of an integer array.

This is an unbiased estimator of σ^2, and is defined by

$$\hat{\sigma}^2 = \frac{1}{(N-1)} \sum (x_i - \hat{\mu})^2 \tag{5.5}$$

where x_i are the elements of the array *data*.

Note that the normalisation factor of $1/(N-1)$ is chosen to make $\hat{\sigma}^2$ an unbiased estimator. (A factor of $1/N$ produces variances which, on the average, are too small since correlations between x_i and $\hat{\mu}$ tend to reduce the sample variance).

For a gaussian distribution the variance of $\hat{\sigma}^2$ is $2\sigma^4/N$. The one standard-deviation error on the variance is $\sqrt{2}\sigma^2/\sqrt{N}$.

If the underlying distribution has long tails then the variance may be subject to large fluctations, or may not converge.

Both these functions compute the mean via a call to `gsl_stats_mean` or `gsl_stats_int_mean`. If you have already computed the mean then you can pass it directly to one of the following two functions, `gsl_stats_est_variance_with_mean` and `gsl_stats_int_est_variance_with_mean`.

`double` **gsl_stats_est_variance_with_mean** (`const double` *data*[], `size_t` *n*, `double` *mean*) — Statistics

`double` **gsl_stats_int_est_variance_with_mean** (`const int` *data*[], `size_t` *n*, `double` *mean*) — Statistics

The function `gsl_stats_est_variance_with_mean` returns the sample variance of *data* relative to the given value of *mean*, and `gsl_stats_int_est_variance` does the same for an integer array. The functions are computed with $\hat\mu$ replaced by the value of *mean* that you supply,

$$\hat\sigma^2 = \frac{1}{(N-1)} \sum (x_i - mean)^2 \tag{5.6}$$

`double` **gsl_stats_est_sd** (`const double` *data*[], `size_t` *n*) — Statistics

`double` **gsl_stats_int_est_sd** (`const int` *data*[], `size_t` *n*) — Statistics

`double` **gsl_stats_est_sd_with_mean** (`const double` *data*[], `size_t` *n*, `double` *mean*) — Statistics

`double` **gsl_stats_int_est_sd_with_mean** (`const int` *data*[], `size_t` *n*, `double` *mean*) — Statistics

The standard deviation is defined as the square root of the variance. These functions return the square root of the corresponding variance functions above.

`double` **gsl_stats_variance** (`const double` *data*[], `size_t` *n*) — Statistics

`double` **gsl_stats_int_variance** (`const int` *data*[], `size_t` *n*) — Statistics

`double` **gsl_stats_variance_with_mean** (`const double` *data*[], `size_t` *n*, `double` *mean*) — Statistics

`double` **gsl_stats_int_variance_with_mean** (`const int` *data*[], `size_t` *n*, `double` *mean*) — Statistics

These functions compute an unbiased estimate of the variance when the population mean of a distribution is known *a priori*.

In this case the estimator for the variance requires a factor $1/N$ and the sample mean $\hat\mu$ must be replaced by the known population mean μ,

$$\hat\sigma^2 = \frac{1}{N} \sum (x_i - \mu)^2 \tag{5.7}$$

`double` **gsl_stats_sd** (`const double` *data*[], `size_t` *n*) — Statistics

`double` **gsl_stats_int_sd** (`const int` *data*[], `size_t` *n*) — Statistics

`double` **gsl_stats_sd_with_mean** (`const double` *data*[], `size_t` *n*, `double` *mean*) — Statistics

`double` **gsl_stats_int_sd_with_mean** (`const int` *data*[], `size_t` *n*, `double` *mean*) — Statistics

The functions `gsl_stats_sd`, `gsl_stats_int_sd`, `gsl_stats_sd_with_mean` and `gsl_stats_int_sd_with_mean` calculate the standard deviation, which is simply the square root of the corresponding variance function.

5.3 Absolute deviation

`double` **gsl_stats_absdev** (`const double` *data*[], `size_t` *n*) Statistics
`double` **gsl_stats_int_absdev** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_absdev` computes the absolute deviation from the mean of *data*, a double precision array of length *n*.

The absolute deviation from the mean is defined as

$$absdev = \frac{1}{N} \sum |x_i - \hat{\mu}| \tag{5.8}$$

where x_i are the elements of the array *data*.

The absolute deviation from the mean provides a more robust measure of the width of a distribution than the variance.

Both these functions compute the mean of *data* via a call to `gsl_stats_mean` or `gsl_stats_int_mean`.

`double` **gsl_stats_absdev_with_mean** (`const double` *data*[], `size_t` *n*, `double` *mean*) Statistics
`double` **gsl_stats_int_absdev_with_mean** (`const int` *data*[], `size_t` *n*, `double` *mean*) Statistics

The functions `gsl_stats_absdev_with_mean` and `gsl_stats_int_absdev_with_mean` compute the absolute deviation of the array *data* relative to the given value of *mean*

$$absdev = \frac{1}{N} \sum |x_i - mean| \tag{5.9}$$

These functions are useful if you have already computed the mean of *data* (and want to avoid recomputing it), or wish to calculate the absolute deviation relative to another value (such as zero, or the median).

5.4 Higher moments (skewness and kurtosis)

`double` **gsl_stats_skew** (`const double` *data*[], `size_t` *n*) Statistics
`double` **gsl_stats_int_skew** (`const int` *data*[], `size_t` *n*, `double` *mean*) Statistics

The functions `gsl_stats_skew` and `gsl_stats_int_skew` compute the skewness of *data*, an array of length *n*.

The skewness is defined as

$$skew = \frac{1}{N} \sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^3 \tag{5.10}$$

where x_i are the elements of the array *data*.

The skewness measures the asymmetry of the tails of a distribution.

Both these functions compute the mean and standard deviation of *data* via calls to `gsl_stats_mean` and `gsl_stats_est_sd` or `gsl_stats_int_mean` and `gsl_stats_int_est_sd`.

double **gsl_stats_skew_with_mean_and_sd** (const double *data*[], size_t *n*, double *mean*, double *sd*) Statistics

double **gsl_stats_int_skew_with_mean_and_sd** (const int *data*[], size_t *n*, double *mean*, double *sd*) Statistics

The functions `gsl_stats_skew_with_mean_and_sd` and `gsl_stats_int_skew_with_mean_and_sd` compute the skewness of the array *data* using the given values of the mean *mean* and standard deviation *sd*.

$$skew = \frac{1}{N} \sum \left(\frac{x_i - mean}{sd} \right)^3 \tag{5.11}$$

These functions are useful if you have already computed the mean and standard deviation of *data* and want to avoid recomputing them.

double **gsl_stats_kurtosis** (const double *data*[], size_t *n*) Statistics

double **gsl_stats_int_kurtosis** (const int *data*[], size_t *n*, double *mean*) Statistics

The functions `gsl_stats_kurtosis` and `gsl_stats_int_kurtosis` compute the kurtosis of *data*, an array of length *n*.

The kurtosis is defined as

$$kurtosis = \left(\frac{1}{N} \sum \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right)^4 \right) - 3 \tag{5.12}$$

The kurtosis measures how sharply peaked a distribution is, relative to its width. The kurtosis is normalised to zero for a gaussian distribution by the -3 in the defintion.

double **gsl_stats_kurtosis_with_mean_and_sd** (const double *data*[], size_t *n*, double *mean*, double *sd*) Statistics

double **gsl_stats_int_kurtosis_with_mean_and_sd** (const int *data*[], size_t *n*, double *mean*, double *sd*) Statistics

The functions `gsl_stats_kurtosis_with_mean_and_sd` and `gsl_stats_int_kurtosis_with_mean_and_sd` compute the kurtosis of the array *data* using the given values of the mean *mean* and standard deviation *sd*.

$$kurtosis = \frac{1}{N} \left(\sum \left(\frac{x_i - mean}{sd} \right)^4 \right) - 3 \tag{5.13}$$

These functions are useful if you have already computed the mean and standard deviation of *data* and want to avoid recomputing them.

5.5 Maximum and Minimum values

`double` **gsl_stats_max** (`const double` *data*[], `size_t` *n*) Statistics
`int` **gsl_stats_int_max** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_max` returns the maximum value in *data*, a double-precision array of length *n*. The function `gsl_stats_int_stats_max` returns the maximum value of an integer array. The maximum value is defined as the value of the element x_i which satisfies $x_i >= x_j$ for all j.

If you want instead to find the element with the largest absolute magnitude you will need to apply `fabs` or `abs` to your data before calling this function.

`double` **gsl_stats_min** (`const double` *data*[], `size_t` *n*) Statistics
`int` **gsl_stats_int_min** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_min` returns the minimun value in *data*, a double-precision array of length *n*. The function `gsl_stats_int_stats_min` returns the minimum value of an integer array. The minimum value is defined as the value of the element x_i which satisfies $x_i <= x_j$ for all j.

If you want instead to find the element with the smallest absolute magnitude you will need to apply `fabs` or `abs` to your data before calling this function.

`size_t` **gsl_stats_max_index** (`const double` *data*[], `size_t` *n*) Statistics
`size_t` **gsl_stats_int_max_index** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_max_index` returns the index of the maximum value in *data*, a double-precision array of length *n*. The function `gsl_stats_int_stats_max` returns the index of the maximum value of an integer array. The maximum value is defined as the value of the element x_i which satisfies $x_i >= x_j$ for all j. When there are several equal maximum elements then the first one is chosen.

`size_t` **gsl_stats_min_index** (`const double` *data*[], `size_t` *n*) Statistics
`size_t` **gsl_stats_int_min_index** (`const int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_min_index` returns the index of the minimum value in *data*, a double-precision array of length *n*. The function `gsl_stats_int_stats_min` returns the index of the minimum value of an integer array. The minimum value is defined as the value of the element x_i which satisfies $x_i >= x_j$ for all j. When there are several equal minimum elements then the first one is chosen.

5.6 Median and Percentiles

The median and percentile functions described in this section operate on sorted data. For convenience we use *quantiles*, measured on a scale of 0 to 1, instead of percentiles (which use a scale of 0 to 100).

`void` **gsl_stats_sort_data** (`double` *data*[], `size_t` *n*) Statistics
`void` **gsl_stats_int_sort_data** (`int` *data*[], `size_t` *n*) Statistics

The function `gsl_stats_sort_data` sorts the elements of *data*, a double-precision array of length *n*, into ascending numerical order in-place (i.e. after sorting, the minimum value of data is in *data[0]* and the maximum value is in *data[n-1]*). The system `qsort` function is used to perform the sorting.

The function `gsl_stats_int_sort_data` sorts an integer array.

`double` **gsl_stats_median_from_sorted_data** (`const double` *sorted_data*[], `size_t` *n*) Statistics
`double` **gsl_stats_int_median_from_sorted_data** (`const int` *sorted_data*[], `size_t` *n*) Statistics

The function `gsl_stats_median_from_sorted_data` returns the median value of *sorted_data*, a double-precision array of length *n* with its elements in ascending numerical order. There are no checks to see whether the data are sorted, so the function `gsl_stats_sort_data` should always be used first.

When the array has an odd number of elements the median is the value of element $(n-1)/2$. When the array has an even number of elements the median is the mean of the two nearest middle values, elements $(n-1)/2$ and $n/2$.

The function `gsl_stats_int_median_from_sorted_data` returns the median of an integer array. Since the median may have to be found by interpolation the function `gsl_stats_int_median_from_sorted_data` always returns a floating-point number.

`double` **gsl_stats_quantile_from_sorted_data** (`const double` *sorted_data*[], `size_t` *n*, `double` *f*) Statistics
`double` **gsl_stats_int_quantile_from_sorted_data** (`const int` *sorted_data*[], `size_t` *n*, `double` *f*) Statistics

The function `gsl_stats_median_from_sorted_data` returns a quantile value of *sorted_data*, a double-precision array of length *n* with its elements in ascending numerical order. The quantile is determined by the *f*, a fraction between 0 and 1. For example, to compute the value of the 75th percentile *f* should have the value 0.75.

There are no checks to see whether the data are sorted, so the function `gsl_stats_sort_data` should always be used first.

The quantile is found by interpolation, using the formula

$$quantile = (1-\delta) data[i] + \delta data[i+1] \tag{5.14}$$

where i is `floor`$((n-1)f)$ and δ is $(n-1)f - i$.

Thus the minimum value of the array (*data[0]*) is given by f equal to zero, the maximum value (*data[n-1]*) is given by f equal to one and the median value is given by f equal to 0.5.

The function `gsl_stats_int_quantile_from_sorted_data` returns a quantile of an integer array. Since the quantile is found by interpolation the function `gsl_stats_int_quantile_from_sorted_data` always returns a floating-point number.

5.7 Statistical tests

FIXME, do more work on the statistical tests

5.8 Example statistical programs

Here is a basic example of how to use the statistical functions:

```
#include <stdio.h>
#include <gsl_statistics.h>

int main()
{
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6} ;
  double mean, variance, largest, smallest;

  mean     = gsl_stats_mean(data, 5);
  variance = gsl_stats_variance(data, 5);
  largest  = gsl_stats_max(data, 5);
  smallest = gsl_stats_min(data, 5);

  printf("The dataset is %g, %g, %g, %g, %g\n",
        data[0], data[1], data[2], data[3], data[4]);

  printf("The sample mean is %g\n", mean) ;
  printf("The estimated variance is %g\n", variance) ;
  printf("The largest value is %g\n", largest) ;
  printf("The smallest value is %g\n", smallest) ;
}
```

The program should produce the following output,

```
The dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sample mean is 16.54
The estimated variance is 4.2984
The largest value is 18.3
The smallest value is 12.6
```

Here is an example using sorted data,

```
#include <stdio.h>
#include <gsl_statistics.h>

int main()
{
  double data[5] = {17.2, 18.1, 16.5, 18.3, 12.6} ;
  double median, upperq, lowerq;

  printf("The original dataset is %g, %g, %g, %g, %g\n",
        data[0], data[1], data[2], data[3], data[4]);

  gsl_stats_sort_data(data, 5) ;
```

```
    printf("The sorted dataset is %g, %g, %g, %g, %g\n",
           data[0], data[1], data[2], data[3], data[4]);

    median   = gsl_stats_median_from_sorted_data(data, 5);
    upperq   = gsl_stats_quantile_from_sorted_data(data, 5, 0.75);
    lowerq   = gsl_stats_quantile_from_sorted_data(data, 5, 0.25);

    printf("The median is %g\n", median) ;
    printf("The upper quartile is %g\n", upperq) ;
    printf("The lower quartile is %g\n", lowerq) ;
  }
```

This program should produce the following output,

```
The original dataset is 17.2, 18.1, 16.5, 18.3, 12.6
The sorted dataset is 12.6, 16.5, 17.2, 18.1, 18.3
The median is 17.2
The upper quartile is 18.1
The lower quartile is 16.5
```

5.9 Statistics References and Further Reading

The standard reference for almost any topic in statistics is the multi-volume *Advanced Theory of Statistics* by Kendall and Stuart.

Maurice Kendall, Alan Stuart, and J. Keith Ord. *The Advanced Theory of Statistics* (multiple volumes) reprinted as *Kendall's Advanced Theory of Statistics*. Wiley, ISBN 047023380X.

Many statistical concepts can be more easily understood by a Bayesian approach. The recent book by Gelman, Carlin, Stern and Rubin gives a comprehensive coverage of the subject, while Jeffreys' *Theory of Probability* gives an older but more pedagogical introduction.

Andrew Gelman, John B. Carlin, Hal S. Stern, Donald B. Rubin. *Bayesian Data Analysis*. Chapman & Hall, ISBN 0412039915.

Sir Harold Jeffreys. *Theory of Probability*. Clarendon Press, ISBN 0198531931

For physicists the Particle Data Group provides useful reviews of Probability and Statistics in the "Mathematical Tools" section of its Annual Review of Particle Physics.

Review of Particle Properties R.M. Barnett et al., Physical Review D54, 1 (1996)

The Review of Particle Physics is available online in postscript and pdf format at `http://pdg.lbl.gov/`.

6 Fast Fourier Transforms (FFTs)

This chapter describes functions for performing Fast Fourier Transforms (FFTs). The library includes radix-2 routines (for lengths which are a power of two) and mixed-radix routines (which work for any length). For efficiency there are separate versions of the routines for real data and for complex data. The mixed-radix routines are a reimplementation of the FFTPACK library by Paul Swarztrauber. Fortran code for FFTPACK is available on Netlib (FFTPACK also includes some routines for sine and cosine transforms but these are currently not available in GSL). For details and derivations of the underlying algorithms consult the document *GSL FFT Algorithms* (see Section 6.4 [FFT References and Further Reading], page 78)

6.1 Mathematical Definitions

Fast Fourier Transforms are efficient algorithms for calculating the discrete fourier transform (DFT),

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi i j k/N) \tag{6.1}$$

The DFT usually arises as an approximation to the continuous fourier transform when functions are sampled at discrete intervals in space or time. The naive evaluation of the discrete fourier transform is a matrix-vector multiplication $W\vec{z}$. A general matrix-vector multiplication takes $O(N^2)$ operations for N data-points. Fast fourier transform algorithms use a divide-and-conquer strategy to factorize the matrix W into smaller sub-matrices, corresponding to the integer factors of the length N. If N can be factorized into a product of integers $f_1 f_2 \ldots f_n$ then the DFT can be computed in $O(N \sum f_i)$ operations. For a radix-2 FFT this gives an operation count of $O(N \log_2 N)$.

All the FFT functions offer three types of transform: forwards, inverse and backwards, based on the same mathematical definitions. The definition of the *forward fourier transform*, x=FFT(z), is,

$$x_j = \sum_{k=0}^{N-1} z_k \exp(-2\pi i j k/N) \tag{6.2}$$

and the definition of the *inverse fourier transform*, x=IFFT(z), is,

$$z_j = \frac{1}{N} \sum_{k=0}^{N-1} x_k \exp(2\pi i j k/N). \tag{6.3}$$

The factor of $1/N$ makes this a true inverse. For example, a call to `gsl_fft_complex_forward` followed by a call to `gsl_fft_complex_inverse` should return the original data (within numerical errors).

In general there are two possible choices for the sign of the exponential in the transform/inverse-transform pair. GSL follows the same convention as FFTPACK, using a negative exponential for the forward transform. The advantage of this convention is that the inverse transform recreates the original function with simple fourier synthesis. Numerical Recipes uses the opposite convention, a positive exponential in the forward transform.

The *backwards FFT* is simply our terminology for an unscaled version of the inverse FFT,

$$z_j^{backwards} = \sum_{k=0}^{N-1} x_k \exp(2\pi ijk/N). \tag{6.4}$$

When the overall scale of the result is unimportant it is often convenient to use the backwards FFT instead of the inverse to save unnecessary divisions.

6.2 FFTs of complex data

The inputs and outputs for the radix-2 and mixed-radix complex FFT routines are simply vectors of complex elements. For example,

```
gsl_complex data[128] ;
```

The array indices are the same as those in the definition of the DFT — i.e. there are no index transformations or permutations of the data. The `gsl_complex` type is defined as `struct {double real ; double imag}`.

For physical applications it is important to remember that the index appearing in the DFT does not correspond directly to a physical frequency. If the time-step of the DFT is Δ then the frequency-domain includes both positive and negative frequencies, ranging from $-1/(2\Delta)$ through 0 to $+1/(2\Delta)$. The positive frequencies are stored from the beginning of the array up to the middle, and the negative frequencies are stored backwards from the end of the array.

Here is a table which shows the layout of the array *data*, and the correspondence between the time-domain data z, and the frequency-domain data x.

```
index    z               x = FFT(z)

0        z(t = 0)        x(f = 0)
1        z(t = 1)        x(f = 1/(N Delta))
2        z(t = 2)        x(f = 2/(N Delta))
.        ........        ..................
N/2      z(t = N/2)      x(f = +1/(2 Delta),-1/(2 Delta))
.        ........        ..................
N-3      z(t = N-3)      x(f = -3/(N Delta))
N-2      z(t = N-2)      x(f = -2/(N Delta))
N-1      z(t = N-1)      x(f = -1/(N Delta))
```

When N is even the location $N/2$ contains the most positive and negative frequencies $(+1/(2\Delta),-1/(2\Delta))$ which are equivalent. If N is odd then general structure of the table above still applies, but $N/2$ does not appear.

6.2.1 Radix-2 FFT routines for complex data

The radix-2 algorithms described in this section are simple and compact, although not necessarily the most efficient. They use the Cooley-Tukey algorithm to compute in-place complex FFTs for lengths which are a power of 2 — no additional storage is required. The corresponding self-sorting mixed-radix routines offer better performance at the expense of requiring additional scratch space.

All these functions are declared in the header file '`gsl_fft_complex.h`'.

`int` **gsl_fft_complex_radix2_forward** (`gsl_complex` *data*[], `size_t` *n*) Function

`int` **gsl_fft_complex_radix2_backward** (`gsl_complex` *data*[], `size_t` *n*) Function

`int` **gsl_fft_complex_radix2_inverse** (`gsl_complex` *data*[], `size_t` *n*) Function

These functions compute the forward, backward and inverse FFTs of *data*, a complex array of length *n*, using an in-place radix-2 decimation-in-time algorithm. The length of the data *n* is restricted to powers of two.

The functions return a value of `0` if no errors were detected, and `-1` in the case of error. The following `gsl_errno` condition is defined for these functions:

`GSL_EDOM` The length of the data *n* is not a power of two.

`int` **gsl_fft_complex_radix2_dif_forward** (`gsl_complex` *data*[], `size_t` *n*) Function

`int` **gsl_fft_complex_radix2_dif_backward** (`gsl_complex` *data*[], `size_t` *n*) Function

`int` **gsl_fft_complex_radix2_dif_inverse** (`gsl_complex` *data*[], `size_t` *n*) Function

These are decimation-in-frequency versions of the radix-2 FFT functions.

