GNU Scientific Library

Reference Manual
Third edition, for GSL Version 1.12

Mark Galassi
Los Alamos National Laboratory

Jim Davies
Department of Computer Science, Georgia Institute of Technology

James Theiler
Astrophysics and Radiation Measurements Group,
Los Alamos National Laboratory

Brian Gough
Network Theory Limited

Gerard Jungman
Theoretical Astrophysics Group, Los Alamos National Laboratory

Patrick Alken
Department of Physics, University of Colorado at Boulder

Michael Booth
Department of Physics and Astronomy, The Johns Hopkins University

Fabrice Rossi
University of Paris-Dauphine

Published by Network Theory Ltd.

A catalogue record for this book is available from the British Library.

Third edition, First printing, January 2009 for version 1.12.

Published by Network Theory Limited.

Email: info@network-theory.co.uk

ISBN: 0-9546120-7-8

This book supersedes the previous edition for version 1.8 (ISBN 0-9541617-3-4).

Original cover design by David Nicholls.

Further information about this book is available from
`http://www.network-theory.co.uk/gsl/manual/`

This book has an unconditional guarantee. If you are not fully satisfied with your purchase for any reason, please contact the publisher at the address above.

The texinfo source files for this manual are available from
`http://www.network-theory.co.uk/gsl/manual/src/`

Table of Contents

Preface

This manual documents the use of the GNU Scientific Library, a numerical library for C and C++ programmers.

The GNU Scientific Library is *free software*. The term "free software" is sometimes misunderstood—it has nothing to do with price. It is about freedom. It refers to your freedom to run, copy, distribute, study, change and improve the software. With the GNU Scientific Library you have all these freedoms.

The GNU Scientific Library is part of the GNU Project. The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. It was conceived as a way of bringing back the cooperative spirit that prevailed in the computing community in earlier days, by removing the obstacles to cooperation imposed by the owners of proprietary software.

The Free Software Foundation is a tax-exempt charity that raises funds for work on the GNU Project and is dedicated to promoting the freedom to modify and redistribute computer programs. You can support the GNU Project by becoming an associate member of the Free Software Foundation and paying regular membership dues. For more information, visit the website `www.fsf.org`.

Brian Gough
Publisher
December 2008

1 Introduction

The GNU Scientific Library (GSL) is a collection of routines for numerical computing. The routines have been written from scratch in C, and present a modern Applications Programming Interface (API) for C programmers, allowing wrappers to be written for very high level languages. The source code is distributed under the GNU General Public License.

1.1 Routines available in GSL

The library covers a wide range of topics in numerical computing. Routines are available for the following areas,

Complex Numbers	Roots of Polynomials
Special Functions	Vectors and Matrices
Permutations	Combinations
Sorting	BLAS Support
Linear Algebra	CBLAS Library
Fast Fourier Transforms	Eigensystems
Random Numbers	Quadrature
Random Distributions	Quasi-Random Sequences
Histograms	Statistics
Monte Carlo Integration	N-Tuples
Differential Equations	Simulated Annealing
Numerical Differentiation	Interpolation
Series Acceleration	Chebyshev Approximations
Root-Finding	Discrete Hankel Transforms
Least-Squares Fitting	Minimization
IEEE Floating-Point	Physical Constants
Basis Splines	Wavelets

The use of these routines is described in this manual. Each chapter provides detailed definitions of the functions, followed by example programs and references to the articles on which the algorithms are based.

Where possible the routines have been based on reliable public-domain packages such as FFTPACK and QUADPACK, which the developers of GSL have reimplemented in C with modern coding conventions.

1.2 GSL is Free Software

The subroutines in the GNU Scientific Library are "free software"; this means that everyone is free to use them, and to redistribute them in other free programs. The library is not in the public domain; it is copyrighted and there are conditions on its distribution. These conditions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of the software that they might get from you.

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

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of any code which uses the GNU Scientific Library, you must give the recipients all the rights that you have received. You must make sure that they, too, receive or can get the source code, both to the library and the code which uses it. And you must tell them their rights. This means that the library should not be redistributed in proprietary programs.

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

The precise conditions for the distribution of software related to the GNU Scientific Library are found in the GNU General Public License (see [GNU General Public License], page 523). Further information about this license is available from the GNU Project webpage *Frequently Asked Questions about the GNU GPL*,

    http://www.gnu.org/copyleft/gpl-faq.html

The Free Software Foundation also operates a license consulting service for commercial users (contact details available from `http://www.fsf.org/`).

1.3 Obtaining GSL

The source code for the library can be obtained in different ways, by copying it from a friend, purchasing it on CDROM or downloading it from the internet. A list of public ftp servers which carry the source code can be found on the GNU website,

    http://www.gnu.org/software/gsl/

The preferred platform for the library is a GNU system, which allows it to take advantage of additional features in the GNU C compiler and GNU C library. However, the library is fully portable and should compile on most systems with a C compiler.

