CML

Release 1.9.10

Ulises Jeremias Cornejo Fandos

Feb 26, 2018

Contents

CHAPTER 1

Introduction

The C Math Library (CML) 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 MIT License.

1.1 Routines available in CML

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

Complex Numbers	Special Functions	Vectors and Matrices
Quaternions	Differential Equations	Numerical Differentiation
IEEE Floating-Point	Physical Constants	Easing Functions

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.

1.2 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`).

CHAPTER 2

Using the Library

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

2.1 An Example Program

The following short program demonstrates the use of the library

```
#include <stdlib.h>
#include <stdio.h>
#include <cml.h>

int
main(int argc, char const *argv[])
{
        cml_complex_t z, w;

        z = complex(1.0, 2.0);
        w = csin(z);

        printf("%g\n", sin(2.0));
        printf("%g\n", atan2(2.0, 3.0));
        printf("%g\n", creal(w));
        printf("%g\n", cimag(w));

        return 0;
}
```

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 `cml` directory. You should write any preprocessor include statements with a `cml/` directory prefix thus:

```
#include <cml/math.h>
```

or simply requiring all the modules in the following way:

```
#include <cml.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 main header file `cml.h` and the `cml` directory is `/usr/local/include`. 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 CML is installed in its default location.

2.2.1 Linking programs with the library

The library is installed as a single file, `libcml.a`. A shared version of the library `libcml.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. The following example shows how to link an application with the library:

```
$ gcc -L/usr/local/lib example.o -lcml
```

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

For a tutorial introduction to the GNU C Compiler and related programs, see "An Introduction to GCC" (ISBN 0954161793).[1]

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:
libcml.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.

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

[1] http://www.network-theory.co.uk/gcc/intro/
[2] `/etc/ld.so.conf` on GNU/Linux systems

```
$ 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 -lcml
```

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 CML 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 `cml_`, while exported macros have the prefix `CML_`.

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.

By default, the actual form of the inline keyword is `extern inline`, which is a `gcc` extension that eliminates unnecessary function definitions.

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.

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 every 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 structures 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 CML 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 CML 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 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++.

2.8 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.

CHAPTER 3

Error Handling

This chapter describes the way that CML 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 chapter are declared in the header file `cml/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 = cml_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 `cml_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:

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

The purpose of the `cml_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 `cml/errno.h`. They all have the prefix `CML_` 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 **CML_EDOM**
: 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 **CML_ERANGE**
: 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 **CML_ENOMEM**
: 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 CML routine encounters problems when trying to allocate memory with `malloc()`.

int **CML_EINVAL**
: 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 *cml_strerror()*.

const char * **cml_strerror** (const int *cml_errno*)
: This function returns a pointer to a string describing the error code `cml_errno`. For example:

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

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

3.3 Error Handlers

The default behavior of the CML 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 CML error handlers have the type `cml_error_handler_t`, which is defined in `cml_errno.h`,

cml_error_handler_t
This is the type of CML 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 cml_errno)
```

To request the use of your own error handler you need to call the function *cml_set_error_handler()* which is also declared in cml_errno.h,

cml_error_handler_t * **cml_set_error_handler** (*cml_error_handler_t* * *new_handler*)
This function sets a new error handler, new_handler, for the CML 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 = cml_set_error_handler (&my_handler);

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

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

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

```
old_handler = cml_set_error_handler (NULL);
```

cml_error_handler_t * **cml_set_error_handler_off** ()
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 CML_ERROR macro in the file cml_errno.h.

3.4 Using CML error reporting in your own functions

If you are writing numerical functions in a program which also uses CML 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 cml_error() with a string describing the error and then return an appropriate error code from cml_errno.h, or a special value, such as NaN. For convenience the file cml_errno.h defines two macros which carry out these steps:

CML_ERROR (reason, cml_errno)
This macro reports an error using the CML conventions and returns a status value of cml_errno. It expands to the following code fragment:

```
cml_error (reason, __FILE__, __LINE__, cml_errno);
return cml_errno;
```

The macro definition in `cml_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 `CML_ETOL`:

```
if (residual > tolerance)
  {
    CML_ERROR("residual exceeds tolerance", CML_ETOL);
  }
```

CML_ERROR_VAL (reason, cml_errno, value)

