e_float User's Manual

Christopher Kormanyos

Version 1.0 from November 3, 2010

e_float@yahoo.com

Contents

1 About e_float

1.1 Introduction

There are many multiple precision (MP) packages available to the scientific and engineering community. Each package has individual strengths within its range of application. However, most MP packages lack a uniform interface for high precision algorithm design. They also offer little or no special function support. Also, many MP packages have limited portability. There is no portable standalone C++ system which offers a wide variety of high precision special functions and handles large function parameters.

The e_float system (extended float) not only addresses these weaknesses but also significantly advances MP technology. It uses several MP packages and provides a uniform C++ interface for high precision algorithm design, independent of the underlying MP implementation. In addition, e_float supports a large collection of high performance MP functions which are entirely portable, solidly designed and can be used with any suitably prepared MP type. Furthermore, the e_float system provides software interfaces for seamless interoperability with other high level languages. No MP system other than e_float offers such a high degree of portability, such a wide range of functions, and such a rich selection of powerful interoperabilities. The e_float library can be used for a variety of purposes such as high precision function evaluation, numerical verification of compilers, function libraries or other computational systems as well as investigations of compiler quality, optimization methods and computer hardware.

e_float is unique because it is designed from the ground up utilizing generic and object–oriented design methods to create an efficient and flexible numerical software architecture [5], [31], [36]. The standard containers and algorithms of the C++ STL and TR1 are consistently used to reduce programmatic complexity [3], [14], [15], [20]. Static and dynamic polymorphism are used to implement a standardized interface to MP types allowing for the interchangeable use of different MP implementations.

e_float is written in the C++ language making broad use of the C++ core language, much of the STL and some of TR1, as specified in ISO/IEC 14882:2003 and ISO/IEC 19768:2007 [14, 15]. It is emphasized that the C++ compiler must closely adhere to these standards in order to successfully compile and link e_float. The source codes have been implemented according to safe programming practices and reliability standards originating from the automotive industry [26, 27]. A great effort has been invested in advanced C++ optimization techniques in order to improve system performance. Generic and object–oriented programming methods have been used to create an efficient and flexible numerical software architecture. In addition, consistent use of standard containers and algorithms of the STL and TR1 [20, 3] has significantly reduced and evenly distributed the computational complexities of the entire program.

Advanced programming techniques have been used to implement interfaces to other high level languages, including the Microsoft® CLR, Python, IronPython and Wolfram's Mathematica® (see [17, 25, 29, 6, 23, 34]). This means that it is possible to combine the calculating power of several systems to obtain a hybrid system which is more powerful than either one of its parts. For example, e_float can be used with the C# language [16], targeting the CLR in the Microsoft®.NET Framework. This exposes the efficient calculating power of e_float to the high level graphical–user–interface (GUI) design regime of C# in the CLR. At the same time the GUI design is separated from complex MP algorithms. This is a very powerful hybrid system based on effective distribution of computational complexity.

The e_float software project consists of approximately 110 manually written source files with

$\sim$20,000 lines of code in addition to about 20 automatically generated files with $\sim$200,000 lines of code. The source codes of e_float are implemented in accordance with safe programming practices and reliability standards originating from the automotive industry [26], [27].

The e_float source code is released under the 'Boost Software License' (BSL) from boost [4]. Unlike the the 'General Public License' (GPL) from GNU [11], the BSL permits the creation of works derived from e_float for any commercial, or non-commercial, or non-profit use with no legal requirement to release the e_float source code [4].

1.2 Configurations

Config	`e_float` Class	Capabilities	Dependencies
efx	`efx::e_float`	functions, algorithm design, test and benchmark	—
gmp	`gmp::e_float`	” ”	GMP library
mpfr	`mpfr::e_float`	” ”	GMP library, MPFR library
f90	`mpfr::e_float`	” ”	`REAL(KIND=16)` wrap, Fortran run–time libraries
pyd	any `e_float`	functions, algorithm design, rapid prototyping, high level Python scripting	those of its `e_float` type, `boost.python` library, Python library
clr	any `e_float`	functions, algorithm design, rapid prototyping, high level scripting, CLR GUI development	those of its `e_float` type, common language runtime
cas	any `e_float`	computer algebra interoperability	those of its `e_float` type, computer algebra system

Table 1: The system configurations of e_float are shown.

The e_float system supports a variety of configurations using several `e_float` classes as well as other libraries and systems. These are shown in Table 1. Details about the capabilities and dependencies of the configurations are also included in the table.

1.3 The e_float System Architecture

The e_float system architecture is robust and flexible. With this architecture, both the integration of other MP types as well as the addition of more functions and interoperabilities can be done with ease. The system architecture is shown in Figure 1. It has four layers and two additional blocks, the test block and the tools block. Layers 1–4 have successively increasing levels of abstraction. They build up very high level functionalities in a stable, stepwise fashion. Note that "*e_float*" is not only the name of the system but also the name of several `e_float` classes.

Layer 1, the low level MP layer, ensures that each implementation–dependent, possibly non–portable MP implementation is suitably prepared to conform with the C++ class requirements of Layer 2. Each individual MP type is encapsulated within a specific `e_float` C++ class, each one of which defined within its own unique namespace. For example, classes such as `efx::e_float`, `gmp::e_float` and others are implemented. Each of these classes is derived from a common abstract base class called `::e_float_base`. The abstract base class defines public and pure virtual functions which implement arithmetic primitives such as self–multiplication, self–compare, special numbers like NaN, and string operations. These arithmetic primitives fulfill the requirements necessary for elementary mathematics. They simultaneously conform with Layer 2. Thus, via inheritance and implementation of virtual functions, each individual `e_float` class supports elementary mathematics and also complies with Layer 2.

Some MP types include their own versions of various functions. Layer 1 accommodates this with the "*has–its–own*"–mechanism. This mechanism allows the C++ interface of a given MP type to use the MP's own algorithm for a particular function. It uses virtual Boolean functions prefixed with "`has_its_own`". For example, `has_its_own_sin` returns `true` in order to use the MP's own implementation of $\sin(x)$, $x \in \mathbb{R}$. The performance of a specific MP class can be optimized by selectively activating these functions. MPFR [7] has its own implementations of most elementary functions and several higher transcendental functions. The elementary functions are quite efficient and these are, in fact, used to optimize the performance of `mpfr::e_float`.

Layer 2 implements the uniform C++ interface. The `e_float` type from layer 1, which will be used in the project configuration, is selected with a compile–time option. The functions of this `e_float` class are used to implement all arithmetic operations, numeric limits and basic I/O mechanisms. Thus, layer 2 provides a uniform C++ interface which is generic and fully equipped with all the basic functions necessary for high level MP algorithm design in a C++ environment.

Layer 3 is the C++ mathematical layer. It adds the class `ef_complex`, *i.e.* the complex data type. This layer uses both the selected `e_float` type as well as `ef_complex` to implement e_float's rich collection of elementary functions and higher transcendental functions.

Layer 4, the interoperability user layer, exposes all of the functions and capabilities of layers 2 and 3 to other high level languages. Marshaling techniques [30] are used to create managed C++ classes and wrapper functions which embody the full functionality of layers 2 and 3. These are compiled to a single CLR assembly which can be used with all Microsoft® CLR languages including C#, managed C++/CLI, IronPython, *etc*. Compatibility with the Microsoft®.NET Framework 3.5 has been tested. Another interoperability employs the `boost.python` library [1] to expose the functionality of layers 2 and 3 to Python. Compatibilities with Python 2.6.4 and boost $\geq$ 1.39 have been tested.

Another layer 4 interoperability targets Mathematica®. A sparse architecture has been developed to create a generic interface for interacting with computer algebra systems. The compatibility of this interface with Mathematica® 7.1 has been tested. The interoperabilities of layer 4 are very powerful mechanisms based on highly advanced programming techniques. They can be used for very high level designs such as scripting, rapid algorithm prototyping and result visualization.

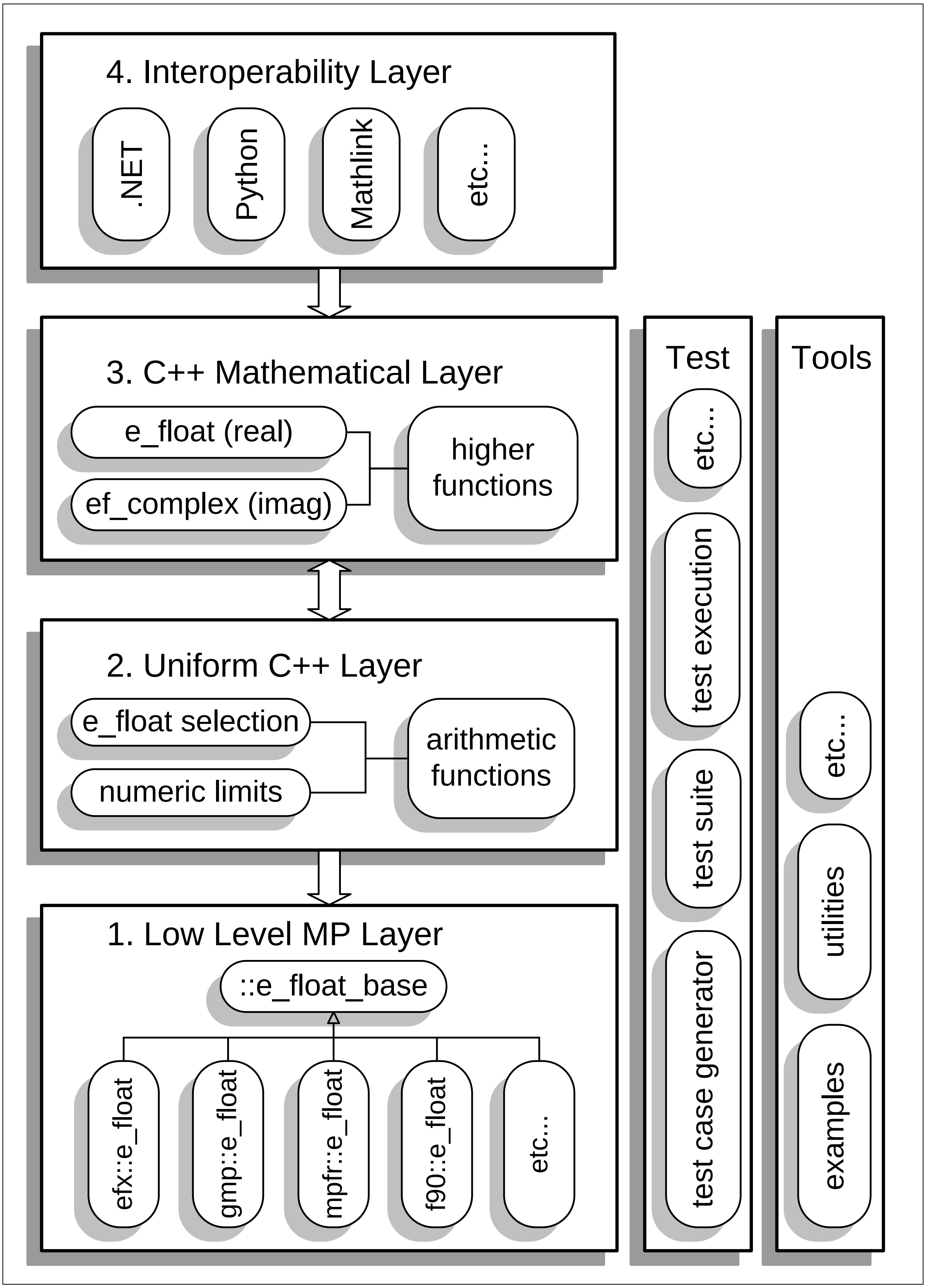


Figure 1: The e_float system architecture is shown.

The test block contains several hundred automatically generated test files which have been specifically designed to test all algorithms and convergence regions of the entire e_float system. The test block includes an automatic test execution system and an automatic test case generator. This block allows for fully reproducible automated testing of the system.

The tool block contains a variety of utilities and examples. The utilities predominantly consist of generic templates for standard mathematical operations such as numerical differentiation, root finding, recursive quadrature, *etc*. The examples show practical, non–trivial uses of the e_float system involving both high level algorithm design as well as interoperability.

The e_float architecture exemplifies how layered design can be leveraged to find the right granularity to distribute a very large computational complexity among smaller constituents which have manageable software scope. For example, there are vast software distances between the hand–optimized assembler routines of GNU MP [10] and, for example, the Hurwitz zeta function, or a high level GUI in C#. The e_float architecture elegantly navigates these distances to build up high level functionalities in a controlled, stepwise fashion.