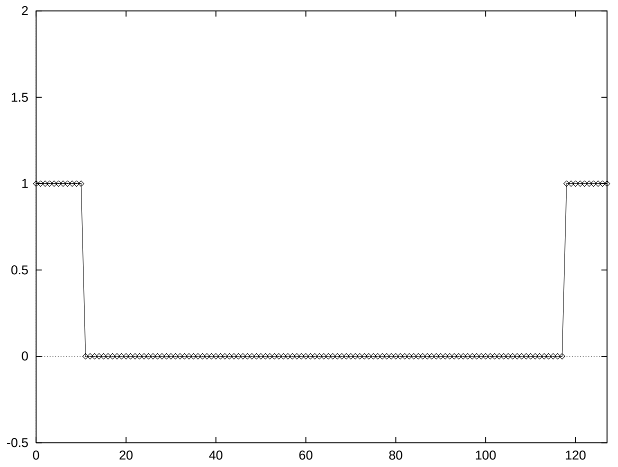
6.2.2 Example of using radix-2 FFT routines for complex data

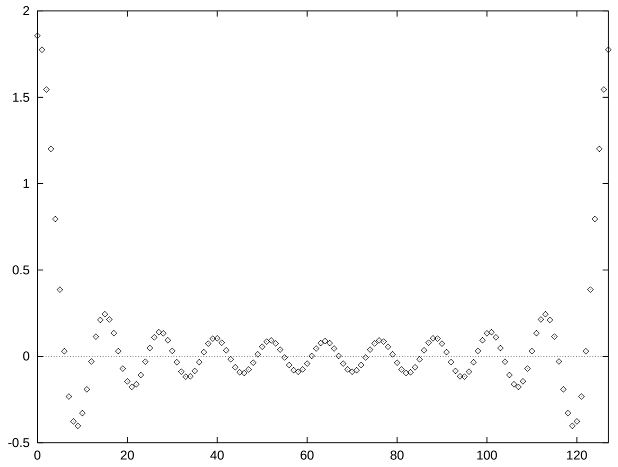
Here is an example program which computes the FFT of a short pulse in a sample of length 128. To make the resulting fourier transform real the pulse is defined for equal positive and negative times ($-10 \ldots 10$) (the negative times wrap around the end of the array).

```
#include <stdio.h>
#include <math.h>
#include <gsl_errno.h>
#include <gsl_fft_complex.h>

int main ()
{
  int i;
  gsl_complex data[128];

  for (i = 0; i < 128; i++)
```

```
        {
           data[i].real = 0.0;
           data[i].imag = 0.0;
        }

      data[0].real = 1.0;

      for (i = 1; i <= 10; i++)
        {
           data[i].real = data[128-i].real = 1.0;
        }

      for (i = 0; i < 128; i++)
        {
          printf ("%d %e %e\n", i, data[i].real, data[i].imag);
        }
      printf ("\n");

      gsl_fft_complex_radix2_forward (data, 128);

      for (i = 0; i < 128; i++)
        {
          printf ("%d %e %e\n", i, data[i].real/sqrt(128),
                                   data[i].imag/sqrt(128));
        }

    }
```

Note that have assumed that the program is using the default `gsl` error handler (which calls `abort` for any errors). If you are not using a safe error handler you would need to check the return status of `gsl_fft_complex_radix2_forward`.





A pulse (left) and its discrete fourier transform (right), output from the example program. The transformed data is rescaled by $1/\sqrt{N}$ so that it fits on the same plot as the input. Only the real part is shown, by the choice of the input data the imaginary part is zero.

Allowing for the wrap-around of negative times at $t = 128$, and working in units of k/N, the DFT approximates the continuum fourier transform of a modulated sin function,

$$\int_{-a}^{+a} e^{-2\pi ikx} dx = \frac{\sin(2\pi ka)}{\pi k}.$$

6.2.3 Mixed-radix FFT routines for complex data

This section describes mixed-radix FFT algorithms for complex data. The mixed-radix functions work for FFTs of any length. They are a reimplementation of the Fortran FFT-PACK library by Paul Swarztrauber. The theory is explained in the review article *Self-sorting Mixed-radix FFTs* by Clive Temperton. The routines here use the same indexing scheme and basic algorithms as FFTPACK.

The mixed-radix algorithm is based on sub-transform modules – highly optimized small length FFTs which are combined to create larger FFTs. There are efficient modules for factors of 2, 3, 4, 5, 6 and 7. The modules for the composite factors of 4 and 6 are faster than combining the modules for $2 * 2$ and $2 * 3$.

For factors which are not implemented as modules there is a fall-back to a general length-n module which uses Singleton's method for efficiently computing a DFT. This module is $O(n^2)$, and slower than a dedicated module would be but works for any length n. Of course, lengths which use the general length-n module will still be factorized as much as possible. For example, a length of 143 will be factorized into $11 * 13$. Large prime factors are the worst case scenario, e.g. as found in $n = 2 * 3 * 99991$, and should be avoided because their $O(n^2)$ scaling will dominate the run-time (consult *GSL FFT Algorithms* for possible ways around this problem).

The mixed-radix initialization function `gsl_fft_complex_init` returns the list of factors chosen by the library for a given length N. It can be used to check how well the length has been factorized, and estimate the run-time. To a first approximation the run-time scales as $N \sum f_i$, where the f_i are the factors of N. For programs under user control you may wish to issue a warning that the transform will be slow when the length is poorly factorized. If you frequently encounter data lengths which cannot be factorized using the existing small-prime modules consult *GSL FFT Algorithms* for details on adding support for other factors.

All these functions are declared in the header file '`gsl_fft_complex.h`'.

`gsl_fft_complex_wavetable *` Function
gsl_fft_complex_wavetable_alloc (`size_t` *n*);

This function allocates space for a `gsl_fft_complex_wavetable` struct, and a scratch area and trigonometric lookup table, both of size *n* complex elements. The relevant components of the wavetable are initialized to point to the newly allocated memory.

The function returns a pointer to the newly allocated `gsl_fft_complex_wavetable` if no errors were detected, and 0 in the case of error. The following `gsl_errno` conditions are defined for this function:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`GSL_ENOMEM`
: The requested memory could not be allocated (`malloc` returned a null pointer).

`int` **gsl_fft_complex_init** (`size_t` *n*, `gsl_fft_complex_wavetable *` *wavetable*); Function

This function initializes *wavetable*. It selects a factorization of the length *n* into the implemented subtransforms, storing the details of the factorization in *wavetable*. Using this factorization it then prepares a trigonometric lookup table in the memory previously allocated by `gsl_fft_complex_wavetable_alloc`.

The wavetable is computed using direct calls to `sin` and `cos`, for accuracy. It could be computed faster using recursion relations if necessary. If an application performs many FFTs of the same length then computing the wavetable is a one-off overhead which does not affect the final throughput.

The wavetable structure can be used repeatedly for any transform of the same length. The table is not modified by calls to any of the other FFT functions. The same wavetable can be used for both forward and backward (or inverse) transforms of a given length.

The function returns a value of `0` if no errors were detected, and `-1` in the case of error. The following `gsl_errno` conditions are defined for this function:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`GSL_EFACTOR`
: The length *n* could not be factorized (this shouldn't happen).

`GSL_EFAILED`
: A failure was detected in the wavetable generation. This could be caused by an inconsistency in a user-supplied *wavetable* structure.

`void` **gsl_fft_complex_wavetable_free** (`gsl_fft_complex_wavetable *` *wavetable*); Function

This function frees the blocks of memory previously allocated by `gsl_fft_complex_wavetable_alloc` and pointed to by the components of *wavetable*.

The wavetable should be freed if no further FFTs of the same length will be needed.

The functions `gsl_fft_complex_init`, `gsl_fft_wavetable_alloc` and `gsl_fft_wavetable_free` all operate on a structure `gsl_fft_complex_wavetable`. This structure contains internal parameters for the FFT.

It is not necessary to set any of the members directly except for advanced usage. However, it can useful to examine the wavetable returned by `gsl_fft_complex_init`. For example, the factorization chosen by `gsl_fft_complex_init` is given and can be used to provide an estimate of the run-time or numerical error.

Here is the wavetable structure. It is declared in the header file '`gsl_fft_complex.h`'.

struct gsl_fft_complex_wavetable Data Type

This is a structure that holds the factorization and pointers to the scratch area and trigonometric lookup tables for the mixed radix fft algorithm. It has the following members:

`size_t n` This is the number of complex data points

`size_t nf` This is the number of factors that the length `n` was decomposed into.

`size_t factor[64]`
: This is the array of factors. Only the first `nf` elements are used.

`gsl_complex * scratch`
: This is a pointer to a scratch area of `n` complex elements, capable of holding intermediate copies of the original data set.

`gsl_complex * trig`
: This is a pointer to a preallocated trigonometric lookup table of `n` complex elements.

`gsl_complex * twiddle[64]`
: This is an array of pointers into `trig`, giving the twiddle factors for each pass.

`int` **gsl_fft_complex_forward** (`gsl_complex` *data*[], `size_t` *n*, `const gsl_fft_wavetable *` *wavetable*) Function

`int` **gsl_fft_complex_inverse** (`gsl_complex` *data*[], `size_t` *n*, `const gsl_fft_complex_wavetable *` *wavetable*) Function

`int` **gsl_fft_complex_backward** (`gsl_complex` *data*[], `size_t` *n*, `const gsl_fft_complex_wavetable *` *wavetable*) Function

These functions compute the forward, backward and inverse FFT of *data*, a complex array of length *n*, using a mixed radix decimation-in-frequency algorithm.

The caller must supply a *wavetable* containing the chosen factorization, trigonometric lookup tables and scratch area. The wavetable can be easily prepared using the functions `gsl_fft_complex_init` and `gsl_fft_complex_alloc`.

There is no restriction on the length *n*. Efficient modules are provided for subtransforms of length 2, 3, 4, 5, 6 and 7. Any remaining factors are computed with a slow, $O(n^2)$, general-n module.

The functions return a value of `0` if no errors were detected, and `-1` in the case of error. The following `gsl_errno` conditions are defined for these functions:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`GSL_EINVAL`
: The length of the data *n* and the length used to compute the given *wavetable* do not match.

6.2.4 Example of using mixed-radix FFT routines for complex data

Here is an example program which computes the FFT of a short pulse in a sample of length 630 ($= 2 * 3 * 3 * 5 * 7$) using the mixed-radix algorithm.

```
#include <stdio.h>
#include <math.h>
#include <gsl_errno.h>
#include <gsl_fft_complex.h>

int main ()
{
  int i;
  const int n = 630 ;
  gsl_complex data[n];

  gsl_fft_complex_wavetable * wavetable;

  for (i = 0; i < n; i++)
    {
      data[i].real = 0.0 ;
      data[i].imag = 0.0 ;
    }

  data[0].real = 1.0 ;

  for (i = 1; i <= 10; i++)
    {
      data[i].real = data[n-i].real = 1.0 ;
    }

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e %e\n", i, data[i].real, data[i].imag);
    }
  printf ("\n");

  wavetable = gsl_fft_complex_wavetable_alloc (n);
  gsl_fft_complex_init (n, wavetable);

  for (i = 0; i < wavetable->nf; i++)
    {
       printf("# factor %d: %d\n", i, wavetable->factor[i]);
    }

  gsl_fft_complex_forward (data, n, wavetable);

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e %e\n", i, data[i].real, data[i].imag);
    }

  gsl_fft_complex_wavetable_free (wavetable);

}
```

Note that we have assumed that the program is using the default `gsl` error handler (which calls `abort` for any errors). If you are not using a safe error handler you would need to check the return status of of all the `gsl` routines.

6.3 FFTs of real data

The functions for real data are similar to those for complex data. However, there is an important difference between forward and inverse transforms. The fourier transform of a real sequence is not real. It is a complex sequence with a special symmetry:

$$z_k = z^*_{N-k} \tag{6.5}$$

A sequence with this symmetry is called *conjugate-complex* or *half-complex*. This different structure requires different storage-layouts for the forward transform (from real to half-complex) and inverse transform (from half-complex back to real). As a consequence the routines are divided into two sets: functions in `gsl_fft_real` which operate on real sequences and functions in `gsl_fft_halfcomplex` which operate on half-complex sequences.

Functions in `gsl_fft_real` compute the frequency coefficients of a real sequence. The half-complex coefficients c of a real sequence x are given by fourier analysis,

$$c_k = \sum_{j=0}^{N-1} x_k \exp(-2\pi i j k/N) \tag{6.6}$$

Functions in `gsl_fft_halfcomplex` compute inverse or backwards transforms. They reconstruct real sequences by fourier synthesis from their half-complex frequency coefficients, c,

$$x_j = \frac{1}{N} \sum_{k=0}^{N-1} c_k \exp(2\pi i j k/N) \tag{6.7}$$

The symmetry of the half-complex sequence implies that only half of the complex numbers in the output need to be stored. The remaining half can be reconstructed using the half-complex symmetry condition. (This works for all lengths, even and odd. When the length is even the middle value, where $k = N/2$, is also real). Thus only N real numbers are required to store the half-complex sequence, and the transform of a real sequence can be stored in the same size array as the original data.

The precise storage arrangements depend on the algorithm, and are different for radix-2 and mixed-radix routines. The radix-2 function operates in-place, which constrain the locations where each element can be stored. The restriction forces real and imaginary parts to be stored far apart. The mixed-radix algorithm does not have this restriction, and it stores the real and imaginary parts of a given term in neighboring locations. This is desirable for better locality of memory accesses.

6.3.1 Radix-2 FFT routines for real data

This section describes radix-2 FFT algorithms for real data. They use the Cooley-Tukey algorithm to compute in-place FFTs for lengths which are a power of 2.

The radix-2 FFT functions for real data are declared in the header files '`gsl_fft_real.h`'

`int` **gsl_fft_real_radix2** (`double` *data*[], `size_t` *n*) Function

This function computes an in-place radix-2 FFT of *data*, a real array of length *n*. The output is a half-complex sequence, which is stored in-place. The arrangement of the half-complex terms uses the following scheme: for $k < N/2$ the real part of the k-th term is stored in location k, and the corresponding imaginary part is stored in location $N - k$. Terms with $k > N/2$ can be reconstructed using the symmetry $z_k = z^*_{N-k}$. The terms for $k = 0$ and $k = N/2$ are both purely real, and count as a special case. Their real parts are stored in locations 0 and $N/2$ respectively, while their imaginary parts which are zero are not stored.

The following table shows the correspondence between the output *data* and the equivalent results obtained by considering the input data as a complex sequence with zero imaginary part,

```
complex[0].real    =    data[0]
complex[0].imag    =    0
complex[1].real    =    data[1]
complex[1].imag    =    data[N-1]
...............         ................
complex[k].real    =    data[k]
complex[k].imag    =    data[N-k]
...............         ................
complex[N/2].real  =    data[N/2]
complex[N/2].real  =    0
...............         ................
complex[k'].real   =    data[k]        k' = N - k
complex[k'].imag   =   -data[N-k]
...............         ................
complex[N-1].real  =    data[1]
complex[N-1].imag  =   -data[N-1]
```

The radix-2 FFT functions for halfcomplex data are declared in the header file '`gsl_fft_halfcomplex.h`'.

`int` **gsl_fft_halfcomplex_radix2_inverse** (`double` *data*[], `size_t` *n*) Function

`int` **gsl_fft_halfcomplex_radix2_backwards** (`double` *data*[], `size_t` *n*) Function

These functions compute the inverse or backwards in-place radix-2 FFT of the half-complex sequence *data*, a real of length *n* stored according the output scheme used by `gsl_fft_real_radix2`. The result is a real array stored in natural order.

6.3.2 Mixed-radix FFT routines for real data

This section describes mixed-radix FFT algorithms for real data. The mixed-radix functions work for FFTs of any length. They are a reimplementation of the real-FFT routines in the Fortran FFTPACK library by Paul Swarztrauber. The theory behind the algorithm is explained in the article *Fast Mixed-Radix Real Fourier Transforms* by Clive Temperton. The routines here use the same indexing scheme and basic algorithms as FFTPACK.

The functions use the FFTPACK storage convention for half-complex sequences. In this convention the half-complex transform of a real sequence is stored in with frequencies in increasing order, starting at zero, with the real and imaginary parts of each frequency in neighboring locations. When a value is known to be real the imaginary part is not stored. The imaginary part of the zero-frequency component is never stored. It is known to be zero (since the zero frequency component is simply the sum of the input data (all real)). For a sequence of even length the imaginary part of the frequency $n/2$ is not stored either, since the symmetry $z_k = z^*_{N-k}$ implies that this is purely real too.

The storage scheme is best shown by some examples. The table below shows the output for an odd-length sequence, $n = 5$. The two columns give the correspondence between the 5 values in the half-complex sequence returned by `gsl_fft_real`, *halfcomplex[]* and the values *complex[]* that would be returned if the same real input sequence were passed to `gsl_fft_complex_backward` as a complex sequence (with imaginary parts set to `0`),

```
complex[0].real  =  halfcomplex[0]
complex[0].imag  =  0
complex[1].real  =  halfcomplex[1]
complex[1].imag  =  halfcomplex[2]
complex[2].real  =  halfcomplex[3]
complex[2].imag  =  halfcomplex[4]
complex[3].real  =  halfcomplex[3]
complex[3].imag  = -halfcomplex[4]
complex[4].real  =  halfcomplex[1]
complex[4].imag  = -halfcomplex[2]
```

The upper elements of the *complex* array, `complex[3]` and `complex[4]` are filled in using the symmetry condition. The imaginary part of the zero-frequency term `complex[0].imag` is known to be zero by the symmetry.

The next table shows the output for an even-length sequence, $n = 5$ In the even case both the there are two values which are purely real,

```
complex[0].real  =  halfcomplex[0]
complex[0].imag  =  0
complex[1].real  =  halfcomplex[1]
complex[1].imag  =  halfcomplex[2]
complex[2].real  =  halfcomplex[3]
complex[2].imag  =  halfcomplex[4]
complex[3].real  =  halfcomplex[5]
complex[3].imag  =  0
```

```
complex[4].real  =  halfcomplex[3]
complex[4].imag  = -halfcomplex[4]
complex[5].real  =  halfcomplex[1]
complex[5].imag  = -halfcomplex[2]
```

The upper elements of the *complex* array, `complex[4]` and `complex[5]` are be filled in using the symmetry condition. Both `complex[0].imag` and `complex[3].imag` are known to be zero.

All these functions are declared in the header files '`gsl_fft_real.h`' and '`gsl_fft_halfcomplex.h`'.

`gsl_fft_real_wavetable *` **gsl_fft_real_wavetable_alloc** (`size_t` *n*); Function

`gsl_fft_halfcomplex_wavetable *` **gsl_fft_halfcomplex_wavetable_alloc** (`size_t` *n*); Function

These functions allocate space for a wavetable struct, a scratch area of size n real elements and a trigonometric lookup table, of size $n/2$ complex elements. The relevant components of the wavetable are modified to point to the newly allocated memory.

These functions return a pointer to the newly allocated struct if no errors were detected, and `0` in the case of error. The following `gsl_errno` conditions are defined for these functions:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`GSL_ENOMEM`
The requested memory could not be allocated (`malloc` returned a null pointer).

`int` **gsl_fft_real_init** (`size_t` *n*, `gsl_fft_real_wavetable *` *wavetable*); Function

`int` **gsl_fft_halfcomplex_init** (`size_t` *n*, `gsl_fft_halfcomplex_wavetable *` *wavetable*); Function

These functions initialize *wavetable*. They first select a factorization of the length *n* into the implemented subtransforms, storing the details of the factorization in *wavetable*.

Using this factorization they then prepare a trigonometric lookup table in the memory previously allocated by `gsl_fft_real_wavetable_alloc` or `gsl_fft_halfcomplex_wavetable_alloc`. The wavetable is computed using direct calls to `sin` and `cos`, for accuracy. It could be computed faster using recursion relations if necessary. If an application performs many FFTs of the same length then computing the wavetable is a one-off overhead which does not affect the final throughput.

The wavetable structure can be used repeatedly for any transform of the same length. The table is not modified by calls to any of the other FFT functions. The same wavetable can be used for both forward and backward (or inverse) transforms of a given length.

The functions return a value of `0` if no errors were detected, and `-1` in the case of error. The following `gsl_errno` conditions are defined for these functions:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`GSL_EFACTOR`
: The length *n* could not be factorized (this shouldn't happen).

`GSL_EFAILED`
: A failure was detected in the wavetable generation. This could be caused by an inconsistency in a user-supplied *wavetable* structure.

`void` **gsl_fft_real_wavetable_free** (`gsl_fft_real_wavetable *` *wavetable*); Function

`void` **gsl_fft_halfcomplex_wavetable_free** (`gsl_fft_halfcomplex_wavetable *` *wavetable*); Function

These functions free the blocks of memory previously allocated by `gsl_fft_real_wavetable_alloc` or `gsl_fft_halfcomplex_wavetable_alloc` and pointed to by the components of *wavetable*.

The wavetable should be freed if no further FFTs of the same length will be needed.

`int` **gsl_fft_real** (`double` *data*[], `size_t` *n*, `const gsl_fft_real_wavetable *` *wavetable*) Function

`int` **gsl_fft_halfcomplex** (`double` *data*[], `size_t` *n*, `const gsl_fft_halfcomplex_wavetable *` *wavetable*) Function

These functions compute the FFT of *data*, a real or half-complex array of length *n*, using a mixed radix decimation-in-frequency algorithm. For `gsl_fft_real` *data* is an array of time-ordered real data. For `gsl_fft_halfcomplex` *data* contains fourier coefficients in the half-complex ordering described above.

The caller must supply a *wavetable* containing the chosen factorization, trigonometric lookup tables and scratch area. The wavetable can be easily prepared using the functions `gsl_fft_real_alloc` and `gsl_fft_real_init` or `gsl_fft_halfcomplex_alloc` and `gsl_fft_halfcomplex_init`.

There is no restriction on the length *n*. Efficient modules are provided for subtransforms of length 2, 3, 4 and 5. Any remaining factors are computed with a slow, $O(n^2)$, general-n module.

The functions return a value of `0` if no errors were detected, and `-1` in the case of error. The following `gsl_errno` conditions are defined for these functions:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`GSL_EINVAL`
: The length of the data *n* and the length used to compute the given *wavetable* do not match.

