CML

Release 1.10.4

Ulises Jeremias Cornejo Fandos

Jan 13, 2019

Contents

CHAPTER 1

Introduction

The C Math Library (CML) is a pure mathematical C library with a wide variety of mathematical functions that seeks to be close to complying with ANSI C for portability. It's a collection of routines for numerical computing written from scratch in C. The routines present a modern API for C programmers, allowing wrappers to be written for very high level languages. It is free software under the MIT License.

1.1 Routines available in CML

Routines are available for the following areas,

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

Each chapter of this manual provides detailed definitions of the functions, followed by examples and references to the articles and other resources on which the algorithms are based.

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

Mathematical Functions

For the development of this module, the functions present in many of the system libraries are taken as reference with the idea of offering them in CML as an option for when they are not present.

This chapter describes basic mathematical functions.

The functions and macros described in this chapter are defined in the header file `cml/math.h`.

3.1 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

3.2 Infinities and Not-a-number

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

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

CML_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.

3.3 Elementary Functions

The following routines provide portable implementations of functions found in the BSD math library, e.g. 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_x*)
This function computes the value of $\sqrt{x^2+y^2+x^2}$ 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)`.

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

double **cml_pow** (double *x*, double *a*)
The function returns the number `x` raised to the double-precision power `a`, x^a. This is computed as $\exp(\log(x) * a)$ using logarithms and exponentials.

double **cml_exp** (double *x*)
This function returns the exponential of the number `x`, $\exp(x)$.

double **cml_log** (double *x*)
This function returns the natural logarithm (base e) of the number `x`, $\log(x)$. The branch cut is the negative real axis.

double **cml_log10** (double *x*)
This function returns the base-10 logarithm of the number `x`, $\log_{10}(x)$.

double **cml_log_b** (double *x*, double *b*)
This function returns the base-`b` logarithm of the double-precision number `x`, $\log_b(x)$. This quantity is computed as the ratio $\log(x)/\log(b)$.

3.4 Trigonometric Functions

double **cml_sin** (double *x*)
This function returns the sine of the number `x`, $\sin(x)$.

double **cml_cos** (double *x*)
This function returns the cosine of the number `x`, $\cos(x)$.

double **doublean** (double *x*)
This function returns the tangent of the number `x`, $\tan(x)$.

double **cml_sec** (double *x*)
This function returns the secant of the number `x`, $\sec(x) = 1/\cos(x)$.

double **cml_csc** (double *x*)
This function returns the cosecant of the number `x`, $\csc(x) = 1/\sin(x)$.

double **cml_cot** (double *x*)
This function returns the cotangent of the number `x`, $\cot(x) = 1/\tan(x)$.

3.5 Inverse Trigonometric Functions

double **cml_asin** (double *x*)
This function returns the arcsine of the number `x`, $\arcsin(x)$.

double **cml_acos** (double *x*)
This function returns the arccosine of the number x, $\arccos(x)$.

double **cml_atan** (double *x*)
This function returns the arctangent of the number x, $\arctan(x)$.

double **cml_asec** (double *x*)
This function returns the arcsecant of the number x, $\text{arcsec}(x) = \arccos(1/x)$.

double **cml_acsc** (double *x*)
This function returns the arccosecant of the number x, $\text{arccsc}(x) = \arcsin(1/x)$.

double **cml_acot** (double *x*)
This function returns the arccotangent of the number x, $\text{arccot}(x) = \arctan(1/x)$.