This macro is the same as `CML_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 `CML_ERROR_VAL` macro:

```
if (x == 0)
  {
    CML_ERROR_VAL("argument lies on singularity", CML_ERANGE, CML_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 <cml/cml_errno.h>
#include <cml/cml_fft_complex.h>

...
  int status;
  size_t n = 37;

  cml_set_error_handler_off();

  status = cml_fft_cml_complex_radix2_forward (data, stride, n);

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

The function `cml_fft_cml_complex_radix2_forward()` 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 `CML_EINVAL`, indicating that the length argument is invalid. The function call to *`cml_set_error_handler_off()`* stops the default error handler from aborting the program. The `else` clause catches any other possible errors.

CHAPTER 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 `cml/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

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

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

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

bool **cml_isnan** (double *x*)
This function returns 1 if `x` is not-a-number.

bool **cml_isinf** (double *x*)
This function returns $+1$ if `x` is positive infinity, -1 if `x` is negative infinity and 0 otherwise.[1]

bool **cml_isfinite** (double *x*)
This function returns 1 if `x` is a real number, and 0 if it is infinite or not-a-number.

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 portability-functions).

double **cml_log1p** (double *x*)
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 **cml_expm1** (double *x*)
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 **cml_hypot** (double *x*, double *y*)
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 **cml_hypot3** (double *x*, double *y*, double *cml_z*)
This function computes the value of $\sqrt{x^2+y^2+z^2}$ in a way that avoids overflow.

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

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

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

[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 CML 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 `isinf()`.

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

double **cml_frexp** (double *x*, int *e)
This function splits the number `x` into its normalized fraction f and exponent e, such that $x = f * 2^e$ and $0.5 <= f < 1$. 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. CML provides some simple functions to fill this gap. For reasons of efficiency, these functions do not check for overflow or underflow conditions.

double **cml_pow_int** (double *x*, int *n*)
double **cml_pow_uint** (double *x*, unsigned int *n*)
These routines 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.

double **cml_pow_2** (double *x*)
double **cml_pow_3** (double *x*)
double **cml_pow_4** (double *x*)
double **cml_pow_5** (double *x*)
double **cml_pow_6** (double *x*)
double **cml_pow_7** (double *x*)
double **cml_pow_8** (double *x*)
double **cml_pow_9** (double *x*)
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 <cml.h>
[...]
double y = pow_4(3.141);  /* compute 3.141**4 */
```

4.5 Testing the Sign of Numbers

double **cml_sgn** (double *x*)
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 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).

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

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

4.7 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).

bool **cml_cmp**(double *x*, double *y*, double *epsilon*)
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.

CHAPTER 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.

The complex types, functions and arithmetic operations are defined in the header file `cml/complex.h`.

5.1 Representation of complex numbers

Complex numbers are represented using the type `cml_complex_t`. 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 `cml_complex_t` type is given by the following struct:

```
typedef struct _complex
{
        union
        {
                double p[2];
                double parts[2];
                struct
                {
                        double re;
                        double im;
                };
                struct
                {
                        double real;
                        double imaginary;
                };
        };
} cml_complex_t;
```

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, `parts[0]` and `parts[1]`, allowing the struct to be mapped correctly onto packed complex arrays.

cml_complex_t **complex** (double *x*, double *y*)
: This function uses the rectangular Cartesian components (x, y) to return the complex number $z = x + yi$. An inline version of this function is used when `HAVE_INLINE` is defined.

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

creal (`z`)
cimag (`z`)
: These macros return the real and imaginary parts of the complex number `z`.

5.2 Properties of complex numbers

double **cml_complex_arg** (cml_complex_t z)
: This function returns the argument of the complex number `z`, $\arg(z)$, where $-\pi < \arg(z) <= \pi$.

double **cml_complex_abs** (cml_complex_t z)
: This function returns the magnitude of the complex number `z`, $|z|$.

double **cml_complex_abs2** (cml_complex_t z)
: This function returns the squared magnitude of the complex number `z`, $|z|^2$.

double **cml_complex_logabs** (cml_complex_t z)
: 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(cml_complex_abs(z))` would lead to a loss of precision in this case.

5.3 Complex arithmetic operators

cml_complex_t **cml_complex_add** (cml_complex_t *a*, cml_complex_t *b*)
: This function returns the sum of the complex numbers `a` and `b`, $z = a + b$.

cml_complex_t **cml_complex_sub** (cml_complex_t *a*, cml_complex_t *b*)
: This function returns the difference of the complex numbers `a` and `b`, $z = a - b$.

cml_complex_t **cml_complex_mul** (cml_complex_t *a*, cml_complex_t *b*)
: This function returns the product of the complex numbers `a` and `b`, $z = ab$.

cml_complex_t **cml_complex_div** (cml_complex_t *a*, cml_complex_t *b*)
: This function returns the quotient of the complex numbers `a` and `b`, $z = a/b$.

cml_complex_t **cml_complex_add_real** (cml_complex_t *a*, double *x*)
: This function returns the sum of the complex number `a` and the real number `x`, $z = a + x$.

cml_complex_t **cml_complex_sub_real** (cml_complex_t *a*, double *x*)
: This function returns the difference of the complex number `a` and the real number `x`, $z = a - x$.

cml_complex_t **cml_complex_mul_real** (cml_complex_t *a*, double *x*)
: This function returns the product of the complex number `a` and the real number `x`, $z = ax$.

cml_complex_t **cml_complex_div_real** (cml_complex_t *a*, double *x*)
: This function returns the quotient of the complex number `a` and the real number `x`, $z = a/x$.

cml_complex_t **cml_complex_add_imag** (cml_complex_t *a*, double *y*)
This function returns the sum of the complex number a and the imaginary number iy, $z = a + iy$.

cml_complex_t **cml_complex_sub_imag** (cml_complex_t *a*, double *y*)
This function returns the difference of the complex number a and the imaginary number iy, $z = a - iy$.

cml_complex_t **cml_complex_mul_imag** (cml_complex_t *a*, double *y*)
This function returns the product of the complex number a and the imaginary number iy, $z = a * (iy)$.

cml_complex_t **cml_complex_div_imag** (cml_complex_t *a*, double *y*)
This function returns the quotient of the complex number a and the imaginary number iy, $z = a/(iy)$.

cml_complex_t **cml_complex_conj** (cml_complex_t *z*)
This function returns the complex conjugate of the complex number z, $z^* = x - yi$.

cml_complex_t **cml_complex_inverse** (cml_complex_t *z*)
This function returns the inverse, or reciprocal, of the complex number z, $1/z = (x - yi)/(x^2 + y^2)$.

cml_complex_t **cml_complex_negative** (cml_complex_t *z*)
This function returns the negative of the complex number z, $-z = (-x) + (-y)i$.