`int` **gsl_fft_real_unpack** (`const double` *real_coefficient*[], `gsl_complex` *complex_coefficient*[], `size_t` *n*) Function

This function converts a single real array, *real_coefficient* into an equivalent complex array, *complex_coefficient*, (with imaginary part set to zero), suitable for `gsl_fft_complex` routines. The algorithm for the conversion is simply,

```
for (i = 0; i < n; i++)
    {
      complex_coefficient[i].real = real_coefficient[i];
      complex_coefficient[i].imag = 0.0;
    }
```

The function returns a value of `0` if no errors were detected, and `-1` in the case of error. There is only one `gsl_errno` condition defined for this function:

`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

`int` **gsl_fft_halfcomplex_unpack** (`const double` *halfcomplex_coefficient*[], `gsl_complex` *complex_coefficient*[], `size_t` *n*) Function

This function converts *halfcomplex_coefficient*, an array of half-complex coefficients as returned by `gsl_fft_real`, into an ordinary complex array, *complex_coefficient*. It fills in the complex array using the symmetry $z_k = z^*_{N-k}$ to reconstruct the redundant elements. The algorithm for the conversion is,

```
complex_coefficient[0].real = halfcomplex_coefficient[0];
complex_coefficient[0].imag = 0.0;

for (i = 1; i < n - i; i++)
  {
    const double hc_real = halfcomplex_coefficient[2 * i - 1];
    const double hc_imag = halfcomplex_coefficient[2 * i];
    complex_coefficient[i].real = hc_real;
    complex_coefficient[i].imag = hc_imag;
    complex_coefficient[n - i].real = hc_real;
    complex_coefficient[n - i].imag = -hc_imag;
  }
 if (i == n - i)
  {
    complex_coefficient[i].real = halfcomplex_coefficient[n - 1];
    complex_coefficient[i].imag = 0.0;
  }
```

The function returns a value of `0` if no errors were detected, and `-1` in the case of error. There is only one `gsl_errno` condition defined for this function:

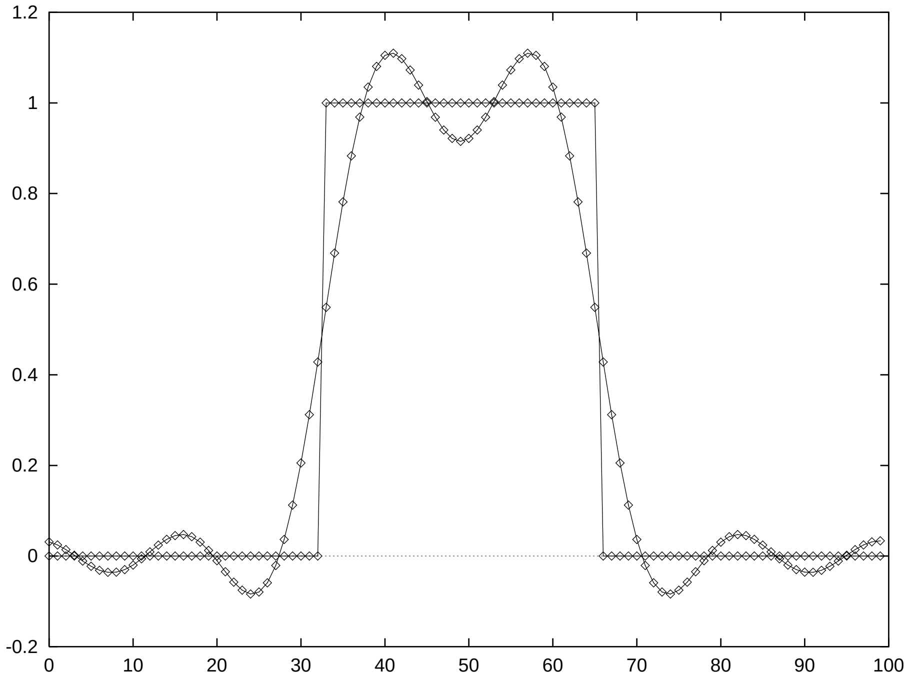
`GSL_EDOM` The length of the data *n* is not a positive integer (i.e. *n* is zero).

6.3.3 Example of using mixed-radix FFT routines for real data

Here is an example program using `gsl_fft_real` and `gsl_fft_halfcomplex`. It generates a real signal in the shape of a square pulse. The pulse is fourier transformed to frequency space, and all but the lowest ten frequency components are removed from the array of fourier coefficients returned by `gsl_fft_real`.

The remaining fourier coefficients are transformed back to the time-domain, to give a filtered version of the square pulse. Since fourier coefficients are stored using the half-complex symmetry both positive and negative frequencies are removed and the final filtered signal is also real.

```
#include <stdio.h>
#include <math.h>
#include <gsl_errno.h>
#include <gsl_fft_real.h>
#include <gsl_fft_halfcomplex.h>

int main ()
{
  int i, n = 100;
  double data[n];

  gsl_fft_real_wavetable * real_wavetable;
  gsl_fft_halfcomplex_wavetable * halfcomplex_wavetable;

  for (i = 0; i < n; i++)
    {
      data[i] = 0.0;
    }

  for (i = n / 3; i < 2 * n / 3; i++)
    {
      data[i] = 1.0;
    }

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e\n", i, data[i]);
    }
  printf ("\n");

  real_wavetable = gsl_fft_real_wavetable_alloc (n);
  gsl_fft_real_init (n, real_wavetable);
  gsl_fft_real (data, n, real_wavetable);
  gsl_fft_real_wavetable_free (real_wavetable);

  for (i = 11; i < n; i++)
    {
      data[i] = 0;
    }

  halfcomplex_wavetable = gsl_fft_halfcomplex_wavetable_alloc (n);
  gsl_fft_halfcomplex_init (n, halfcomplex_wavetable);
  gsl_fft_halfcomplex_inverse (data, n, halfcomplex_wavetable);
  gsl_fft_halfcomplex_wavetable_free (halfcomplex_wavetable);

  for (i = 0; i < n; i++)
    {
      printf ("%d: %e\n", i, data[i]);
    }
}
```



Low-pass filtered version of a real pulse, output from the example program.

6.4 FFT References and Further Reading

A good starting point for learning more about the FFT is the review article *Fast Fourier Transforms: A Tutorial Review and A State of the Art* by Duhamel and Vetterli,

P. Duhamel and M. Vetterli. Fast fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19:259–299, 1990.

To find out about the algorithms used in the GSL routines you may want to consult the latex document *GSL FFT Algorithms* (it is included in GSL, as '`doc/fftalgorithms.tex`'). This has general information on FFTs and explicit derivations of the implementation for each routine. There are also references to the relevant literature. For convenience some of the more important references are reproduced below.

There are several introductory books on the FFT with example programs, such as *The Fast Fourier Transform* by Brigham and *DFT/FFT and Convolution Algorithms* by Burrus and Parks,

E. Oran Brigham. *The Fast Fourier Transform*. Prentice Hall, 1974.

C. S. Burrus and T. W. Parks. *DFT/FFT and Convolution Algorithms*. Wiley, 1984.

Both these introductory books cover the radix-2 FFT in some detail. The mixed-radix algorithm at the heart of the FFTPACK routines is reviewed in Clive Temperton's paper,

Clive Temperton. Self-sorting mixed-radix fast fourier transforms. *Journal of Computational Physics*, 52(1):1–23, 1983.

The derivation of FFTs for real-valued data is explained in the following two articles,

> Henrik V. Sorenson, Douglas L. Jones, Michael T. Heideman, and C. Sidney Burrus. Real-valued fast fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849–863, 1987.
>
> Clive Temperton. Fast mixed-radix real fourier transforms. *Journal of Computational Physics*, 52:340–350, 1983.

In 1979 the IEEE published a compendium of carefully-reviewed Fortran FFT programs in *Programs for Digital Signal Processing*. It is a useful reference for implementations of many different FFT algorithms,

> Digital Signal Processing Committee and IEEE Acoustics, Speech, and Signal Processing Committee, editors. *Programs for Digital Signal Processing*. IEEE Press, 1979.

There is also an annotated bibliography of papers on the FFT and related topics by Burrus,

> C. S. Burrus. Notes on the FFT.

The notes are available from `http://www-dsp.rice.edu/res/fft/fftnote.asc`.

7 Root finding

This chapter describes functions for finding a root of an arbitrary one-dimensional function which you provide. The header file '`gsl_roots.h`' contains prototypes for the root finding functions and related declarations.

7.1 Root Finding Overview

Root finding algorithms can be divided into two classes, those which are guaranteed to converge and those which converge only if started "close enough" to a root. Algorithms with guaranteed convergence start with a bounded region known to contain a root. The size of this bounded region is reduced iteratively until it is reaches a desired tolerance. This provides a rigorous error estimate for the location of the root.

Algorithms without guaranteed convergence sacrifice rigorous error bounds for speed. By approximating the behavior of a function in the vicinity of a root they attempt to find a higher order improvement of an initial guess. This technique is often referred to as "root polishing". When the behavior of the function is known to be compatible with the algorithm and a good initial guess is available, perhaps from a systematic approximation, these algorithms can provide rapid convergence.

Note that root finding functions can only search for one root at a time. When there are several roots in the search area, the first root to be found will be returned; however it is difficult to predict which of the roots this will be. *In most cases, no error will be reported if you try to find a root in an area where there is more than one.*

Care must be taken when a function may have a root of second-order or higher multiplicity (such as $f(x) = (x-c)^2$ or $f(x) = (x-c)^3$). Routines which maintain a strict bound on the root should not have problems with odd-multiplicity roots (e.g. cubic, quintic, ...), since they make use only of the occurrence zero-crossings and not the behavior of the function. However it is impossible to use these routines for even-multiplicity roots because they require a bound on the initial root which guarantees a zero-crossing (below zero at one end of the bound and above zero at the other end). Roots with even-multiplicity do not cross zero, but only touch it instantaneously. In general these functions need to be approached on a case-by-case basis using knowledge of the algorithm to be used. Root polishing algorithms generally work with higher multiplicity roots but with a reduced rate of convergence.

While it is not absolutely required that f have a root within the search region, numerical root finding functions should not be used haphazardly to check for the *existence* of roots. There are better ways to do this! Because it is so easy to create situations where numerical root finders go awry, it is a bad idea to throw a root finder at a function you do not know much about. In general it is best to examine the function visually by plotting before searching for a root.

7.2 Root Finder Exit Values

Since the return value of the root finding functions is reserved for the error status, you must provide storage for the location of the found root.

`double * root` **root** Function Argument

A pointer to a place for the root finder to store the location of the found root. This must be a valid pointer; the root finders will not allocate any memory for you.

If a root finder succeeds, it will return `0` and store the location of the found root in `*root`.

If a root finder fails, it will return `-1` and set `gsl_errno` to a diagnostic value. See Section 7.11 [Root Finder Error Handling], page 90, for a discussion of possible error codes. Nothing useful will be stored in `*root` if the function failed.

7.3 Providing the Function to Search

You must provide a continous function of one variable for the root finder(s) to operate on, and, sometimes, its first derivative.

Recall that when passing pointers to functions, you give the name of the function you are passing. For example:

```
foo = i_take_a_function_pointer(my_function);
```

`double (*` **f**`)(double)` Function Argument

A pointer to the function whose root you are searching for. It is called by the root finding function many times during its search. It must be continous within the region of interest.

Here is an example function which you could pass to a root finder:

```
double
my_f (double x) {
   return sin (2 * x) + 2 * cos (x);
}
```

`double (*` **df**`)(double)` Function Argument

A pointer to the first derivative of the function whose root you are searching for.

If we were looking for a root of the function in the previous example, this is what we would use for `df`:

```
double
my_df (double x) {
   return 2 * cos (2 * x) - 2 * sin (x);
}
```

`void (*` **fdf**`)(double *, double *, double, int, int)` Function Argument

A pointer to a function which calculates both the value of the function under search and the value of its first derivative. Because many terms of a function and its derivative are the same, it is often faster to use this method as opposed to providing $f(x)$ and $f'(x)$ separately. However, it is more complicated.

It stores $f(x)$ in its first argument and $f'(x)$ in its second.

Here's an example where $f(x) = 2\sin(2x)\cos(x)$:

```
void
my_fdf (double * y, double * yprime, double x,
        int y_wanted, int yprime_wanted) {
   double sin2x, cosx;

   sin2x = sin (2 * x);
   cosx = cos (x);

   if (y_wanted)
      *y = 2 * sin2x * cos (x);
   if (yprime_wanted)
      *yprime = 2 * sin2x * -sin (x) + 2 * cos (2 * x) * cosx);
}
```

Low level functions return errors and roots and are provided functions to search in the same manner as the high level functions; see Section 7.2 [Root Finder Exit Values], page 80, and Section 7.3 [Providing the Function to Search], page 81, respectively.

7.4 Search Bounds and Guesses

When using low level functions, you can specify and monitor the region being searched more precisely than you can when using high level functions. You provide either search bounds or one or two guesses; this section explains how search bounds and guesses work and how function arguments control them.

Search bounds are the endpoints of the search interval which is iterated smaller and smaller until the length of the interval is smaller than the requested precision or one of the endpoints converges; a guess is an x value which is iterated around until the it is within the desired precision of a root. Two guesses behave similarly to one; there are just two x values wandering about instead of one.

In low level functions, these arguments are defined as pointers to `double` rather than simply `double`s for two reasons. First, if the root finding function fails, it is very useful to have the final values of your iterated variables available to help diagnose why it failed. Second, it makes it possible to preserve the state of the root finder, enabling it to be restarted in the same place if needed. A situation where this could be useful is if the function under search is very costly to evaluate.

Note that these arguments must be valid pointers; the root finders will not allocate any memory for you.

`double *` **lower_bound** Low Level Function Argument
`double *` **upper_bound** Low Level Function Argument

The initial upper and lower bounds of the interval in which to search for a root. `lower_bound` must be less than `upper_bound`.

These arguments are modified during execution of the root finding function; if you need to preserve their initial values, you must make copies of them. See the third paragraph of this section for the reasoning behind this behavior.

`double * `**guess** Low Level Function Argument
`double * `**guess2** Low Level Function Argument

One or two initial values for the guess(es) iterated by the root finding function.

These arguments are modified during execution of the root finding function; if you need to preserve their initial values, you must make copies of them. See the third paragraph of this section for the reasoning behind this behavior.

7.5 Search Stopping Parameters

The root finding functions (and numerical root finding functions in general) stop when one of the following conditions is true:

- A root has been found to within the user-specified precision.
- A user-specified maximum number of iterations has executed.
- An error has occured.

Whenever you call a low level root finding function, you must specify precisely absolute and/or relative tolerances and the maximum number of iterations.

The stopping criterion decides that two values a and b. with relative tolerance R and absolute tolerance A, are close enough if the following relation is true:

$$|a-b| \leq R\min(|a|,|b|)+A \tag{7.1}$$

You can set either R or A to zero, but be aware that the library will signal an error if the search moves into an area where both R and A are meaningless; assuming a and b are the endpoints of the region of interest, the following must be true an error will be returned:

$$R\min(|a|,|b|)+A \geq 10\max(|a|,|b|)\times \texttt{DBL\_EPSILON} \tag{7.2}$$

(We introduce a buffer of 10 to protect against roundoff error.)

For the sake of efficient resource use, do not ask for more precision than you need, especially if your function is costly to evaluate.

`double `**abs_epsilon** Low Level Function Argument

The maximum permissible absolute error in root finder answers.

The only static limit on `abs_epsilon` is that it must be positive; see above for other restrictions, however.

`double `**rel_epsilon** Low Level Function Argument

The maximum permissible relative error in root finder answers.

`rel_epsilon` must be greater than or equal to `DBL_EPSILON`$\times 10$ (note the buffer factor to protect against roundoff error). See above for additional non-static restrictions.

`unsigned int` **max_iterations** — Function Argument

The maximum number of iterations a root finder is allowed to perform. This must be greater than or equal to 1, as performing a negative number of iterations is extremely difficult and not doing any iterations is rather useless.

Do not set `max_iterations` too large. If there is a problem, you want to know about it as soon as possible; you don't want your program chugging away for many cycles in error.

In addition, the root finding functions which extrapolate (Newton's Method, (Section 7.10 [Newtons Method], page 89, and Secant Method, Section 7.9 [Secant Method], page 88) accept an additional argument:

`double` **max_step_size** — Function Argument

The maximum step size an extrapolating algorithm is allowed to take. This is to prevents the algorithm from landing on a place where the test function's derivative is very small and zooming off to infinity or into a different solution basin.

FIXME: talk about minimum value for max_step_size.

For example, if while solving $\sin(x) = 0$, x_n of Newton's Method (see Section 7.10 [Newtons Method], page 89) landed on 1.570700000 ($\pi/2 \approx 1.570796327$), then x_{n+1} would be approximately -10000, which is definitely not what we wanted! We want the root finder to recognize this step as "too big" and flag an error.

The alarm bell will ring if the following relation in true:

$$|\frac{d}{dx}f(x)| < |\frac{f(x)}{\texttt{max\_step\_size}}| \tag{7.3}$$

Note that while Secant Method (see Section 7.9 [Secant Method], page 88) does not deal with derivatives directly, when extrapolating it approximates them numerically.

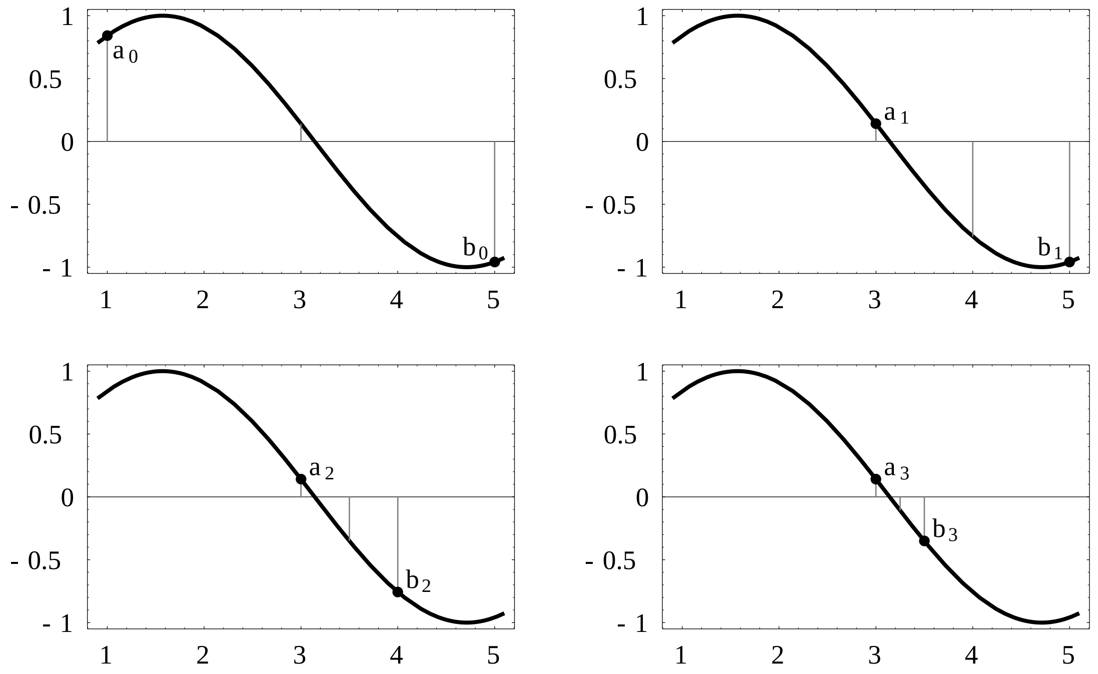
Do not set `max_step_size` too large; that will defeat its purpose. In the $\sin(x) = 0$ example, π would be a good value for `max_step_size`; any step larger than that would certainly be headed astray. A good understanding of the problem is especially important for `max_step_size`.

7.6 Bisection

Bisection is a simple and robust method of finding roots of a function f; when its arguments are valid, it cannot fail. However, it is the slowest algorithm provided by the library, and it cannot find roots of even degree. Its convergence is linear.

One begins the algorithm with an interval which is guaranteed by the Intermediate Value Theorem to contain a root: where a and b are the endpoints of the interval, $f(a)$ must differ in sign from $f(b)$. (If you're a bit fuzzy on the Intermediate Value Theorem, consult any elementary calculus textbook.)

Each iteration, bisection chops its interval in half and discards the interval which does not contain a root. Once the interval is smaller than the requested epsilon, iteration stops and the root location is returned.



Four iterations of bisection, where a_n is nth position of the beginning of the interval and b_n is the nth position of the end. The midpoint of each interval is also indicated.

`int` **gsl_root_bisection** (`double *` *root*, `double (*` *f*`)(double)`, `double *` *lower_bound*, `double *` *upper_bound*, `double` *rel_epsilon*, `double` *abs_epsilon*, `unsigned int` *max_iterations*, `double` *max_deltay*) Function

Search for a zero of `f` using bisection. Returns `0` if successful, `-1` on error (see Section 7.11 [Root Finder Error Handling], page 90).

`f(*lower_bound)` and `f(*upper_bound)` must differ in sign.

Arguments:

`root` A place to store the root location once it is found. See Section 7.2 [Root Finder Exit Values], page 80.

`f` A user defined function to search for a root. See Section 7.3 [Providing the Function to Search], page 81.

`lower_bound`, `upper_bound`
Lower and upper bounds of the interval to search. See Section 7.4 [Search Bounds and Guesses], page 82.

`rel_epsilon`, `abs_epsilon`
Maximum permitted relative and absolute error. See Section 7.5 [Search Stopping Parameters], page 83.

`max_iterations`
: The maximum allowed number of iterations. See Section 7.5 [Search Stopping Parameters], page 83.

`max_deltay`
: The maximum allowed difference between `f(*lower_bound)` and `f(*upper_bound)`. See Section 7.5 [Search Stopping Parameters], page 83.

7.7 Brent-Dekker Method

Brent's method is a simple and robust algorithm of finding roots of a function f; when its arguments are valid, it cannot fail.

One begins the algorithm with an interval which is guaranteed by the Intermediate Value Theorem to contain a root: where a and b are the endpoints of the interval, $f(a)$ must differ in sign from $f(b)$.

Each iteration, brent's method chops its interval in half and discards the interval which does not contain a root. Once the interval is smaller than the requested epsilon, iteration stops and the root location is returned.

`int` **gsl_root_brent** `(double *` *root*`, double (*` *f*`)(double), double *` *lower_bound*`, double *` *upper_bound*`, double` *rel_epsilon*`, double` *abs_epsilon*`, unsigned int` *max_iterations*`)` Function

Search for a zero of `f` using Brent's method. Returns `0` if successful, `-1` on error (see Section 7.11 [Root Finder Error Handling], page 90).

`f(*lower_bound)` and `f(*upper_bound)` must differ in sign.

Arguments:

`root`
: A place to store the root location once it is found. See Section 7.2 [Root Finder Exit Values], page 80.

`f`
: A user defined function to search for a root. See Section 7.3 [Providing the Function to Search], page 81.

`lower_bound`, `upper_bound`
: Lower and upper bounds of the interval to search. See Section 7.4 [Search Bounds and Guesses], page 82.

`rel_epsilon`, `abs_epsilon`
: Maximum permitted relative and absolute error. See Section 7.5 [Search Stopping Parameters], page 83.

`max_iterations`
: The maximum allowed number of iterations. See Section 7.5 [Search Stopping Parameters], page 83.

7.8 False Position

False position is a robust method of finding roots of a function f; if its arguments are valid, it cannot fail. However, it cannot find roots of even degree. Its convergence is linear, but it is usually faster than bisection.