Announcements of new releases, updates and other relevant events are made on the `info-gsl@gnu.org` mailing list. To subscribe to this low-volume list, send an email of the following form:

```
To: info-gsl-request@gnu.org
Subject: subscribe
```

You will receive a response asking you to reply in order to confirm your subscription.

1.4 No Warranty

The software described in this manual has no warranty, it is provided "as is". It is your responsibility to validate the behavior of the routines and their accuracy using the source code provided, or to purchase support and warranties from commercial redistributors. Consult the GNU General Public license for further details (see [GNU General Public License], page 523).

1.5 Reporting Bugs

A list of known bugs can be found in the '`BUGS`' file included in the GSL distribution or online in the GSL bug tracker.[1] Details of compilation problems can be found in the '`INSTALL`' file.

If you find a bug which is not listed in these files, please report it to `bug-gsl@gnu.org`.

All bug reports should include:

- The version number of GSL
- The hardware and operating system
- The compiler used, including version number and compilation options
- A description of the bug behavior
- A short program which exercises the bug

It is useful if you can check whether the same problem occurs when the library is compiled without optimization. Thank you.

Any errors or omissions in this manual can also be reported to the same address.

---

(1) `http://savannah.gnu.org/bugs/?group=gsl`

1.6 Further Information

Additional information, including online copies of this manual, links to related projects, and mailing list archives are available from the website mentioned above.

Any questions about the use and installation of the library can be asked on the mailing list `help-gsl@gnu.org`. To subscribe to this list, send an email of the following form:

```
To: help-gsl-request@gnu.org
Subject: subscribe
```

This mailing list can be used to ask questions not covered by this manual, and to contact the developers of the library.

If you would like to refer to the GNU Scientific Library in a journal article, the recommended way is to cite this reference manual, e.g. *M. Galassi et al, GNU Scientific Library Reference Manual (3rd Ed.), ISBN 0-9546120-7-8.*

If you want to give a url, use "`http://www.gnu.org/software/gsl/`".

1.7 Conventions used in this manual

This manual contains many examples which can be typed at the keyboard. A command entered at the terminal is shown like this,

```
$ command
```

The first character on the line is the terminal prompt, and should not be typed. The dollar sign '`$`' is used as the standard prompt in this manual, although some systems may use a different character.

The examples assume the use of the GNU operating system. There may be minor differences in the output on other systems. The commands for setting environment variables use the Bourne shell syntax of the standard GNU shell (`bash`).

2 Using the library

This chapter describes how to compile programs that use GSL, and introduces its conventions.

2.1 An Example Program

The following short program demonstrates the use of the library by computing the value of the Bessel function $J_0(x)$ for $x = 5$,

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>

int
main (void)
{
  double x = 5.0;
  double y = gsl_sf_bessel_J0 (x);
  printf ("J0(%g) = %.18e\n", x, y);
  return 0;
}
```

The output is shown below, and should be correct to double-precision accuracy,[1]

```
J0(5) = -1.775967713143382920e-01
```

The steps needed to compile this program are described in the following sections.

2.2 Compiling and Linking

The library header files are installed in their own 'gsl' directory. You should write any preprocessor include statements with a 'gsl/' directory prefix thus,

```
#include <gsl/gsl_math.h>
```

If the directory is not installed on the standard search path of your compiler you will also need to provide its location to the preprocessor as a command line flag. The default location of the 'gsl' directory is '/usr/local/include/gsl'. A typical compilation command for a source file 'example.c' with the GNU C compiler gcc is,

```
$ gcc -Wall -I/usr/local/include -c example.c
```

This results in an object file 'example.o'. The default include path for gcc searches '/usr/local/include' automatically so the -I option can actually be omitted when GSL is installed in its default location.

---

(1) The last few digits may vary slightly depending on the compiler and platform used—this is normal.

2.2.1 Linking programs with the library

The library is installed as a single file, '`libgsl.a`'. A shared version of the library '`libgsl.so`' is also installed on systems that support shared libraries. The default location of these files is '`/usr/local/lib`'. If this directory is not on the standard search path of your linker you will also need to provide its location as a command line flag.

To link against the library you need to specify both the main library and a supporting CBLAS library, which provides standard basic linear algebra subroutines. A suitable CBLAS implementation is provided in the library '`libgslcblas.a`' if your system does not provide one. The following example shows how to link an application with the library,

```
$ gcc -L/usr/local/lib example.o -lgsl -lgslcblas -lm
```

The default library path for `gcc` searches '`/usr/local/lib`' automatically so the `-L` option can be omitted when GSL is installed in its default location.

2.2.2 Linking with an alternative BLAS library

The following command line shows how you would link the same application with an alternative CBLAS library called '`libcblas`',