5.4 Elementary Complex Functions

cml_complex_t **cml_complex_sqrt** (cml_complex_t *z*)
This function returns the square root of the complex number z, $\sqrt{z}$. The branch cut is the negative real axis. The result always lies in the right half of the complex plane.

cml_complex_t **cml_complex_sqrt_real** (double *x*)
This function returns the complex square root of the real number x, where x may be negative.

cml_complex_t **cml_complex_pow** (cml_complex_t *z*, cml_complex_t *a*)
The function returns the complex number z raised to the complex power a, z^a. This is computed as $\exp(\log(z) * a)$ using complex logarithms and complex exponentials.

cml_complex_t **cml_complex_pow_real** (cml_complex_t *z*, double *x*)
This function returns the complex number z raised to the real power x, z^x.

cml_complex_t **cml_complex_exp** (cml_complex_t *z*)
This function returns the complex exponential of the complex number z, $\exp(z)$.

cml_complex_t **cml_complex_log** (cml_complex_t *z*)
This function returns the complex natural logarithm (base e) of the complex number z, $\log(z)$. The branch cut is the negative real axis.

cml_complex_t **cml_complex_log10** (cml_complex_t *z*)
This function returns the complex base-10 logarithm of the complex number z, $\log_{10}(z)$.

cml_complex_t **cml_complex_log_b** (cml_complex_t *z*, cml_complex_t *b*)
This function returns the complex base-b logarithm of the complex number z, $\log_b(z)$. This quantity is computed as the ratio $\log(z)/\log(b)$.

5.5 Complex Trigonometric Functions

cml_complex_t **cml_complex_sin** (cml_complex_t *z*)
This function returns the complex sine of the complex number z, $\sin(z) = (\exp(iz) - \exp(-iz))/(2i)$.

cml_complex_t **cml_complex_cos** (cml_complex_t *z*)
This function returns the complex cosine of the complex number z, $\cos(z) = (\exp(iz) + \exp(-iz))/2$.

cml_complex_t **cml_complex_tan**(cml_complex_t z)
This function returns the complex tangent of the complex number z, $\tan(z) = \sin(z)/\cos(z)$.

cml_complex_t **cml_complex_sec**(cml_complex_t z)
This function returns the complex secant of the complex number z, $\sec(z) = 1/\cos(z)$.

cml_complex_t **cml_complex_csc**(cml_complex_t z)
This function returns the complex cosecant of the complex number z, $\csc(z) = 1/\sin(z)$.

cml_complex_t **cml_complex_cot**(cml_complex_t z)
This function returns the complex cotangent of the complex number z, $\cot(z) = 1/\tan(z)$.

5.6 Inverse Complex Trigonometric Functions

cml_complex_t **cml_complex_asin**(cml_complex_t z)
This function returns the complex arcsine of the complex number z, $\arcsin(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

cml_complex_t **cml_complex_asin_real**(double z)
This function returns the complex arcsine of the real number z, $\arcsin(z)$. For z between -1 and 1, the function returns a real value in the range $[-\pi/2, \pi/2]$. For z less than -1 the result has a real part of $-\pi/2$ and a positive imaginary part. For z greater than 1 the result has a real part of $\pi/2$ and a negative imaginary part.

cml_complex_t **cml_complex_acos**(cml_complex_t z)
This function returns the complex arccosine of the complex number z, $\arccos(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

cml_complex_t **cml_complex_acos_real**(double z)
This function returns the complex arccosine of the real number z, $\arccos(z)$. For z between -1 and 1, the function returns a real value in the range $[0, \pi]$. For z less than -1 the result has a real part of π and a negative imaginary part. For z greater than 1 the result is purely imaginary and positive.

cml_complex_t **cml_complex_atan**(cml_complex_t z)
This function returns the complex arctangent of the complex number z, $\arctan(z)$. The branch cuts are on the imaginary axis, below $-i$ and above i.

cml_complex_t **cml_complex_asec**(cml_complex_t z)
This function returns the complex arcsecant of the complex number z, $\text{arcsec}(z) = \arccos(1/z)$.

cml_complex_t **cml_complex_asec_real**(double z)
This function returns the complex arcsecant of the real number z, $\text{arcsec}(z) = \arccos(1/z)$.

cml_complex_t **cml_complex_acsc**(cml_complex_t z)
This function returns the complex arccosecant of the complex number z, $\text{arccsc}(z) = \arcsin(1/z)$.

cml_complex_t **cml_complex_acsc_real**(double z)
This function returns the complex arccosecant of the real number z, $\text{arccsc}(z) = \arcsin(1/z)$.

cml_complex_t **cml_complex_acot**(cml_complex_t z)
This function returns the complex arccotangent of the complex number z, $\text{arccot}(z) = \arctan(1/z)$.