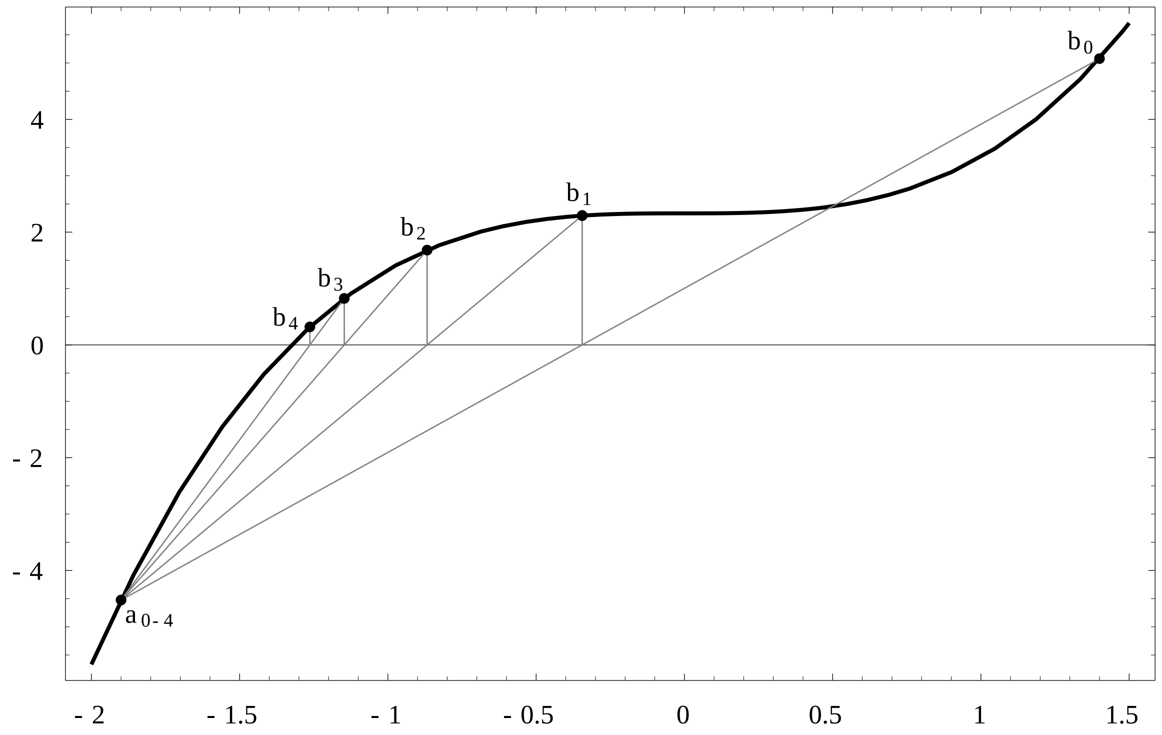
One begins the algorithm with an interval which is guaranteed by the Intermediate Value Theorem to contain a root: where a and b are the endpoints of the interval, $f(a)$ must differ in sign from $f(b)$. (If you're a bit fuzzy on the Intermediate Value Theorem, consult any elementary calculus textbook.)

Each iteration, false position draws a line between $f(a)$ and $f(b)$; the x position where this line crosses the x axis is where the interval is split. The part of the interval which contains the root is taken to be the new interval, and the process is repeated until one of the following is true:

$$|a - b| \leq \varepsilon \tag{7.4}$$

$$a_n - a_{n-1} = 0 \quad \text{and} \quad |b_n - b_{n-1}| \leq \varepsilon \tag{7.5}$$

$$b_n - b_{n-1} = 0 \quad \text{and} \quad |a_n - a_{n-1}| \leq \varepsilon \tag{7.6}$$



Several iterations of false position, where a_n is nth position of the beginning of the interval and b_n is the nth position of the end.

`int` **gsl_root_falsepos** (`double *` *root*, `double (*` *f*`)(double)`, `double *` *lower_bound*, `double *` *upper_bound*, `double` *rel_epsilon*, `double` *abs_epsilon*, `unsigned int` *max_iterations*, `double` *max_deltay*) Function

Search for a zero of `f` using false position. Return `0` if successful, `-1` on error (see Section 7.11 [Root Finder Error Handling], page 90.

`f(*lower_bound)` and `f(*upper_bound)` must differ in sign.

Arguments:

`root` A place to store the root location once it is found. See Section 7.2 [Root Finder Exit Values], page 80.

`f` A user defined function to search for a root. See Section 7.3 [Providing the Function to Search], page 81.

`lower_bound`, `upper_bound`
Lower and upper bounds of the interval to search. See Section 7.4 [Search Bounds and Guesses], page 82.

`rel_epsilon`, `abs_epsilon`
Maximum permitted relative and absolute error. See Section 7.5 [Search Stopping Parameters], page 83.

`max_iterations`
The maximum allowed number of iterations. See Section 7.5 [Search Stopping Parameters], page 83.

`max_deltay`
The maximum allowed difference between `f(*lower_bound)` and `f(*upper_bound)`. See Section 7.5 [Search Stopping Parameters], page 83.

7.9 Secant Method

Secant Method is a somewhat fragile method of finding roots. On single roots, its convergence is of order $(1+\sqrt{5})/2$ (approximately 1.62). On multiple roots, converges linearly.

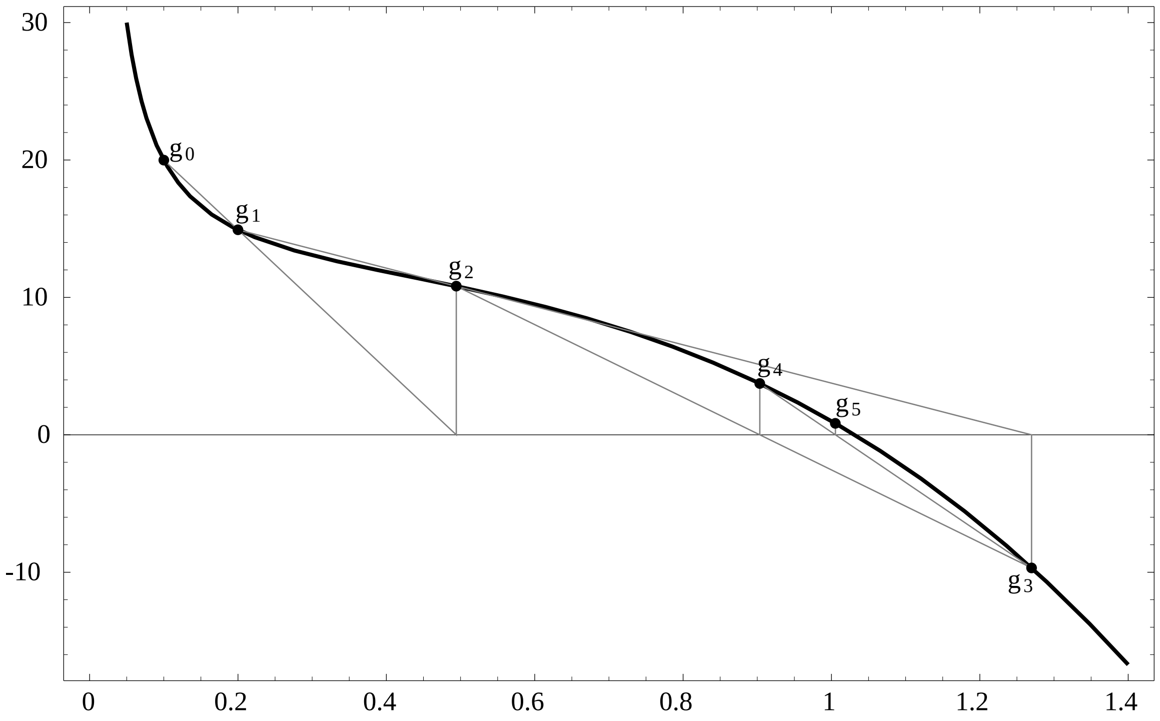
One begins the algorithm with two guesses for the value of the root, g_0 and g_1. The root may be either inside or outside the interval defined by g_0 and g_1.

Each iteration, Secant Method draws a line through $f(g_{n-1})$ and $f(g_n)$. The x position where this line crosses the x axis becomes g_{n+1}. g_{n-1} is discarded, n is incremented, and the process is repeated until:

$$|g_n - g_{n-1}| \leq \varepsilon \tag{7.7}$$

Note that g_{n+1} may be obtained by either interpolation or extrapolation and that Secant Method cannot fail during interpolation.

Secant Method breaks in the same situations that Newton's Method does, though it is somewhat less sensitive. (See Section 7.10 [Newtons Method], page 89.)



Several iterations of Secant Method, where g_n is the nth guess.

int **gsl_root_secant** (`double *` *root*, `double (*` *f*`)(double)`, `double *` *guess*, `double *` *guess2*, `double` *epsilon*, `unsigned int` *max_iterations*, `double` *max_step_size*) Function

Search for a zero of `f` using Secant Method, with `*guess` and `*guess2` being the guesses.

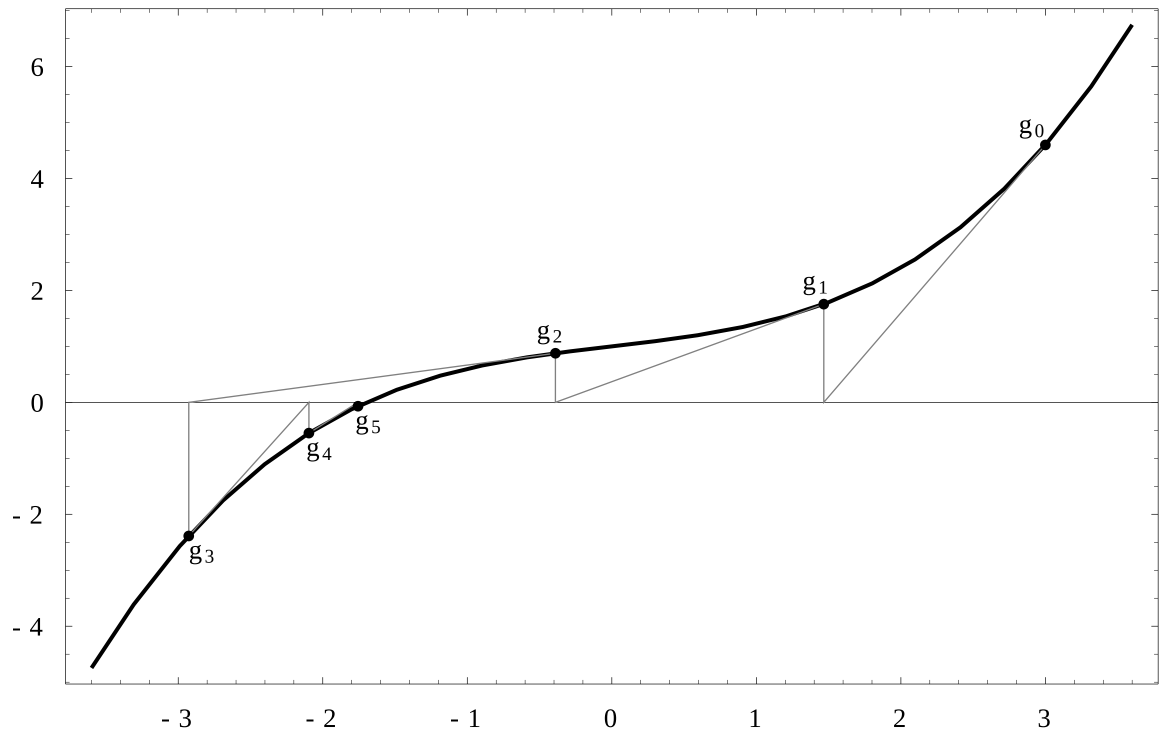
arguments. See Section 7.11 [Root Finder Error Handling], page 90, for a discussion of possible error codes.

7.10 Newtons Method

Newton's Method is a fast but somewhat fragile method of finding roots. On single roots, it converges quadratically; however, on multiple roots it converges linearly.

One begins the algorithm with one guess g for the value of the root. Each iteration, Newton's Method draws a line tangent to f (the function whose root you are searching for); the x position where this line crosses the x axis becomes the new g. The process is repeated until:

$$|g_n - g_{n-1}| \leq \varepsilon \qquad (7.8)$$



Several iterations of Newton's Method, where g_n is the nth guess.

`int` **gsl_root_newton** (`double *` *root*, `void (*` *fdf*`)(double *, double *, double, int, int)`, `double *` *guess*, `double` *epsilon*, `unsigned int` *max_iterations*, `double` *max_step_size*) Function

Search for a zero of `f` using Newton's Method, with `*guess` being the guess.

See Section 7.11 [Root Finder Error Handling], page 90, for a discussion of possible error codes.

7.11 Root Finder Error Handling

When successful, the root finding functions return `0`; on error, they return `-1` and set the global variable `gsl_errno` to a diagnostic value. (See Chapter 3 [Error handling in GSL], page 3, for a general discussion of GSL error handling.)

When using low-level functions, you can examine `*guess`, `*guess2`, `*lower_bound`, or `*upper_bound` (see Section 7.4 [Search Bounds and Guesses], page 82) to help determine why a root finder failed.

The root finders can set `gsl_errno` to the following macros. Some errors can only be encountered by low level functions; they are marked by *[Low Level Only]*.

`GSL_EINVAL`

One or more of the input arguments is invalid because at least one of these conditions is true:

- *[Low Level Only]* `max_iterations` is equal to 0. See Section 7.5 [Search Stopping Parameters], page 83.
- The lower bound of the search interval is not less than the upper bound.
- You supplied a pointer argument which was null.

- f(*lower_bound*) and f(*upper_bound*) do not differ in sign, and the function that you are using requires that they do.

`GSL_EBADFUNC`
: The function under search (or its derivative) did not return a valid number when it was called by the the root finder. (Instead, it returned `NAN` or `INF`.)

`GSL_ERUNAWAY`
: *[Low Level Only]* A root finder tried to take a step larger than `max_step_size` (see Section 7.5 [Search Stopping Parameters], page 83). This happens when an extrapolating algorithm lands on a place where the derivative is too small.

`GSL_EMAXITER`
: The number of iterations executed exceeded `max_iterations` (see Section 7.5 [Search Stopping Parameters], page 83).

`GSL_EBADTOL`
: *[Low Level Only]* You specified an invalid error tolerance in one or more of the following ways (see Section 7.5 [Search Stopping Parameters], page 83):

  - `rel_epsilon` was too small.
  - `rel_epsilon` and `abs_epsilon` were both zero.
  - `rel_epsilon` was zero, and the search converged to a place too far from zero for `abs_epsilon` to be useful.
  - `abs_epsilon` was zero, and the search converged to a place too close to zero for `rel_epsilon` to be useful.
  - **FIXME: add stuff for delta.**

`GSL_EZERODIV`
: *[Low Level Only]* A function detected that any further iterations would result in division by zero. This most often happens when Newton's Method (see Section 7.10 [Newtons Method], page 89) or Secant Method (see Section 7.9 [Secant Method], page 88) lands on an extremum.

8 Special Functions

This chapter describes the GSL special function library.

8.1 Airy Functions

The Airy functions $Ai(x)$ and $Bi(x)$ are defined by the integral representations

$$Ai(x) = \frac{1}{\pi}\int_0^\infty \cos(\frac{1}{3}t^3 + xt)dt, \tag{8.1}$$

$$Bi(x) = \frac{1}{\pi}\int_0^\infty (e^{-t^3/3} + \sin(\frac{1}{3}t^3 + xt)dt. \tag{8.2}$$

They are linearly independent solutions of the equation $f(x)'' = xf(x)$.

Being functions of a single variable, they are quite easy to evaluate in production code, using fits to Chebyshev polynomials in various regions. Such fits provide a uniform approximation over the full domain of the function. The GSL implementation is a re-implementation of the Airy Chebyshev fits in the SLATEC Fortran library.

`double` **gsl_sf_airy_Ai(double** `x`**)** Function
`double` **gsl_sf_airy_Bi(double** `x`**)** Function
`double` **gsl_sf_airy_Bi_scaled(double** `x`**)** Function

8.2 Bessel Functions

8.3 Chebyshev Polynomials

The Chebyshev polynomials $T_n(x) = \cos(n \arccos x)$ provide an orthogonal basis of polynomials on the interval $[-1, 1]$, with the weight function $\frac{1}{\sqrt{1-x^2}}$. The first few such polynomials are

$$T_0(x) = 1, \tag{8.3}$$

$$T_1(x) = x, \tag{8.4}$$

$$T_2(x) = 2x^2 - 1. \tag{8.5}$$

By construction, $T_n(x)$ has precisely n zeroes in the interval $[-1, 1]$, located at $x_k^{(n)} = \cos(\frac{\pi}{n}(k-1/2))$. It is the nature of these zeroes and the behaviour of the set of polynomials at these special points which makes the Chebyshev polynomials especially useful in numerical approximation of functions.

8.4 Coulomb Wave Functions

8.5 Dilogarithm

8.6 Error Function

8.7 Fermi-Dirac Function

8.8 Gamma Function

8.9 Laguerre Functions

8.10 Legendre Functions and Spherical Harmonics

8.11 Logarithm (Complex)

8.12 Power Function

A common complaint about the standard C library is its lack of a function for calculating (small) integer powers. GSL provides a simple function to fill this gap.

`double` **gsl_sf_pow_int(double** `x`, `int n`) Function

```
#include <gsl_sf_pow_int.h>
double y = gsl_sf_pow_int(3., 12)
```

8.13 Psi (DiGamma) Function

8.14 Trigonometric Functions (Complex)

9 Series Acceleration

The functions described in this chapter accelerate the convergence of a series using the Levin u-transform. This method takes a small number of terms from the start of a series and uses a systematic approximation to compute an extrapolated value and an estimate of its error. The u-transform works for both convergent and divergent series, including asymptotic series.

These functions are declared in the header file '`gsl_sum.h`'.

9.1 Acceleration functions

`int` **gsl_sum_levin_u** (`const double *` *array*, `size_t` *array_size*, `double *` *q_num*, `double *` *q_den*, `double *` *dq_num*, `double *` *dq_den*, `double *` *dsum*, `double *` *sum_accel*, `double *` *sum_plain*, `double *` *precision*) Function

This function takes the terms of a series in *array* of size *array_size* and computes the extrapolated limit of the series using a Levin u-transform. The extrapolated sum is stored in *sum_accel*, with an estimate of the relative error stored in *precision*. The actual term-by-term sum is returned in *sum_plain*. Additional working space must be provided in the arrays *q_num*, *q_den*, *dsum* of size *array_size* elements each and in *dq_num*, *dq_den* of size *array_size***2. The algorithm estimates both truncation error and round-off error to choose an optimal number of terms for the extrapolation.

9.2 Example of accelerating a series

The following code calculates an estimate of $\zeta(2) = \pi^2/6$ using the series,

$$\zeta(2) = 1 + 1/2^2 + 1/3^2 + 1/4^2 + \dots \tag{9.1}$$

After N terms the error in the sum is $O(1/N)$, making direct summation of the series converge slowly.

```
#include <stdio.h>
#include <gsl_math.h>
#include <gsl_sum.h>

#define N 20

int
main (void)
{
  double t[N], qnum[N], qden[N], dsum[N], dqnum[N * N], dqden[N * N];
  double sum_accel, sum_plain, prec;
  double sum = 0;
  size_t n_used ;
  int n;
```

```
    const double zeta_2 = M_PI * M_PI / 6.0;

    /* terms for zeta(2) = \sum_{n=0}^{\infty} 1/n^2 */

    for (n = 0; n < N; n++)
      {
        double np1 = n + 1.0;
        t[n] = 1.0 / (np1 * np1);
        sum += t[n] ;
      }

    gsl_sum_levin_u_accel (t, N, qnum, qden, dqnum, dqden, dsum,
   &sum_accel, &n_used, &sum_plain, &prec);

    printf("term-by-term sum = %.16f using %d terms\n", sum, N) ;
    printf("term-by-term sum = %.16f using %d terms\n", sum_plain, n_used) ;
    printf("accelerated sum  = %.16f using %d terms\n", sum_accel, n_used) ;
    printf("exact value      = %.16f\n", zeta_2) ;
    printf("estimated error  = %.16f\n", prec * sum_accel) ;
    printf("actual error     = %.16f\n", sum_accel - zeta_2) ;

    return 0;
  }
```

The Levin u-transform is able to obtain an estimate of the sum to 1 part in 10^{10} using the first eleven terms of the series. The error estimate returned by the function safely bounds the correct value and is conservatively large. By comparison a direct summation would require 10^{10} terms to achieve the same precision.

```
bjg|vvv> ./a.out
term-by-term sum = 1.5961632439130233 using 20 terms
term-by-term sum = 1.5649766384209025 using 11 terms
accelerated sum  = 1.6449340668936467 using 11 terms
exact value      = 1.6449340668482264
estimated error  = 0.0000000002517837
actual error     = 0.0000000000454203
```

9.3 Series Acceleration References

The algorithms used by these functions are described in the following papers,

T. Fessler, W.F. Ford, D.A. Smith, HURRY: An acceleration algorithm for scalar sequences and series *ACM Transactions on Mathematical Software*, 9(3):346–354, 1983. and Algorithm 602 9(3):355–357, 1983.

The theory of the u-transform was presented by Levin,

D. Levin, Development of Non-Linear Transformations for Improving Covergence of Sequences, *Intern. J. Computer Math.* B3:371–388, 1973

10 Simulated Annealing

Stochastic search techniques are used when the structure of a space is not well understood or is not smooth, so that techniques like Newton's method (which requires calculating Jacobian derivative matrices) cannot be used.

In particular, these techniques are frequently used to solve *combinatorial optimization* problems, such as the traveling salesman problem.

The basic problem layout is that we are looking for a point in the space at which a real valued *energy function* (or *cost function*) is minimized.

Simulated annealing is a technique which has given good results in avoiding local minima; it is based on the idea of taking a random walk through the space at successively lower temperatures, where the probability of taking a step is given by a Boltzmann distribution.

10.1 Simulated Annealing algorithm

We take random walks through the problem space, looking for points with low energies; in these random walks, the probability of taking a step is determined by the Boltzmann distribution

$$p = e^{-(E_{i+1}-E_i)/(kT)} \tag{10.1}$$

if $E_{i+1} < E_i$, and $p = 0$ when $E_{i+1} \geq E_i$.

In other words, a step *will* occur if the new energy is lower. If the new energy is higher, the transition can still occur, and its likelyhood is proportional to the temperature T and inversely proportional to the energy difference $E_{i+1} - E_i$.

The temperature T is initially set to a high value, and a random walk is carried out at that temperature. Then the temperature is lowered very slightly (according to a *cooling schedule*) and another random walk is taken.

This slight probability of taking a step that gives *higher* energy is what allows simulated annealing to frequently get out of local minima.

An initial guess is supplied. At each step, a point is chosen at a random distance from the current one, where the random distance r is distributed according to a Boltzmann distribution $r = \exp^{-E/kT}$. After a few search steps using this distribution, the temperature T is lowered according to some scheme, for example $T \to T/\mu_T$ where mu_T is slightly greater than 1.