The e_float system architecture is a significant technological milestone in MP programming technology. While other MP packages do sometimes provide a specialized C++ interface for their own specific implementations, they are mostly incompatible with each other. However, e_float's uniform C++ layer creates a generic interface which can be used with any underlying MP type. Assuming that a given MP type can be brought into conformance with layer 2, it can be used in a portable fashion with all of e_float's capabilities including arithmetic, elementary functions, special functions, interoperabilities, automatic tests, utilities, and additional user–created extensions.

1.4 MP Types

Class Name	Internal Representation	Digits	Padding
efx::e_float	data: UINT32 base–10^8 array Boolean negative sign INT64 base–10 exponent floating point class (finite, NaN, *etc.*)	30–300	15%
gmp::e_float	data: GMP's type ::mpf_t floating point class (finite, NaN, *etc.*)	30–300	15%
mpfr::e_float	data: MPFR's type ::mpfr_t	30–300	15%
f90::e_float	data: Fortran's type REAL(KIND=16)	~ 30	4–5 digits

Table 2: The MP classes in e_float are shown. Details about the internal representation as well as the digit range and the added internal extra digits (*i.e.* the padding) are given.

Four MP types have been selected for inclusion in e_float. These are listed in Table 2. The table also includes information about the internal representations of the individual MP classes and their digit ranges. The three main e_float classes are efx::e_float, mpfr::e_float and gmp::e_float. They are designed for high precision calculations with large exponent range.

GNU MP and MPFR have been included because of their high performance, their general acceptance in the scientific community and their widespread availability (at least for Unix/Linux–GNU systems). Unfortunately, these libraries are not easily ported to compiler and build systems other than GCC and GNUmake. However, for the e_float development GNU MP 4.2.4 and MPFR 2.4.1 have been ported to Microsoft® Visual Studio® 2008, based in part on Gladman's [9] port.

A new MP type called EFX, *extended–float–x*, has been created for the e_float system. It is written entirely in C++ and uses base–10^8 data elements. Since GNU MP and MPFR use more efficient base–2^n data elements and because they take advantage of hand–optimized assembler for the most time–critical inner loops, EFX does not quite reach the performance of either GNU MP or MPFR. However, EFX has other advantages — a base–10 representation as well as a data field which is created on the stack, not using dynamic memory allocation. Therefore, the base–10 numerical value can be viewed in a human recognizable form with a graphical debugger, without awkward print operations or code modifications. This makes EFX well suited for algorithm development. EFX has been used for all early algorithm prototyping during the e_float development.

The final MP type is called F90. It uses a skinny Fortran 90/C++ layer to wrap the Fortran quadruple–precision data type REAL(KIND=16). The range of this MP type is limited to that of the Fortran type — about 30 digits of precision with an exponent of about $\pm\,4000$. Due to its limited range, f90::e_float only passes about 75% of the test cases defined in the test suite. If this range sufficient for the application, then f90::e_float is extremely fast because it uses a native data type.

2 Building e_float

Every effort has been made to ensure that building the e_float system is straightforward. The default build supports execution of the test suite with the selected `e_float` type. Simple operations or other spot–tests can be carried out by deactivating the subroutine `::test_real_imag` and activating the subroutine `test::spot::test_spot` in the `main` program which is implemented in the file `test/test.cpp`. Designers of advanced applications can remove the test suite entirely and create a custom build.

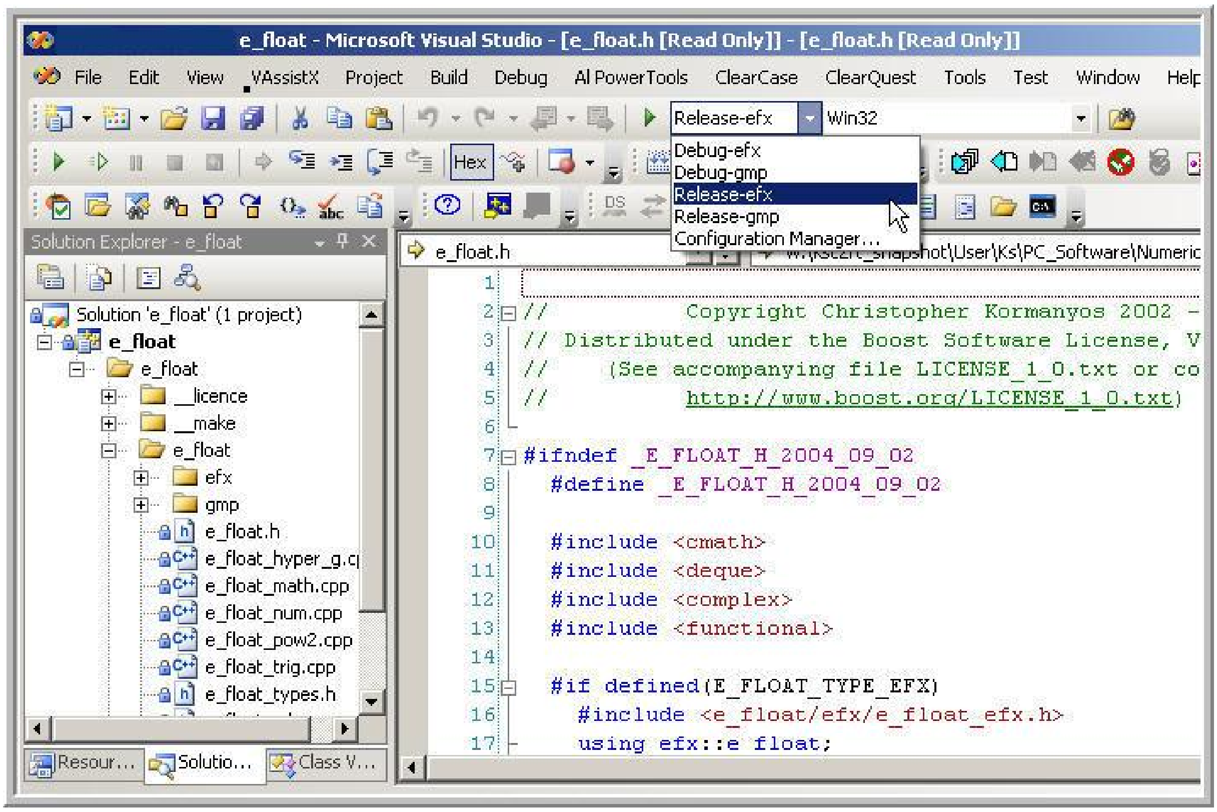
2.1 Compiler Systems

Compiler	Compatible	Final Test	Build System
Microsoft® Visual C++® x86 Microsoft® Visual C++® x64	≥ 9 with SP1	9 with SP1	Microsoft® NMAKE
GNU GCC i686–pc–cygwin GNU GCC x86_64–linux–gnu	≥ 4.2.2	4.4.2	GNUmake 3.81
Intel® ICC x86 Intel® ICC x64	≥ 11.1.046	11.1.051	Microsoft® NMAKE

Table 3: The compilers and build systems supported by e_float are shown.

Several compiler systems have been used at their highest warning settings in order to achieve very high levels of language standards adherence, portability and reliability. The compilers and build systems in Table 3 have been used to develop, build and test e_float. Tools from Microsoft®, Intel® and GNU are supported (see [22, 24, 13, 8, 12]). The tools in the “Compatible” column have been used to successfully build and execute the system. Those in the “Final Test” column have been used not only to successfully build and execute the system, but also to test and verify e_float using the entire test suite for three different MP types, each tested at 30, 50, 100, 200 and 300 digits of precision.

2.2 Windows® Environment


e_float - Microsoft Visual Studio - [e_float.h [Read Only]] - [e_float.h [Read Only]]
File Edit View VAssistX Project Build Debug AI PowerTools ClearCase ClearQuest Tools Test Window Help
Release-efx
Win32
Debug-efx
Debug-gmp
Release-efx
Release-gmp
Configuration Manager...
Hex
Solution Explorer - e_float
Solution 'e_float' (1 project)
e_float
e_float
__licence
__make
e_float
efx
gmp
e_float.h
e_float_hyper_g.c
e_float_math.cpp
e_float_num.cpp
e_float_pow2.cpp
e_float_trig.cpp
e_float_types.h
Resour...
Solutio...
Class V...
e_float.h
W:\Ks\Ez\t_snapshot\User\Ks\PC_Software\Numeric

```
//          Copyright Christopher Kormanyos 2002 -
// Distributed under the Boost Software License, V
//    (See accompanying file LICENSE_1_0.txt or co
//          http://www.boost.org/LICENSE_1_0.txt)

#ifndef _E_FLOAT_H_2004_09_02
  #define _E_FLOAT_H_2004_09_02

  #include <cmath>
  #include <deque>
  #include <complex>
  #include <functional>

  #if defined(E_FLOAT_TYPE_EFX)
    #include <e_float/efx/e_float_efx.h>
    using efx::e_float;
```

Figure 2: The e_float configuration selection with Microsoft® Visual Studio® 2008 Professional (+SP1) is shown. The project configurations listed in Table 1 can be selected and built.

e_float can be built within a Windows® environment using Microsoft® Visual Studio® 2008 Professional with Service Pack 1 (SP1), as shown in Figure 2.

2.2.1 Building the Configurations ‘efx’, ‘gmp’, ‘mpfr’ and ‘f90’ in Windows®

- Open the solution workspace file `e_float/e_float.sln`, as shown in Figure 2.
- Select the appropriate project configuration such as “release–mpfr”, as shown in Figure 2 (see Table 1).
- Select 32–bit or 64–bit targets by selecting either “Win32” or “x64” using the “Solution Platforms” manager.
- Build the solution with the menu item `Build...Build Solution`, or use the corresponding build button.
- Rebuild the solution with the menu item `Build...Rebuild Solution`, or use the corresponding rebuild button.
- The configurations `gmp` and `mpfr` require GNU MP and GNU MP + MPFR, respectively. Therefore, GNU MP and MPFR have been ported to Microsoft® Visual Studio® 2008 and built for the Windows® environment, based in part on the original porting work provided by

Gladman [9]. Separate builds are provided for the 32–bit Intel® IA32 architecture as well as the 64–bit Intel® Core™2 architecture (x64). The prebuilt libraries `gmp.lib` and `mpfr.lib` for both IA32 as well as x64 have been copied to the respective directories `p4` and `x64`, which are subdirectories of the directory `e_float/gmp/4-2-4/vc9`. The link commands are already present within the project file.

- The configuration `f90` requires the Fortran 90 wrapper as well as the corresponding Fortran run–time libraries to be linked with the project. The Fortran 90 wrapper `libf90quad.lib` has been prebuilt for both IA32 as well as for x64 using the Intel® Fortran compiler. In addition, the Fortran run–time libraries have been extracted from the Intel® Fortran installation directories for both IA32 as well as x64. All of these libraries are stored in the respective directories `p4` and `x64`, which are subdirectories of `e_float/f90/libf90quad/vc9`. The necessary link commands are already present within the Microsoft® Visual Studio® project file.

2.2.2 Building the Configuration ‘clr’ in Windows®

- Open the solution workspace file `e_float/e_float.sln`,
- Select the project configuration “release–clr”.
- Build the solution with the menu item `Build...Build Solution`, or use the corresponding build button.
- Rebuild the solution with the menu item `Build...Rebuild Solution`, or use the corresponding rebuild button.
- This build configuration produces a Microsoft® Windows® .NET assembly which can be used with any .NET language in the Microsoft® CLR.

2.2.3 Building the Configuration ‘pyd’ in Windows®

- Open the solution workspace file `e_float/e_float.sln`,
- Select the project configuration “release–pyd”.
- Build and rebuild the solution as described above.
- This configuration is only available for the Win32 platform.

2.2.4 Building the Configuration ‘cas’ in Windows®

- This configuration depends strongly on the selected computer algebra system.
- An example in given for Mathematica® 7.1 for the Win32 platform.
- Contact the author for detailed instructions when interfacing with a computer algebra system.