5.7 Complex Hyperbolic Functions

cml_complex_t **cml_complex_sinh**(cml_complex_t z)
This function returns the complex hyperbolic sine of the complex number z, $\sinh(z) = (\exp(z) - \exp(-z))/2$.

cml_complex_t **cml_complex_cosh** (cml_complex_t z)
This function returns the complex hyperbolic cosine of the complex number z, $\cosh(z) = (\exp(z) + \exp(-z))/2$.

cml_complex_t **cml_complex_tanh** (cml_complex_t z)
This function returns the complex hyperbolic tangent of the complex number z, $\tanh(z) = \sinh(z)/\cosh(z)$.

cml_complex_t **cml_complex_sech** (cml_complex_t z)
This function returns the complex hyperbolic secant of the complex number z, $\mathrm{sech}(z) = 1/\cosh(z)$.

cml_complex_t **cml_complex_csch** (cml_complex_t z)
This function returns the complex hyperbolic cosecant of the complex number z, $\mathrm{csch}(z) = 1/\sinh(z)$.

cml_complex_t **cml_complex_coth** (cml_complex_t z)
This function returns the complex hyperbolic cotangent of the complex number z, $\coth(z) = 1/\tanh(z)$.

5.8 Inverse Complex Hyperbolic Functions

cml_complex_t **cml_complex_asinh** (cml_complex_t z)
This function returns the complex hyperbolic arcsine of the complex number z, $\mathrm{arcsinh}(z)$. The branch cuts are on the imaginary axis, below $-i$ and above i.

cml_complex_t **cml_complex_acosh** (cml_complex_t z)
This function returns the complex hyperbolic arccosine of the complex number z, $\mathrm{arccosh}(z)$. The branch cut is on the real axis, less than 1. Note that in this case we use the negative square root in formula 4.6.21 of Abramowitz & Stegun giving $\mathrm{arccosh}(z) = \log(z - \sqrt{z^2 - 1})$.

cml_complex_t **cml_complex_acosh_real** (double z)
This function returns the complex hyperbolic arccosine of the real number z, $\mathrm{arccosh}(z)$.

cml_complex_t **cml_complex_atanh** (cml_complex_t z)
This function returns the complex hyperbolic arctangent of the complex number z, $\mathrm{arctanh}(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

cml_complex_t **cml_complex_atanh_real** (double z)
This function returns the complex hyperbolic arctangent of the real number z, $\mathrm{arctanh}(z)$.

cml_complex_t **cml_complex_asech** (cml_complex_t z)
This function returns the complex hyperbolic arcsecant of the complex number z, $\mathrm{arcsech}(z) = \mathrm{arccosh}(1/z)$.

cml_complex_t **cml_complex_acsch** (cml_complex_t z)
This function returns the complex hyperbolic arccosecant of the complex number z, $\mathrm{arccsch}(z) = \mathrm{arcsinh}(1/z)$.

cml_complex_t **cml_complex_acoth** (cml_complex_t z)
This function returns the complex hyperbolic arccotangent of the complex number z, $\mathrm{arccoth}(z) = \mathrm{arctanh}(1/z)$.

5.9 References and Further Reading

The implementations of the elementary and trigonometric functions are based on the following papers,

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, "Implementing Complex Elementary Functions Using Exception Handling", ACM Transactions on Mathematical Software, Volume 20 (1994), pp 215–244, Corrigenda, p553

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, "Implementing the complex arcsin and arccosine functions using exception handling", ACM Transactions on Mathematical Software, Volume 23 (1997) pp 299–335

The general formulas and details of branch cuts can be found in the following books,

- Abramowitz and Stegun, Handbook of Mathematical Functions, "Circular Functions in Terms of Real and Imaginary Parts", Formulas 4.3.55–58, "Inverse Circular Functions in Terms of Real and Imaginary Parts", Formulas 4.4.37–39, "Hyperbolic Functions in Terms of Real and Imaginary Parts", Formulas 4.5.49–52, "Inverse Hyperbolic Functions—relation to Inverse Circular Functions", Formulas 4.6.14–19.
- Dave Gillespie, Calc Manual, Free Software Foundation, ISBN 1-882114-18-3

CHAPTER 6

Numerical Differentiation

The functions described in this chapter compute numerical derivatives by finite differencing. An adaptive algorithm is used to find the best choice of finite difference and to estimate the error in the derivative.

The functions described in this chapter are declared in the header file `cml/deriv.h`.

6.1 Functions

int **`cml_deriv_central`** (const cml_function_t **f*, double *x*, double *h*, double **result*, double **abserr*)

This function computes the numerical derivative of the function `f` at the point `x` using an adaptive central difference algorithm with a step-size of `h`. The derivative is returned in `result` and an estimate of its absolute error is returned in `abserr`.

The initial value of `h` is used to estimate an optimal step-size, based on the scaling of the truncation error and round-off error in the derivative calculation. The derivative is computed using a 5-point rule for equally spaced abscissae at $x - h$, $x - h/2$, x, $x + h/2$, $x + h$, with an error estimate taken from the difference between the 5-point rule and the corresponding 3-point rule $x - h$, x, $x + h$. Note that the value of the function at x does not contribute to the derivative calculation, so only 4-points are actually used.

int **`cml_deriv_forward`** (const cml_function_t **f*, double *x*, double *h*, double **result*, double **abserr*)

This function computes the numerical derivative of the function `f` at the point `x` using an adaptive forward difference algorithm with a step-size of `h`. The function is evaluated only at points greater than `x`, and never at `x` itself. The derivative is returned in `result` and an estimate of its absolute error is returned in `abserr`. This function should be used if $f(x)$ has a discontinuity at `x`, or is undefined for values less than `x`.