10.2 Simulated Annealing functions

gsl_Efunc_t Simulated annealing

```
typedef double (*gsl_Efunc_t) (void *xp);
```

gsl_siman_step_t Simulated annealing

```
typedef void (*gsl_siman_step_t) (void *xp, double step_size);
```

gsl_siman_metric_t Simulated annealing

```
typedef double (*gsl_siman_metric_t) (void *xp, void *yp);
```

gsl_siman_print_t Simulated annealing

```
typedef void (*gsl_siman_print_t) (void *xp);
```

gsl_siman_params_t Simulated annealing

These are the parameters that control a run of `gsl_siman_solve`.

```
/* this structure contains all the information needed to structure
   the search, beyond the energy function, the step function and the
   initial guess. */
struct s_siman_params {
  int n_tries;          /* how many points to try for each step */
  int iters_fixed_T; /* how many iterations at each temperature? */
  double step_size; /* max step size in the random walk */
  /* the following parameters are for the Boltzmann distribution */
  double k, t_initial, mu_t, t_min;
};

typedef struct s_siman_params gsl_siman_params_t;
```

`void` **gsl_siman_solve** (`void *`*x0_p*, `gsl_Efunc_t` *Ef*, `gsl_siman_metric_t` *distance*, `gsl_siman_print_t` *print_position*, `size_t` *element_size*, `gsl_siman_params_t params`) Simulated annealing

Does a *simulated annealing* search through a given space. The space is specified by providing the functions *Ef*, *distance*, *print_position*, *element_size*.

The *params* structure (described above) controls the run by providing the temperature schedule and other tunable parameters to the algorithm (see Section 10.1 [Simulated Annealing algorithm], page 96). p The result (optimal point in the space) is placed in `*`*x0_p*.

If *print_position* is not null, a log will be printed to the screen with the following columns:

```
number_of_iterations temperature x x-(*x0_p) Ef(x)
```

If *print_position* is null, no information is printed to the screen.

10.3 Examples with Simulated Annealing

GSL's Simulated Annealing package is clumsy, and it has to be because it is written in C, for C callers, and tries to be polymorphic at the same time. But here we provide some examples which can be pasted into your application with little change and should make things easier.

10.3.1 Trivial example

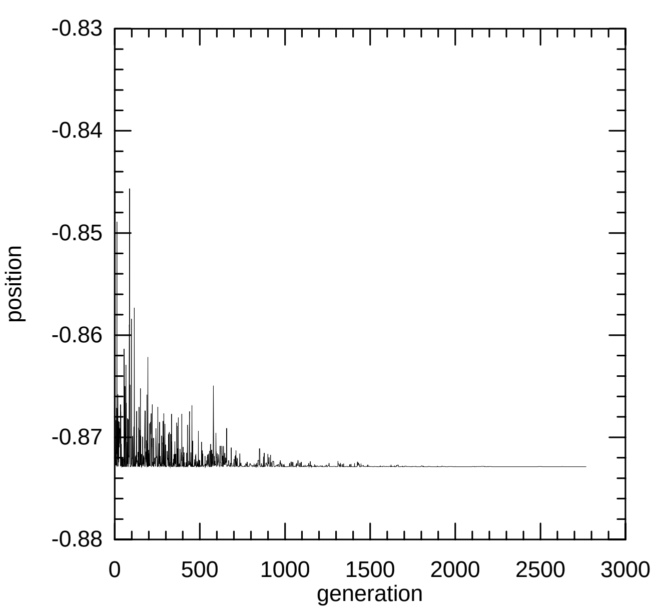
The first example, in one dimensional cartesian space, sets up an energy function which is a damped sine wave; this has many local minima, but only one global minimum, somewhere between 1.0 and 1.5. The initial guess given is 15.5, which is several local minima away from the global minimum.

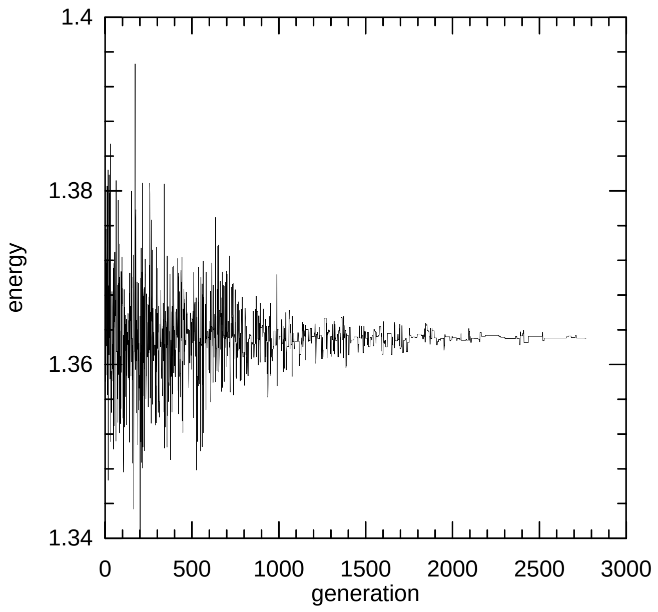
```
/* set up parameters for this simulated annealing run */
#define N_TRIES 200 /* how many points do we try before stepping */
#define ITERS_FIXED_T 10 /* how many iterations for each T? */
#define STEP_SIZE 10 /* max step size in random walk */
#define K 1.0 /* Boltzmann constant */
#define T_INITIAL 0.002 /* initial temperature */
#define MU_T 1.005 /* damping factor for temperature */
#define T_MIN 2.0e-6

gsl_siman_params_t params = {N_TRIES, ITERS_FIXED_T, STEP_SIZE,
     K, T_INITIAL, MU_T, T_MIN};

/* now some functions to test in one dimension */
double E1(void *xp)
{
  double x = * ((double *) xp);

  return exp(-square(x-1))*sin(8*x);
}

double M1(void *xp, void *yp)
{
  double x = *((double *) xp);
  double y = *((double *) yp);

  return fabs(x - y);
}

void S1(void *xp, double step_size)
{
  double r;
  double old_x = *((double *) xp);
  double new_x;

  r = gsl_ran_uniform();
  new_x = r*2*step_size - step_size + old_x;

  memcpy(xp, &new_x, sizeof(new_x));
}

void P1(void *xp)
{
  printf("%12g", *((double *) xp));
}

int main(int argc, char *argv[])
{
  Element x0; /* initial guess for search */
```

```
    double x_initial = 15.5;

    gsl_siman_solve(&x_initial, E1, S1, M1, P1, sizeof(double), params);
    return 0;
  }
```

Here are a couple of plots that are generated by running `siman_test` in the following way:

```
./siman\_test | grep -v "^#" | xyplot -xyil -y -0.88 -0.83 -d "x...y" | xyps -d > sima
./siman\_test | grep -v "^#" | xyplot -xyil -xl "generation" -yl "energy" -d "x..y" | 
```





Example of a simulated annealing run: at higher temperatures (early in the plot) you see that the solution can fluctuate, but at lower temperatures it converges.

10.3.2 Traveling Salesman Problem

The TSP (*Traveling Salesman Problem*) is the classic combinatorial optimization problem. I have provided a very simple version of it, based on the coordinates of twelve cities in the southwestern United States. This should maybe be called the *Flying Salesman Problem*, since I am using the great-circle distance between cities, rather than the driving distance. Also: I assume the earth is a sphere, so I don't use geoid distances.

The `gsl_siman_solve()` routine finds a route which is 3490.62 Kilometers long; this is confirmed by an exhaustive search of all possible routes with the same initial city.

The full code can be found in '`siman/siman_tsp.c`', but I include here some plots generated with in the following way:

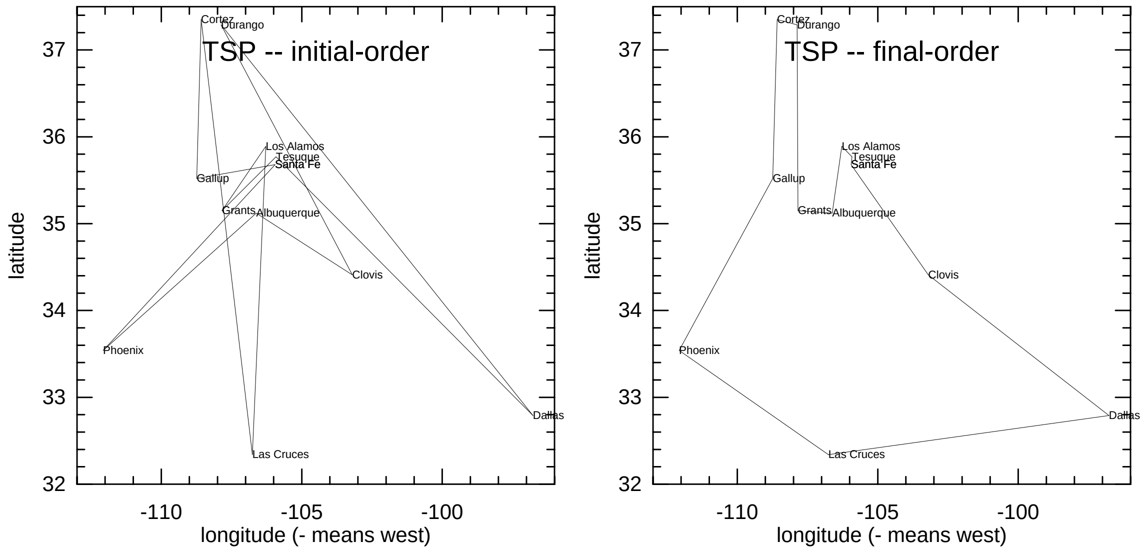
```
./siman_tsp > tsp.output
grep -v "^#" tsp.output  | xyplot -xyil -d "x...............y" -lx "generation" -ly "
grep initial_city_coord tsp.output | awk '{print $2, $3, $4, $5}' | xyplot -xyil -lb0 
grep final_city_coord tsp.output | awk '{print $2, $3, $4, $5}' | xyplot -xyil -lb0 -c
```

This is the output showing the initial order of the cities; longitude is negative, since it is west and I want the plot to look like a map.

```
# initial coordinates of cities (longitude and latitude)
###initial_city_coord: -105.95 35.68 Santa Fe
###initial_city_coord: -112.07 33.54 Phoenix
###initial_city_coord: -106.62 35.12 Albuquerque
###initial_city_coord: -103.2 34.41 Clovis
###initial_city_coord: -107.87 37.29 Durango
###initial_city_coord: -96.77 32.79 Dallas
###initial_city_coord: -105.92 35.77 Tesuque
###initial_city_coord: -107.84 35.15 Grants
###initial_city_coord: -106.28 35.89 Los Alamos
###initial_city_coord: -106.76 32.34 Las Cruces
###initial_city_coord: -108.58 37.35 Cortez
###initial_city_coord: -108.74 35.52 Gallup
###initial_city_coord: -105.95 35.68 Santa Fe
```

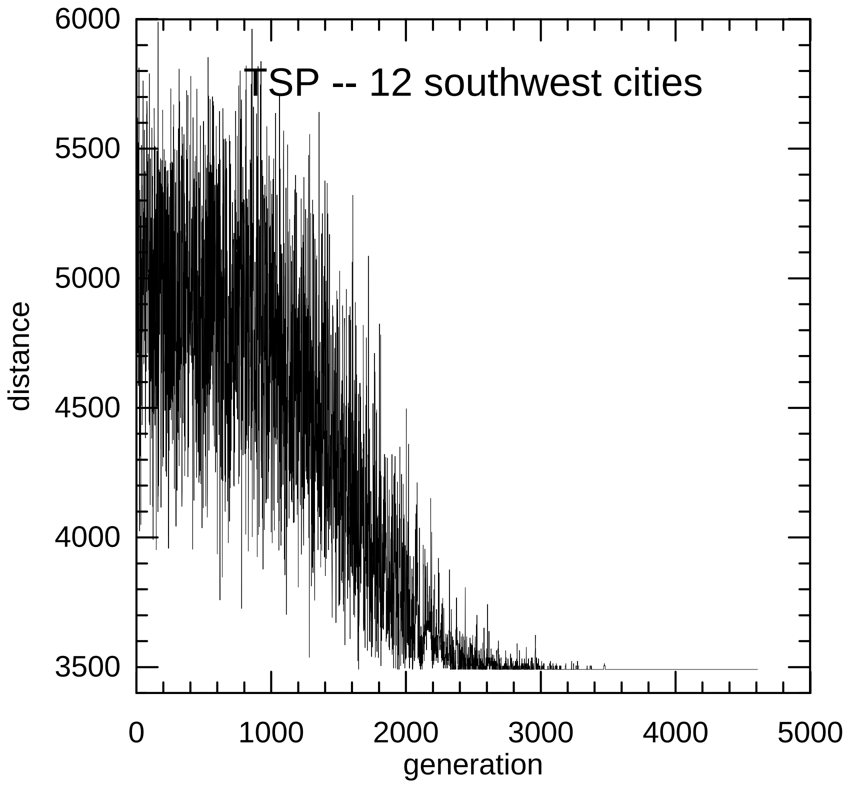
The optimal route turns out to be:

```
# final coordinates of cities (longitude and latitude)
###final_city_coord: -105.95 35.68 Santa Fe
###final_city_coord: -106.28 35.89 Los Alamos
###final_city_coord: -106.62 35.12 Albuquerque
###final_city_coord: -107.84 35.15 Grants
###final_city_coord: -107.87 37.29 Durango
###final_city_coord: -108.58 37.35 Cortez
###final_city_coord: -108.74 35.52 Gallup
###final_city_coord: -112.07 33.54 Phoenix
###final_city_coord: -106.76 32.34 Las Cruces
###final_city_coord: -96.77 32.79 Dallas
###final_city_coord: -103.2 34.41 Clovis
###final_city_coord: -105.92 35.77 Tesuque
###final_city_coord: -105.95 35.68 Santa Fe
```



Initial and final (optimal?) route for the 12 southwestern cities Flying Salesman Problem.

Here's a plot of the cost function (energy) versus generation (point in the calculation at which a new temperature is set) for this problem:



Example of a simulated annealing run for the 12 southwestern cities Flying Salesman Problem.

11 Vectors and Matrices

The functions described in this chapter provide a simple vector and matrix interface to ordinary C arrays. By writing your functions in terms of vectors and matrices you can pass a single structure containing both data and dimensions without needing additional function arguments.

11.1 The vector struct

Vectors are defined by a `gsl_vector` structure which contains two members, the size and a pointer to a block of memory where the elements of the vector are stored. The `gsl_vector` structure is very simple and looks like this,

```
typedef struct
{
  size_t size;
  double * data;
} gsl_vector ;
```

The library also defines three other types of vectors, for single-precision floating point numbers, complex numbers of type `gsl_complex` and for integers which have the names `gsl_vector_float`, `gsl_vector_complex` and `gsl_vector_int`.

11.2 Vector allocation

The functions for allocating memory to a vector follow the style of `malloc` and `free`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a vector then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every `alloc`.

`gsl_vector *` **gsl_vector_alloc** (`size_t` *n*) Function

This function allocates memory for a vector and its *n* elements, returning a pointer to a newly initialized vector struct. The elements of the vector are not initialized and so their values are undefined. Use the function `gsl_vector_calloc` if you want to ensure that all the elements are initialized to zero.

A null pointer is returned if insufficient memory is available to store the vector.

`gsl_vector *` **gsl_vector_calloc** (`size_t` *n*) Function

This function allocates memory for a vector and initializes all the elements of the vector to zero.

`void` **gsl_vector_free** (`gsl_vector *` *v*) Function

This function frees the memory used by a vector *v* previously allocated with `gsl_vector_alloc` or `gsl_vector_calloc`.

The following functions perform the same tasks as the functions above for single-precision and integer vectors.

`gsl_vector_float *` **gsl_vector_float_alloc** (`size_t` *n*) Function
`gsl_vector_float *` **gsl_vector_float_calloc** (`size_t` *n*) Function
`void` **gsl_vector_float_free** (`gsl_vector_float *` *v*) Function

These functions perform memory management for single-precision floating point vectors defined with the struct `gsl_vector_float`.

`gsl_vector_complex *` **gsl_vector_complex_alloc** (`size_t` *n*) Function
`gsl_vector_complex *` **gsl_vector_complex_calloc** (`size_t` *n*) Function
`void` **gsl_vector_complex_free** (`gsl_vector_complex *` *v*) Function

These functions perform memory management for complex vectors defined with the struct `gsl_vector_complex`.

`gsl_vector_int *` **gsl_vector_int_alloc** (`size_t` *n*) Function
`gsl_vector_int *` **gsl_vector_int_calloc** (`size_t` *n*) Function
`void` **gsl_vector_int_free** (`gsl_vector_int *` *v*) Function

These functions perform memory management for integer vectors defined with the struct `gsl_vector_int`.

11.3 Accessing vector elements

Unlike FORTRAN, the C language does not provide support for range checking of vectors and matrices. However, the functions `gsl_vector_get` and `gsl_vector_set` can perform range checking for you and report an error if you attempt to access elements outside the allowed range.

The functions for accessing the elements of a vector or matrix are defined in '`gsl_vector.h`' and declared `extern inline` to eliminate function-call overhead. If necessary you can turn off range checking completely without modifying any source files by recompiling your program with the preprocessor definition `GSL_RANGE_CHECK_OFF`. Provided your compiler supports inline functions the effect of turning off range checking is to replace calls to `gsl_vector_get(v,i)` by `v->data[i]` and and calls to `gsl_vector_set(v,i,x)` by `v->data[i] = x`. Thus there should be no performance penalty at all for using the library functions when range checking is turned off.

`double` **gsl_vector_get** (`const gsl_vector *` *v*, `size_t` *i*) Function

This function returns the *i*th element of a vector *v*. If *i* lies outside the allowed range of 0 to *n-1* then the error handler is invoked and 0 is returned.

`void` **gsl_vector_set** (`gsl_vector *` *v*, `size_t` *i*, `double` *x*) Function

This function sets the value of the *i*th element of a vector *v* to *x*. If *i* lies outside the allowed range of 0 to *n-1* then the error handler is invoked.

The following functions perform the same tasks for vectors of the type `gsl_vector_float` and `gsl_vector_int`.

`float` **gsl_vector_float_get** (`const gsl_vector_float *` *v*, `size_t` *i*) Function

`void` **gsl_vector_float_set** (`gsl_vector_float *` *v*, `size_t` *i*, `float` *x*) Function

These functions access the elements of a single-precision vector.

`gsl_complex` **gsl_vector_complex_get** (`const gsl_vector_complex *` *v*, `size_t` *i*) Function

`void` **gsl_vector_complex_set** (`gsl_vector_complex *` *v*, `size_t` *i*, `gsl_complex` *x*) Function

These functions access the elements of a complex vector.

`int` **gsl_vector_int_get** (`const gsl_vector_int *` *v*, `size_t` *i*) Function

`void` **gsl_vector_int_set** (`gsl_vector_int *` *v*, `size_t` *i*, `int` *x*) Function

These functions access the elements of an integer vector.

11.4 Reading and writing vectors

The library provides functions for reading and writing vectors to a file as binary data or formatted text.

`int` **gsl_vector_fwrite** (`FILE *` *stream*, `const gsl_vector *` *v*) Function

This function writes the elements of the vector *v* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

`int` **gsl_vector_fread** (`FILE *` *stream*, `gsl_vector *` *v*) Function

This function reads into the vector *v* from the open stream *stream* in binary format. The vector *v* must be preallocated with the correct length since the function uses the size of *v* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

`int` **gsl_vector_fprintf** (`FILE *` *stream*, `const gsl_vector *` *v*, `const char *` *format*) Function

This function writes the elements of the vector *v* line-by-line to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats for floating point numbers and `%d` for integers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file.

`int` **gsl_vector_fscanf** (`FILE *` *stream*, `gsl_vector *` *v*) Function

This function reads formatted data from the stream *stream* into the vector *v*. The vector *v* must be preallocated with the correct length since the function uses the size of *v* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

The following functions read and write single-precision and integer vectors with the types `gsl_vector_float` and `gsl_vector_int`.

`int` **gsl_vector_float_fwrite** (`FILE *` *stream*, `const gsl_vector_float *` *v*) Function

`int` **gsl_vector_float_fread** (`FILE *` *stream*, `gsl_vector_float *` *v*) Function

`int` **gsl_vector_float_fprintf** (`FILE *` *stream*, `const gsl_vector_float *` *v*, `const char *` *format*) Function

`int` **gsl_vector_float_fscanf** (`FILE *` *stream*, `gsl_vector_float *` *v*) Function

These functions read and write single-precision vectors as binary data or formatted text.

`int` **gsl_vector_complex_fwrite** (`FILE *` *stream*, `const gsl_vector_complex *` *v*) Function

`int` **gsl_vector_complex_fread** (`FILE *` *stream*, `gsl_vector_complex *` *v*) Function

`int` **gsl_vector_complex_fprintf** (`FILE *` *stream*, `const gsl_vector_complex *` *v*, `const char *` *format*) Function

`int` **gsl_vector_complex_fscanf** (`FILE *` *stream*, `gsl_vector_complex *` *v*) Function

These functions read and write complex vectors as binary data or formatted text.

`int` **gsl_vector_int_fwrite** (`FILE *` *stream*, `const gsl_vector_int *` *v*) Function

`int` **gsl_vector_int_fread** (`FILE *` *stream*, `gsl_vector_int *` *v*) Function

`int` **gsl_vector_int_fprintf** (`FILE *` *stream*, `const gsl_vector_int *` *v*, `const char *` *format*) Function

`int` **gsl_vector_int_fscanf** (`FILE *` *stream*, `gsl_vector_int *` *v*) Function

These functions read and write integer vectors as binary data or formatted text.

11.5 Example programs for vectors

This program shows how to allocate, initialize and read from a vector using the functions `gsl_vector_alloc`, `gsl_vector_set` and `gsl_vector_get`.

```
#include <stdio.h>
#include <gsl_vector.h>

int main ()
{
  int i;
  gsl_vector * v = gsl_vector_alloc (3) ;
  for (i = 0; i < 3; i++)
    {
      gsl_vector_set (v, i, 1.23 + i);
    }

  for (i = 0; i < 100; i++)
    {
      printf("v_%d = %g\n", i, gsl_vector_get (v, i));
    }
}
```

Here is the output from the program. The final loop attempts to read outside the range of the vector `v`, and the error is trapped by the range-checking code in `gsl_vector_get`.

```
v_0 = 1.23
v_1 = 2.23
v_2 = 3.23
gsl: vector_source.c:12: ERROR: index out of range
IOT trap/Abort (core dumped)
```

The next program shows how to write a vector to a file.

```
#include <stdio.h>
#include <gsl_vector.h>

int main ()
{
  int i;
  gsl_vector * v = gsl_vector_calloc (100) ;

  for (i = 0; i < 100; i++)
    {
      gsl_vector_set (v, i, 1.23 + i);
    }

  {
     FILE * f = fopen("test.dat", "w") ;
     gsl_vector_fprintf (f, v, "%.5g");
     fclose (f);
  }
}
```

After running this program the file '`test.dat`' should contain the elements of `v`, written using the format specifier `%.5g`. The vector could then be read back in using the function `gsl_vector_fscanf (f, v)`.