```
$ gcc example.o -lgsl -lcblas -lm
```

For the best performance an optimized platform-specific CBLAS library should be used for `-lcblas`. The library must conform to the CBLAS standard. The ATLAS package provides a portable high-performance BLAS library with a CBLAS interface. It is free software and should be installed for any work requiring fast vector and matrix operations. The following command line will link with the ATLAS library and its CBLAS interface,

```
$ gcc example.o -lgsl -lcblas -latlas -lm
```

If the ATLAS library is installed in a non-standard directory use the `-L` option to add it to the search path, as described above.

For more information about BLAS functions see Chapter 12 [BLAS Support], page 137.

2.3 Shared Libraries

To run a program linked with the shared version of the library the operating system must be able to locate the corresponding '`.so`' file at runtime. If the library cannot be found, the following error will occur:

```
$ ./a.out
./a.out: error while loading shared libraries:
libgsl.so.0: cannot open shared object file: No such
file or directory
```

To avoid this error, either modify the system dynamic linker configuration(2) or define the shell variable `LD_LIBRARY_PATH` to include the directory where the library is installed.

---

(2) '`/etc/ld.so.conf`' on GNU/Linux systems.

For example, in the Bourne shell (/bin/sh or /bin/bash), the library search path can be set with the following commands:

```
$ LD_LIBRARY_PATH=/usr/local/lib
$ export LD_LIBRARY_PATH
$ ./example
```

In the C-shell (/bin/csh or /bin/tcsh) the equivalent command is,

```
% setenv LD_LIBRARY_PATH /usr/local/lib
```

The standard prompt for the C-shell in the example above is the percent character '%', and should not be typed as part of the command.

To save retyping these commands each session they can be placed in an individual or system-wide login file.

To compile a statically linked version of the program, use the -static flag in gcc,

```
$ gcc -static example.o -lgsl -lgslcblas -lm
```

2.4 ANSI C Compliance

The library is written in ANSI C and is intended to conform to the ANSI C standard (C89). It should be portable to any system with a working ANSI C compiler.

The library does not rely on any non-ANSI extensions in the interface it exports to the user. Programs you write using GSL can be ANSI compliant. Extensions which can be used in a way compatible with pure ANSI C are supported, however, via conditional compilation. This allows the library 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 at compile-time. This should make it impossible to link a program that would use these functions and give incorrect results.

To avoid namespace conflicts all exported function names and variables have the prefix gsl_, while exported macros have the prefix GSL_.

2.5 Inline functions

The inline keyword is not part of the original ANSI C standard (C89) so the library does not export any inline function definitions by default. Inline functions were introduced officially in the newer C99 standard but most C89 compilers have also included inline as an extension for a long time.

To allow the use of inline functions, the library provides optional inline versions of performance-critical routines by conditional compilation in the exported header files. The inline versions of these functions can be included by defining the macro HAVE_INLINE when compiling an application,

```
$ gcc -Wall -c -DHAVE_INLINE example.c
```

If you use `autoconf` this macro can be defined automatically. If you do not define the macro `HAVE_INLINE` then the slower non-inlined versions of the functions will be used instead.

By default, the actual form of the inline keyword is `extern inline`, which is a `gcc` extension that eliminates unnecessary function definitions. If the form `extern inline` causes problems with other compilers a stricter autoconf test can be used, see Appendix C [Autoconf Macros], page 501.

When compiling with `gcc` in C99 mode (`gcc -std=c99`) the header files automatically switch to C99-compatible inline function declarations instead of `extern inline`. With other C99 compilers, define the macro `GSL_C99_INLINE` to use these declarations.

2.6 Long double

In general, the algorithms in the library are written for double precision only. The `long double` type is not supported for actual computation.

One reason for this choice is that the precision of `long double` is platform dependent. The IEEE standard only specifies the minimum precision of extended precision numbers, while the precision of `double` is the same on all platforms.

However, it is sometimes necessary to interact with external data in long-double format, so the vector and matrix datatypes include long-double versions.

It should be noted that in some system libraries the `stdio.h` formatted input/output functions `printf` and `scanf` are not implemented correctly for `long double`. Undefined or incorrect results are avoided by testing these functions during the `configure` stage of library compilation and eliminating certain GSL functions which depend on them if necessary. The corresponding line in the `configure` output looks like this,

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

Consequently when `long double` formatted input/output does not work on a given system it should be impossible to link a program which uses GSL functions dependent on this.

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

2.7 Portability functions

To help in writing portable applications GSL provides some implementations of functions that are found in other libraries, such as the BSD math library. You can write your application to use the native versions of these functions, and substitute the GSL versions via a preprocessor macro if they are unavailable on another platform.

For example, after determining whether the BSD function `hypot` is available you can include the following macro definitions in a file '`config.h`' with your application,