2.3 UNIX–like Environment

e_float can be built within a UNIX–like environment, such as native UNIX, native Linux, mingw or cygwin. Building e_float within a UNIX–like environment requires GCC version 4.3.1 or higher and GNUmake version 3.81 or higher. In order to successfully build e_float with GCC, it must be verified that both the version of GCC as well as the version of GNU make are high enough. These checks can be done by querying the version of GCC (or g++) and the version of GNUmake as shown below.

```
chris@desktop:~$ g++ --version
g++ (GCC) 4.3.3
Copyright (C) 2008 Free Software Foundation, Inc.
...
```

```
chris@desktop:~$ make --version
GNU make 3.81
Copyright (C) 2006 Free Software Foundation, Inc.
...
```

2.3.1 Building the Configurations ‘efx’, ‘gmp’ and ‘mpfr’ in UNIX

- For UNIX–like operating systems, e_float can be built using either of the three MP implementations, `efx`, `gmp` or `mpfr`.
- The makefile is designed to accept the selection of the `MP` type as an input parameter, using either `MP=efx` or `MP=gmp`. If the `MP` flag is not provided, the default behavior is to build with `MP=efx`.
- After entering the make command (*i.e.* starting the build), the build of the project should begin and run successfully to completion.
- The `gmp` build configuration of e_float requires an installed version of GNU MP because it links with `libgmp.lib` from the GNU MP (*i.e.* uses the link command line switch `−lgmp`).
- The `mpfr` build configuration of e_float requires installed versions of GNU MP as well as GNU MP + MPFR because it links with `libgmp.lib` and `libmpfr.lib` (*i.e.* uses the link command line switches `−lgmp −lmpfr`).

A sample build command line using `efx::e_float` is shown in the command line sequence of the sample bash session below.

```
chris@desktop:~$ cd e_float
chris@desktop:~$ make MP=efx
```

A sample build command line using `gmp::e_float` is shown in the command line sequence of the sample bash session below.

```
chris@desktop:~$ cd e_float
chris@desktop:~$ make MP=gmp
```

The clean option of the e_float build is also supported. Be sure to use the correct value of the build flag MP for clean operations because the clean only cleans intermediate build results of the selected build configuration. A sample build command line for cleaning the project with the configuration gmp is shown in the bash command line sequence below.

```
chris@desktop:~$ cd e_float
chris@desktop:~$ make clean MP=gmp
```

2.3.2 Building the Configuration ‘clr’ in UNIX

The clr configuration is not currently supported in UNIX.

2.3.3 Building the Configuration ‘pyd’ in UNIX

- Build, rebuild and clean the solution using make with the make option MP=pyd. A dynamic library is built.
- This build links with the boost python library and the python library as shown in the makefile. Ensure that these libraries are properly installed for this build.

2.3.4 Building the Configuration ‘cas’ in Windows®

The cas configuration is not currently supported in UNIX. Specialized support must be developed for the desired computer algebra system.

3 Using e_float

Using and adapting the e_float system is intuitive and straightforward. Simple operations used for testing and benchmarking the e_float library can be carried out by modifying the test file `test.cpp` and rebuilding e_float, as described in Section 2 above. Integration of e_float in another independent project would, however, require a separate build.

3.1 Real–Numbered Arithmetic with e_float

Using `e_float` objects while coding is natural and intuitive because `e_float` objects can be used in the same way as conventional plain–old–data (POD) floating-point data types are used. Complete compatibility with the usual C++ semantics for real–numbered arithmetic has been implemented. There are also some extra utilities designed for standard situations which commonly arise in numerical programming. For example, there is as a globally defined digit tolerance called `ef::tol(void)` within the namespace '`ef`'. There is also a convenient `e_float` class member function called `order`, which returns the base–10 'big–O' order of an `e_float` object.

The interface to real–numbered arithmetic with e_float objects is defined in the C++ header file `<e_float/e_float.h>`. Real valued functions are defined within the namespace '`ef`'. A complete synopsis of the e_float class syntax is shown in Chapter 4. There is no real valued arithmetic interface to any language other than C++. There is no real valued arithmetic interface to the C language. The code chunk below illustrates an example of non-trivial, real valued arithmetic with e_float by showing a possible implementation of the small-argument Taylor series expansion of $\sin(x)$, $x \in \mathbb{R}$. Common arithmetic operations and usage of some of the utility functions are displayed.

```cpp
#include <iostream>
#include <iomanip>
#include <e_float/e_float.h>

// Compute the sin(x) for x near zero.
e_float sin_small_arg(const e_float& x)
{
  const e_float x_squared            = x * x;
        e_float sum                  = x;
        e_float k_fact               = ef::one();
        e_float x_pow_two_k_plus_one = x;

  bool b_neg_term = true;

  // Do the Taylor series expansion.
  for(INT32 k = 3; k < ef::max_iteration(); k += 2)
  {
    k_fact *= static_cast<INT32>((k - 1) * k);
    x_pow_two_k_plus_one *= x_squared;

    const e_float term = x_pow_two_k_plus_one / k_fact;

    if((term.order() - sum.order()) < -ef::tol())
    {
      // The required precision has been reached.
      break;
    }

    !b_neg_term ? sum += term : sum -= term;
    b_neg_term = !b_neg_term;
  }

  return sum;
}

int main(void)
{
  // Calculate the value of sin(1/10).
  static const e_float x = ef::tenth();
  static const e_float y = sin_small_arg(x);

  // Set the output stream precision and print the result.
  std::cout << std::setprecision(std::numeric_limits<e_float>::digits10)
            << y << std::endl;
}
```

The number of decimal digits is fixed at compile–time. The precision can be dynamically changed during run–time for intermediate calculation steps, but never increased to more than the fixed number of digits. The number of digits is set by setting the value of the preprocessor symbol `E_FLOAT _DIGITS10` which is used to initialize the value of `ef_digits10_setting`, which is a static constant 32-bit signed integer defined in the public interface of `e_float_base`.

A specialization of the template class `std::numeric_limits<e_float>` has been defined using the usual semantics. This means that it is possible to query details such as the number of decimal digits of precision or the maximum `e_float` value in the 'usual' manner for the C++ language. Support for formatted string output using `std::ostream` objects is defined using the usual semantics. The output stream must be appropriately set in order to display full precision. A code sample showing numeric limits, setting output stream precision and printing to output stream is shown below.

```
#include <iostream>
#include <iomanip>
#include <e_float/e_float.h>

int main(void)
{
  // Query the number of base-10 digits and print its value.
  static const int d = std::numeric_limits<e_float>::digits10;
  std::cout << d << std::endl;

  // Query the maximum value.
  static const e_float m = std::numeric_limits<e_float>::max();

  // Set the output stream precision and print the maximum value.
  std::cout << std::setprecision(std::numeric_limits<e_float>::digits10)
            << m << std::endl;
}
```

3.2 Mixed–Mode Arithmetic with e_float

All single–argument `e_float` class constructors have been declared with the `explicit` qualifier. This prevents unwanted side–effects such as the automatic compile–time conversion between POD types and `e_float` objects. If, for example, the single–argument `e_float` class constructor for `double` were non–explicit, then this

```
using ef::sin;

```

```
const e_float y = ef::pi() / sin(1.1); // Correctly results in compiler error.
```

would be ambiguous because the compiler would use conventional double–precision for the sin function and subsequently convert to an `e_float` result. This would compromise the precision of the final result because one of its intermediate calculation values would only have the precision of `double`.

However, even though all constructors `e_float` class constructors from POD types are explicit, there is support for global operators for multiplication and division with constant `INT32`. These operations are exact and non–ambiguous because the representation of `INT32` is exact and non–ambiguous. Using the global `INT32` multiplication operator is shown in the code sample below.

```
// Compute the exp(1.1 * x) / 12 for x real.
const e_float one_pt_one(1.1);
const e_float exp_x_times_one_pt_one = ef::exp(x * one_pt_one);

const e_float y = exp_x_times_one_pt_one / static_cast<INT32>(12);
```

Remarking further on the code sample above, some users might expect `e_float` class constructors from integer types to be non–explicit because the representation of integer types is exact and non–ambiguous. However, the `e_float` class constructors from integer types are nonetheless declared explicit. Otherwise some common C++ code sequences would lead to non–intuitive, confusing results. For example, this sequence

```

const e_float one_seventh = 1/7; // Properly gives a compiler error
```

might be expected to produce the value of $\frac{1}{7}$ to full precision. This might be expected from the perspective of a symbolic system such as a computer algebra system. But in fact, the C++ compiler actually reduces the value of $\frac{1}{7}$ to 0 in this case. The subsequent C++ initialization is carried out with the pure–integer value 0 instead of the full decimal value of $\frac{1}{7}$. This situation is somewhat non–intuitive and confusing. In order to facilitate non–ambiguity in these kinds of situations and to avoid unexpected compiler side–efects, all `e_float` constructors from integer types are declared `explicit`. Therefore when writing code for these kinds of situations, the proper way to initialize fractions such as $\frac{1}{7}$ or $\frac{1}{12}$ is like this

```

const e_float one_seventh = e_float(1) / static_cast<INT32>(7);
const e_float one_twelfth = ef::one()  / static_cast<INT32>(12);
```

The strict use of the C++ style static cast could be considered optional from a stylistic point of view. Nonetheless, the author prefers to include the static cast because this improves code portability by reducing the degrees of freedom which exist for compiler interpretation of the plain integer data type.

3.3 Complex–Numbered Arithmetic with e_float

Complex valued arithmetic is supported with the class `ef_complex`, which has the same public interface as the STL template class `std::complex<`*typename T*`>`. Complex valued functions are defined within the namespace '`efz`'. The interface to complex valued arithmetic with ef_complex objects is defined in the C++ header file <e_float/e_float_z.h>. Complex ef_complex objects can be used with each other in normal arithmetic expressions and also mixed together with real e_float objects. There is no complex valued arithmetic interface to any language other than C++. There is no complex valued arithmetic interface to the C language. The following code snippet displays the basic use of ef_complex objects.

```
#include <iostream>
#include <iomanip>
#include <functions/complex/e_float_complex.h>
#include <functions/functions.h>

int main(void)
{
  const ef_complex z1(ef::quarter(), ef::third());
  const ef_complex z2 = efz::sqrt(z1) + ef::tenth();
  const ef_complex z3 = efz::riemann_zeta(z2);

  std::cout << std::setprecision(std::numeric_limits<e_float>::digits10)
            << z3
            << std::endl;

  std::cout << std::setprecision(std::numeric_limits<e_float>::digits10)
            << efz::abs(z3)
            << std::endl;
}
```

3.4 Using the e_float Functions

Using the e_float library of functions is straightforward. All of the function prototypes are declared in C++ header files. These have been collected in a single header file called <functions/functions.h> which contains both real valued functions as well as complex valued functions. The function prototypes are C++ function prototypes written in the C++ language. There is no functional interface to any language other than C++. There is no functional interface to the C language. A complete listing of the functions supported by e_float is provided in Section 5. A use of functions is shown in the code snippet below.

```
#include <iostream>
#include <iomanip>
#include <e_float/e_float.h>
#include <functions/functions.h>

int main(void)
{
  // Calculate some real numbered function values...
  static const e_float x = ef::pi() / 7;
  static const e_float s = ef::sin(x);
  static const e_float g = ef::gamma(x);
  static const e_float b = ef::cyl_bessel_j(ef::third(), x);

  // ...and print them to full precision.
  std::cout.precision(std::numeric_limits<e_float>::digits10);

  std::cout << s << std::endl;
  std::cout << g << std::endl;
  std::cout << b << std::endl;
}
```

3.5 Using the STL with e_float objects

Real e_float objects and complex `ef_complex` objects can be used without limitation in standard STL containers and algorithms. Both e_float and `ef_complex` objects as well as pointers to these objects can be stored in all standard STL and TR1 containers such as STL's template `std::vector<`*typename T*`>` class or TR1's template `std::tr1::array<`*typename T*`, std::size_t` *N*`>` class. This provides for powerful manipulation with STL algorithms such as sorting, accumulating, streaming to output streams, etc. A use of e_float with a container, an algorithm, iterators, and stream output is shown in the code sample below.

```
#include <iostream>
#include <iomanip>
#include <iterator>
#include <deque>
#include <algorithm>
#include <e_float/e_float.h>
#include <functions/functions.h>

int main(void)
{
  // Show e_float STL usage.
  std::deque<e_float> z;

  // Compute three zeros of a Bessel function with very high order
  // and store these in a container.
  ef::cyl_bessel_j_zero(ef::million() + ef::pi(), 3u, z);

  std::cout.precision(std::numeric_limits<e_float>::digits10);

  // Copy the zeros to the output using an algorithm.
  std::copy(z.begin(), z.end(), std::ostream_iterator<e_float>(std::cout, "\n"));
}
```

3.6 Additional Examples Using e_float

Additional practical applications of e_float are provided by seven hand–written example files. The example files are stored in `examples/example*.cpp`, and they are part of the tools block.

Example 1 shows real–numbered usage in combination with a timing measurement. It calculates 21 non–trivial values of $P_\nu^\mu(x)$, with $\nu, \mu, x \in \mathbb{R}$. Example 2 shows complex–numbered usage in combination with a timing measurement. It calculates 21 non–trivial values of $\zeta(s)$, with $s \in \mathbb{Z}$.