11.6 The matrix struct

Matrices are defined by a `gsl_matrix` structure which contains three members, the two dimensions of the matrix and a pointer to a block of memory where the elements of the matrix are stored. The `gsl_matrix` structure is very simple and looks like this,

```
typedef struct
{
  size_t size1;
  size_t size2;
  double * data;
} gsl_matrix ;
```

The library also defines three other types of matrices, for single-precision floating point numbers, complex numbers of type `gsl_complex` and integers which have the names `gsl_matrix_float`, `gsl_matrix_complex` and `gsl_matrix_int`.

11.7 Matrix allocation

The functions for allocating memory to a matrix follow the style of `malloc` and `free`. They also perform their own error checking. If there is insufficient memory available to allocate a vector then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every `alloc`.

`gsl_matrix *` **gsl_matrix_alloc** (`size_t` *n1*, `size_t` *n2*) Function

These functions allocate memory for a matrix and its *n1 n2* elements, returning a pointer to a newly initialized matrix struct. The elements of the matrix are not initialized and so their values are undefined. Use the function `gsl_matrix_calloc` if you want to ensure that all the elements are initialized to zero.

A null pointer is returned if insufficient memory is available to store the matrix.

`gsl_matrix *` **gsl_matrix_calloc** (`size_t` *n1*, `size_t` *n2*) Function

These functions allocate memory for a matrix and initializes all the elements of the matrix to zero.

`void` **gsl_matrix_free** (`gsl_matrix *` *m*) Function

These functions free a matrix *m* previously allocated with `gsl_matrix_alloc` or `gsl_matrix_calloc`.

The following functions perform the same tasks as the functions above for single-precision and integer matrices.

`gsl_matrix_float *` **gsl_matrix_float_alloc** (`size_t` *n1*, `size_t` *n2*) Function

`gsl_matrix_float *` **gsl_matrix_float_calloc** (`size_t` *n1*, `size_t` *n2*) Function

`void` **gsl_matrix_float_free** (`gsl_matrix_float *` *m*) Function

These functions perform memory management for single-precision floating point matrices defined with the struct `gsl_matrix_float`.

`gsl_matrix_complex *` **gsl_matrix_complex_alloc** (`size_t` *n1*, `size_t` *n2*) Function

`gsl_matrix_complex *` **gsl_matrix_complex_calloc** (`size_t` *n1*, `size_t` *n2*) Function

`void` **gsl_matrix_complex_free** (`gsl_matrix_complex *` *m*) Function

These functions perform memory management for complex matrices defined with the struct `gsl_matrix_complex`.

`gsl_matrix_int *` **gsl_matrix_int_alloc** (`size_t` *n1*, `size_t` *n2*) Function

`gsl_matrix_int *` **gsl_matrix_int_calloc** (`size_t` *n1*, `size_t` *n2*) Function

`void` **gsl_matrix_int_free** (`gsl_matrix_int *` *m*) Function

These functions perform memory management for integer matrices defined with the struct `gsl_matrix_int`.

11.8 Accessing matrix elements

The functions for accessing the elements of a matrix use the same range checking system as vectors. You turn off range checking by recompiling your program with the preprocessor definition `GSL_RANGE_CHECK_OFF`.

The elements of the matrix are stored in "C-order", where the second index moves continuously through memory. More precisely, the element accessed by the function `gsl_matrix_get(m,i,j)` and `gsl_matrix_set(m,i,j,x)` is

```
m->data[i * n2 + j]
```

where *n2* is the second dimension of the matrix.

`double` **gsl_matrix_get** (`const gsl_matrix *` *m*, `size_t` *i*, `size_t` *j*) Function

These functions return the (*i*,*j*)th element of a matrix *m*. If *i* or *j* lie outside the allowed range of 0 to *n1-1* and 0 to *n2-1* then the error handler is invoked and 0 is returned.

`void` **gsl_matrix_set** (`gsl_matrix *` *m*, `size_t` *i*, `size_t` *j*, `double` *x*) Function

These functions set the value of the (*i*,*j*)th element of a matrix *m* to *x*. If *i* or *j* lies outside the allowed range of 0 to *n1-1* and 0 to *n2-1* then the error handler is invoked.

The following functions perform the same tasks for matrices of the type `gsl_matrix_float` and `gsl_matrix_int`.

`float` **gsl_matrix_float_get** (`const gsl_matrix_float *` *m*, `size_t` *i*, `size_t` *j*) Function

`void` **gsl_matrix_float_set** (`gsl_matrix_float *` *m*, `size_t` *i*, `size_t` *j*, `float` *x*) Function

These functions access the elements of a single-precision matrix.

`gsl_complex` **gsl_matrix_complex_get** (`const gsl_matrix_complex *` *m*, `size_t` *i*, `size_t` *j*) Function

`void` **gsl_matrix_complex_set** (`gsl_matrix_complex *` *m*, `size_t` *i*, `size_t` *j*, `gsl_complex` *x*) Function

These functions access the elements of a complex matrix.

`int` **gsl_matrix_int_get** (`const gsl_matrix_int *` *m*, `size_t` *i*, `size_t` *j*) Function

`void` **gsl_matrix_int_set** (`gsl_matrix_int *` *m*, `size_t` *i*, `size_t` *j*, `int` *x*) Function

These functions access the elements of an integer matrix.

11.9 Reading and writing matrices

The library provides functions for reading and writing matrices to a file as binary data or formatted text.

`int` **gsl_matrix_fwrite** (`FILE *` *stream*, `const gsl_matrix *` *m*) Function

This function writes the elements of the matrix *m* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

`int` **gsl_matrix_fread** (`FILE *` *stream*, `gsl_matrix *` *m*) Function

This function reads into the matrix *m* from the open stream *stream* in binary format. The matrix *m* must be preallocated with the correct length since the function uses the size of *m* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

`int` **gsl_matrix_fprintf** (`FILE *` *stream*, `const gsl_matrix *` *m*, `const char *` *format*) Function

This function writes the elements of the matrix *m* line-by-line to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats for floating point numbers and `%d` for integers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file.

`int` **gsl_matrix_fscanf** (`FILE *` *stream*, `gsl_matrix *` *m*) Function

This function reads formatted data from the stream *stream* into the matrix *m*. The matrix *m* must be preallocated with the correct length since the function uses the size of *m* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

The following functions read and write single-precision and integer matrices with the types `gsl_matrix_float` and `gsl_matrix_int`.

int **gsl_matrix_float_fwrite** (FILE * *stream*, const gsl_matrix_float * *m*) Function
int **gsl_matrix_float_fread** (FILE * *stream*, gsl_matrix_float * *m*) Function
int **gsl_matrix_float_fprintf** (FILE * *stream*, const gsl_matrix_float * *m*, const char * *format*) Function
int **gsl_matrix_float_fscanf** (FILE * *stream*, gsl_matrix_float * *m*) Function

These functions read and write single-precision matrices as binary data or formatted text.

int **gsl_matrix_complex_fwrite** (FILE * *stream*, const gsl_matrix_complex * *m*) Function
int **gsl_matrix_complex_fread** (FILE * *stream*, gsl_matrix_complex * *m*) Function
int **gsl_matrix_complex_fprintf** (FILE * *stream*, const gsl_matrix_complex * *m*, const char * *format*) Function
int **gsl_matrix_complex_fscanf** (FILE * *stream*, gsl_matrix_complex * *m*) Function

These functions read and write complex matrices as binary data or formatted text.

int **gsl_matrix_int_fwrite** (FILE * *stream*, const gsl_matrix_int * *m*) Function
int **gsl_matrix_int_fread** (FILE * *stream*, gsl_matrix_int * *m*) Function
int **gsl_matrix_int_fprintf** (FILE * *stream*, const gsl_matrix_int * *m*, const char * *format*) Function
int **gsl_matrix_int_fscanf** (FILE * *stream*, gsl_matrix_int * *m*) Function

These functions read and write integer matrices as binary data or formatted text.

11.10 Example programs for matrices

This program shows how to allocate, initialize and read from a matrix using the functions gsl_matrix_alloc, gsl_matrix_set and gsl_matrix_get.

```
#include <stdio.h>
#include <gsl_matrix.h>

int main ()
{
  int i, j;
  gsl_matrix * m = gsl_matrix_alloc (10,3) ;

  for (i = 0; i < 10; i++)
    for (j = 0; j < 3; j++)
```

```
        gsl_matrix_set (m, i, j, 0.23 + 100*i + j);

    for (i = 0; i < 100; i++)
      for (j = 0; j < 3; j++)
        printf("m_(%d,%d) = %g\n", i, j, gsl_matrix_get (m, i, j));
  }
```

Here is the output from the program. The final loop attempts to read outside the range of the matrix `m`, and the error is trapped by the range-checking code in `gsl_matrix_get`.

```
m_(0,0) = 0.23
m_(0,1) = 1.23
m_(0,2) = 2.23
m_(1,0) = 100.23
m_(1,1) = 101.23
m_(1,2) = 102.23
...
m_(9,2) = 902.23
gsl: matrix_source.c:13: ERROR: first index out of range
IOT trap/Abort (core dumped)
```

The next program shows how to write a matrix to a file.

```
#include <stdio.h>
#include <gsl_matrix.h>

int main ()
{
  int i, j, differences = 0;
  gsl_matrix * m = gsl_matrix_calloc (100,100) ;
  gsl_matrix * a = gsl_matrix_calloc (100,100) ;

  for (i = 0; i < 100; i++)
    for (j = 0 ; j < 100; j++)
      gsl_matrix_set (m, i, j, 0.23 + i + j);

  {
     FILE * f = fopen("test.dat", "w") ;
     gsl_matrix_fwrite (f, m);
     fclose (f);
  }

  {
     FILE * f = fopen("test.dat", "r") ;
     gsl_matrix_fread (f, a);
     fclose (f);
  }

  for (i = 0; i < 100; i++)
    for (j = 0 ; j < 100; j++)
        if (gsl_matrix_get(m, i, j) != gsl_matrix_get(a, i, j))
           differences ++ ;
  printf("differences = %d (should be zero)\n", differences) ;

}
```

After running this program the file '`test.dat`' should contain the elements of `m`, written in binary format. The matrix which is read back in using the function `gsl_matrix_fread` should be exactly equal to the original matrix.

12 Histograms

This chapter describes functions for creating histograms. Histograms provide a convenient way of summarizing the distribution of a set of data. A histogram consists of a set of *bins* which count the number of events falling into a given range of a continuous variable x. In GSL the bins of a histogram contain floating-point numbers, so they can be used to record both integer and non-integer distributions. The bins can use arbitrary sets of ranges (uniformly spaced bins are the default). Both one and two-dimensional histograms are supported.

Once a histogram has been created it can also be converted into a probability distribution function. The library provides efficient routines for selecting random samples from probability distributions. This can be useful for generating simulations based real data.

12.1 The histogram struct

A histogram is defined by the following struct,

gsl_histogram Data Type

`size_t n` This is the number of histogram bins

`double * range`
The ranges of the bins are stored in an array of *n+1* elements pointed to by *range*.

`double * bin`
The counts for each bin are stored in an array of *n* elements pointed to by *bin*. The bins are floating-point numbers, so you can increment them by non-integer values if necessary.

The range for *bin*[i] is given by *range*[i] to *range*[i+1]. For n bins there are $n+1$ entries in the array *range*. Each bin is inclusive at the lower end and exclusive at the upper end. Mathematically this means that the bins are defined by the following inequality,

bin[i] corresponds to $range[i] \le x < range[i+1]$

Here is a diagram of the correspondence between ranges and bins on the number-line for x,

```
   r[0]      r[1]      r[2]      r[3]      r[4]      r[5]
 ---|---------|---------|---------|---------|---------|---  x
    [ bin[0] )[ bin[1] )[ bin[2] )[ bin[3] )[ bin[5] )
```

In this picture the values of the *range* array are denoted by r. On the left-hand side of each bin the square bracket `"["` denotes an inclusive lower bound ($r \le x$), and the round parentheses `")"` on the right-hand side denote an exclusive upper bound ($x < r$). Thus any samples which fall on the upper end of the histogram are excluded. If you want to include this value for the last bin you will need to add an extra bin to your histogram.

The `gsl_histogram` struct and its associated functions are defined in the header file '`gsl_histogram.h`'.

12.2 Histogram allocation

The functions for allocating memory to a histogram follow the style of `malloc` and `free`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a histogram then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every histogram `alloc`.

`gsl_histogram *` **gsl_histogram_calloc** (`size_t` *n*) Function

This function allocates memory for a histogram with *n* bins, and returns a pointer to its newly initialized `gsl_histogram` struct. The bins are uniformly spaced with a total range of $0 \le x < n$, as shown in the table below.

bin[0] corresponds to $0 \le x < 1$
bin[1] corresponds to $1 \le x < 2$
......
bin[n-1] corresponds to $n - 1 \le x < n$

The bins are initialized to zero so the histogram is ready for use.

If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`.

`gsl_histogram *` **gsl_histogram_calloc_uniform** (`size_t` *n*, `double` *xmin*, `double` *xmax*) Function

This function allocates memory for a histogram with *n* uniformly spaced bins from *xmin* to *xmax*, and returns a pointer to the newly initialized `gsl_histogram` struct. The bins are shown in the table below,

bin[0] corresponds to $xmin \le x < xmin + d$
bin[1] corresponds to $xmin + d \le x < xmin + 2d$
......
bin[n-1] corresponds to $xmin + (n-1)d \le x < xmax$

where d is the bin spacing, $(xmax - xmin)/n$. Each bin is initialized to zero.

If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`.

To create a histogram with non-uniform bins you will need to call `gsl_histogram_calloc` to prepare a new histogram struct and then modify the `range` array to use your desired bin limits. The ranges can be arbitrary, subject to the restriction that they are monotonically increasing.

For example, the following code fragment shows how to create a histogram with logarithmic bins from 1—10, 10—100 and 100—1000.

```
gsl_histogram * h = gsl_histogram_calloc (3) ;

h->range[0] = 1.0 ;     /* bin[0] covers the range 1 <= x < 10 */
h->range[1] = 10.0 ;    /* bin[1] covers the range 10 <= x < 100 */
h->range[2] = 100.0 ;   /* bin[2] covers the range 100 <= x < 1000 */
h->range[3] = 1000.0 ;
```

Note that the size of the *range* array is automatically defined as `double range[4]` by `gsl_histogram_calloc`, and is one element bigger than the array of bins `double bin[3]`. Thus the range array safely includes extra space for the final upper value, *range[3]*.

`void` **gsl_histogram_free** (`gsl_histogram * h`) Function

This function frees the histogram *h* and all of the memory associated with it.

12.3 Updating and accessing histogram elements

There are two ways to access histogram bins, either by specifying an x coordinate or by using the bin-index directly. The functions for accessing the histogram through x coordinates use a binary search to identify the bin which covers the appropriate range.

`int` **gsl_histogram_increment** (`gsl_histogram * h, double x`) Function

This function updates the histogram *h* by adding one (1.0) to the bin whose range contains the coordinate *x*.

If *x* lies in the valid range of the histogram then the function returns zero to indicate success. If *x* is less than the lower limit of the histogram then the function returns `GSL_EDOM`, and none of bins are modified. Similarly, if the value of *x* is greater than or equal to the upper limit of the histogram then the function returns `GSL_EDOM`, and none of the bins are modified. The error handler is not called, however, since it is often necessary to compute histogram for a small range of a larger dataset, ignoring the values outside the range of interest.

`int` **gsl_histogram_accumulate** (`gsl_histogram *` *h*, `double` *x*, `double` *weight*) Function

This function is similar to `gsl_histogram_increment` but increases the value of the appropriate bin in the histogram *h* by the floating-point number *weight*.

`double` **gsl_histogram_get** (`const gsl_histogram *` *h*, `size_t` *i*) Function

This function returns the contents of the *i*th bin of the histogram *h*. If *i* lies outside the valid range of indices for the histogram then the error handler is called with an error code of `GSL_EDOM` and the function returns 0.

`int` **gsl_histogram_get_range** (`const gsl_histogram *` *h*, `size_t` *i*, `double *` *lower*, `double *` *upper*) Function

This function finds the upper and lower range limits of the *i*th bin of the histogram *h*. If the index *i* is valid then the corresponding range limits are stored in *lower* and *upper*. The lower limit is inclusive (i.e. events with this coordinate are included in the bin) and the upper limit is exclusive (i.e. events with the coordinate of the upper limit are excluded and fall in the neighboring higher bin, if it exists). The function returns 0 to indicate success. If *i* lies outside the valid range of indices for the histogram then the error handler is called and the function returns an error code of `GSL_EDOM`.

`double` **gsl_histogram_max** (`const gsl_histogram *` *h*) Function
`double` **gsl_histogram_min** (`const gsl_histogram *` *h*) Function
`size_t` **gsl_histogram_bins** (`const gsl_histogram *` *h*) Function

These functions return the maximum upper and mimimum lower range limits and the number of bins of the histogram *h*. They provide a way of determining these values without accessing the `gsl_histogram` struct directly.

`void` **gsl_histogram_reset** (`gsl_histogram *` *h*) Function

This function resets all the bins in the histogram *h* to zero.

12.4 Searching histogram ranges

The following functions are used by the access and update routines to locate the bin which corresponds to a given x coordinate.

`int` **gsl_histogram_find_impl** (`size_t` *n*, `const double` *range*[], `double` *x*, `size_t *` *i*) Function

This function finds and sets the index *i* to the offset in the array *range* of size *n* which bounds the value of *x*, such that $range[i] \le x < range[i+1]$. The binary search function `bsearch` from the system C-library is used to locate the appropriate range. If a suitable value of *i* is found then the function returns 0 to indicate success. If *x* is less than the lower limit the function returns -1, and if *x* is greater than or equal to the upper limit it returns +1. The error handler is not called.

`int` **gsl_histogram_find** (`const gsl_histogram *` *h*, `double` *x*, `size_t *` *i*) Function

This function uses `gsl_histogram_find_impl` to set the index *i* to the bin number which covers the coordinate *x* in the histogram *h*. If *x* is found then the function sets the index *i* and returns zero to indicate success. If *x* lies outside the valid range of the histogram then the function returns `GSL_EDOM` and the error handler is invoked.

12.5 Reading and writing histograms

The library provides functions for reading and writing histograms to a file as binary data or formatted text.

`int` **gsl_histogram_fwrite** (`FILE *` *stream*, `const gsl_histogram *` *h*) Function

This function writes the ranges and bins of the histogram *h* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

`int` **gsl_histogram_fread** (`FILE *` *stream*, `gsl_histogram *` *h*) Function

This function reads into the histogram *h* from the open stream *stream* in binary format. The histogram *h* must be preallocated with the correct size since the function uses the number of bins in *h* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

`int` **gsl_histogram_fprintf** (`FILE *` *stream*, `const gsl_histogram *` *h*, `const char *` *range_format*, `const char *` *bin_format*) Function

This function writes the ranges and bins of the histogram *h* line-by-line to the stream *stream* using the format specifiers *range_format* and *bin_format*. These should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The histogram output is formatted in three columns, and the columns are separated by spaces, like this,

```
range[0] range[1] bin[0]
range[1] range[2] bin[1]
range[2] range[3] bin[2]
....
range[n-1] range[n] bin[n-1]
```

The values of the ranges are formatted using *range_format* and the value of the bins are formatted using *bin_format*. Each line contains the lower and upper limit of the range of the bins and the value of the bin itself. Since the upper limit of one bin is the lower limit of the next there is duplication of these values between lines but this allows the histogram to be manipulated with line-oriented tools.

`int` **gsl_histogram_fscanf** (`FILE *` *stream*, `gsl_histogram *` *h*) Function

This function reads formatted data from the stream *stream* into the histogram *h*. The data is assumed to be in the three-column format used by `gsl_histogram_fprintf`. The histogram *h* must be preallocated with the correct length since the function uses the size of *h* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

12.6 Resampling from histograms

A histogram made by counting events can be regarded as a measurement of a probability distribution. Allowing for statistical error, the height of each bin represents the probability of an event where the value of x falls in the range of that bin. The probability distribution function has the one-dimensional form $p(x)dx$ where,

$$p(x) = n_i/(N w_i) \qquad (12.1)$$

In this equation n_i is the number of events in the bin which contains x, w_i is the width of the bin and N is the total number of events. The distribution of events within each bin is assumed to be uniform.

12.7 The histogram probability distribution struct

The probability distribution function for a histogram consists of a set of *bins* which measure the probability of an event falling into a given range of a continuous variable x. A probability distribution function is defined by the following struct, which actually stores the cumulative probability distribution function. This is the natural quantity for generating samples via the inverse transform method, because there is a one-to-one mapping between the cumulative probability distribution and the range [0,1]. It can be shown that by taking a uniform random number in this range and finding its corresponding coordinate in the cumulative probability distribution we obtain samples with the desired probability distribution.

gsl_histogram_pdf Data Type

`size_t n` This is the number of bins used to approximate the probability distribution function.

`double * range`
The ranges of the bins are stored in an array of *n+1* elements pointed to by *range*.

`double * sum`
The cumulative probability for the bins is stored in an array of *n* elements pointed to by *sum*.

The following functions allow you to create a `gsl_histogram_pdf` struct which represents this probability distribution and generate random samples from it.

`gsl_histogram_pdf *` **gsl_histogram_pdf_alloc** (`const gsl_histogram *` *h*) Function

This function allocates memory for a probability distribution calculated from the histogram *h* and returns a pointer to a newly initialized `gsl_histogram_pdf` struct. If any of the bins of *h* are negative then a null pointer is returned and the error handler is invoked with an error code of `GSL_EDOM` because a probability distribution cannot contain negative values.

If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`.

`void` **gsl_histogram_pdf_free** (`gsl_histogram_pdf *` *p*) Function

This function frees the probability distribution function *p* and all of the memory associated with it.

`double` **gsl_histogram_pdf_sample** (`const gsl_histogram_pdf *` *p*, `double` *r*) Function

This function uses *r*, a uniform random number between zero and one, to compute a single random sample from the probability distribution *p*. The algorithm used to compute the sample *s* is given by the following formula,

$$s = range[i] + delta * (range[i+1] - range[i]) \tag{12.2}$$

where i is the index which satisfies $sum[i] \le r < sum[i+1]$ and $delta$ is $(r - sum[i])/(sum[i+1] - sum[i])$.

12.8 Example programs for histograms

The following program shows how to make a simple histogram of a column of numerical data supplied on `stdin`. The program takes three arguments, specifying the upper and lower bounds of the histogram and the number of bins. It then reads numbers from `stdin`, one line at a time, and adds them to the histogram. When there is no more data to read it prints out the accumulated histogram using `gsl_histogram_fprintf`.