```
/* Substitute gsl_hypot for missing system hypot */

#ifndef HAVE_HYPOT
#define hypot gsl_hypot
#endif
```

The application source files can then use the include command `#include <config.h>` to replace each occurrence of `hypot` by `gsl_hypot` when `hypot` is not available. This substitution can be made automatically if you use `autoconf`, see Appendix C [Autoconf Macros], page 501.

In most circumstances the best strategy is to use the native versions of these functions when available, and fall back to GSL versions otherwise, since this allows your application to take advantage of any platform-specific optimizations in the system library. This is the strategy used within GSL itself.

2.8 Alternative optimized functions

The main implementation of some functions in the library will not be optimal on all architectures. For example, there are several ways to compute a Gaussian random variate and their relative speeds are platform-dependent. In cases like this the library provides alternative implementations of these functions with the same interface. If you write your application using calls to the standard implementation you can select an alternative version later via a preprocessor definition. It is also possible to introduce your own optimized functions this way while retaining portability. The following lines demonstrate the use of a platform-dependent choice of methods for sampling from the Gaussian distribution,

```
#ifdef SPARC
#define gsl_ran_gaussian gsl_ran_gaussian_ratio_method
#elif INTEL
#define gsl_ran_gaussian my_gaussian
#endif
```

These lines would be placed in the configuration header file '`config.h`' of the application, which should then be included by all the source files. Note that the alternative implementations will not produce bit-for-bit identical results, and in the case of random number distributions will produce an entirely different stream of random variates.

2.9 Support for different numeric types

Many functions in the library are defined for different numeric types. This feature is implemented by varying the name of the function with a type-related modifier—a primitive form of C++ templates. The modifier is inserted into the function name after the initial module prefix. The following table shows the function names defined for all the numeric types of an imaginary module `gsl_foo` with function `fn`,

`gsl_foo_fn`	`double`
`gsl_foo_long_double_fn`	`long double`
`gsl_foo_float_fn`	`float`
`gsl_foo_long_fn`	`long`
`gsl_foo_ulong_fn`	`unsigned long`
`gsl_foo_int_fn`	`int`
`gsl_foo_uint_fn`	`unsigned int`
`gsl_foo_short_fn`	`short`
`gsl_foo_ushort_fn`	`unsigned short`
`gsl_foo_char_fn`	`char`
`gsl_foo_uchar_fn`	`unsigned char`

The normal numeric precision `double` is considered the default and does not require a suffix. For example, the function `gsl_stats_mean` computes the mean of double precision numbers, while the function `gsl_stats_int_mean` computes the mean of integers.

A corresponding scheme is used for library defined types, such as `gsl_vector` and `gsl_matrix`. In this case the modifier is appended to the type name. For example, if a module defines a new type-dependent struct or typedef `gsl_foo` it is modified for other types in the following way,

`gsl_foo`	`double`
`gsl_foo_long_double`	`long double`
`gsl_foo_float`	`float`
`gsl_foo_long`	`long`
`gsl_foo_ulong`	`unsigned long`
`gsl_foo_int`	`int`
`gsl_foo_uint`	`unsigned int`
`gsl_foo_short`	`short`
`gsl_foo_ushort`	`unsigned short`
`gsl_foo_char`	`char`
`gsl_foo_uchar`	`unsigned char`

When a module contains type-dependent definitions the library provides individual header files for each type. The filenames are modified as shown in the below. For convenience the default header includes the definitions for all the types. To include only the double precision header file, or any other specific type, use its individual filename.

```
#include <gsl/gsl_foo.h>               All types
#include <gsl/gsl_foo_double.h>        double
#include <gsl/gsl_foo_long_double.h>   long double
#include <gsl/gsl_foo_float.h>         float
#include <gsl/gsl_foo_long.h>          long
```