3.6 Hyperbolic Functions

double **cml_sinh** (double *x*)
This function returns the hyperbolic sine of the number x, $\sinh(x) = (\exp(x) - \exp(-x))/2$.

double **cml_cosh** (double *x*)
This function returns the hyperbolic cosine of the number x, $\cosh(x) = (\exp(x) + \exp(-x))/2$.

double **doubleanh** (double *x*)
This function returns the hyperbolic tangent of the number x, $\tanh(x) = \sinh(x)/\cosh(x)$.

double **cml_sech** (double *x*)
This function returns the hyperbolic secant of the double-precision number x, $\text{sech}(x) = 1/\cosh(x)$.

double **cml_csch** (double *x*)
This function returns the hyperbolic cosecant of the double-precision number x, $\text{csch}(x) = 1/\sinh(x)$.

double **cml_coth** (double *x*)
This function returns the hyperbolic cotangent of the double-precision number x, $\coth(x) = 1/\tanh(x)$.

3.7 Inverse Hyperbolic Functions

double **cml_asinh** (double *x*)
This function returns the hyperbolic arcsine of the number x, $\text{arcsinh}(x)$.

double **cml_acosh** (double *x*)
This function returns the hyperbolic arccosine of the double-precision number x, $\text{arccosh}(x)$.

double **cml_atanh** (double *x*)
This function returns the hyperbolic arctangent of the double-precision number x, $\text{arctanh}(x)$.

double **cml_asech** (double *x*)
This function returns the hyperbolic arcsecant of the double-precision number x, $\text{arcsech}(x) = \text{arccosh}(1/x)$.

double **cml_acsch** (double *x*)
This function returns the hyperbolic arccosecant of the double-precision number x, $\text{arccsch}(x) = \text{arcsinh}(1/x)$.

double **cml_acoth** (double *x*)
This function returns the hyperbolic arccotangent of the double-precision number x, $\text{arccoth}(x) = \text{arctanh}(1/x)$.

3.8 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/math.h>
[...]
double y = pow_4(3.141);  /* compute 3.141**4 */
```

3.9 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).

3.10 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))`.

3.11 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 4

Complex Numbers

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

4.1 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
{
        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`.

4.2 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.

4.3 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, $-z = (-x) + (-y)i$.

4.4 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 $\log(z)/\log(b)$.

4.5 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, $\cot(z) = 1/\tan(z)$.

4.6 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, $\mathrm{arcsec}(z) = \arccos(1/z)$.

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

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

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

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

4.7 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, $\coth(z) = 1/\tanh(z)$.

4.8 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, $\mathrm{arccoth}(z) = \mathrm{arctanh}(1/z)$.

CHAPTER 5

Quaternions

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

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

5.1 Representation of quaternions

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

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

```
typedef struct
{
        union
        {
                double q[4];
                struct
                {
                        double w, x, y, z;
                };
                struct
                {
                        double a, i, j, k;
                };
        };
} cml_quaternion_t;
```

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.

Again, the development of this module is inspired by the same present in GSL looking to adapt it completely to the practices and tools present in CML.

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

double
f(double x, void *params)
{
        (void) params; /* avoid unused parameter warning */
        return cml_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
```

(continues on next page)

(continued from previous page)

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

6.3 References and Further Reading

This work is a spiritual descendent of the Differentiation module in GSL.

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.

7.1 References and Further Reading

This work is a spiritual descendent (not to say derivative work) of works done by Robert Penner. So, the main references could be found in http://robertpenner.com/easing/

- http://robertpenner.com/easing/penner_chapter7_tweening.pdf

- http://gilmoreorless.github.io/sydjs-preso-easing/
- http://upshots.org/actionscript/jsas-understanding-easing
- http://sol.gfxile.net/interpolation/

CHAPTER 8

Physical Constants

This module is inspired by the constants module present in GSL.

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_CML_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).

CHAPTER 9

IEEE floating-point arithmetic

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,

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

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,

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

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:

```
seeeeeeeefffffffffffffffffffffff

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:

```
seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff

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");
```

(continues on next page)

(continued from previous page)

```
        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.

CHAPTER 10

Statistics

This chapter describes the statistical functions in the library. The basic statistical functions include routines to compute the mean, variance and standard deviation. More advanced functions allow you to calculate absolute deviations, skewness, and kurtosis as well as the median and arbitrary percentiles. The algorithms use recurrence relations to compute average quantities in a stable way, without large intermediate values that might overflow.