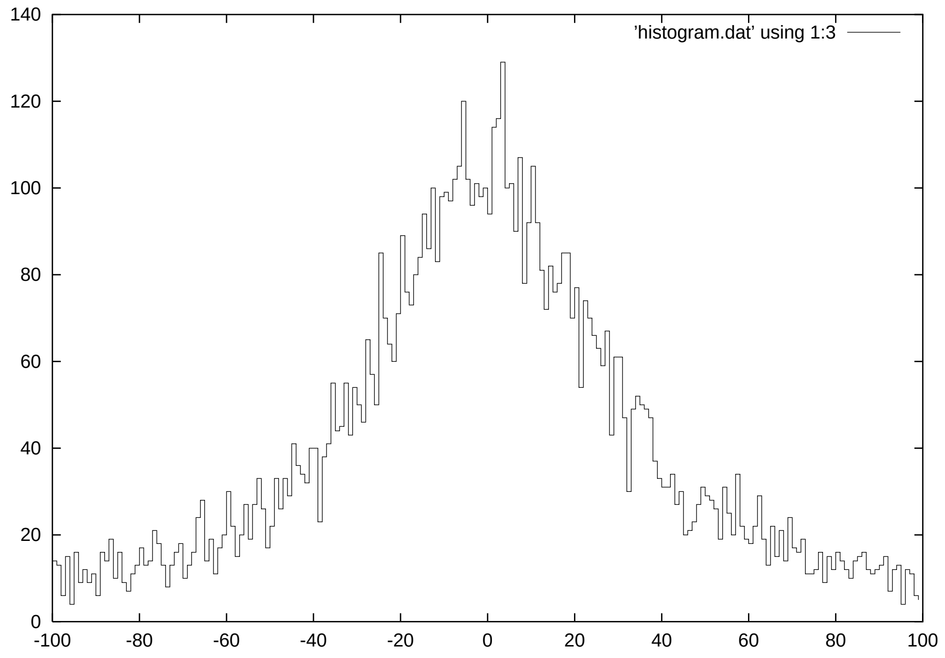
```
#include <stdio.h>
#include <stdlib.h>
#include <gsl_histogram.h>

int
main (int argc, char **argv)
{
  double a, b ;
  size_t n;

  if (argc != 4)
    {
      printf ("Usage: gsl-histogram xmin xmax n\n"
              "Computes a histogram of the data on stdin"
              "using n bins from xmin to xmax\n");
      exit (0);
    }

  a = atof (argv[1]);
  b = atof (argv[2]);
  n = atoi (argv[3]);

  {
    gsl_histogram * h = gsl_histogram_calloc_uniform (n, a, b) ;
    int status ;

    do {
      double x ;
      status = fscanf(stdin, "%lg", &x) ;

      gsl_histogram_increment (h, x) ;

    } while (status == 1) ;

    gsl_histogram_fprintf (stdout, h, "%g", "%g") ;
  }

  exit (0) ;
}
```

Here is an example of the program in use. We generate 10000 random samples from a lorentz distribution with a width of 30 and histogram them over the range -100 to 100, using 200 bins.

```
gsl-lorentz 30 10000 | gsl-histogram -100 100 200 > histogram.dat
```

A plot of the resulting histogram shows the familiar shape of the lorentz distribution and the fluctuations caused by the finite sample size.

```
gnuplot> plot 'histogram.dat' using 1:3 with step
```



12.9 Two dimensional histograms

A two dimensional histogram consists of a set of *bins* which count the number of events falling in a given area of the (x, y) plane. The simplest way to use a two dimensional histogram is to record two-dimensional position information, $n(x, y)$. Another possibility is to form a *joint distribution* by recording related variables. For example a detector might record both the position of an event (x) and the amount of energy it deposited E. These could be histogrammed as the joint distribution $n(x, E)$.

12.10 The 2D histogram struct

Two dimensional histograms are defined by the following struct,

gsl_histogram2d Data Type

`size_t nx, ny`

This is the number of histogram bins in the x and y directions.

`double * xrange`
: The ranges of the bins in the x-direction are stored in an array of *nx + 1* elements pointed to by *xrange*.

`double * yrange`
: The ranges of the bins in the y-direction are stored in an array of *ny + 1* pointed to by *yrange*.

`double * bin`
: The counts for each bin are stored in an array pointed to by *bin*. The bins are floating-point numbers, so you can increment them by non-integer values if necessary. The array *bin* stores the two dimensional array of bins in a single block of memory according to the mapping *bin(i,j)* = `bin[i * ny + j]`.

The range for *bin(i,j)* is given by *xrange[i]* to *xrange[i+1]* in the x-direction and *yrange[j]* to *yrange[j+1]* in the y-direction. Each bin is inclusive at the lower end and exclusive at the upper end. Mathematically this means that the bins are defined by the following inequality,

```
bin(i,j) corresponds to xrange[i] \le  x < xrange[i+1]
                    and yrange[j] \le  y < yrange[j+1]
```

Note that any samples which fall on the upper sides of the histogram are excluded. If you want to include these values for the side bins you will need to add an extra row or column to your histogram.

The `gsl_histogram2d` struct and its associated functions are defined in the header file '`gsl_histogram2d.h`'.

12.11 2D Histogram allocation

The functions for allocating memory to a 2D histogram follow the style of `malloc` and `free`. In addition they also perform their own error checking. If there is insufficient memory available to allocate a histogram then the functions call the GSL error handler (with an error number of `GSL_ENOMEM`) in addition to returning a null pointer. Thus if you use the library error handler to abort your program then it isn't necessary to check every 2D histogram `alloc`.

`gsl_histogram2d *` **gsl_histogram2d_calloc** (`size_t` *nx*, `size_t` *ny*) Function

This function allocates memory for a two-dimensional histogram with *nx* bins in the x direction and *ny* bins in the y direction. The function returns a pointer to a newly initialized `gsl_histogram2d` struct. The bins are uniformly spaced with a total range of $0 \le x < nx$ in the x-direction and $0 \le y < ny$ in the y-direction, as shown in the table below.

The bins are initialized to zero so the histogram is ready for use.

If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`.

`gsl_histogram2d *` **gsl_histogram2d_calloc_uniform** (`size_t` *nx*, `size_t` *ny*, `double` *xmin*, `double` *xmax*, `double` *ymin*, `double` *ymax*) — Function

`void` **gsl_histogram2d_free** (`gsl_histogram2d *` *h*) — Function

This function frees the 2D histogram *h* and all of the memory associated with it.

12.12 Updating and accessing 2D histogram elements

You can access the bins of a two-dimensional histogram either by specifying a pair of (x, y) coordinates or by using the bin indices (i, j) directly. The functions for accessing the histogram through (x, y) coordinates use binary searches in the x and y directions to identify the bin which covers the appropriate range.

`int` **gsl_histogram2d_increment** (`gsl_histogram2d *` *h*, `double` *x*, `double` *y*) — Function

This function updates the histogram *h* by adding one (1.0) to the bin whose x and y ranges contain the coordinates (*x*,*y*).

If the point (x, y) lies inside the valid ranges of the histogram then the function returns zero to indicate success. If (x, y) lies outide the limits of the histogram then the function returns `GSL_EDOM`, and none of bins are modified. The error handler is not called, since it is often necessary to compute histogram for a small range of a larger dataset, ignoring any coordinates outside the range of interest.

`int` **gsl_histogram2d_accumulate** (`gsl_histogram2d *` *h*, `double` *x*, `double` *y*, `double` *weight*) — Function

This function is similar to `gsl_histogram2d_increment` but increases the value of the appropriate bin in the histogram *h* by the floating-point number *weight*.

`double` **gsl_histogram2d_get** (`const gsl_histogram2d *` *h*, `size_t` *i*, `size_t` *j*) — Function

This function returns the contents of the (*i*,*j*)th bin of the histogram *h*. If (*i*,*j*) lies outside the valid range of indices for the histogram then the error handler is called with an error code of `GSL_EDOM` and the function returns 0.

`int` **gsl_histogram2d_get_xrange** (`const gsl_histogram2d *` *h*, `size_t` *i*, `double *` *xlower*, `double *` *xupper*) — Function

`int` **gsl_histogram2d_get_yrange** (`const gsl_histogram2d *` *h*, `size_t` *j*, `double *` *ylower*, `double *` *yupper*) — Function

These functions find the upper and lower range limits of the *i*th and *j*th bins in the x and y directions of the histogram *h*. The range limits are stored in *xlower* and *xupper* or *ylower* and *yupper*. The lower limits are inclusive (i.e. events with these coordinates are included in the bin) and the upper limits are exclusive (i.e. events with the value of the upper limit are not included and fall in the neighboring higher bin, if it exists). The functions return 0 to indicate success. If *i* or *j* lies outside the valid range of indices for the histogram then the error handler is called with an error code of `GSL_EDOM`.

`double` **gsl_histogram2d_xmax** (`const gsl_histogram2d *` *h*) Function
`double` **gsl_histogram2d_xmin** (`const gsl_histogram2d *` *h*) Function
`size_t` **gsl_histogram2d_nx** (`const gsl_histogram2d *` *h*) Function
`double` **gsl_histogram2d_ymax** (`const gsl_histogram2d *` *h*) Function
`double` **gsl_histogram2d_ymin** (`const gsl_histogram2d *` *h*) Function
`size_t` **gsl_histogram2d_ny** (`const gsl_histogram2d *` *h*) Function

These functions return the maximum upper and mimimum lower range limits and the number of bins for the x and y directions of the histogram *h*. They provide a way of determining these values without accessing the `gsl_histogram2d` struct directly.

`void` **gsl_histogram2d_reset** (`gsl_histogram2d *` *h*) Function

This function resets all the bins of the histogram *h* to zero.

12.13 Searching 2D histogram ranges

The following functions are used by the access and update routines to locate the bin which corresponds to a given (x, y) coordinate.

`int` **gsl_histogram2d_find_impl** (`const gsl_histogram2d *` *h*, `double` *x*, `double` *y*, `size_t *` *i*, `size_t *` *j*) Function

This function finds and sets the indices *i* and *j* to the offsets in the arrays *xrange* and *yrange* which bound the value of (x, y), such that $xrange[i] \leq x < xrange[i+1]$ and $yrange[j] \leq y < yrange[j+1]$. The binary search function `bsearch` from the system C-library is used to locate the appropriate ranges. If suitable values of *i* and *j* are found then the function returns 0 to indicate success. If *x* is less than the lower limit the function returns -1, and if *x* is greater than or equal to the upper limit it returns +1. If *x* is valid then the same checks are applied to *y*, with return values of -1 or +1 indicating and error. The error handler is not called.

`int` **gsl_histogram2d_find** (`const gsl_histogram2d *` *h*, `double` *x*, `double` *y*, `size_t *` *i*, `size_t *` *j*) Function

This function uses `gsl_histogram2d_find_impl` to set the indices *i* and *j* to the bin which covers the coordinates (x, y) in the histogram *h*. If (x, y) is found then the function sets the indices (*i*,*j*) and returns zero to indicate success. If (x, y) lies outside the valid range of the histogram then the function returns `GSL_EDOM` and the error handler is invoked.

12.14 Reading and writing 2D histograms

The library provides functions for reading and writing two dimensional histograms to a file as binary data or formatted text.

`int` **gsl_histogram2d_fwrite** (`FILE *` *stream*, `const gsl_histogram2d *` *h*) Function

This function writes the ranges and bins of the histogram *h* to the stream *stream* in binary format. The return value is 0 for success and `GSL_EFAILED` if there was a problem writing to the file. Since the data is written in the native binary format it may not be portable between different architectures.

int **gsl_histogram2d_fread** (FILE * *stream*, gsl_histogram2d * *h*) Function

This function reads into the histogram *h* from the stream *stream* in binary format. The histogram *h* must be preallocated with the correct size since the function uses the number of x and y bins in *h* to determine how many bytes to read. The return value is 0 for success and `GSL_EFAILED` if there was a problem reading from the file. The data is assumed to have been written in the native binary format on the same architecture.

int **gsl_histogram2d_fprintf** (FILE * *stream*, const gsl_histogram2d * *h*, const char * *range_format*, const char * *bin_format*) Function

This function writes the ranges and bins of the histogram *h* line-by-line to the stream *stream* using the format specifiers *range_format* and *bin_format*. These should be one of the `%g`, `%e` or `%f` formats for floating point numbers. The function returns 0 for success and `GSL_EFAILED` if there was a problem writing to the file. The histogram output is formatted in five columns, and the columns are separated by spaces, like this,

```
xrange[0] xrange[1] yrange[0] yrange[1] bin(0,0)
xrange[0] xrange[1] yrange[1] yrange[2] bin(0,1)
xrange[0] xrange[1] yrange[2] yrange[3] bin(0,2)
....
xrange[0] xrange[1] yrange[ny-1] yrange[ny] bin(0,ny-1)

xrange[1] xrange[2] yrange[0] yrange[1] bin(1,0)
xrange[1] xrange[2] yrange[1] yrange[2] bin(1,1)
xrange[1] xrange[2] yrange[1] yrange[2] bin(1,2)
....
xrange[1] xrange[2] yrange[ny-1] yrange[ny] bin(1,ny-1)

....

xrange[nx-1] xrange[nx] yrange[0] yrange[1] bin(nx-1,0)
xrange[nx-1] xrange[nx] yrange[1] yrange[2] bin(nx-1,1)
xrange[nx-1] xrange[nx] yrange[1] yrange[2] bin(nx-1,2)
....
xrange[nx-1] xrange[nx] yrange[ny-1] yrange[ny] bin(nx-1,ny-1)
```

Each line contains the lower and upper limits of the bin and the contents of the bin. Since the upper limits of the each bin are the lower limits of the neighbouring bins there is duplication of these values but this allows the histogram to be manipulated with line-oriented tools.

int **gsl_histogram2d_fscanf** (FILE * *stream*, gsl_histogram2d * *h*) Function

This function reads formatted data from the stream *stream* into the histogram *h*. The data is assumed to be in the five-column format used by `gsl_histogram_fprintf`. The histogram *h* must be preallocated with the correct lengths since the function uses the sizes of *h* to determine how many numbers to read. The function returns 0 for success and `GSL_EFAILED` if there was a problem reading from the file.

12.15 Resampling from 2D histograms

As in the one-dimensional case, a two-dimensional histogram made by counting events can be regarded as a measurement of a probability distribution. Allowing for statistical error, the height of each bin represents the probability of an event where (*x*,*y*) falls in the range of that bin. For a two-dimensional histogram the probability distribution takes the form $p(x,y)dxdy$ where,

$$p(x,y) = n_{ij}/(NA_{ij}) \tag{12.3}$$

In this equation n_{ij} is the number of events in the bin which contains (x,y), A_{ij} is the area of the bin and N is the total number of events. The distribution of events within each bin is assumed to be uniform.

gsl_histogram2d_pdf Data Type

`size_t nx, ny`
: This is the number of histogram bins used to approximate the probability distribution function in the x and y directions.

`double * xrange`
: The ranges of the bins in the x-direction are stored in an array of *nx + 1* elements pointed to by *xrange*.

`double * yrange`
: The ranges of the bins in the y-direction are stored in an array of *ny + 1* pointed to by *yrange*.

`double * sum`
: The cumulative probability for the bins is stored in an array of *nx***ny* elements pointed to by *sum*.

The following functions allow you to create a `gsl_histogram2d_pdf` struct which represents a two dimensional probability distribution and generate random samples from it.

gsl_histogram2d_pdf * **gsl_histogram2d_pdf_alloc** (const gsl_histogram2d * *h*) Function

This function allocates memory for a two-dimensional probability distribution calculated from the histogram *h* and returns a pointer to a newly initialized `gsl_histogram2d_pdf` struct. If any of the bins of *h* are negative then a null pointer is returned and the error handler is invoked with an error code of `GSL_EDOM` because a probability distribution cannot contain negative values.

If insufficient memory is available a null pointer is returned and the error handler is invoked with an error code of `GSL_ENOMEM`.

`void` **gsl_histogram2d_pdf_free** (`gsl_histogram2d_pdf *` *p*) Function

This function frees the two-dimensional probability distribution function *p* and all of the memory associated with it.

`int` **gsl_histogram2d_pdf_sample** (`const gsl_histogram2d_pdf *` *p*, `double` *r1*, `double` *r2*, `double *` *x*, `double *` *y*) Function

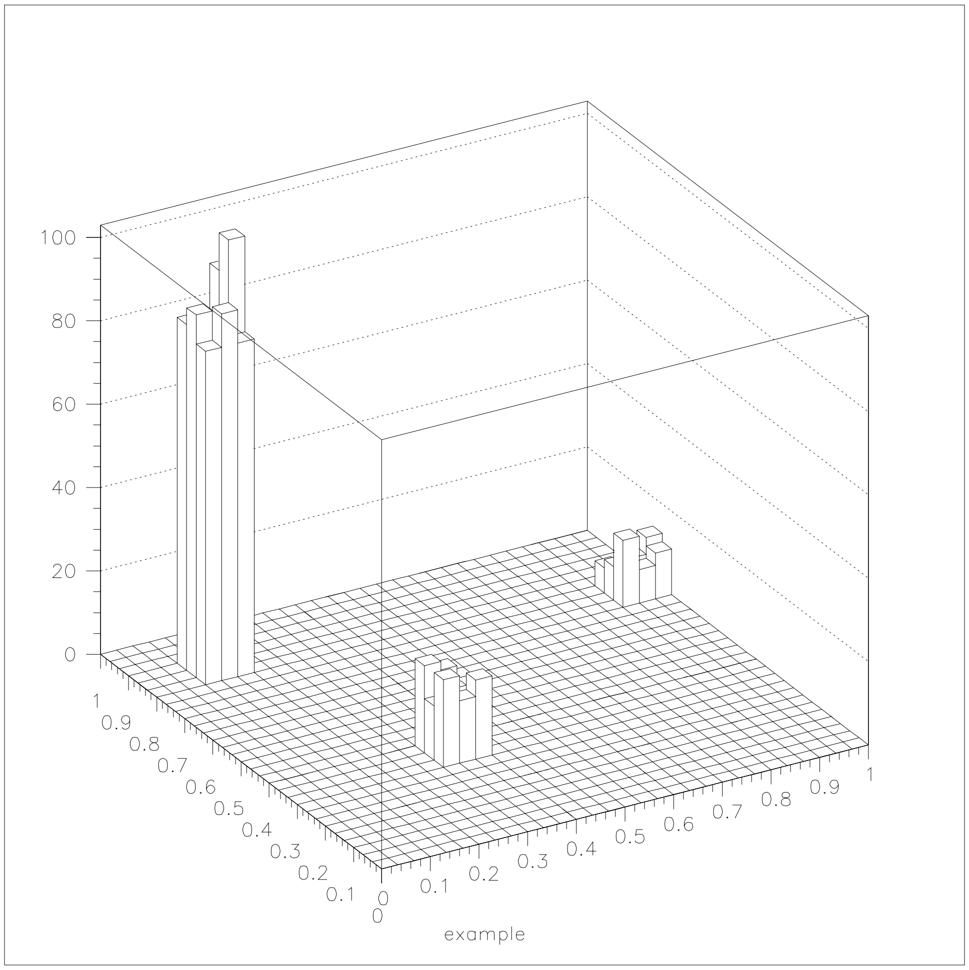
This function uses two uniform random numbers between zero and one, *r1* and *r2*, to compute a single random sample from the two-dimensional probability distribution *p*.

12.16 Example programs for 2D histograms

This program demonstrates two features of two-dimensional histograms. First a 10 by 10 2d-histogram is created with x and y running from 0 to 1. Then a few sample points are added to the histogram, at (0.3,0.3) with a height of 1, at (0.8,0.1) with a height of 5 and at (0.7,0.9) with a height of 0.5. This histogram with three events is used to generate a random sample of 1000 simulated events, which are printed out.

```
#include <stdio.h>
#include <gsl_histogram2d.h>

int
main ()
{
  gsl_histogram2d * h = gsl_histogram2d_calloc_uniform (10, 10,
0, 1, 0, 1) ;

  gsl_histogram2d_accumulate (h, 0.3, 0.3, 1) ;
  gsl_histogram2d_accumulate (h, 0.8, 0.1, 5) ;
  gsl_histogram2d_accumulate (h, 0.7, 0.9, 0.5) ;

  {
    int i ;
    gsl_histogram2d_pdf * p = gsl_histogram2d_pdf_alloc (h) ;

    for (i = 0 ; i < 1000 ; i++) {
      double x, y ;
      double u = ((double) rand ()) / RAND_MAX;
      double v = ((double) rand ()) / RAND_MAX;

      int status = gsl_histogram2d_pdf_sample (p, u, v, &x, &y) ;

      printf("%g %g\n", x, y) ;
    }
  }

 return 0 ;
}
```

The following plot shows the distribution of the simulated events. Using a higher resolution grid we can see the original underlying histogram and also the statistical fluctuations caused by the events being uniformly distributed over the the area of these original bins.



13 Numerical Integration

To be written

13.1 Numerical integration References and Further Reading

The complete story of QUADPACK is given in the book of the same name written by the original developers. It describes the algorithms used in QUADPACK and includes test programs, examples and useful advice on numerical integration. It also has many references to the numerical integration literature.

R. Piessens, E. de Doncker-Kapenga, C.W. Uberhuber, D.K. Kahaner. QUADPACK *A subroutine package for automatic integration* Springer Verlag, 1983.

14 Monte Carlo Integration

This chapter describes the Monte Carlo integration routines in the library. At the moment the algorithms implemented are: plain non-adaptive, Monte Carlo, VEGAS (following Peter Lepage) and MISER (following Numerical recipes). We will dispense with an introduction and simply describe the algorithms and interfaces. Note that the documentation (as well as the code) is still evolving - the code at a faster rate.

14.1 Algorithms

14.1.1 PLAIN (or Simple) Monte Carlo

We begin by establishing some notation. Let $I(f)$ denote the integral of the function f (for convenience, we take $I(1) = 1$) We will write the Monte Carlo estimate of the integral as $E(f;N)$ for the

$$E(f;N) = \frac{1}{N}\sum_{i}^{N} f(x_i). \tag{14.1}$$

for N randomly distributed points x_i Similarly, we will denote the variance of f by

$$\sigma^2(f) = I(f^2) - I(f)^2 \tag{14.2}$$

and the variance of the estimate by

$$Var(f;N) = E(f^2;N) - (E(f;N))^2. \tag{14.3}$$

The fundamental point of Monte Carlo integration is that for large N,

$$E(f;N) - I(f) \approx \frac{\sigma^2(f)}{N} \tag{14.4}$$

and furthermore, by the same argument,

$$\sigma^2(f) \approx Var(f;N) \tag{14.5}$$

That's all there is to simple Monte Carlo.

14.1.2 MISER

The starting point of all stratified sampling techniques is the observation that for two disjoint regions a and b with Monte Carlo estimates of the integral $E_a(f)$ and $E_b(f)$ of the integral of f in those regions and variances $\sigma_a^2(f)$ and $\sigma_b^2(f)$, the variance $Var(f)$ of the combined estimate $E(f) = \frac{1}{2}E_a(f) + \frac{1}{2}E_b(f)$ is given by

$$Var(f) = \frac{\sigma_a^2(f)}{4N_a} + \frac{\sigma_b^2(f)}{4N_b} \tag{14.6}$$

This variance is minimized (subject to the constrain that $N_a + N_b$ is fixed) by choosing (assuming one can) N_a s.t.

$$\frac{N_a}{N_a + N_b} = \frac{\sigma_a}{\sigma_a + \sigma_b} \tag{14.7}$$

For such a choice, the variance is given by

$$Var(f) = \frac{(\sigma_a + \sigma_b)^2}{4N} \tag{14.8}$$

In words, the variance of the estimate (and hence the error) is minimized by concentrating points in regions where the variance of the function f is large.