```
#include <gsl/gsl_foo_ulong.h>        unsigned long
#include <gsl/gsl_foo_int.h>          int
#include <gsl/gsl_foo_uint.h>         unsigned int
#include <gsl/gsl_foo_short.h>        short
#include <gsl/gsl_foo_ushort.h>       unsigned short
#include <gsl/gsl_foo_char.h>         char
#include <gsl/gsl_foo_uchar.h>        unsigned char
```

2.10 Compatibility with C++

The library header files automatically define functions to have `extern "C"` linkage when included in C++ programs. This allows the functions to be called directly from C++.

To use C++ exception handling within user-defined functions passed to the library as parameters, the library must be built with the additional `CFLAGS` compilation option '`-fexceptions`'.

2.11 Aliasing of arrays

The library assumes that arrays, vectors and matrices passed as modifiable arguments are not aliased and do not overlap with each other. This removes the need for the library to handle overlapping memory regions as a special case, and allows additional optimizations to be used. If overlapping memory regions are passed as modifiable arguments then the results of such functions will be undefined. If the arguments will not be modified (for example, if a function prototype declares them as `const` arguments) then overlapping or aliased memory regions can be safely used.

2.12 Thread-safety

The library can be used in multi-threaded programs. All the functions are thread-safe, in the sense that they do not use static variables. Memory is always associated with objects and not with functions. For functions which use *workspace* objects as temporary storage the workspaces should be allocated on a per-thread basis. For functions which use *table* objects as read-only memory the tables can be used by multiple threads simultaneously. Table arguments are always declared `const` in function prototypes, to indicate that they may be safely accessed by different threads.

There are a small number of static global variables which are used to control the overall behavior of the library (e.g. whether to use range-checking, the function to call on fatal error, etc). These variables are set directly by the user, so they should be initialized once at program startup and not modified by different threads.

2.13 Deprecated Functions

From time to time, it may be necessary for the definitions of some functions to be altered or removed from the library. In these circumstances the functions will first be declared *deprecated* and then removed from subsequent versions of the library. Functions that are deprecated can be disabled in the current release by setting the preprocessor definition `GSL_DISABLE_DEPRECATED`. This allows existing code to be tested for forwards compatibility.

2.14 Code Reuse

Where possible the routines in the library have been written to avoid dependencies between modules and files. This should make it possible to extract individual functions for use in your own applications, without needing to have the whole library installed. You may need to define certain macros such as `GSL_ERROR` and remove some `#include` statements in order to compile the files as standalone units. Reuse of the library code in this way is encouraged, subject to the terms of the GNU General Public License.

3 Error Handling

This chapter describes the way that GSL functions report and handle errors. By examining the status information returned by every 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.

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

3.1 Error Reporting

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

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

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

The 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 functions that you call.

Whenever a 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. The caller can examine the return code and decide what action to take, including ignoring the error if it is not considered serious.

In addition to reporting errors by return codes the library also has an error handler function `gsl_error`. This function is called by other library functions when they report an error, just before they return to the caller. The default behavior of the error handler is to print a message and abort the program,

```
gsl: file.c:67: ERROR: invalid argument supplied by user
Default GSL error handler invoked.
Aborted
```

The purpose of the `gsl_error` handler is to provide a function where a breakpoint can be set that will catch library errors when running under the debugger. It is not intended for use in production programs, which should handle any errors using the return codes.

3.2 Error Codes

The error code numbers returned by library functions are defined in the file 'gsl_errno.h'. They all have the prefix `GSL_` and expand to non-zero constant integer values. Error codes above 1024 are reserved for applications, and are not used by the library. Many of the error codes use the same base name as the corresponding error code in the 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_ENOMEM` 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).

The error codes can be converted into an error message using the function `gsl_strerror`.

`const char * gsl_strerror (const int` *gsl_errno*`)` Function

This function returns a pointer to a string describing the error code *gsl_errno*. For example,

```
printf ("error: %s\n", gsl_strerror (status));
```

would print an error message like `error: output range error` for a status value of `GSL_ERANGE`.

3.3 Error Handlers

The default behavior of the GSL error handler is to print a short message and call `abort`. When this default is in use programs will stop with a core-dump whenever a library routine reports an error. This is intended as a fail-safe default for programs which 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 it is your responsibility to check the return values of routines and handle them yourself. You can also customize the error behavior by providing a new error handler. For example, an alternative error handler could log all errors to a file, ignore certain error conditions (such as underflows), or start the debugger and attach it to the current process when an error occurs.

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

`gsl_error_handler_t` Data Type

This is the type of GSL error handler functions. An error handler will be passed four arguments which specify the reason for the error (a string), the name of the source file in which it occurred (also a string), the line number in that file (an integer) and the error number (an integer). 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,
              int gsl_errno)
```

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 function 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 be only one error handler per program. This function should be not be used in multi-threaded programs except to set up a program-wide error handler from a master thread. The following example shows how to set and restore a new error handler,

```
/* save original handler, install new handler */
old_handler = gsl_set_error_handler (&my_handler);

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

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

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

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

`gsl_error_handler_t * gsl_set_error_handler_off ()` Function

This function turns off the error handler by defining an error handler which does nothing. This will cause the program to continue after any error, so the return values from any library routines must be checked. This is the recommended behavior for production programs. The previous handler is returned (so that you can restore it later).

The error behavior can be changed for specific applications by recompiling the library with a customized definition of the `GSL_ERROR` macro in the file '`gsl_errno.h`'.

3.4 Using GSL error reporting in your own functions

If you are writing numerical functions in a 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 the file '`gsl_errno.h`' defines two macros which 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 tolerance", GSL_ETOL);
  }
```

`GSL_ERROR_VAL` (*reason*, *gsl_errno*, *value*) Macro

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

The following example shows how to return a `NaN` at a mathematical singularity using the `GSL_ERROR_VAL` macro,