Example 3 shows mixed mode, real/integer operation by calculating the real–numbered Jahnke–Emden–Lambda function, $\Lambda_\nu(x) = \{\Gamma(\nu+1)\, J_\nu(x)\} \,/\, \left(\frac{1}{2}x\right)^\nu$ [19]. The small–argument series expansion employs arithmetic operations as well as mixed–mode calculations. It also shows how to effectively use STL containers of `e_float` objects with STL algorithms.

Examples 4 and 5 use template utilities from the tools block. Example 4 performs a numerical differentiation, $\frac{\partial}{\partial \nu} J_\nu(151+\gamma)\big|_{(\nu=123+C)}$, where γ is Euler's constant and C is Catalan's constant. The derivative central difference rule is ill–conditioned and does not maintain full precision. Example 5 calculates a numerical integral, $J_0(x) = \frac{1}{\pi}\int_0^\pi \cos(x\,\sin t)\,dt$, with $x = (12+\gamma)$ using a recursive trapezoid rule. The utilities in Examples 4 and 5 make use of advanced object–oriented templates. These examples show how static and dynamic polymorphism can be combined to make efficient, elegant implementations for standard numerical tasks.

In example 6, some well–known conventional algorithms are extended to high precision and redesigned for use with real as well as complex numbers. Luke [21] developed quadruple–precision algorithms in Fortran 77 which calculate coefficients for expansions of hypergeometric functions in series of Chebyshev polynomials. Luke's algorithms `CCOEF2` for ${}_2F_1(a, b; c; z)$ on page 59, `CCOEF3` for ${}_1F_1(a; b; z)$ on page 74, and `CCOEF6` for ${}_1F_2(a; b, c; z)$ on page 85 have been extended to high precision. The inner loops have been simplified through analyses with a computer algebra system. Furthermore, these algorithms have been implemented as templates for simultaneous use with real as well as complex parameters. Table 4 shows that e_float's implementations of hypergeometric functions have strong limitations on their parameter ranges. Example 6 takes a promising first step toward extending these parameter ranges.

Example 7 shows e_float's interoperability with the MathLink® facility of Mathematica®. Template programs combined with e_float's interface to computer algebra systems are used to implement the generalized complex polygamma function $\psi_\nu(x)$, with $\nu, x \in \mathbb{R}$ or $\mathbb{Z}$. This extends the functionality of e_float since its native algorithm, $\psi_n(x)$, only supports real parameters.

3.7 Interoperability In Detail

3.7.1 Python Export

The e_float export to Python produces a dynamic, shared library. It is called "`e_float_pyd.*`", where the file ending will be either "`pyd`" or "`so`" for Windows® or Unix/Linux–GNU systems respectively. The shared library can be loaded into Python and the functions and classes of e_float can be used with its high level scripting capabilities such as list manipulation. Interoperability with mpmath [28] can be achieved with Python strings. The Python session below shows how to load the shared library and generate a list of function values. The final print command needs two carriage returns.

```
>>> import e_float_pyd
>>> from e_float_pyd import(e_float, ef_complex, ef, efz)
>>> lst=[ef.cyl_bessel_j(ef.third(), k + ef.quarter()) for k in range(10)]
>>> for y in lst: print y
```

3.7.2 Microsoft® CLR Export

The e_float export to the Microsoft® CLR creates a CLR assembly for the Microsoft®.net Framework. The name of the assembly is "`e_float_clr.dll`". It can be used with all Microsoft® CLR languages including C#, managed C++/CLI, IronPython, *etc*. The code below illustrates the syntax of e_float in the C# language.

```
using e_float_clr.cli;
namespace test
{
  class Program
  {
    static void Main(string[] args)
    {
      e_float x = ef.half();
      e_float y = new e_float(123);
      ef_complex z = efz.riemann_zeta(new ef_complex(x, y));

      System.Console.WriteLine(z.get_str());
    }
  }
}
```

3.7.3 Computer Algebra Systems

Using a computer algebra system from within e_float, in particular extending e_float with Mathematica®, has been discussed previously in Example 7. The architecture for this is in the directory `interop/cas`. It uses an abstract base class called `ComputerAlgebraSystemObject` whose public interface defines generic methods which exchange information with a computer algebra system using STL strings and containers of `e_float` objects. The other interoperability direction, using e_float from within Mathematica®, is not yet supported.

4 The e_float Class Architecture

The class architecture of the e_float system has been described in Section 1.3. The architecture is based on three main components: the `e_float_base` class, the `e_float` class and the `e_float` global arithmetic interface. The synopsis of these components is described in the following sections.

4.1 The e_float_base Class

The `e_float_base` class is a partly abstract base class from which the `e_float` class is derived. The 33 public pure–virtual interface functions of the `e_float_base` class describe all mathematical primitives needed for real–numbered arithmetic. The synopsis of the public interface to the `e_float_base` class is shown below.

```
// Synopsis of the e_float_base class.
class e_float_base
{
public:  // Public interface.

  // Digit settings
  static const INT32 ef_digits10_setting;
  static const INT32 ef_digits10;
  static const INT32 ef_digits10_extra;
  static const INT32 ef_digits10_tol;

  static const std::string::size_type& width_of_exponent_field(void);

  // This predominantly abstract base class has no constructors.

  // Virtual destructor.
  virtual ~e_float_base() { }

  // Specific special values.
  virtual const e_float_base& my_value_nan(void) const = 0;
  virtual const e_float_base& my_value_inf(void) const = 0;
  virtual const e_float_base& my_value_max(void) const = 0;
  virtual const e_float_base& my_value_min(void) const = 0;

  virtual INT32 cmp(const e_float&) const = 0;

  virtual void precision(const INT32) = 0;

  // Basic operations.
  virtual e_float_base& operator= (const e_float&) = 0;
  virtual e_float_base& operator+=(const e_float&) = 0;
  virtual e_float_base& operator-=(const e_float&) = 0;
  virtual e_float_base& operator*=(const e_float&) = 0;
  virtual e_float_base& operator/=(const e_float&) = 0;
  virtual e_float_base& mul_by_int(const INT32) = 0;
  virtual e_float_base& div_by_int(const INT32) = 0;

  virtual e_float_base& calculate_inv (void) = 0;
  virtual e_float_base& calculate_sqrt(void) = 0;

  // Comparison functions
  virtual bool isnan    (void) const = 0;
  virtual bool isinf    (void) const = 0;
  virtual bool isfinite(void) const = 0;

  virtual bool iszero (void) const = 0;
  virtual bool isone  (void) const = 0;
  virtual bool isint  (void) const = 0;
  virtual bool isneg  (void) const = 0;
          bool ispos  (void) const { return !isneg(); }

  virtual e_float_base& negate(void) = 0;

  // Operators pre-increment and pre-decrement
  virtual e_float_base& operator++(void)  = 0;
  virtual e_float_base& operator--(void)  = 0;

  // Argument range and check functions
  virtual INT64 order(void) const = 0;

  // Conversion routines
  virtual void    extract_parts       (double&, INT64&) const = 0;
  virtual double  extract_double      (void) const = 0;
  virtual INT64   extract_int64       (void) const = 0;
  virtual e_float extract_integer_part(void) const = 0;
  virtual e_float extract_decimal_part(void) const = 0;

  // Formated input and output routines.
  virtual void wr_string(std::string&, std::ostream&) const = 0;
  virtual bool rd_string(const char* const) = 0;

    // The implementation of the "has_its_own"-mechanism.
  virtual bool has_its_own_cbrt         (void) const;
  virtual bool has_its_own_rootn        (void) const;
  virtual bool has_its_own_exp          (void) const;
  virtual bool has_its_own_log          (void) const;
  virtual bool has_its_own_sin          (void) const;
  virtual bool has_its_own_cos          (void) const;
  virtual bool has_its_own_tan          (void) const;
  virtual bool has_its_own_asin         (void) const;
  virtual bool has_its_own_acos         (void) const;
  virtual bool has_its_own_atan         (void) const;
  virtual bool has_its_own_sinh         (void) const;
  virtual bool has_its_own_cosh         (void) const;
  virtual bool has_its_own_tanh         (void) const;
  virtual bool has_its_own_asinh        (void) const;
  virtual bool has_its_own_acosh        (void) const;
  virtual bool has_its_own_atanh        (void) const;
  virtual bool has_its_own_gamma        (void) const;
  virtual bool has_its_own_riemann_zeta (void) const;
  virtual bool has_its_own_cyl_bessel_jn(void) const;
  virtual bool has_its_own_cyl_bessel_yn(void) const;

  static e_float my_cbrt         (const e_float&);
  static e_float my_rootn        (const e_float&, const UINT32);
  static e_float my_exp          (const e_float&);
  static e_float my_log          (const e_float&);
  static e_float my_sin          (const e_float&);
  static e_float my_cos          (const e_float&);
  static e_float my_tan          (const e_float&);
  static e_float my_asin         (const e_float&);
  static e_float my_acos         (const e_float&);
  static e_float my_atan         (const e_float&);
  static e_float my_sinh         (const e_float&);
  static e_float my_cosh         (const e_float&);
  static e_float my_tanh         (const e_float&);
  static e_float my_asinh        (const e_float&);
  static e_float my_acosh        (const e_float&);
  static e_float my_atanh        (const e_float&);
  static e_float my_gamma        (const e_float&);
  static e_float my_riemann_zeta (const e_float&);
  static e_float my_cyl_bessel_jn(const INT32, const e_float&);
  static e_float my_cyl_bessel_yn(const INT32, const e_float&);

  // Utility get-functions to be used for Python exports.
  e_float     get_pos(void) const;
  e_float     get_neg(void) const;
  e_float     get_abs(void) const;
  INT64       get_int(void) const;
  double      get_flt(void) const;
  std::string get_str(void) const;
};

// Global operators with iostream objects.
std::ostream& operator<<(std::ostream& os, const e_float_base& f);
std::istream& operator>>(std::istream& is, e_float_base& f);
```

4.2 The e_float Class

The `e_float` class implements all of the pure–virtual functions of its base class and adds additional numerical constants and local private utility functions. The synopsis of the public interface to the `e_float` class is shown below.

```
// Synopsis of the e_float class.
namespace my_ef_space
{
  class e_float
  {
  private: // Private implementation details.
  public:  // Public interface.

    // Digit characteristics.
    static const INT64 ef_max_exp;
    static const INT64 ef_min_exp;
    static const INT64 ef_max_exp10;
    static const INT64 ef_min_exp10;

    // Constructors and destructor.
    e_float(void);
    explicit e_float(const INT32 n);
    explicit e_float(const INT64 n);
    explicit e_float(const UINT32 u);
    explicit e_float(const UINT64 u);
    explicit e_float(const double d);
    explicit e_float(const char* const s);
    explicit e_float(const std::string& str);
    e_float(const e_float& f);
    e_float(const double mantissa, const INT64 exponent);
    ~e_float();

    // Special values and precision.
    virtual const e_float& my_value_nan(void) const;
    virtual const e_float& my_value_inf(void) const;
    virtual const e_float& my_value_max(void) const;
    virtual const e_float& my_value_min(void) const;

    virtual void precision(const INT32 prec_digits);

    // Basic mathematical operations.
    virtual e_float& operator= (const e_float& v);
    virtual e_float& operator+=(const e_float& v);
    virtual e_float& operator-=(const e_float& v);
    virtual e_float& operator*=(const e_float& v);
    virtual e_float& operator/=(const e_float& v);

    virtual e_float& mul_by_int(const INT32 n);
    virtual e_float& div_by_int(const INT32 n);

    virtual e_float& calculate_inv (void);
    virtual e_float& calculate_sqrt(void);

    // Comparison functions.
    virtual INT32 cmp(const e_float& v) const;

    virtual bool isnan   (void) const;
    virtual bool isinf   (void) const;
    virtual bool isfinite(void) const;

    virtual bool iszero  (void) const;
    virtual bool isone   (void) const;
    virtual bool isint   (void) const;
    virtual bool isneg   (void) const;

    // Operators negate, post increment/decrement.
    virtual e_float& negate(void);
    virtual e_float& operator++(void);
    virtual e_float& operator--(void);

    // Extraction of the integer and decimal parts.
    virtual void    extract_parts       (double& mantissa, INT64& exponent) const;
    virtual double  extract_double      (void) const;
    virtual INT64   extract_int64       (void) const;
    virtual e_float extract_integer_part(void) const;
    virtual e_float extract_decimal_part(void) const;

    // The base-10 order of the e_float.
    virtual INT64 order(void) const;

    // The implementation of the "has_its_own"-mechanism.
    virtual bool has_its_own_cbrt (void) const;
    virtual bool has_its_own_rootn(void) const;
    virtual bool has_its_own_exp  (void) const;
    virtual bool has_its_own_log  (void) const;
    virtual bool has_its_own_sin  (void) const;
    virtual bool has_its_own_cos  (void) const;
    virtual bool has_its_own_tan  (void) const;
    virtual bool has_its_own_asin (void) const;
    virtual bool has_its_own_acos (void) const;
    virtual bool has_its_own_atan (void) const;
    virtual bool has_its_own_sinh (void) const;
    virtual bool has_its_own_cosh (void) const;
    virtual bool has_its_own_tanh (void) const;
    virtual bool has_its_own_asinh(void) const;
    virtual bool has_its_own_acosh(void) const;
    virtual bool has_its_own_atanh(void) const;

    static e_float my_cbrt        (const e_float& x);
    static e_float my_rootn       (const e_float& x, const UINT32 p);
    static e_float my_exp         (const e_float& x);
    static e_float my_log         (const e_float& x);
    static e_float my_sin         (const e_float& x);
    static e_float my_cos         (const e_float& x);
    static e_float my_tan         (const e_float& x);
    static e_float my_asin        (const e_float& x);
    static e_float my_acos        (const e_float& x);
    static e_float my_atan        (const e_float& x);
    static e_float my_sinh        (const e_float& x);
    static e_float my_cosh        (const e_float& x);
    static e_float my_tanh        (const e_float& x);
    static e_float my_asinh       (const e_float& x);
    static e_float my_acosh       (const e_float& x);
    static e_float my_atanh       (const e_float& x);
  };
}
```