10.1 Data Types

The functions are available in versions for datasets in the standard floating-point and integer types. The versions for double precision floating-point data have the prefix `cml_stats` and are declared in the header file `cml/statistics/double.h`. The versions for integer data have the prefix `cml_stats_int` and are declared in the header file `cml/statistics/int.h`. All the functions operate on C arrays with a `stride` parameter specifying the spacing between elements. The full list of available types is given below,

Prefix	Type
cml_stats	double
cml_stats_float	float
cml_stats_long_double	long double
cml_stats_int	int
cml_stats_uint	unsigned int
cml_stats_long	long
cml_stats_ulong	unsigned long
cml_stats_short	short
cml_stats_ushort	unsigned short
cml_stats_char	char
cml_stats_uchar	unsigned char
cml_stats_complex	complex double
cml_stats_complex_float	complex float
cml_stats_complex_long_double	complex long double

10.2 Mean, Standard Deviation and Variance

double **cml_stats_mean** (const double *data[]*, size_t *stride*, size_t *n*)

This function returns the arithmetic mean of `data`, a dataset of length `n` with stride `stride`. The arithmetic mean, or *sample mean*, is denoted by $\hat{\mu}$ and defined as,

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

where x_i are the elements of the dataset `data`. For samples drawn from a gaussian distribution the variance of $\hat{\mu}$ is σ^2/N.

double **cml_stats_variance** (const double *data[]*, size_t *stride*, size_t *n*)

This function returns the estimated, or *sample*, variance of `data`, a dataset of length `n` with stride `stride`. The estimated variance is denoted by $\hat{\sigma}^2$ and is defined by,

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

where x_i are the elements of the dataset `data`. Note that the normalization factor of $1/(N-1)$ results from the derivation of $\hat{\sigma}^2$ as an unbiased estimator of the population variance σ^2. For samples drawn from a Gaussian distribution the variance of $\hat{\sigma}^2$ itself is $2\sigma^4/N$.

This function computes the mean via a call to *`cml_stats_mean()`*. If you have already computed the mean then you can pass it directly to *`cml_stats_variance_m()`*.

double **cml_stats_variance_m** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*)

This function returns the sample variance of `data` relative to the given value of `mean`. The function is computed with $\hat{\mu}$ replaced by the value of `mean` that you supply,

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

double **cml_stats_sd** (const double *data[]*, size_t *stride*, size_t *n*)
double **cml_stats_sd_m** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*)

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

double **cml_stats_tss** (const double *data[]*, size_t *stride*, size_t *n*)
double **cml_stats_tss_m** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*)

These functions return the total sum of squares (TSS) of `data` about the mean. For *`cml_stats_tss_m()`* the user-supplied value of `mean` is used, and for *`cml_stats_tss()`* it is computed using *`cml_stats_mean()`*.

$$\text{TSS} = \sum (x_i - mean)^2$$

double **cml_stats_variance_with_fixed_mean** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*)

This function computes an unbiased estimate of the variance of `data` when the population mean `mean` of the underlying distribution is known *a priori*. In this case the estimator for the variance uses the factor $1/N$ and the sample mean $\hat{\mu}$ is replaced by the known population mean μ,

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

double **cml_stats_sd_with_fixed_mean** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*)

This function calculates the standard deviation of `data` for a fixed population mean `mean`. The result is the square root of the corresponding variance function.