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

3.5 Examples

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

```
#include <stdio.h>
#include <gsl/gsl_errno.h>
#include <gsl/gsl_fft_complex.h>

...
  int status;
  size_t n = 37;

  gsl_set_error_handler_off();

  status = gsl_fft_complex_radix2_forward (data, stride, n);

  if (status) {
    if (status == GSL_EINVAL) {
       fprintf (stderr, "invalid argument, n=%d\n", n);
    } 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 `n` 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 function call to `gsl_set_error_handler_off` stops the default error handler from aborting the program. The `else` clause catches any other possible errors.

4 Mathematical Functions

This chapter describes basic mathematical functions. Some of these functions are present in system libraries, but the alternative versions given here can be used as a substitute when the system functions are not available.

The functions and macros described in this chapter are defined in the header file 'gsl_math.h'.

4.1 Mathematical Constants

The library ensures that the standard BSD mathematical constants are defined. For reference, here is a list of the constants:

`M_E`
: The base of exponentials, e

`M_LOG2E`
: The base-2 logarithm of e, $\log_2(e)$

`M_LOG10E`
: The base-10 logarithm of e, $\log_{10}(e)$

`M_SQRT2`
: The square root of two, $\sqrt{2}$

`M_SQRT1_2`
: The square root of one-half, $\sqrt{1/2}$

`M_SQRT3`
: The square root of three, $\sqrt{3}$

`M_PI`
: The constant pi, π

`M_PI_2`
: Pi divided by two, $\pi/2$

`M_PI_4`
: Pi divided by four, $\pi/4$

`M_SQRTPI`
: The square root of pi, $\sqrt{\pi}$

`M_2_SQRTPI`
: Two divided by the square root of pi, $2/\sqrt{\pi}$

`M_1_PI`
: The reciprocal of pi, $1/\pi$

`M_2_PI`
: Twice the reciprocal of pi, $2/\pi$

`M_LN10`
The natural logarithm of ten, $\ln(10)$

`M_LN2`
The natural logarithm of two, $\ln(2)$

`M_LNPI`
The natural logarithm of pi, $\ln(\pi)$

`M_EULER`
Euler's constant, γ

4.2 Infinities and Not-a-number

`GSL_POSINF` Macro
This macro contains the IEEE representation of positive infinity, $+\infty$. It is computed from the expression `+1.0/0.0`.

`GSL_NEGINF` Macro
This macro contains the IEEE representation of negative infinity, $-\infty$. It is computed from the expression `-1.0/0.0`.

`GSL_NAN` Macro
This macro contains the IEEE representation of the Not-a-Number symbol, `NaN`. It is computed from the ratio `0.0/0.0`.

int gsl_isnan (const double *x*) Function
This function returns 1 if x is not-a-number.

int gsl_isinf (const double *x*) Function
This function returns $+1$ if x is positive infinity, -1 if x is negative infinity and 0 otherwise.[1]

int gsl_finite (const double *x*) Function
This function returns 1 if x is a real number, and 0 if it is infinite or not-a-number.

[1] Note that the C99 standard only requires the system `isinf` function to return a non-zero value, without the sign of the infinity. The implementation in some earlier versions of GSL used the system `isinf` function and may have this behavior on some platforms. Therefore, it is advisable to test the sign of x separately, if needed, rather than relying the sign of the return value from `gsl_isinf()`.

4.3 Elementary Functions

The following routines provide portable implementations of functions found in the BSD math library. When native versions are not available the functions described here can be used instead. The substitution can be made automatically if you use `autoconf` to compile your application (see Section 2.7 [Portability functions], page 11).

`double gsl_log1p (const double x)` Function

This function computes the value of $\log(1+x)$ in a way that is accurate for small x. It provides an alternative to the BSD math function `log1p(x)`.

`double gsl_expm1 (const double x)` Function

This function computes the value of $\exp(x)-1$ in a way that is accurate for small x. It provides an alternative to the BSD math function `expm1(x)`.

`double gsl_hypot (const double x, const double y)` Function

This function computes the value of $\sqrt{x^2+y^2}$ in a way that avoids overflow. It provides an alternative to the BSD math function `hypot(x,y)`.

`double gsl_hypot3 (const double x, const double y, const double z)` Function

This function computes the value of $\sqrt{x^2+y^2+z^2}$ in a way that avoids overflow.

`double gsl_acosh (const double x)` Function

This function computes the value of $\operatorname{arccosh}(x)$. It provides an alternative to the standard math function `acosh(x)`.

`double gsl_asinh (const double x)` Function

This function computes the value of $\operatorname{arcsinh}(x)$. It provides an alternative to the standard math function `asinh(x)`.

`double gsl_atanh (const double x)` Function

This function computes the value of $\operatorname{arctanh}(x)$. It provides an alternative to the standard math function `atanh(x)`.

`double gsl_ldexp (double x, int e)` Function

This function computes the value of $x * 2^e$. It provides an alternative to the standard math function `ldexp(x,e)`.

`double gsl_frexp (double x, int * e)` Function

This function splits the number x into its normalized fraction f and exponent e, such that $x = f * 2^e$ and $0.5 \le f < 1$. The function returns f and stores the exponent in e. If x is zero, both f and e are set to zero. This function provides an alternative to the standard math function `frexp(x, e)`.

4.4 Small integer powers

A common complaint about the standard C library is its lack of a function for calculating (small) integer powers. GSL provides some simple functions to fill this gap. For reasons of efficiency, these functions do not check for overflow or underflow conditions.

`double gsl_pow_int (double x, int n)` — Function

This routine computes the power x^n for integer n. The power is computed efficiently—for example, x^8 is computed as $((x^2)^2)^2$, requiring only 3 multiplications. A version of this function which also computes the numerical error in the result is available as `gsl_sf_pow_int_e`.

`double gsl_pow_2 (const double x)` — Function
`double gsl_pow_3 (const double x)` — Function
`double gsl_pow_4 (const double x)` — Function
`double gsl_pow_5 (const double x)` — Function
`double gsl_pow_6 (const double x)` — Function
`double gsl_pow_7 (const double x)` — Function
`double gsl_pow_8 (const double x)` — Function
`double gsl_pow_9 (const double x)` — Function

These functions can be used to compute small integer powers x^2, x^3, etc. efficiently. The functions will be inlined when `HAVE_INLINE` is defined, so that use of these functions should be as efficient as explicitly writing the corresponding product expression.