4.3 The e_float Global Arithmetic Interface

The e_float global arithmetic interface defines global functions for arithmetic. The synopsis of the e_float global arithmetic interface is shown below.

```
// Synopsis of the e_float global arithmetic interface.
// Global operators post-increment and post-decrement
e_float operator++(e_float& u, int);
e_float operator--(e_float& u, int);

// Global unary operators of e_float reference.
      e_float  operator-(const e_float& u);
      e_float& operator+(      e_float& u);
const e_float& operator+(const e_float& u);

// Global add/sub/mul/div of const e_float reference with const e_float reference
e_float operator+(const e_float& u, const e_float& v);
e_float operator-(const e_float& u, const e_float& v);
e_float operator*(const e_float& u, const e_float& v);
e_float operator/(const e_float& u, const e_float& v);

// Specialization for global add/sub/mul/div of const e_float reference with INT32
e_float operator+(const e_float& u, const INT32 n);
e_float operator-(const e_float& u, const INT32 n);
e_float operator*(const e_float& u, const INT32 n);
e_float operator/(const e_float& u, const INT32 n);

e_float operator+(const INT32 n, const e_float& u);
e_float operator-(const INT32 n, const e_float& u);
e_float operator*(const INT32 n, const e_float& u);
e_float operator/(const INT32 n, const e_float& u);

// Specializations of global self-add/sub/mul-div of e_float reference with INT32
e_float& operator+=(e_float& u, const INT32 n);
e_float& operator-=(e_float& u, const INT32 n);
e_float& operator*=(e_float& u, const INT32 n);
e_float& operator/=(e_float& u, const INT32 n);

// Global comparison operators of const e_float reference with const e_float reference
bool operator< (const e_float& u, const e_float& v);
bool operator<=(const e_float& u, const e_float& v);
bool operator==(const e_float& u, const e_float& v);
bool operator!=(const e_float& u, const e_float& v);
bool operator>=(const e_float& u, const e_float& v);
bool operator> (const e_float& u, const e_float& v);

// Specializations of global comparison of e_float reference with INT32
bool operator< (const e_float& u, const INT32 n);
bool operator<=(const e_float& u, const INT32 n);
bool operator==(const e_float& u, const INT32 n);
bool operator!=(const e_float& u, const INT32 n);
bool operator>=(const e_float& u, const INT32 n);
bool operator> (const e_float& u, const INT32 n);

bool operator< (const INT32 n, const e_float& u);
bool operator<=(const INT32 n, const e_float& u);
bool operator==(const INT32 n, const e_float& u);
bool operator!=(const INT32 n, const e_float& u);
bool operator>=(const INT32 n, const e_float& u);
bool operator> (const INT32 n, const e_float& u);
```

4.4 Numeric Limits of the e_float Class

In order to maintain consistency with C++ programming styles and idioms, a template specialization of STL's standard numeric limits class has been implemented. It is called `std::numeric_limits<e_float>`. The class synopsis is shown below.

```
// Specialization of std::numeric_limits<e_float>.
namespace std
{
  template <> class numeric_limits<e_float>
  {
  public: // Implement some "usual" public members for floating point types.
    static const bool  is_specialized;
    static const bool  is_signed;
    static const bool  is_integer;
    static const bool  is_exact;
    static const bool  is_bounded;
    static const bool  is_modulo;
    static const bool  is_iec559;
    static const int   digits10;
    static const int   digits;
    static const INT64 max_exponent;
    static const INT64 min_exponent;
    static const INT64 max_exponent10
    static const INT64 min_exponent10;
    static const int   radix;
    static const int   round_style;
    static const bool  has_infinity;
    static const bool  has_quiet_NaN;
    static const bool  has_signaling_NaN;
    static const int   has_denorm;
    static const bool  has_denorm_loss;
    static const bool  traps;
    static const bool  tinyness_before;

    static const e_float& (min)       (void) throw();
    static const e_float& (max)       (void) throw();
    static const e_float& epsilon     (void) throw();
    static const e_float& round_error(void) throw();
    static const e_float& infinity    (void) throw();
    static const e_float& quiet_NaN   (void) throw();
  };
}
```

4.5 Adapting Another MP Implementation for use with e_float

From a software–architectural standpoint, it is quite straightforward to adapt another MP implementation for use with e_float. In order to be used with e_float, the MP needs to be implemented in a class which is called `e_float`. This adapted MP class must be derived from the `e_float_base` class and it must also reside within a unique namespace. Furthermore, the adapted MP class must fully implement all of the 33 pure–virtual functions of `e_float_base`, which are listed in Section 4.1. Although architecturally simple enough, the actual implementation details of the functions needed for the virtual interface — such as fast multiplication — can be technically very challenging to implement in an efficient fashion.

Any optional number of the virtual functions needed for the "*h*as–its–own" mechanism should also be implemented such that they return `true` if the MP's own function should be used. Be sure to also implement the corresponding static functions.

When the adapted MP type is fully implemented, it can be selected with a compile–time definition and used as a drop–in replacement for the selected MP type in e_float.

4.6 The Complex Class Interface

The `ef_complex` class defines the e_float interface to complex–numbered values. The synopsis of the public interface to the `ef_complex` class is shown below.

```
// Synopsis of the ef_complex public interface.
  class ef_complex
  {
  private:
  public:

    explicit ef_complex(const  INT32 n);
    explicit ef_complex(const  INT64 n);
    explicit ef_complex(const UINT32 u);
    explicit ef_complex(const UINT64 u);
    explicit ef_complex(const double d);

    ef_complex(const e_float& re = ef::zero(), const e_float& im = ef::zero());

    ef_complex(const ef_complex& z);

    e_float real(void) const;
    e_float imag(void) const;

    static e_float real(const ef_complex& z);
    static e_float imag(const ef_complex& z);

    e_float norm(void) const;

    ef_complex& operator= (const ef_complex& v);
    ef_complex& operator= (const e_float& v);
    ef_complex& operator+=(const ef_complex& v);
    ef_complex& operator-=(const ef_complex& v);
    ef_complex& operator*=(const ef_complex& v);
    ef_complex& operator/=(const ef_complex& v);

    ef_complex& operator+=(const e_float& v);
    ef_complex& operator-=(const e_float& v);
    ef_complex& operator*=(const e_float& v);
    ef_complex& operator/=(const e_float& v);

    // Operators pre-increment and post-increment
    const ef_complex& operator++(void);
          ef_complex  operator++(int);

    // Operators pre-decrement and post-decrement
    const ef_complex& operator--(void);
          ef_complex  operator--(int);

    // Unary operators.
    ef_complex  operator-(void) const;
    ef_complex& operator+(void);

    // Operators with integer.
    ef_complex& operator+=(const INT32 n);
    ef_complex& operator-=(const INT32 n);
    ef_complex& operator*=(const INT32 n);
    ef_complex& operator/=(const INT32 n);

    bool isnan   (void) const;
    bool isinf   (void) const;
    bool isfinite(void) const;
    bool isneg   (void) const;
    bool ispos   (void) const;
    bool isint   (void) const;
    bool isone   (void) const;
    bool iszero  (void) const;

    // Utility get-functions to be used for Python exports.
    ef_complex  get_pos(void) const;
    ef_complex  get_neg(void) const;
    e_float     get_abs(void) const;
    INT64       get_int(void) const;
    double      get_flt(void) const;
    std::string get_str(void) const;
  };

  inline std::ostream& operator<<(std::ostream& os, const ef_complex& z) { return os << '(' << z.↩
      real() << ',' << z.imag() << ')'; }
```

5 The e_float Functions

5.1 Supported Functions and Known Limitations

The functions supported by e_float including parameter ranges and known limitations are listed in Table 4. There is support for many functions and also for wide ranges of function parameters. Several functions such as Bessel functions and Legendre functions include very large parameter ranges, unavailable from other systems. The e_float system has been designed for 30 to 300 decimal digits of precision. Complex arithmetic is supported and elementary functions are implemented for real and complex numbers. The special functions are implemented primarily for real parameters, but some also accept complex parameters.

Symbol	Name	Parameters	Known Limitations
$+, -, \times, \div, =$, *etc*	operations	$x, y \in \mathbb{R}$; $x, y \in \mathbb{Z}$	
$+=, -=, \times=, \div=$, *etc*	self–operations	$x, y \in \mathbb{R}$; $x, y \in \mathbb{Z}$	
$<, \leq, >, \geq, ==, !=$	comparisons	$x, y \in \mathbb{R}$	
$x \rightarrow$ `INT64`, `double`, *etc*	convert $\mathbb{R}$	$x \in \mathbb{R}$	
$x \rightarrow$ `std::complex<double>`	convert $\mathbb{Z}$	$x \in \mathbb{Z}$	
$x \rightarrow$ `std::string`, *etc*	convert `char`	$x \in \mathbb{R}$; $x \in \mathbb{Z}$	
`std::ostream <<` x	output stream	$x \in \mathbb{R}$; $x \in \mathbb{Z}$	
fabs(x), floor(x), *etc*	math convert	$x, y \in \mathbb{R}$	
$x_{\texttt{frac}}$, $x_{\texttt{int}}$	constituents	$x \in \mathbb{R}$	
1, 2, $\frac{1}{3}$, $\texttt{INT32}_{\texttt{max}}$, *etc*	rationals	$\in \mathbb{Q}$	
$\sqrt{2}$, π, e, $\log(2)$, γ, *etc*	constants	$\in \mathbb{R}$, $\notin \mathbb{Q}$	
$n!$, $n!!$, B_n, E_n, *etc*	int functions	$n \in \mathbb{N}^+$	$n \lesssim 10^6$
P_n	primes	$n \in \mathbb{N}^+$	$n \lesssim 10^6$
$n = \prod P_i$	prime factors	$n \in \mathbb{N}^+$	$n \lesssim 10^9$
e^x, x^a	power	$x, a \in \mathbb{R}$; $x, a \in \mathbb{Z}$	
$\log(x), \log_a(x)$	logarithmic	$x, a \in \mathbb{R}$; $x, a \in \mathbb{Z}$	
$\sin(x)$, $\cos(x)$, *etc*	trigonometric	$x \in \mathbb{R}$; $x \in \mathbb{Z}$	$\lvert\mathfrak{Re}(x)\rvert \lesssim 10^{20}$
asin(x), acos(x), *etc*		$x \in \mathbb{R}$; $x \in \mathbb{Z}$	
$\sinh(x)$, $\cosh(x)$, *etc*	hyperbolic	$x \in \mathbb{R}$; $x \in \mathbb{Z}$	
asinh(x), acosh(x), *etc*		$x \in \mathbb{R}$; $x \in \mathbb{Z}$	
$Ai(x), Bi(x), Ai'(x), Bi'(x)$	Airy	$x \in \mathbb{R}$	$-10^{10} \lesssim x \lesssim 10^4$
$J_\nu(x), Y_\nu(x), K_\nu(x), I_\nu(x)$	Bessel	$x \in \mathbb{R}$; $\nu \in \mathbb{R}$	$\lvert\nu\rvert \lesssim 10^9$
$T_n(x), U_n(x), L_n(x), H_n(x)$	polynomial	$x \in \mathbb{R}$; $x \in \mathbb{Z}$; $n \in \mathbb{N}^+$	$\lvert n\rvert \lesssim 10^5$
$\Gamma(x)$, $(x)_a$, *etc*	gamma	$x, a \in \mathbb{R}$; $x, a \in \mathbb{Z}$	
$\Gamma(x, a)$	incomp gamma	$x, a \in \mathbb{R}$	$\lvert a\rvert \lesssim 10^2$
$\psi^n(x)$	polygamma	$x \in \mathbb{R}$; $n \in \mathbb{N}^+$,	$0 \leq n \lesssim 10^2$ for $x < 0$
${}_1F_1(a, b; x)$	conf hyperg	$x \in \mathbb{R}$	$\lvert a\rvert, \lvert b\rvert \lesssim 10^2$
${}_pF_q(a_1, \ldots a_p; b_1, \ldots b_q; x)$	hyperg	$x \in \mathbb{R}$; $p, q \in \mathbb{N}^+$; $a_n, b_n \in \mathbb{R}$;	$\lvert x\rvert \leq 1$; only partial support for $a_n \in \mathbb{N}^-$; $\lvert a\rvert, \lvert b\rvert \lesssim 10$
$P_\nu^\mu(x), Q_\nu^\mu(x)$	Legendre	$x \in \mathbb{R}$; $\nu, \mu \in \mathbb{R}$	$\lvert x\rvert \leq 1$; $\lvert\nu\rvert, \lvert\mu\rvert \lesssim 10^5$; $Q_\nu^\mu(x)$ could be NaN if ν, $\mu \ni \mathbb{N}$, but $\nu + \mu \in \mathbb{N}$
$L_\nu^\lambda(x)$	Laguerre	$x \in \mathbb{R}$; $\nu, \lambda \in \mathbb{R}$	$\lvert\nu\rvert, \lvert\lambda\rvert \lesssim 10^2$
$D_\nu(x)$	parabolic cyl	$x \in \mathbb{R}$; $\nu \in \mathbb{R}$	$\lvert\nu\rvert \lesssim 10^5$
$H_\nu(x)$	Hermite	$x \in \mathbb{R}$; $\nu \in \mathbb{R}$	$\lvert\nu\rvert \lesssim 10^5$
$E(\phi, m)$, $F(\phi, m)$, $K(m)$	Elliptic	$\phi, m \in \mathbb{R}$	$0 \leq m \leq 1$
$\zeta(s)$	Riemann zeta	$s \in \mathbb{R}$; $s \in \mathbb{Z}$	$\lvert\mathfrak{Im}(s)\rvert \lesssim 10^6$
$\zeta(s, a)$	Hurwitz zeta	$s, a \in \mathbb{R}$; $s, a \in \mathbb{Z}$	$\lvert\mathfrak{Im}(s)\rvert \lesssim 10^6$; $\mathfrak{Re}(s) \gtrsim -10$
$\mathrm{Li}_n(x)$	polylog	$x \in \mathbb{R}$; $x \leq 1$; $n \in \mathbb{N}$	$n \in \mathbb{N}^+$ for $x < 0$