10.3 Absolute deviation

double **cml_stats_absdev** (const double *data[]*, size_t *stride*, size_t *n*)

This function computes the absolute deviation from the mean of `data`, a dataset of length `n` with stride `stride`. The absolute deviation from the mean is defined as,

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

where x_i are the elements of the dataset `data`. The absolute deviation from the mean provides a more robust measure of the width of a distribution than the variance. This function computes the mean of `data` via a call to `cml_stats_mean()`.

double **cml_stats_absdev_m** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*)

This function computes the absolute deviation of the dataset `data` relative to the given value of `mean`,

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

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

10.4 Higher moments (skewness and kurtosis)

double **cml_stats_skew** (const double *data[]*, size_t *stride*, size_t *n*)

This function computes the skewness of `data`, a dataset of length `n` with stride `stride`. The skewness is defined as,

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

where x_i are the elements of the dataset `data`. The skewness measures the asymmetry of the tails of a distribution.

The function computes the mean and estimated standard deviation of `data` via calls to `cml_stats_mean()` and `cml_stats_sd()`.

double **cml_stats_skew_m_sd** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*, double *sd*)

This function computes the skewness of the dataset `data` using the given values of the mean `mean` and standard deviation `sd`,

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

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

double **cml_stats_kurtosis** (const double *data[]*, size_t *stride*, size_t *n*)

This function computes the kurtosis of `data`, a dataset of length `n` with stride `stride`. The kurtosis is defined as,

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

The kurtosis measures how sharply peaked a distribution is, relative to its width. The kurtosis is normalized to zero for a Gaussian distribution.

double **cml_stats_kurtosis_m_sd** (const double *data[]*, size_t *stride*, size_t *n*, double *mean*, double *sd*)

This function computes the kurtosis of the dataset `data` using the given values of the mean `mean` and standard deviation `sd`,

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

This function is useful if you have already computed the mean and standard deviation of `data` and want to avoid recomputing them.

10.5 Autocorrelation

double **cml_stats_lag1_autocorrelation** (const double *data[]*, const size_t *stride*, const size_t *n*)

This function computes the lag-1 autocorrelation of the dataset `data`.

$$a_1 = \frac{\sum_{i=2}^{n}(x_i - \hat{\mu})(x_{i-1} - \hat{\mu})}{\sum_{i=1}^{n}(x_i - \hat{\mu})(x_i - \hat{\mu})}$$

double **cml_stats_lag1_autocorrelation_m** (const double *data[]*, const size_t *stride*, const size_t *n*, const double *mean*)

This function computes the lag-1 autocorrelation of the dataset `data` using the given value of the mean `mean`.

10.6 Covariance

double **cml_stats_covariance** (const double *data1[]*, const size_t *stride1*, const double *data2[]*, const size_t *stride2*, const size_t *n*)

This function computes the covariance of the datasets `data1` and `data2` which must both be of the same length `n`.

$$covar = \frac{1}{(n-1)}\sum_{i=1}^{n}(x_i - \hat{x})(y_i - \hat{y})$$

double **cml_stats_covariance_m** (const double *data1[]*, const size_t *stride1*, const double *data2[]*, const size_t *stride2*, const size_t *n*, const double *mean1*, const double *mean2*)

This function computes the covariance of the datasets `data1` and `data2` using the given values of the means, `mean1` and `mean2`. This is useful if you have already computed the means of `data1` and `data2` and want to avoid recomputing them.

10.7 Correlation

double **cml_stats_correlation** (const double *data1[]*, const size_t *stride1*, const double *data2[]*, const size_t *stride2*, const size_t *n*)

This function efficiently computes the Pearson correlation coefficient between the datasets `data1` and `data2` which must both be of the same length `n`.

$$r = \frac{cov(x,y)}{\hat{\sigma}_x\hat{\sigma}_y} = \frac{\frac{1}{n-1}\sum(x_i - \hat{x})(y_i - \hat{y})}{\sqrt{\frac{1}{n-1}\sum(x_i - \hat{x})^2}\sqrt{\frac{1}{n-1}\sum(y_i - \hat{y})^2}}$$

double **cml_stats_spearman** (const double *data1[]*, const size_t *stride1*, const double *data2[]*, const size_t *stride2*, const size_t *n*, double *work[]*)

This function computes the Spearman rank correlation coefficient between the datasets `data1` and `data2` which must both be of the same length `n`. Additional workspace of size 2 * `n` is required in `work`. The Spearman rank correlation between vectors x and y is equivalent to the Pearson correlation between the ranked vectors x_R and y_R, where ranks are defined to be the average of the positions of an element in the ascending order of the values.