The initial value of `h` is used to estimate an optimal step-size, based on the scaling of the truncation error and round-off error in the derivative calculation. The derivative at x is computed using an "open" 4-point rule for equally spaced abscissae at $x + h/4$, $x + h/2$, $x + 3h/4$, $x + h$, with an error estimate taken from the difference between the 4-point rule and the corresponding 2-point rule $x + h/2$, $x + h$.

int **`cml_deriv_backward`** (const cml_function_t **f*, double *x*, double *h*, double **result*, double **abserr*)

This function computes the numerical derivative of the function `f` at the point `x` using an adaptive backward difference algorithm with a step-size of `h`. The function is evaluated only at points less than `x`, and never at `x`

itself. The derivative is returned in `result` and an estimate of its absolute error is returned in `abserr`. This function should be used if $f(x)$ has a discontinuity at `x`, or is undefined for values greater than `x`.

This function is equivalent to calling *cml_deriv_forward()* with a negative step-size.

6.2 Examples

The following code estimates the derivative of the function $f(x) = x^{3/2}$ at $x = 2$ and at $x = 0$. The function $f(x)$ is undefined for $x < 0$ so the derivative at $x = 0$ is computed using *cml_deriv_forward()*.

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

double
f(double x, void *params)
{
        (void)(params); /* avoid unused parameter warning */
        return pow(x, 1.5);
}

int
main(void)
{
        cml_function_t F;
        double result, abserr;

        F.function = &f;
        F.params = 0;

        printf("f(x) = x^(3/2)\n");

        cml_deriv_central(&F, 2.0, 1e-8, &result, &abserr);
        printf("x = 2.0\n");
        printf("f'(x) = %.10f +/- %.10f\n", result, abserr);
        printf("exact = %.10f\n\n", 1.5 * sqrt(2.0));

        cml_deriv_forward (&F, 0.0, 1e-8, &result, &abserr);
        printf("x = 0.0\n");
        printf("f'(x) = %.10f +/- %.10f\n", result, abserr);
        printf("exact = %.10f\n", 0.0);

        return 0;
}
```

Here is the output of the program,

```
f(x) = x^(3/2)
x = 2.0
f'(x) = 2.1213203120 +/- 0.0000005006
exact = 2.1213203436

x = 0.0
f'(x) = 0.0000000160 +/- 0.0000000339
exact = 0.0000000000
```

6.3 References and Further Reading

The algorithms used by these functions are described in the following sources:

- Abramowitz and Stegun, *Handbook of Mathematical Functions*, Section 25.3.4, and Table 25.5 (Coefficients for Differentiation).
- S.D. Conte and Carl de Boor, *Elementary Numerical Analysis: An Algorithmic Approach*, McGraw-Hill, 1972.

CHAPTER 7

Easings Functions

The functions described in this chapter are declared in the header file `cml/easings.h`.

The easing functions are an implementation of the functions presented in http://easings.net/, useful particularly for animations. Easing is a method of distorting time to control apparent motion in animation. It is most commonly used for slow-in, slow-out. By easing time, animated transitions are smoother and exhibit more plausible motion.

Easing functions take a value inside the range `[0.0, 1.0]` and usually will return a value inside that same range. However, in some of the easing functions, the returned value extrapolate that range http://easings.net/ to see those functions).

The following types of easing functions are supported:

```
Linear
Quadratic
Cubic
Quartic
Quintic
Sine
Circular
Exponential
Elastic
Bounce
Back
```

The core easing functions are implemented as C functions that take a time parameter and return a progress parameter, which can subsequently be used to interpolate any quantity.

CHAPTER 8

Physical Constants

This chapter describes macros for the values of physical constants, such as the speed of light, c, and gravitational constant, G. The values are available in different unit systems, including the standard MKSA system (meters, kilograms, seconds, amperes) and the CGSM system (centimeters, grams, seconds, gauss), which is commonly used in Astronomy.

The full list of constants is described briefly below. Consult the header files themselves for the values of the constants used in the library.

8.1 Fundamental Constants

CML_CONST_MKSA_SPEED_OF_LIGHT
The speed of light in vacuum, c.

CML_CONST_MKSA_VACUUM_PERMEABILITY
The permeability of free space, μ_0. This constant is defined in the MKSA system only.

CML_CONST_MKSA_VACUUM_PERMITTIVITY
The permittivity of free space, ϵ_0. This constant is defined in the MKSA system only.

CML_CONST_MKSA_PLANCKS_CONSTANT_H
Planck's constant, h.

CML_CONST_MKSA_PLANCKS_CONSTANT_HBAR
Planck's constant divided by 2π, $\hbar$.

CML_CONST_NUM_AVOGADRO
Avogadro's number, N_a.

CML_CONST_MKSA_FARADAY
The molar charge of 1 Faraday.

CML_CONST_MKSA_BOLTZMANN
The Boltzmann constant, k.

CML_CONST_MKSA_MOLAR_GAS
The molar gas constant, R_0.

CML_CONST_MKSA_STANDARD_GAS_VOLUME
The standard gas volume, V_0.

CML_CONST_MKSA_STEFAN_BOLTZMANN_CONSTANT
The Stefan-Boltzmann radiation constant, σ.

CML_CONST_MKSA_GAUSS
The magnetic field of 1 Gauss.

8.2 Astronomy and Astrophysics

CML_CONST_MKSA_ASTRONOMICAL_UNIT
The length of 1 astronomical unit (mean earth-sun distance), au.

CML_CONST_MKSA_GRAVITATIONAL_CONSTANT
The gravitational constant, G.

CML_CONST_MKSA_LIGHT_YEAR
The distance of 1 light-year, ly.

CML_CONST_MKSA_PARSEC
The distance of 1 parsec, pc.

CML_CONST_MKSA_GRAV_ACCEL
The standard gravitational acceleration on Earth, g.

CML_CONST_MKSA_SOLAR_MASS
The mass of the Sun.

8.3 Atomic and Nuclear Physics

CML_CONST_MKSA_ELECTRON_CHARGE
The charge of the electron, e.

CML_CONST_MKSA_ELECTRON_VOLT
The energy of 1 electron volt, eV.

CML_CONST_MKSA_UNIFIED_ATOMIC_MASS
The unified atomic mass, amu.

CML_CONST_MKSA_MASS_ELECTRON
The mass of the electron, m_e.

CML_CONST_MKSA_MASS_MUON
The mass of the muon, m_μ.

CML_CONST_MKSA_MASS_PROTON
The mass of the proton, m_p.

CML_CONST_MKSA_MASS_NEUTRON
The mass of the neutron, m_n.

CML_CONST_NUM_FINE_STRUCTURE
The electromagnetic fine structure constant α.

CML_CONST_MKSA_RYDBERG
The Rydberg constant, Ry, in units of energy. This is related to the Rydberg inverse wavelength R_∞ by $Ry = hcR_\infty$.

CML_CONST_MKSA_BOHR_RADIUS
The Bohr radius, a_0.

CML_CONST_MKSA_ANGSTROM
The length of 1 angstrom.

CML_CONST_MKSA_BARN
The area of 1 barn.

CML_CONST_MKSA_BOHR_MAGNETON
The Bohr Magneton, μ_B.

CML_CONST_MKSA_NUCLEAR_MAGNETON
The Nuclear Magneton, μ_N.

CML_CONST_MKSA_ELECTRON_MAGNETIC_MOMENT
The absolute value of the magnetic moment of the electron, μ_e. The physical magnetic moment of the electron is negative.

CML_CONST_MKSA_PROTON_MAGNETIC_MOMENT
The magnetic moment of the proton, μ_p.

CML_CONST_MKSA_THOMSON_CROSS_SECTION
The Thomson cross section, σ_T.

CML_CONST_MKSA_DEBYE
The electric dipole moment of 1 Debye, D.

8.4 Measurement of Time

CML_CONST_MKSA_MINUTE
The number of seconds in 1 minute.

CML_CONST_MKSA_HOUR
The number of seconds in 1 hour.

CML_CONST_MKSA_DAY
The number of seconds in 1 day.

CML_CONST_MKSA_WEEK
The number of seconds in 1 week.

8.5 Imperial Units

CML_CONST_MKSA_INCH
The length of 1 inch.

CML_CONST_MKSA_FOOT
The length of 1 foot.

CML_CONST_MKSA_YARD
The length of 1 yard.

CML_CONST_MKSA_MILE
The length of 1 mile.

CML_CONST_MKSA_MIL
The length of 1 mil (1/1000th of an inch).

8.6 Speed and Nautical Units

CML_CONST_MKSA_KILOMETERS_PER_HOUR
The speed of 1 kilometer per hour.

CML_CONST_MKSA_MILES_PER_HOUR
The speed of 1 mile per hour.

CML_CONST_MKSA_NAUTICAL_MILE
The length of 1 nautical mile.

CML_CONST_MKSA_FATHOM
The length of 1 fathom.

CML_CONST_MKSA_KNOT
The speed of 1 knot.

8.7 Printers Units

CML_CONST_MKSA_POINT
The length of 1 printer's point (1/72 inch).

CML_CONST_MKSA_TEXPOINT
The length of 1 TeX point (1/72.27 inch).

8.8 Volume, Area and Length

CML_CONST_MKSA_MICRON
The length of 1 micron.

CML_CONST_MKSA_HECTARE
The area of 1 hectare.