```
#include <gsl/gsl_math.h>
double y = gsl_pow_4 (3.141)  /* compute 3.141**4 */
```

4.5 Testing the Sign of Numbers

`GSL_SIGN (x)` — Macro

This macro returns the sign of x. It is defined as `((x) >= 0 ? 1 : -1)`. Note that with this definition the sign of zero is positive (regardless of its IEEE sign bit).

4.6 Testing for Odd and Even Numbers

`GSL_IS_ODD (n)` — Macro

This macro evaluates to 1 if n is odd and 0 if n is even. The argument n must be of integer type.

`GSL_IS_EVEN (n)` — Macro

This macro is the opposite of `GSL_IS_ODD(n)`. It evaluates to 1 if n is even and 0 if n is odd. The argument n must be of integer type.

4.7 Maximum and Minimum functions

Note that the following macros perform multiple evaluations of their arguments, so they should not be used with arguments that have side effects (such as a call to a random number generator).

`GSL_MAX (a, b)` — Macro

This macro returns the maximum of *a* and *b*. It is defined as `((a) > (b) ? (a):(b))`.

`GSL_MIN (a, b)` — Macro

This macro returns the minimum of *a* and *b*. It is defined as `((a) < (b) ? (a):(b))`.

`extern inline double GSL_MAX_DBL (double a, double b)` — Function

This function returns the maximum of the double precision numbers *a* and *b* using an inline function. The use of a function allows for type checking of the arguments as an extra safety feature. On platforms where inline functions are not available the macro `GSL_MAX` will be automatically substituted.

`extern inline double GSL_MIN_DBL (double a, double b)` — Function

This function returns the minimum of the double precision numbers *a* and *b* using an inline function. The use of a function allows for type checking of the arguments as an extra safety feature. On platforms where inline functions are not available the macro `GSL_MIN` will be automatically substituted.

`extern inline int GSL_MAX_INT (int a, int b)` — Function
`extern inline int GSL_MIN_INT (int a, int b)` — Function

These functions return the maximum or minimum of the integers *a* and *b* using an inline function. On platforms where inline functions are not available the macros `GSL_MAX` or `GSL_MIN` will be automatically substituted.

`extern inline long double GSL_MAX_LDBL (long double a, long double b)` — Function
`extern inline long double GSL_MIN_LDBL (long double a, long double b)` — Function

These functions return the maximum or minimum of the long doubles *a* and *b* using an inline function. On platforms where inline functions are not available the macros `GSL_MAX` or `GSL_MIN` will be automatically substituted.

4.8 Approximate Comparison of Floating Point Numbers

It is sometimes useful to be able to compare two floating point numbers approximately, to allow for rounding and truncation errors. The following function implements the approximate floating-point comparison algorithm proposed by D.E. Knuth in Section 4.2.2 of *Seminumerical Algorithms* (3rd edition).

`int gsl_fcmp (double x, double y, double epsilon)` Function

This function determines whether x and y are approximately equal to a relative accuracy *epsilon*.

The relative accuracy is measured using an interval of size 2δ, where $\delta = 2^k \epsilon$ and k is the maximum base-2 exponent of x and y as computed by the function `frexp`.

If x and y lie within this interval, they are considered approximately equal and the function returns 0. Otherwise if $x < y$, the function returns -1, or if $x > y$, the function returns $+1$.

Note that x and y are compared to relative accuracy, so this function is not suitable for testing whether a value is approximately zero.

The implementation is based on the package `fcmp` by T.C. Belding.

5 Complex Numbers

The functions described in this chapter provide support for complex numbers. The algorithms take care to avoid unnecessary intermediate underflows and overflows, allowing the functions to be evaluated over as much of the complex plane as possible.

For multiple-valued functions the branch cuts have been chosen to follow the conventions of Abramowitz and Stegun in the *Handbook of Mathematical Functions*. The functions return principal values which are the same as those in GNU Calc, which in turn are the same as those in *Common Lisp, The Language (Second Edition)*[1] and the HP-28/48 series of calculators.

The complex types are defined in the header file '`gsl_complex.h`', while the corresponding complex functions and arithmetic operations are defined in '`gsl_complex_math.h`'.

5.1 Representation of complex numbers

Complex numbers are represented using the type `gsl_complex`. The internal representation of this type may vary across platforms and should not be accessed directly. The functions and macros described below allow complex numbers to be manipulated in a portable way.

For reference, the default form of the `gsl_complex` type is given by the following struct,