Table 4: The functions supported by e_float, including their parameters and known limitations (other than overflow and underflow), are listed.

5.2 Function Details

The following sections briefly describe the functions which are implemented in the e_float library. Only very terse mathematical descriptions of the functions are provided, for example the primary function definition or a single series expansion at one point such as zero or one. The brief descriptions and remarks are only meant to uniquely define the functions and do not provide complete descriptions of the algorithms and computations. The actual calculations in the e_float library use many additional algorithms and function representations, often involving hypergeometric series or precomputed tables, to calculate function values over wide parameter ranges. Real valued functions are defined within the namespace '`ef`'. Utility functions are defined within the namespace '`ef`'. Complex valued functions are defined within the namespace '`efz`'.

5.2.1 Integer and Constant Functions

```
const e_float& zero         (void);
const e_float& one          (void);
const e_float& two          (void);
const e_float& three        (void);
const e_float& four         (void);
const e_float& five         (void);
const e_float& six          (void);
const e_float& seven        (void);
const e_float& eight        (void);
const e_float& nine         (void);
const e_float& ten          (void);
const e_float& twenty       (void);
const e_float& thirty       (void);
const e_float& forty        (void);
const e_float& fifty        (void);
const e_float& hundred      (void);
const e_float& two_hundred  (void);
const e_float& three_hundred(void);
const e_float& four_hundred (void);
const e_float& five_hundred (void);
const e_float& thousand     (void);
const e_float& two_k        (void);
const e_float& three_k      (void);
const e_float& four_k       (void);
const e_float& five_k       (void);
const e_float& ten_k        (void);
const e_float& twenty_k     (void);
const e_float& thirty_k     (void);
const e_float& forty_k      (void);
const e_float& fifty_k      (void);
const e_float& hundred_k    (void);
const e_float& million      (void);
const e_float& ten_M        (void);
const e_float& hundred_M    (void);
const e_float& billion      (void);
const e_float& int32max     (void);
const e_float& int32min     (void);
const e_float& int64max     (void);
const e_float& int64min     (void);
const e_float& one_minus    (void);
const e_float& tenth        (void);
const e_float& fifth        (void);
const e_float& quarter      (void);
const e_float& third        (void);
const e_float& half         (void);
const e_float& two_third    (void);
const e_float& four_third   (void);
const e_float& three_half   (void);
```

Returns: These functions return a static constant reference to the precomputed value of the respective rational number.

```
const e_float& sqrt2        (void);
const e_float& sqrt3        (void);
const e_float& pi           (void);
const e_float& pi_half      (void);
const e_float& pi_quarter   (void);
const e_float& pi_squared   (void);
const e_float& two_pi       (void);
const e_float& sqrt_pi      (void);
const e_float& degree       (void);
const e_float& exp1         (void);
const e_float& ln2          (void);
const e_float& ln3          (void);
const e_float& ln10         (void);
const e_float& log10_2      (void);
const e_float& golden_ratio (void);
const e_float& euler_gamma  (void);
const e_float& catalan      (void);
const e_float& khinchin     (void);
const e_float& glaisher     (void);
```

Returns: These functions return a static constant reference to the precomputed value of the respective mathematical constant.

```
void prime(const UINT32 n, std::deque<UINT32>& primes);
```

Effects: This function calculates the first `n` prime numbers and stores them in the container `primes`.

```
void integer_factors(const UINT32 n, std::deque<Util::point_nm>& pf);
```

Effects: This function calculates the prime factorization of the unsigned integer `n` and stores the results as ordered sequence of integer pairs in the container `pf`.

Remark: The functions `prime` and `integer_factors` are implemented as utility functions. They are not intended for high-performance use.

```
e_float euler    (const UINT32 n);
e_float bernoulli(const UINT32 n);
e_float stirling2(const UINT32 n, const UINT32 k);
```

Effects: These functions calculate the integer function of the respective arguments.

Returns: The `euler` function returns the signed integer number which can be defined by the identity [33]

$$\frac{1}{\cosh(x)} = \frac{2}{e^x + e^{-x}} \equiv \sum_{n=0}^{\infty} \frac{E_n x^n}{n!}. \tag{1}$$

Returns: The `bernoulli` function returns the signed rational number which can be defined by the identity [35]

$$\frac{x}{e^x - 1} \equiv \sum_{n=0}^{\infty} \frac{B_n x^n}{n!}. \tag{2}$$

Returns: The `stirling2` function returns the unsigned integer Stirling number of the second kind given by the formula [33]

$$S(n,k) = \left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{j=0}^{k} (-1)^{k-j} \binom{k}{j} j^n. \tag{3}$$

5.2.2 Elementary Functions

```
// Utility functions for real and complex valued arguments.
bool isnan(const double x);
bool isnan(const e_float& x);
bool isnan(const ef_complex& z);

bool finite(const double x);
bool finite(const e_float& x);
bool finite(const ef_complex& z);

bool isneg(const double x);
bool isneg(const e_float& x);
bool isneg(const ef_complex& z);

e_float fabs(const e_float& x);
e_float abs  (const e_float& x);
e_float abs  (const ef_complex& z);
e_float real(const e_float& x);
e_float real(const ef_complex& z);
e_float imag(const e_float& x);
e_float imag(const ef_complex& z);

bool ispos(const double x);
bool ispos(const e_float& x);
bool ispos(const ef_complex& z);

bool isint(const double x);
bool isint(const e_float& x);
bool isint(const ef_complex& z);

bool isone(const double x);
bool isone(const e_float& x);
bool isone(const ef_complex& z);

bool iszero(const double x);
bool iszero(const e_float& x);
bool iszero(const ef_complex& z);

double to_double(const double& x);
double to_double(const e_float& x);

INT64 to_int64(const double x);
INT64 to_int64(const e_float& x);
INT64 to_int64(const ef_complex& z);

bool small_arg(const double x);
bool small_arg(const e_float& x);
bool small_arg(const ef_complex& z);
```

Returns: These functions return the given utility function for their respective arguments.

Remark: The real valued `real` function returns its argument and the real valued `imag` function returns zero.

Remark: The functions `isint` and `isone` test for integer values within a tolerance.

Remark: The `small_arg` function tests if $|\mathrm{x}| \overset{?}{\lesssim} 10^{-d/6}$, where d is the number of decimal digits of precision of the underlying type.

```
// Elementary real valued functions using the C++ naming convention.
e_float floor      (const e_float& x);
e_float ceil       (const e_float& x);
e_float pow2       (const INT64 p);
e_float pown       (const e_float& x, const INT64 p);
e_float inv        (const e_float& x);
e_float sqrt       (const e_float& x);
e_float sqrt1pm1   (const e_float& x);
e_float cbrt       (const e_float& x);
e_float rootn      (const e_float& x, const INT32 p);
e_float exp        (const e_float& x);
e_float log        (const e_float& x);
e_float log10      (const e_float& x);
e_float loga       (const e_float& x, const e_float& a);
e_float log1p      (const e_float& x);
e_float log101p    (const e_float& x);
e_float loga1p     (const e_float& x, const e_float& a);
e_float log1p1m2   (const e_float& x);
e_float pow        (const e_float& x, const e_float& a);
e_float powm1      (const e_float& x, const e_float& a);
void    sincos     (const e_float& x, e_float* const p_sin,
                                      e_float* const p_cos);
e_float sin        (const e_float& x);
e_float cos        (const e_float& x);
e_float tan        (const e_float& x);
e_float csc        (const e_float& x);
e_float sec        (const e_float& x);
e_float cot        (const e_float& x);
e_float asin       (const e_float& x);
e_float acos       (const e_float& x);
e_float atan       (const e_float& x);
e_float atan2      (const e_float& y, const e_float& x);
void    sinhcosh   (const e_float& x, e_float* const p_sin,
                                      e_float* const p_cos);
e_float sinh       (const e_float& x);
e_float cosh       (const e_float& x);
e_float tanh       (const e_float& x);
e_float asinh      (const e_float& x);
e_float acosh      (const e_float& x);
e_float atanh      (const e_float& x);
```

Returns: These functions return the real value of the corresponding elementary function for their respective real valued arguments.

Remark: Not every function has a counterpart function in C or C++.

Remark: The `sqrt1pm1` function returns $\sqrt{1+x}-1$, for small x.

Remark: The `log1p` function returns $\log(1+x)$. The `log101p` function returns $\log_{10}(1+x)$. The `loga1p` function returns $\log_a(1+x)$.

Remark: The `log1p1m2` function returns $(1/2)\log\left[(1+x)/(1-x)\right]$.

Remark: The `powm1` function returns $x^a - 1$ for x near one.

```
// Elementary complex valued functions using the C++ naming convention.
ef_complex polar   (const e_float& mod, const e_float& arg);
ef_complex conj    (const ef_complex& z);
ef_complex iz      (const ef_complex& z);
ef_complex sin     (const ef_complex& z);
ef_complex cos     (const ef_complex& z);
ef_complex tan     (const ef_complex& z);
void       sincos  (const ef_complex& z, ef_complex* const p_sin,
                                         ef_complex* const p_cos);
ef_complex csc     (const ef_complex& z);
ef_complex sec     (const ef_complex& z);
ef_complex cot     (const ef_complex& z);
ef_complex asin    (const ef_complex& z);
ef_complex acos    (const ef_complex& z);
ef_complex atan    (const ef_complex& z);
ef_complex inv     (const ef_complex& z);
ef_complex sqrt    (const ef_complex& z);
ef_complex exp     (const ef_complex& z);
ef_complex log     (const ef_complex& z);
ef_complex log10   (const ef_complex& z);
ef_complex loga    (const ef_complex& z, const ef_complex& a);
ef_complex pown    (const ef_complex& z, const INT64 p);
ef_complex pow     (const ef_complex& z, const ef_complex& a);
ef_complex rootn   (const ef_complex& z, const INT32 p);
ef_complex sinh    (const ef_complex& z);
ef_complex cosh    (const ef_complex& z);
ef_complex tanh    (const ef_complex& z);
void       sinhcosh(const ef_complex& z, ef_complex* const p_sinh,
                                         ef_complex* const p_cosh);
ef_complex asinh   (const ef_complex& z);
ef_complex acosh   (const ef_complex& z);
ef_complex atanh   (const ef_complex& z);
```

Returns: These functions return the complex value of the corresponding elementary function for their respective complex valued arguments.