CML_CONST_MKSA_ACRE
The area of 1 acre.

CML_CONST_MKSA_LITER
The volume of 1 liter.

CML_CONST_MKSA_US_GALLON
The volume of 1 US gallon.

CML_CONST_MKSA_CANADIAN_GALLON
The volume of 1 Canadian gallon.

CML_CONST_MKSA_UK_GALLON
The volume of 1 UK gallon.

CML_CONST_MKSA_QUART
The volume of 1 quart.

CML_CONST_MKSA_PINT
The volume of 1 pint.

8.9 Mass and Weight

CML_CONST_MKSA_POUND_MASS
The mass of 1 pound.

CML_CONST_MKSA_OUNCE_MASS
The mass of 1 ounce.

CML_CONST_MKSA_TON
The mass of 1 ton.

CML_CONST_MKSA_METRIC_TON
The mass of 1 metric ton (1000 kg).

CML_CONST_MKSA_UK_TON
The mass of 1 UK ton.

CML_CONST_MKSA_TROY_OUNCE
The mass of 1 troy ounce.

CML_CONST_MKSA_CARAT
The mass of 1 carat.

CML_CONST_MKSA_GRAM_FORCE
The force of 1 gram weight.

CML_CONST_MKSA_POUND_FORCE
The force of 1 pound weight.

CML_CONST_MKSA_KILOPOUND_FORCE
The force of 1 kilopound weight.

CML_CONST_MKSA_POUNDAL
The force of 1 poundal.

8.10 Thermal Energy and Power

CML_CONST_MKSA_CALORIE
The energy of 1 calorie.

CML_CONST_MKSA_BTU
The energy of 1 British Thermal Unit, btu.

CML_CONST_MKSA_THERM
The energy of 1 Therm.

CML_CONST_MKSA_HORSEPOWER
The power of 1 horsepower.

8.11 Pressure

CML_CONST_MKSA_BAR
The pressure of 1 bar.

CML_CONST_MKSA_STD_ATMOSPHERE
The pressure of 1 standard atmosphere.

CML_CONST_MKSA_TORR
The pressure of 1 torr.

CML_CONST_MKSA_METER_OF_MERCURY
The pressure of 1 meter of mercury.

CML_CONST_MKSA_INCH_OF_MERCURY
The pressure of 1 inch of mercury.

CML_CONST_MKSA_INCH_OF_WATER
The pressure of 1 inch of water.

CML_CONST_MKSA_PSI
The pressure of 1 pound per square inch.

8.12 Viscosity

CML_CONST_MKSA_POISE
The dynamic viscosity of 1 poise.

CML_CONST_MKSA_STOKES
The kinematic viscosity of 1 stokes.

8.13 Light and Illumination

CML_CONST_MKSA_STILB
The luminance of 1 stilb.

CML_CONST_MKSA_LUMEN
The luminous flux of 1 lumen.

CML_CONST_MKSA_LUX
The illuminance of 1 lux.

CML_CONST_MKSA_PHOT
The illuminance of 1 phot.

CML_CONST_MKSA_FOOTCANDLE
The illuminance of 1 footcandle.

CML_CONST_MKSA_LAMBERT
The luminance of 1 lambert.

CML_CONST_MKSA_FOOTLAMBERT
The luminance of 1 footlambert.

8.14 Radioactivity

CML_CONST_MKSA_CURIE
The activity of 1 curie.

CML_CONST_MKSA_ROENTGEN
The exposure of 1 roentgen.

CML_CONST_MKSA_RAD
The absorbed dose of 1 rad.

8.15 Force and Energy

CML_CONST_MKSA_NEWTON
The SI unit of force, 1 Newton.

CML_CONST_MKSA_DYNE
The force of 1 Dyne = 10^{-5} Newton.

CML_CONST_MKSA_JOULE
The SI unit of energy, 1 Joule.

CML_CONST_MKSA_ERG
The energy 1 erg = 10^{-7} Joule.

8.16 Prefixes

These constants are dimensionless scaling factors.

CML_CONST_NUM_YOTTA
10^{24}

CML_CONST_NUM_ZETTA
10^{21}

CML_CONST_NUM_EXA
10^{18}

CML_CONST_NUM_PETA
10^{15}

CML_CONST_NUM_TERA
10^{12}

CML_CONST_NUM_GIGA
10^{9}

CML_CONST_NUM_MEGA
10^{6}

CML_CONST_NUM_KILO
10^{3}

CML_CONST_NUM_MILLI
10^{-3}

CML_CONST_NUM_MICRO
10^{-6}

CML_CONST_NUM_NANO
10^{-9}

CML_CONST_NUM_PICO
10^{-12}

CML_CONST_NUM_FEMTO
10^{-15}

CML_CONST_NUM_ATTO
10^{-18}

CML_CONST_NUM_ZEPTO
10^{-21}

CML_CONST_NUM_YOCTO
10^{-24}

8.17 Examples

The following program demonstrates the use of the physical constants in a calculation. In this case, the goal is to calculate the range of light-travel times from Earth to Mars.

The required data is the average distance of each planet from the Sun in astronomical units (the eccentricities and inclinations of the orbits will be neglected for the purposes of this calculation). The average radius of the orbit of Mars is 1.52 astronomical units, and for the orbit of Earth it is 1 astronomical unit (by definition). These values are combined with the MKSA values of the constants for the speed of light and the length of an astronomical unit to produce a result for the shortest and longest light-travel times in seconds. The figures are converted into minutes before being displayed.

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

int
main(void)
{
        double c  = CML_CONST_MKSA_SPEED_OF_LIGHT;
        double au = CML_CONST_MKSA_ASTRONOMICAL_UNIT;
        double minutes = CML_CONST_MKSA_MINUTE;

        /* distance stored in meters */
        double r_earth = 1.00 * au;
        double r_mars  = 1.52 * au;

        double t_min, t_max;

        t_min = (r_mars - r_earth) / c;
        t_max = (r_mars + r_earth) / c;

        printf("light travel time from Earth to Mars:\n");
        printf("minimum = %.1f minutes\n", t_min / minutes);
        printf("maximum = %.1f minutes\n", t_max / minutes);

        return 0;
}
```