```
typedef struct
{
  double dat[2];
} gsl_complex;
```

The real and imaginary part are stored in contiguous elements of a two element array. This eliminates any padding between the real and imaginary parts, `dat[0]` and `dat[1]`, allowing the struct to be mapped correctly onto packed complex arrays.

`gsl_complex gsl_complex_rect (double` *x*`, double` *y*`)` Function

This function uses the rectangular cartesian components (x,y) to return the complex number $z = x + iy$. An inline version of this function is used when `HAVE_INLINE` is defined.

`gsl_complex gsl_complex_polar (double` *r*`, double` *theta*`)` Function

This function returns the complex number $z = r\exp(i\theta) = r(\cos(\theta) + i\sin(\theta))$ from the polar representation (r,*theta*).

`GSL_REAL (`*z*`)` Macro
`GSL_IMAG (`*z*`)` Macro

These macros return the real and imaginary parts of the complex number z.

(1) Note that the first edition uses different definitions.

`GSL_SET_COMPLEX` (*zp*, *x*, *y*) Macro

This macro uses the cartesian components (*x*,*y*) to set the real and imaginary parts of the complex number pointed to by *zp*. For example,

```
GSL_SET_COMPLEX(&z, 3, 4)
```

sets z to be $3 + 4i$.

`GSL_SET_REAL` (*zp*,*x*) Macro
`GSL_SET_IMAG` (*zp*,*y*) Macro

These macros allow the real and imaginary parts of the complex number pointed to by *zp* to be set independently.

5.2 Properties of complex numbers

`double gsl_complex_arg` (`gsl_complex` *z*) Function

This function returns the argument of the complex number z, $\arg(z)$, where $-\pi < \arg(z) \leq \pi$.

`double gsl_complex_abs` (`gsl_complex` *z*) Function

This function returns the magnitude of the complex number z, $|z|$.

`double gsl_complex_abs2` (`gsl_complex` *z*) Function

This function returns the squared magnitude of the complex number z, $|z|^2$.

`double gsl_complex_logabs` (`gsl_complex` *z*) Function

This function returns the natural logarithm of the magnitude of the complex number z, $\log|z|$. It allows an accurate evaluation of $\log|z|$ when $|z|$ is close to one. The direct evaluation of `log(gsl_complex_abs(z))` would lead to a loss of precision in this case.

5.3 Complex arithmetic operators

`gsl_complex gsl_complex_add` (`gsl_complex` *a*, `gsl_complex` *b*) Function

This function returns the sum of the complex numbers a and b, $z = a + b$.

`gsl_complex gsl_complex_sub` (`gsl_complex` *a*, `gsl_complex` *b*) Function

This function returns the difference of the complex numbers a and b, $z = a - b$.

`gsl_complex gsl_complex_mul` (`gsl_complex` *a*, `gsl_complex` *b*) Function

This function returns the product of the complex numbers a and b, $z = ab$.

`gsl_complex gsl_complex_div` (`gsl_complex` *a*, `gsl_complex` *b*) Function

This function returns the quotient of the complex numbers a and b, $z = a/b$.

`gsl_complex gsl_complex_add_real` (`gsl_complex` *a*, `double` *x*) Function

This function returns the sum of the complex number a and the real number x, $z = a + x$.

Function Index

C

G

Type and Variable Index

H

I

M

O

R

S

V

Concept Index

C

D

E

F

G

H

I

J

K

L

M

N

R

S

T

U

V

W

Y

Z