Remark: Not every function has a counterpart function in C or C++.

5.2.3 Airy Functions

```
e_float airy_a      (const e_float& x);
e_float airy_a_prime(const e_float& x);
e_float airy_b      (const e_float& x);
e_float airy_b_prime(const e_float& x);
```

Effects: These functions compute the Airy functions and derivatives for their respective argument `x`, with `x` $\in \mathbb{R}$.

Returns: The `airy_a` function returns [35]

$$Ai(x) = \frac{1}{3^{2/3}\Gamma(\frac{2}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{2}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k - \frac{x}{\sqrt[3]{3}\,\Gamma(\frac{1}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{4}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k . \tag{4}$$

Returns: The `airy_a_prime` function returns [35]

$$Ai'(x) = \frac{x^2}{2\,3^{2/3}\Gamma(\frac{2}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{5}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k - \frac{1}{\sqrt[3]{3}\,\Gamma(\frac{1}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{1}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k . \tag{5}$$

Returns: The `airy_b` function returns [35]

$$Bi(x) = \frac{1}{\sqrt[6]{3}\,\Gamma(\frac{2}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{2}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k + \frac{\sqrt[6]{3}}{\Gamma(\frac{1}{3})}\, x \sum_{k=0}^{\infty} \frac{1}{\left(\frac{4}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k . \tag{6}$$

Returns: The `airy_b_prime` function returns [35]

$$Bi'(x) = \frac{\sqrt[6]{3}}{\Gamma(\frac{1}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{1}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k + \frac{x^2}{2\,\sqrt[6]{3}\,\Gamma(\frac{2}{3})} \sum_{k=0}^{\infty} \frac{1}{\left(\frac{5}{3}\right)_k k!} \left(\frac{x^3}{9}\right)^k . \tag{7}$$

```
void airy_a_zero(const UINT32 k, std::deque<e_float>& zeros);
void airy_b_zero(const UINT32 k, std::deque<e_float>& zeros);
```

Effects: These functions compute the first `k` zeros of the Airy functions on the negative real axis, with results stored in the container `zeros`.

5.2.4 Bessel Functions

```
e_float cyl_bessel_i(const INT32 n,     const e_float& x);
e_float cyl_bessel_i(const e_float& v, const e_float& x);
e_float cyl_bessel_j(const INT32 n,     const e_float& x);
e_float cyl_bessel_j(const e_float& v, const e_float& x);
e_float cyl_bessel_k(const INT32 n,     const e_float& x);
e_float cyl_bessel_k(const e_float& v, const e_float& x);
e_float cyl_neumann (const INT32 n,     const e_float& x);
e_float cyl_neumann (const e_float& v, const e_float& x);
```

Effects: These functions compute the cylindrical Bessel functions for their respective arguments `v`, `n`, `x`, with `v`, `x` $\in \mathbb{R}$; `n` $\in \mathbb{N}$.

Returns: The `cyl_bessel_i` function returns [35]

$$I_\nu(x) \;=\; \frac{1}{\Gamma(\nu+1)} \left(\frac{x}{2}\right)^{\nu} \sum_{k=0}^{\infty} \frac{(x/2)^{2k}}{(\nu+1)_k\, k!} \quad . \tag{8}$$

Returns: The `cyl_bessel_j` function returns [35]

$$J_\nu(x) \;=\; \frac{1}{\Gamma(\nu+1)} \left(\frac{x}{2}\right)^{\nu} \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{2k}}{(\nu+1)_k\, k!} \quad . \tag{9}$$

Returns: The `cyl_bessel_k` function returns [35]

$$K_\nu(x) \;=\; \frac{\Gamma(\nu)}{2} \left(\frac{x}{2}\right)^{-\nu} \sum_{k=0}^{\infty} \frac{(x/2)^{2k}}{(1-\nu)_k\, k!} + \frac{\Gamma(-\nu)}{2} \left(\frac{x}{2}\right)^{\nu} \sum_{k=0}^{\infty} \frac{(x/2)^{2k}}{(\nu+1)_k\, k!} \quad . \tag{10}$$

Returns: The `cyl_neumann` function returns [35]

$$\begin{aligned} Y_\nu(x) \;=\; & -\;\frac{\Gamma(\nu)}{\pi}\left(\frac{x}{2}\right)^{-\nu}\sum_{k=0}^{\infty}\frac{(-1)^k(x/2)^{2k}}{(1-\nu)_k\;k!} \\ & -\;\frac{\Gamma(-\nu)\cos(\nu\pi)}{\pi}\left(\frac{x}{2}\right)^{\nu}\sum_{k=0}^{\infty}\frac{(-1)^k(x/2)^{2k}}{(\nu+1)_k\;k!}\quad . \end{aligned} \tag{11}$$

```
e_float cyl_bessel_i_prime(const e_float& v, const e_float& x);
e_float cyl_bessel_i_prime(const INT32 n,    const e_float& x);
e_float cyl_bessel_j_prime(const e_float& v, const e_float& x);
e_float cyl_bessel_j_prime(const INT32 n,    const e_float& x);
e_float cyl_bessel_k_prime(const e_float& v, const e_float& x);
e_float cyl_bessel_k_prime(const INT32 n,    const e_float& x);
e_float cyl_neumann_prime (const e_float& v, const e_float& x);
e_float cyl_neumann_prime (const INT32 n,    const e_float& x);
```

Effects: These functions compute the derivatives of the cylindrical Bessel functions for their respective arguments `v`, `n`, `x`, with `v`, `x` $\in \mathbb{R}$; `n` $\in \mathbb{N}$.

```
void cyl_bessel_j_zero(const e_float& v, const UINT32 k, std::deque<e_float>& zeros);
void cyl_bessel_j_zero(const INT32 n,    const UINT32 k, std::deque<e_float>& zeros);
```

Effects: These functions compute the first `k` positive zeros of the cylindrical Bessel functions $J_\nu(x)$ and $J_n(x)$, with results stored in the container `zeros`.

5.2.5 Orthogonal Polynomials

```
e_float    chebyshev_t(const INT32 n, const e_float& x);
ef_complex chebyshev_t(const INT32 n, const ef_complex& z);
e_float    chebyshev_u(const INT32 n, const e_float& x);
ef_complex chebyshev_u(const INT32 n, const ef_complex& z);
e_float    hermite    (const INT32 n, const e_float& x);
ef_complex hermite    (const INT32 n, const ef_complex& z);
e_float    laguerre   (const INT32 n, const e_float& x);
ef_complex laguerre   (const INT32 n, const ef_complex& z);
e_float    legendre_p (const INT32 n, const e_float& x);
e_float    legendre_q (const INT32 n, const e_float& x);
```

Effects: These functions compute the orthogonal polynomials for their respective arguments `n`, `x`, `z`, with `x` $\in \mathbb{R}$; `z` $\in \mathbb{Z}$; and `n` $\in \mathbb{N}^+$.

5.2.6 Gamma and Related Functions

```
e_float    gamma                (const e_float& x);
ef_complex gamma                (const ef_complex& z);
e_float    incomplete_gamma     (const e_float& a, const e_float& x);
e_float    gen_incomplete_gamma(const e_float& a, const e_float& x0,
                                                  const e_float& x1);
```

Effects: These functions compute the gamma functions for their respective arguments a, x, x0, x1, z, with a, x, x0, x1 $\in \mathbb{R}$; z $\in \mathbb{Z}$.

Returns: The gamma function returns [35]

$$\Gamma(x) \;=\; \int_0^\infty t^{a-1} e^{-t} dt \quad . \tag{12}$$

Returns: The incomplete_gamma function returns [35]

$$\Gamma(a,x) \;=\; \int_x^\infty t^{a-1} e^{-t} dt \quad . \tag{13}$$

Returns: The (generalized) gen_incomplete_gamma function returns [35]

$$\Gamma(a,x1,x2) \;=\; \int_{x1}^{x2} t^{a-1} e^{-t} dt \quad . \tag{14}$$

Remark: The incomplete_gamma function and the gen_incomplete_gamma function are implemented as utility functions. They do not always maintain precision over the entire range of their parameters.

```
e_float beta           (const e_float& a,     const e_float& b);
e_float beta           (const ef_complex& a, const ef_complex& b);
e_float incomplete_beta(const e_float& x,     const e_float& a,
                                             const e_float& b);
```

Effects: These functions compute the beta functions for their respective arguments a, b, x, with a, b, x $\in \mathbb{R}$; or a, b, z $\in \mathbb{Z}$.

Returns: The beta function returns [35]

$$B(a,b) \;=\; \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)} \quad . \tag{15}$$

Returns: The incomplete_beta function returns [35]

$$B_x(a,b) \;=\; \int_0^x t^{a-1}(1-t)^{b-1} dt \quad . \tag{16}$$

Remark: The incomplete_beta function is implemented as a utility function. It does not cover the entire range of its parameters.

```
e_float factorial (const UINT32 n);
e_float factorial2(const  INT32 n);
```

Effects: These functions compute the factorial or double factorial functions for their respective argument n, with n $\in \mathbb{N}^+$.

Returns: The factorial function returns [35]

$$n! \;=\; \prod_{k=1}^{n} k \quad . \tag{17}$$

Returns: The `factorial2` function (double factorial) returns [33]

$$n!! = \begin{cases} 2^{n/2}(n/2)! & \text{if } n \text{ is even,} \\ \binom{\frac{n}{2}}{\frac{n-1}{2}} 2^{(n-1)/2}\left[(n-1)/2\right]! & \text{if } n \text{ is odd.} \end{cases} \tag{18}$$

```cpp
e_float binomial(const UINT32 n,    const UINT32 k);
e_float binomial(const UINT32 n,    const e_float& y);
e_float binomial(const e_float& x, const UINT32 k);
e_float binomial(const e_float& x, const e_float& y);
```

Effects: These functions compute the binomial functions for their respective arguments `n`, `k`, `x`, `y`, with `n`, `k` $\in \mathbb{N}^+$; `x`, `y` $\in \mathbb{R}$.

Returns: The `binomial` function in its integer form returns [35]

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad . \tag{19}$$

Returns: The `binomial` function in its non-integer form returns [35]

$$\binom{x}{y} = \frac{\Gamma(x+1)}{\Gamma(y+1)\Gamma(x-y+1)} \quad . \tag{20}$$

```cpp
e_float    pochhammer(const e_float& x,    const UINT32 n);
ef_complex pochhammer(const ef_complex& z, const UINT32 n);
e_float    pochhammer(const e_float& x,    const e_float&    a);
ef_complex pochhammer(const ef_complex& z, const ef_complex& a);
```

Effects: These functions compute the Pochhammer functions for their respective arguments.

Returns: The `pochhammer` function in its integer form returns [32]

$$x_n = x(x+1)\ldots(x+n-1) \quad . \tag{21}$$

Returns: The `pochhammer` function in its non-integer form returns [32]

$$x_a = \frac{\Gamma(x+a)}{\Gamma(x)} \quad . \tag{22}$$

5.2.7 Hypergeometric Series

```cpp
e_float hyperg_0f0    (const e_float& x);
e_float hyperg_0f1    (const e_float& b, const e_float& x);
e_float hyperg_0f1_reg(const e_float& a, const e_float& x);
e_float hyperg_1f0    (const e_float& a, const e_float& x);
e_float hyperg_1f1    (const e_float& a, const e_float& b, const e_float& x);
e_float hyperg_1f1_reg(const e_float& a, const e_float& b, const e_float& x);
e_float hyperg_2f0    (const e_float& a, const e_float& b, const e_float& x);
e_float hyperg_2f1    (const e_float& a, const e_float& b, const e_float& c,
                       const e_float& x);
e_float hyperg_2f1_reg(const e_float& a, const e_float& b, const e_float& c,
                       const e_float& x);
e_float hyperg_pfq    (const std::deque<e_float>& a, const
                             std::deque<e_float>& b, const e_float& x);
```

Effects: These functions compute the hypergeometric series for their respective real valued arguments.

Remark: There is support for hypergeometric functions, but these are primarily implemented as utilities for other calculations and not intended to cover the complete parameter ranges. These series have limited convergence, generally only for arguments less than unity in magnitude. Some of the series do not check for negative integer numerator parameters. Others do.