The most straightforward stratified sampling routine divides a hyper-cubic integration region into sub-cubes along the coordinate axes. This is simple to do, but gets out of hand for large numbers of dimensions d because the number of sub-cubes grows like K^d (where K is the number of sub-divisions along the axis).

What MISER does (and other recursive stratified samplers) is to bisect the hypercube along one coordinate axis. The direction is chosen by examining all possible bisections (d of them) and picking the one that gives the best combined variance. The same procedure is then done for each of the two half-spaces (choosing N_a and N_b as described above), and so on. Since by assumption one does not know the variance $Var_L(f)$ in the various sub-regions, it is estimated using some fraction of the total number of points alloted to the estimate.

14.1.3 VEGAS

Vegas takes another approach to obtaining good results, namely it tries to sample points from the probability distribution described by the function f. More precisely, suppose we estimate the integral of f with points distributed according to a probability distribution described by the function g. If we call the new estimate $E_g(f; N)$ we have

$$E_g(f; N) = E(f/g; N) \tag{14.9}$$

and similarly

$$Var_g(f; N) = Var(f/g; N) \tag{14.10}$$

It is clear from this that if $g = |f|/I(|f|)$ then the variance $V_g(f; N)$ vanishes. This it was what Vegas attempts to do. It makes several passes, histograming f, and each time using the histogram to define the sampling distribution for the next pass. This basic idea again has the problem that the number of histogram bins grows like K^d. The compromise that Vegas (ie, Peter Lepage) adopts is to assume that g factors: $g(x_1, x_2, \ldots) = g_1(x_1)g_2(x_2)\ldots$ so that the number of bins required is only $d \cdot K$. In this case, it is not hard to show that the optimal distribution is

$$g_1(x_1) = [\int dx_1 \ldots dx_n \frac{f^2(x_1, \ldots, x_n)}{g_2(x_2)\ldots g_n(x_n)}]^{(1/2)} \tag{14.11}$$

Lepage's Vegas is actually a bit fancier than this, combining stratified sampling and importance sampling. The integration region is divided into a number of "boxes", with each box getting in fixed number of points (the goal is 2). Each box can then have a fractional number of bins, but if bins/box is less than two, Vegas switches to a kind variance reduction (rather than importance sampling).

14.2 Interface

All of the integration routines use the same interface. There is an allocator to allocate memory to hold control variables and workspace, a routine to initialize those control variables, the integrator itself, and of course a function to free the space when done. For an integrator algorithm, call it COOL (substitute any of the currently existing algorithms) we then have

`gsl_monte_COOL_state*` **gsl_monte_COOL_alloc(size_t** *dim*) Function

`int` **gsl_monte_COOL_init(gsl_monte_COOL_state*** *s*) Function

`int` **gsl_monte_COOL_free(gsl_monte_COOL_state*** *s*)**,** Function

`int` **gsl_monte_COOL_validate(gsl_monte_COOL_state*** *s***,** Function

double **x_lower*, double **x_upper*, unsigned long *dimension*, unsigned long *function_calls*) `int` **gsl_monte_COOL_integrate(gsl_monte_COOL_state*** *s***,**

double* *x_lower*, double* *x_upper*, unsigned long *dimension*, unsigned long *function_calls*, double* *result*, double* *error*, ...)

Notice the ellipses in the last argument to the actual integration routine. This is because the vegas algorithm (and perhaps others in the future) returns extra information – in the case of vegas, it is the χ^2 of the result.

In addition to the common function interface, the routines also share the state variables

`gsl_rng*` **ranf** Control

which determines which random number generator will be used.

and

`int` **verbose** Control

which says whether to print information about the calculation (though the actual use depends on the algorithm).

In the near future, we expect that there will be a common interface for selecting a "log" stream for reporting various information about the performance of the particular algorithm.

We describe the algorithm-specific variables.

PLAIN: None

MISER:

`double` **alpha** Control

alpha controls how the variances for the two sub-regions are combined. The Numerical Recipes gang argue that for recursive sampling there is no reason to expect that the variance should scale like $1/N$ and so they allow the scaling to depend on α

$$Var(f) = \frac{\sigma_a}{N_a^\alpha} + \frac{\sigma_b}{N_b^\alpha} \tag{14.12}$$

`double` **dither** Control

Rather than exactly bisection the integration region, dither allows the user to introduce a bit of fuzz. This helps in the case when the function to be integrated has some symmetry, say if it is peaked in the center of the hypercube. If needed, dither should be around 0.1.

VEGAS:

`double` **alpha** Control

For Vegas, `alpha` controls the stiffness of the rebinning algorithm: `alpha = 0` means never rebin. It is typically set between one and two.

`double` **acc** Control

Setting `acc` allows vegas to terminate when the desired accuracy has been achieved, rather than after a certain number of function calls. Setting it to a negative value disables this feature.

`long int` **max_it_num** Control

The maximum number of iterations to perform.

`int` **stage** Control

Setting this determines the "stage" of the calculation. Normally, `stage = 0`. Calling vegas with `stage = 1` retains the grid (but not the answers) from the previous run, so that one can "tune" the grid using a relatively small number of points and then do a large run with `stage = 1` on the optimized grid. Setting `stage = 2` keeps the grid and the answers from the previous run and `stage = 3` enters at the main loop, so that nothing is changed – this is like deciding to change `max_it_num` during the run.

`int` **mode** Control

The possible choices are `GSL_VEGAS_MODE_IMPORTANCE`, `GSL_VEGAS_MODE_STRATIFIED`, `GSL_VEGAS_MODE_IMPORTANCE_ONLY`. This determines whether vegas will use importance sampling or stratified sampling, or whether it can pick on its own. In low dimensions Vegas uses strict stratified sampling (more precisely, stratified sampling is chosen if there are fewer than 2 bins per box).

14.3 Example

Here we provide s simple example of using the integration routines. Vegas was chosen at random.

```
#include <math.h>
#include <stdio.h>

#include <gsl_math.h>
#include <gsl_monte_vegas.h>

double f1(double x[])
{
  int i;
  double product = 1.0;

  for (i = 0; i < 10; i++)
    product *= x[i];

  return product;
}

main ()
{
  double xl[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  double xu[10] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1};

  double res = 0;
  double err = 0;
  double chisq = 0;
  int status = 0;
  unsigned long calls = 10000;
  unsigned long dimension = 10;

  gsl_monte_vegas_state* s = gsl_monte_vegas_alloc(10);
  gsl_monte_plain_init(s);

  s->alpha = 1.5; /* default */
  s->verbose = 0; /* default, we prefer to remain ignorant! */
  s->stage = 0; /* default.  Start at the beginning */
  s->acc = -1.0 /* default. don't terminate when accuracy is reached */
  s->max_it_num = 5 /* default.  Do five iterations before quitting */

  status = gsl_monte_vegas_integrate(s, f0, xl, xu, dimension, calls,
                                     &res, &err, &chisq);
  if (status)
    printf("oops!\n");
  else
    printf("vegas(f0) = %f +- %f with \"chisq\" %f\n", res, err, chisq);

}
```

14.4 The Future

In the future, the author of the Monte Carlo routines intends to add more algorithms, greater control over the current ones, more error handling, and a more consistent interface. Probably something like the rng interface will evolve, so that something like

```
integrator = gsl_monte_alloc(gsl_monte_vegas, dimension)
```

will be possible. The old interface will probably stay around for a while (though, since this is *version* < 1, the author does not want to be held to such as statement).

15 The IEEE standard for floating-point arithmetic

This chapter describes functions for examing the representation of floating point numbers and controlling the floating point environment of your program.

15.1 Representation of floating point numbers

void **gsl_ieee_printf_float** (const float * *x*) Function
void **gsl_ieee_printf_double** (const double * *x*) Function

```
#include <stdio.h>
#include <gsl_ieee_utils.h>

main ()
{
  float f = 1.0/3.0 ;
  double d = 1.0/3.0 ;

  double fd = f ; /* promote from float to double */

  printf("     float 1/3 = ") ; gsl_ieee_printf_float(&f) ; printf("\n") ;
  printf("promoted float = ") ; gsl_ieee_printf_double(&fd) ; printf("\n") ;
  printf("    double 1/3 = ") ; gsl_ieee_printf_double(&d) ; printf("\n") ;
}
```

```
     float 1/3 =  1.01010101010101010101011*2^-2
promoted float =  1.0101010101010101010101100000000000000000000000000000*2^-2
    double 1/3 =  1.0101010101010101010101010101010101010101010101010101*2^-2
```

To use these numbers in Calc, precede them by `2#` to indicate binary. In `bc`, work with the mantissa separately from the exponent.

float vs double vs long double (how many digits are available for each)

importance of using 1.234L in long double calculations

```
int main (void)
{
  long double x = 1.0, y = 1.0 ;

  x = x + 0.2 ;
  y = y + 0.2L ;

  printf(" d %.20Lf\n",x) ;
  printf("ld %.20Lf\n",y) ;

  return 1;
}

 d 1.20000000000000001110
ld 1.20000000000000000004
```

15.2 Setting up your IEEE environment

The IEEE standard defines several *modes* for controlling the behavior of floating point operations. These modes specify the important properties of computer arithmetic: the direction used for rounding (e.g. whether numbers should be rounded up, down or to the nearest number), the rouding precision and how the program should handle arithmetic exceptions, such as division by zero.

Unfortunately there is no universal API for controlling these features – each system has its own way of accessing them. For example, the Linux kernel provides the function `__setfpucw` (*set-fpu-control-word*) to set IEEE modes, while HP-UX and Solaris use the functions `fpsetround` and `fpsetmask`. To help you write portable programs GSL allows you to specify modes in a platform-independent using the environment variable `GSL_IEEE_MODE`. The library then takes care of all the necessary machine-specific initializations for you when you call the function `gsl_ieee_env_setup`.

`void` **gsl_ieee_env_setup** () Function

This function reads the environment variable `GSL_IEEE_MODE` and attempts to set up the corresponding specified IEEE modes. The environment variable should be a list of keywords, separated by semicolons, like this,

`GSL_IEEE_MODE` = "*keyword*;*keyword*;..."

where *keyword* is one of the following mode-names,

```
single-precision
double-precision
extended-precision
round-to-nearest
round-down
round-up
round-to-zero
mask-all
mask-invalid
mask-denormalized
mask-division-by-zero
mask-overflow
mask-underflow
trap-inexact
trap-common
```

If `GSL_IEEE_MODE` is empty or undefined then the function returns immediately and no attempt is made to change the system's IEEE mode. When the modes from `GSL_IEEE_MODE` are turned on the function prints a short message showing the new settings to remind you that the results of the program will be affected.

If the requested modes are not supported by the platform being used then the function calls the error handler and returns an error code of `GSL_EUNSUP`.

To demonstrate the effects of different rounding modes consider following the program computes e, the base of natural logarithms, by summing a rapidly-decreasing series,

$$e = 1 + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots = 2.71828182846... \quad (15.1)$$

```
#include <math.h>
#include <stdio.h>
#include <gsl_ieee_utils.h>

int main (void)
{
  double x = 1, oldsum = 0, sum = 0;
  int i = 0 ;

  gsl_ieee_env_setup () ; /* read GSL_IEEE_MODE */

  do
    {
      i++ ;

      oldsum = sum ;
      sum += x ;
      x = x / i ;

      printf("i=%2d sum=%.18f error=%g\n",i, sum, sum - M_E) ;
    }
  while (sum != oldsum) ;

}
```

Here are the results of running the program in `round-to-nearest` mode. This is the IEEE default so it isn't really necessary to specify it here,

```
GSL_IEEE_MODE="round-to-nearest" ./a.out
i= 1 sum=1.000000000000000000 error=-1.71828
i= 2 sum=2.000000000000000000 error=-0.718282
....
i=18 sum=2.718281828459045535 error=4.44089e-16
i=19 sum=2.718281828459045535 error=4.44089e-16
```

After nineteen terms the sum converges to within 4×10^{-16} of the correct value. If we now change the rounding mode to `round-down` the final result is less accurate,

```
GSL_IEEE_MODE="round-down" ./a.out
i= 1 sum=1.000000000000000000 error=-1.71828
....
i=19 sum=2.718281828459041094 error=-3.9968e-15
```

The result is about 4×10^{-15} below the correct value, an order of magnitude worse than the result obtained in the `round-to-nearest` mode.

If we change to rounding mode to `round-up` then the series no longer converges (the reason is that when we add each term to the sum the final result is always rounded up.

This is guaranteed to increase the sum by at least one tick on each iteration). To avoid this problem we would need to use a safer converge criterion, such as `while (fabs(sum - oldsum) > epsilon)`, with a suitably chosen value of epsilon.

Finally we can see the effect of computing the sum using single-precision rounding, in the default `round-to-nearest` mode. In this case the program thinks it is still using double precision numbers but the CPU rounds the result of each floating point operation to single-precision accuracy. This simulates the effect of writing the program using single-precision `float` variables instead of `double` variables. The iteration stops after about half the number of iterations and the final result is much less accurate,

```
GSL_IEEE_MODE="single-precision" ./a.out
....
i=12 sum=2.718281984329223633 error=1.5587e-07
```

with an error of $O(10^{-7})$, which corresponds to single precision accuracy (about 1 part in 10^7). Continuing the iterations further does not decrease the error because all the subsequent results are rounded to the same value.

15.3 IEEE References and Further Reading

David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic. [Comm ACM ??] Vol. 23, No. 1 (March 1991) pages 5-48

Appendix A Debugging Numerical Programs

This chapter describes some tips and tricks for debugging numerical programs which use GSL.

A.1 Using gdb

Any errors reported by the library are routed through the function `gsl_error`. By running your programs under gdb and setting a breakpoint in this function you can automatically catch any library errors. You can add a breakpoint for every session by putting

```
break gsl_error
```

into your '`.gdbinit`' file in the directory where your program is started. If the breakpoint catches an error then you can use a backtrace (`bt`) to see the call-tree, and the arguments which possibly caused the error. By moving into the caller (`up`, `up`) you can investigate the values of variable at that point.

Here is an example from the program `fft/test_trap`, which contains the following line,

```
status = gsl_fft_complex_wavetable_alloc (0, &complex_wavetable);
```

The function `gsl_fft_complex_wavetable_alloc` takes the length of an FFT as its first argument. When this line is executed an error will be generated because the length of an FFT is not allowed to be zero.

To debug this problem we start `gdb`, using the file '`.gdbinit`' to define a breakpoint in `gsl_error`,

```
bjg|zeke> gdb test_trap
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i586-debian-linux), Copyright 1996 Free Software Foundation, Inc...
Breakpoint 1 at 0x8050b1e: file error.c, line 14.
```

When we run the program this breakpoint catches the error and shows the reason for it.

```
(gdb) run
Starting program: /home/bjg/gsl/fft/test_trap

Breakpoint 1, gsl_error (reason=0x8052b0d "length n must be positive integer",
    file=0x8052b04 "c_init.c", line=108, gsl_errno=1) at error.c:14
14        if (gsl_error_handler)
```

The first argument of `gsl_error` is always a string describing the error. Now we can look at the backtrace to see what caused the problem,

```
(gdb) bt
#0  gsl_error (reason=0x8052b0d "length n must be positive integer",
    file=0x8052b04 "c_init.c", line=108, gsl_errno=1) at error.c:14
#1  0x8049376 in gsl_fft_complex_wavetable_alloc (n=0, wavetable=0xbffff778)
    at c_init.c:108
#2  0x8048a00 in main (argc=1, argv=0xbffff9bc) at test_trap.c:94
#3  0x80488be in ___crt_dummy__ ()
```

We can see that the error was generated in the function `gsl_fft_complex_wavetable_alloc` when it was called with an argument of *n=0*. The original call came from line 94 in the file '`test_trap.c`'.

By moving up to the level of the original call we can find the line that caused the error,

```
(gdb) up
#1  0x8049376 in gsl_fft_complex_wavetable_alloc (n=0, wavetable=0xbffff778)
    at c_init.c:108
108           GSL_ERROR ("length n must be positive integer", GSL_EDOM);
(gdb) up
#2  0x8048a00 in main (argc=1, argv=0xbffff9bc) at test_trap.c:94
94        status = gsl_fft_complex_wavetable_alloc (0, &complex_wavetable);
```

Thus we have found the line that caused the problem. From this point we could also print out the values of other variables such as `complex_wavetable`.

A.2 GCC warning options for numerical programs

Writing reliable numerical programs in C requires great care. Uninitialized variables, conversions to and from integers or from signed to unsigned integers can all cause hard-to-find problems. For many non-numerical programs compiling with `gcc`'s warning option `-Wall` provides a good check against common errors. However, for numerical programs `-Wall` is not enough. If you are unconvinced take a look at this program which contains an error that can occur in numerical code,

```
#include <math.h>
#include <stdio.h>

double f (int x) ;

int main ()
{
  double a = 1.5 ;
  double y = f(a) ;
  printf("a = %g, sqrt(a) = %g\n", a, y) ;
  return 0 ;
}

double f(x) {
  return sqrt(x) ;
}
```

This code compiles cleanly with `-Wall` but produces some strange output,

```
bjg|zeke> gcc -Wall tmp.c -lm
bjg|zeke> ./a.out
a = 1.5, sqrt(a) = 1
```

Note that adding `-ansi` does not help here, since the program does not contain any invalid constructs. What is happening is that the prototype for the function `f(int x)` is not consistent with the function call `f(y)`, where `y` is a floating point number. This results in the argument being silently converted to an integer. This is valid C, but in a numerical program it also likely to be a programming error so we would like to be warned about it. (If we genuinely wanted to convert `y` to an integer then we could use an explicit cast, `(int)y`).

Fortunately GCC provides many additional warnings which can alert you to problems such as this. You just have to remember to use them. Here is a set of recommended warning options for numerical programs.

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

It saves time if you to put these options in your '`Makefile`' (for example, under the target `make strict`) or define them as a shell variable or alias. For details of each option consult the manual *Using and porting GCC*. The following table gives a brief explanation of what types of errors these warnings catch.

`-ansi -pedantic`
: Use ANSI C, and reject any non-ANSI extensions. These flags help in writing portable programs that will compile on other systems.

`-Werror`
: Consider warnings to be errors, so that compilation stops. This prevents warnings from scrolling off the top of the screen and being lost. You won't be able to compile the program until it is completely warning-free.

`-Wall`
: This turns on a set of warnings for common programming problems. You need `-Wall`, but it is not enough on its own, as explained above.

`-O4`
: Turn on optimization. The warnings for unitialized variables in `-Wall` rely on the optimizer to analyze the code. If there is no optimization then the warnings aren't generated.

`-W`
: This turns on some extra warnings not included in `-Wall`, such as missing return values and comparisons between signed and unsigned integers.

`-Wmissing-prototypes -Wstrict-prototypes`
: Warn if there are any missing or inconsistent prototypes. If your prototypes are missing then you will never detect problems with incorrect arguments. If your prototypes are inconsistent then you already have a problem.

`-Wtraditional`
: This warns about certain constructs that behave differently in traditional and ANSI C. Whether the traditional or ANSI interpretation is used might be unpredictable on other compilers.

`-Wconversion`
: The main use of this option is to warn about conversions from signed to unsigned integers. For example, `unsigned int x = -1`.. If you need to perform such a conversion you can use an explicit cast.

`-Wshadow`
: This warns whenever a local variable shadows another local variable. If two variables have the same name then it is a potential source of confusion.

`-Wpointer-arith -Wcast-qual -Wcast-align`
: These options warn if you try to do pointer arithmetic for types which don't have a size, such as `void`, if you remove a `const` cast from a pointer, or if you cast a pointer to a type which has a different size, causing an invalid alignment.

`-Wwrite-strings`
: This option gives string constants a `const` qualifier so that it will be a compile-time error to attempt to overwrite them.

`-Waggregate-return`
: Warn if any functions that return structures or unions are defined or called. Some older compilers might have problems with such a construct.

`-fshort-enums`
: This option makes the type of `enum` as short as possible. Normally this makes an `enum` different from an `int`. Consequently any attempts to assign a pointer-to-int to a pointer-to-enum will generate a cast-alignment warning.

`-fno-common`
: This option prevents global variables being simultaneously defined in different object files (you get an error at link time). Such a variable should be defined in one file and referred to in other files with an `extern` declaration.

`-Wnested-externs`
: This warns if an `extern` declaration is encountered within an function.

`-Dinline=`
: The `inline` keyword is not part of ANSI C. Thus if you want to use `-ansi` with a program which uses inline functions you can use this preprocessor definition to remove the `inline` keywords.

`-g`
: It always makes sense to put debugging symbols in the executable so that you can debug it using `gdb`. The only effect of debugging symbols is to increase the size of the file, and you can use the `strip` command to remove them later if necessary.

For comparison, this is what happens when the test program above is compiled with these options.

```
bjg|zeke> gcc -ansi -pedantic -Werror -W -Wall -Wtraditional
-Wconversion -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align
-Wwrite-strings -Waggregate-return -Wstrict-prototypes -fshort-enums
-fno-common -Wmissing-prototypes -Wnested-externs -Dinline=
-g -O4 tmp.c
cc1: warnings being treated as errors
tmp.c:7: warning: function declaration isn't a prototype
tmp.c: In function ‘main':
tmp.c:9: warning: passing arg 1 of ‘f' as integer rather than floating
due to prototype
tmp.c: In function ‘f':
tmp.c:14: warning: type of ‘x' defaults to ‘int'
tmp.c:15: warning: passing arg 1 of ‘sqrt' as floating rather than integer
```

```
     due to prototype
     make: *** [tmp] Error 1
```

The error in the prototype is flagged, plus the fact that we should have defined main as `int main (void)` in ANSI C. Clearly there is some work to do before this program is ready to run.

Appendix B Contributors to GSL

Mark Galassi
: Conceived GSL (with James Theiler) and wrote the design document. Wrote the simulated annealing package and the relevant chapter in the manual.

James Theiler
: Conceived GSL (with Mark Galassi). Wrote the random number generators and the relevant chapter in this manual.

Jim Davies
: Wrote the statistical routines and the relevant chapter in this manual.

Brian Gough
: Wrote the FFT package and the relevant chapter in this manual. Also wrote the error handling infrastructure.

Reid Priedhorsky
: Wrote the root finding package and the relevant chapter in this manual.

Gerry Jungman
: Wrote the special function library.

Appendix C Copying

Concept Index

Function Index

Variable Index

Type Index