10.8 Maximum and Minimum values

The following functions find the maximum and minimum values of a dataset (or their indices). If the data contains `NaN`-s then a `NaN` will be returned, since the maximum or minimum value is undefined. For functions which return an index, the location of the first `NaN` in the array is returned.

double **cml_stats_max** (const double *data[]*, size_t *stride*, size_t *n*)

This function returns the maximum value in `data`, a dataset of length `n` with stride `stride`. The maximum value is defined as the value of the element x_i which satisfies $x_i \geq x_j$ for all j.

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

double **cml_stats_min** (const double *data[]*, size_t *stride*, size_t *n*)

This function returns the minimum value in `data`, a dataset of length `n` with stride `stride`. The minimum value is defined as the value of the element x_i which satisfies $x_i \leq x_j$ for all j.

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

void **cml_stats_minmax** (double * *min*, double * *max*, const double *data[]*, size_t *stride*, size_t *n*)

This function finds both the minimum and maximum values `min`, `max` in `data` in a single pass.

size_t **cml_stats_max_index** (const double *data[]*, size_t *stride*, size_t *n*)

This function returns the index of the maximum value in `data`, a dataset of length `n` with stride `stride`. The maximum value is defined as the value of the element x_i which satisfies $x_i \geq x_j$ for all j. When there are several equal maximum elements then the first one is chosen.

size_t **cml_stats_min_index** (const double *data[]*, size_t *stride*, size_t *n*)

This function returns the index of the minimum value in `data`, a dataset of length `n` with stride `stride`. The minimum value is defined as the value of the element x_i which satisfies $x_i \geq x_j$ for all j. When there are several equal minimum elements then the first one is chosen.

void **cml_stats_minmax_index** (size_t * *min_index*, size_t * *max_index*, const double *data[]*, size_t *stride*, size_t *n*)

This function returns the indexes `min_index`, `max_index` of the minimum and maximum values in `data` in a single pass.

10.9 Median and Percentiles

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

double **cml_stats_median_from_sorted_data** (const double *sorted_data[]*, size_t *stride*, size_t *n*)

This function returns the median value of `sorted_data`, a dataset of length `n` with stride `stride`. The elements of the array must be in ascending numerical order. There are no checks to see whether the data are sorted, so the function `cml_sort()` should always be used first.

When the dataset has an odd number of elements the median is the value of element $(n-1)/2$. When the dataset has an even number of elements the median is the mean of the two nearest middle values, elements $(n-1)/2$ and $n/2$. Since the algorithm for computing the median involves interpolation this function always returns a floating-point number, even for integer data types.

double **cml_stats_quantile_from_sorted_data** (const double *sorted_data[]*, size_t *stride*, size_t *n*, double *f*)

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

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

The quantile is found by interpolation, using the formula

$$\text{quantile} = (1-\delta)x_i + \delta x_{i+1}$$

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

Thus the minimum value of the array (`data[0*stride]`) is given by `f` equal to zero, the maximum value (`data[(n-1)*stride]`) is given by `f` equal to one and the median value is given by `f` equal to 0.5. Since the algorithm for computing quantiles involves interpolation this function always returns a floating-point number, even for integer data types.

10.10 References and Further Reading

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

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

Many statistical concepts can be more easily understood by a Bayesian approach. The following book by Gelman, Carlin, Stern and Rubin gives a comprehensive coverage of the subject.

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

CHAPTER 11

Indices and tables

- genindex

Index

A

B

C

D

E

F

H

I

K

L