Here is the output from the program,

```
light travel time from Earth to Mars:
minimum = 4.3 minutes
maximum = 21.0 minutes
```

8.18 References and Further Reading

The authoritative sources for physical constants are the 2006 CODATA recommended values, published in the article below. Further information on the values of physical constants is also available from the NIST website.

- P.J. Mohr, B.N. Taylor, D.B. Newell, "CODATA Recommended Values of the Fundamental Physical Constants: 2006", Reviews of Modern Physics, 80(2), pp. 633–730 (2008).
- http://www.physics.nist.gov/cuu/Constants/index.html
- http://physics.nist.gov/Pubs/SP811/appenB9.html

CHAPTER 9

IEEE floating-point arithmetic

This chapter describes functions for examining the representation of floating point numbers and controlling the floating point environment of your program. The functions described in this chapter are declared in the header file `cml/ieee.h`.

9.1 Representation of floating point numbers

The IEEE Standard for Binary Floating-Point Arithmetic defines binary formats for single and double precision numbers. Each number is composed of three parts: a *sign bit* (s), an *exponent* (E) and a *fraction* (f). The numerical value of the combination (s, E, f) is given by the following formula,

$$(-1)^s(1 \cdot fffff\ldots)2^E$$

The sign bit is either zero or one. The exponent ranges from a minimum value E_{min} to a maximum value E_{max} depending on the precision. The exponent is converted to an unsigned number e, known as the *biased exponent*, for storage by adding a *bias* parameter,

$$e = E + bias$$

The sequence $fffff$... represents the digits of the binary fraction f. The binary digits are stored in *normalized form*, by adjusting the exponent to give a leading digit of 1. Since the leading digit is always 1 for normalized numbers it is assumed implicitly and does not have to be stored. Numbers smaller than $2^{E_{min}}$ are be stored in *denormalized form* with a leading zero,

$$(-1)^s(0 \cdot fffff\ldots)2^{E_{min}}$$

This allows gradual underflow down to $2^{E_{min}-p}$ for p bits of precision. A zero is encoded with the special exponent of $2^{E_{min}-1}$ and infinities with the exponent of $2^{E_{max}+1}$.

The format for single precision numbers uses 32 bits divided in the following way:

```
seeeeeeeeffffffffffffffffffffff

s = sign bit, 1 bit
e = exponent, 8 bits  (E_min=-126, E_max=127, bias=127)
f = fraction, 23 bits
```

The format for double precision numbers uses 64 bits divided in the following way:

```
seeeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff

s = sign bit, 1 bit
e = exponent, 11 bits  (E_min=-1022, E_max=1023, bias=1023)
f = fraction, 52 bits
```

It is often useful to be able to investigate the behavior of a calculation at the bit-level and the library provides functions for printing the IEEE representations in a human-readable form.

void **cml_ieee754_fprintf_float** (FILE * *stream*, const float * *x*)
void **cml_ieee754_fprintf_double** (FILE * *stream*, const double * *x*)
These functions output a formatted version of the IEEE floating-point number pointed to by `x` to the stream `stream`. A pointer is used to pass the number indirectly, to avoid any undesired promotion from `float` to `double`. The output takes one of the following forms,

`NaN`

the Not-a-Number symbol

`Inf, -Inf`

positive or negative infinity

`1.fffff...*2^E, -1.fffff...*2^E`

a normalized floating point number

`0.fffff...*2^E, -0.fffff...*2^E`

a denormalized floating point number

`0, -0`

positive or negative zero

The output can be used directly in GNU Emacs Calc mode by preceding it with `2#` to indicate binary.

void **cml_ieee754_printf_float** (const float * *x*)
void **cml_ieee754_printf_double** (const double * *x*)
These functions output a formatted version of the IEEE floating-point number pointed to by `x` to the stream `stdout`.

The following program demonstrates the use of the functions by printing the single and double precision representations of the fraction $1/3$. For comparison the representation of the value promoted from single to double precision is also printed.

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

int
main(void)
{
        float f = 1.0/3.0;
        double d = 1.0/3.0;
```

```
        double fd = f; /* promote from float to double */

        printf(" f = ");
        cml_ieee754_printf_float(&f);
        printf("\n");

        printf("fd = ");
        cml_ieee754_printf_double(&fd);
        printf("\n");

        printf(" d = ");
        cml_ieee754_printf_double(&d);
        printf("\n");

        return 0;
}
```

The binary representation of $1/3$ is $0.01010101....$ The output below shows that the IEEE format normalizes this fraction to give a leading digit of 1:

```
 f = 1.01010101010101010101011*2^-2
fd = 1.0101010101010101010101100000000000000000000000000000*2^-2
 d = 1.0101010101010101010101010101010101010101010101010101*2^-2
```

The output also shows that a single-precision number is promoted to double-precision by adding zeros in the binary representation.

9.2 References and Further Reading

The reference for the IEEE standard is,

- ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

A more pedagogical introduction to the standard can be found in the following paper,

- David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, Vol.: 23, No.: 1 (March 1991), pages 5–48.
- Corrigendum: *ACM Computing Surveys*, Vol.: 23, No.: 3 (September 1991), page 413. and see also the sections by B. A. Wichmann and Charles B. Dunham in Surveyor's Forum: "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys*, Vol.: 24, No.: 3 (September 1992), page 319.

A detailed textbook on IEEE arithmetic and its practical use is available from SIAM Press,

- Michael L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM Press, ISBN 0898715717.

CHAPTER 10

Indices and tables

- genindex

Index

D

E

F

H

I

L

M

N

O

P

R

S

T

U

V

W

Z