Remark: The suffix ‘`_reg`’ denotes a ‘regularized’ hypergeometric function. For example the function `hyperg_2f1_reg` is the regularized hypergeometric function, defined in [34] and [35] by

$$ {}_2\widetilde{F}_1(a,b;c;x) = {}_2F_1(a,b;c;x)/\Gamma(c). \tag{23}$$

```
e_float conf_hyperg(const e_float& a, const e_float& c,
                    const e_float& x);
e_float      hyperg(const e_float& a, const e_float& b, const e_float& c,
                    const e_float& x);
```

Remark: The function name of the hypergeometric series `conf_hyperg` is synonymous with the function name of the hypergeometric series `hyperg_1f1`.

Remark: The function name of the hypergeometric series `hyperg` is synonymous with the function name of the hypergeometric series `hyperg_2f1`.

5.2.8 Generalized Legendre Functions

```
e_float legendre_p(const e_float& v, const e_float& x);
e_float legendre_p(const e_float& v, const e_float& u, const e_float& x);
e_float legendre_p(const e_float& v, const INT32    m, const e_float& x);
e_float legendre_p(const INT32    n, const e_float& u, const e_float& x);
e_float legendre_p(const INT32    n, const INT32    m, const e_float& x);
```

Effects: These functions compute the associated Legendre functions of type 1 for their respective arguments `v`, `u`, `n`, `m`, `x`, with `v`, `u`, `x` $\in \mathbb{R}$; `n`, `m` $\in \mathbb{N}$.

Returns: The `assoc_legendre_p` function of non-integer degree and non-integer order returns [35]

$$ P_\nu^\mu(x) = \frac{1}{\Gamma(1-\mu)} \left(\frac{1+x}{1-x}\right)^{\mu/2} \sum_{k=0}^{\infty} \frac{(-\nu)_k(\nu+1)_k}{(1-\mu)_k\, k!} \left(\frac{1-x}{2}\right)^k . \tag{24}$$

Returns: The `assoc_legendre_p` function of pure integer order and pure integer degree returns [32]

$$P_n^m(x) = \frac{(-1)^m}{2^n n!}(1-x^2)^{m/2}\frac{d^{n+m}}{dx^{n+m}}(x^2-1)^n \quad . \tag{25}$$

Remark: The functions $P_\nu^\mu(x)$, $P_n^\mu(x)$ and $P_\nu^m(x)$ are only implemented for real values of x with $|x| \leq 1$.

Remark: There are specialized series representations for Legendre functions with mixed integer, non-integer combinations of degree and order.

```
e_float legendre_q(const e_float& v, const e_float& x);
e_float legendre_q(const e_float& v, const e_float& u, const e_float& x);
e_float legendre_q(const e_float& v, const INT32    m, const e_float& x);
e_float legendre_q(const INT32    n, const e_float& u, const e_float& x);
e_float legendre_q(const INT32    n, const INT32    m, const e_float& x);
```

Effects: These functions compute the associated Legendre functions of the second kind of type 1 for their respective arguments `v`, `u`, `n`, `m`, `x`, with `v`, `u`, `x` $\in \mathbb{R}$; `n`, `m` $\in \mathbb{N}$.

Returns: The `assoc_legendre_q` function of non-integer degree and non-integer order returns [35]

$$\begin{aligned} Q_\nu^\mu(x) \;&=\; \frac{\pi\,\csc(\mu\pi)}{2}\left\{\frac{\cos(\mu\pi)}{\Gamma(1-\mu)}\left(\frac{1+x}{1-x}\right)^{\mu/2}\sum_{k=0}^{\infty}\frac{(-\nu)_k(\nu+1)_k}{(1-\mu)_k\;k!}\left(\frac{1-x}{2}\right)^k\right. \\ &\left. -\;\frac{(\nu-\mu+1)_{2\mu}}{\Gamma(1+\mu)}\left(\frac{1-x}{1+x}\right)^{\mu/2}\sum_{k=0}^{\infty}\frac{(-\nu)_k(\nu+1)_k}{(\mu+1)_k\;k!}\left(\frac{1-x}{2}\right)^k\right\}. \end{aligned} \tag{26}$$

Remark: There are specialized series representations for Legendre functions of the second kind with mixed integer, non-integer combinations of degree and order.

5.2.9 Laguerre, Parabolic Cylinder and Hermite Functions

```
e_float laguerre(const e_float& v, const e_float& x);
e_float laguerre(const e_float& v, const e_float& L, const e_float& x);
e_float laguerre(const INT32 n,    const e_float& L, const e_float& x);
e_float laguerre(const INT32 n,    const INT32 m,    const e_float& x);
```

Effects: These functions compute the associated Laguerre functions for their respective arguments `v`, `m`, `L`, `n`, `x`, with `v`, `L`, `x` $\in \mathbb{R}$; `n`, `m` $\in \mathbb{N}$.

Returns: The `laguerre` function of non-integer degree and non-integer order returns [35]

$$L_\nu^\lambda(x) = \frac{\Gamma(\nu+\lambda+1)}{\Gamma(\nu+1)}\sum_{k=0}^{\infty}\frac{(-\nu)_k\,x^k}{\Gamma(k+\lambda+1)k!}. \tag{27}$$

Returns: The `assoc_laguerre` functions of pure integer order and pure integer degree return [32]

$$L_n^m(x) = (-1)^m\frac{d^m}{dx^m}L_{n+m}(x), \tag{28}$$

with

$$L_n(x) = \frac{e^x}{n!}\,\frac{d^n}{dx^n}(x^n e^{-x}). \tag{29}$$

```
e_float weber_d(const e_float& v, const e_float& x);
e_float weber_d(const INT32& n,   const e_float& x);
```

Effects: These functions compute the Weber parabolic cylinder functions for their respective arguments v, n, x, with v, x $\in \mathbb{R}$; n $\in \mathbb{N}$.

Returns: The weber_d function returns [35]

$$\begin{aligned} D_\nu(x) \quad &= \quad 2^{\nu/2}\sqrt{\pi}e^{-x^2/4}\left\{ \frac{1}{\Gamma\left(\dfrac{1-\nu}{2}\right)} \sum_{k=0}^{\infty} \frac{\left(-\dfrac{\nu}{2}\right)_k}{\left(\dfrac{1}{2}\right)_k k!}\left(\frac{x^2}{2}\right)^k \right. \\ &\qquad \left. - \frac{\sqrt{2}\,x}{\Gamma\left(-\dfrac{\nu}{2}\right)} \sum_{k=0}^{\infty} \frac{\left(\dfrac{1-\nu}{2}\right)_k}{\left(\dfrac{3}{2}\right)_k k!}\left(\frac{x^2}{2}\right)^k \right\}. \end{aligned} \tag{30}$$

```
e_float hermite(const e_float& v, const e_float& x);
```

Effects: This function computes the associated Hermite function for its respective arguments v, x, with v, x $\in \mathbb{R}$.

Returns: The hermite function returns [2]

$$H_\nu(x) = 2^{\nu/2}e^{x^2/4}D_\nu(\sqrt{2}\,x) \tag{31}$$

5.2.10 Elliptic Integrals

```
e_float comp_ellint_1(const e_float& m);
e_float comp_ellint_2(const e_float& m);
e_float      ellint_1(const e_float& m, const e_float& phi);
e_float      ellint_2(const e_float& m, const e_float& phi);
```

Effects: These functions compute the complete elliptic integrals and the incomplete elliptic integrals of the first and second kinds for their respective arguments phi, m, with phi, m $\in \mathbb{R}$ and $|m| \leq 1$.

Returns: The comp_ellint_1 function returns [35]

$$K(m) = F(m, \pi/2) \quad . \tag{32}$$

Returns: The comp_ellint_2 function returns [35]

$$E(m) = E(m, \pi/2) \quad . \tag{33}$$

Returns: The `ellint_1` function returns [35]

$$F(m,\phi)=\int_0^{\phi}\frac{d\theta}{\sqrt{1-m\sin^2\theta}}\quad . \tag{34}$$

Returns: The `ellint_2` function returns [35]

$$E(m,\phi)=\int_0^{\phi}\sqrt{1-m\sin^2\theta}\,d\theta\quad . \tag{35}$$

Remark: There are several popular parameter conventions in use. The naming convention with $|m| = k^2$ is used. The parameter naming convention and the parameter order for the incomplete elliptic integrals in e_float are identical to those used in the C++ special functions draft [18].

5.2.11 Zeta Functions

```
e_float    riemann_zeta(const INT32 n);
e_float    riemann_zeta(const e_float& s);
ef_complex riemann_zeta(const ef_complex& s);
e_float    hurwitz_zeta(const e_float& s,    const INT32 n);
ef_complex hurwitz_zeta(const ef_complex& s, const INT32 n);
e_float    hurwitz_zeta(const e_float& s,    const e_float& a);
ef_complex hurwitz_zeta(const ef_complex& s, const ef_complex& a);
```

Effects: These functions compute the zeta functions for their respective arguments `s`, `a`, `n`, with `s`, `a` $\in \mathbb{Z}$ and `n` $\in \mathbb{N}^+$.

Returns: The `riemann_zeta` function returns [35]

$$\zeta(s)=\sum_{k=0}^{\infty}\frac{1}{k^s}\quad . \tag{36}$$

Returns: The `hurwitz_zeta` function returns [35]

$$\zeta(s,a)=\sum_{k=0}^{\infty}\frac{1}{\left[(a+k)^2\right]^{s/2}}\quad . \tag{37}$$

Remark: The `hurwitz_zeta` function can be inaccurate for negative values of `s` with $\mathfrak{Re}(\mathtt{s}) < -10$.

5.2.12 Polylogarithms

```
e_float poly_logarithm(const INT32 n, const e_float& x);
```

Effects: This function computes the polylogarithm function for ist respective arguments `x`, `n`, with `x` $\in \mathbb{R}$; and `n` $\in \mathbb{N}^+$.

Returns: The `poly_logarithm` function returns [35]

$$\mathrm{Li}_n(x)=\sum_{k=1}^{\infty}\frac{x^k}{k^n}\quad . \tag{38}$$

6 Testing e_float

6.1 The Test System

Testing is essential for establishing the reliability of a system with such high complexity and exactness. Consequently, a great deal of effort has been invested in testing and verification of e_float.

Testing has been performed using several test methods. During the development and verification stages of e_float, there have been countless spot–checks, Wronskian analyses, comparisons among the different `e_float` implementations, as well as comparisons with Mathematica® and MPFR. At times, some algorithms have been verified by temporarily deactivating them via source code modification, subsequently forcing a different computation method such as recursion to be used instead. For example, asymptotic Bessel function calculations have been verified in part by forcing stable recursion over many orders to be used instead of asymptotic expansion and verifying that the results computed from both methods agree to full precision.

A large–scale dedicated automatic test system has been developed to provide for reproducible testing of e_float in combination with detailed performance and code coverage analyses. The automatic test system consists of three parts — the automatic test case generator, the test suite and the test execution system (see Figure 1).

The test case generator creates test cases in the form of source code which are subsequently manually added to the test suite, and then compiled and executed by the test execution system. Every test case is designed to test one or several functions, algorithms or convergence zones. In order to improve function and block coverage in testing, the design of the test cases has been partially guided by code coverage analyses using Intel®'s "codecov" tool. Each test case contains a short, automatically generated C++ code sequence, usually involving a loop calculation, which calculates test values and compares these directly with control values, themselves pre–computed to 400 digits with Mathematica® and directly written in the test case as strings.

The test suite contains about 250 real–numbered test cases and about 30 complex–numbered test cases. Each test case computes $\sim$1–100 individual numerical values, resulting in a total of $\sim$10,000 test results. In addition to automatic verification, the test results are written to log files upon test execution. A test case passes its execution if there is full agreement between each individual numerical result and its corresponding control value, up to and including the very last digit of precision given by the precision setting. If any single digit of any single result does not agree with that of its control value, then the whole test case fails.

All test cases in the entire test suite pass fully for each one of the main three `e_float` classes `efx::e_float`, `gmp::e_float` and `mpfr::e_float` at 30, 50, 100, 200 and 300 digits of precision, using each compiler system listed in Table 3. The afore–mentioned test result is a very significant technical result. All testing evidence indicates that e_float correctly calculates all of the function values for the parameter ranges listed in Table 4 to full precision ranging from 30 to 300 digits using any one of the main three `e_float` classes `efx::e_float`, `gmp::e_float` or `mpfr::e_float`.

Certainly, a great deal of effort has gone into verification of e_float. Nonetheless, some function values and parameter ranges remain poorly tested. The author was not able to find independent control data — or even found conflicting results — for some parameter regions. Several non–confirmed values which have been computed with e_float are shown below.

$$P_{\pi+20{,}000}^{\gamma+10{,}000}\left(\tfrac{2}{3}\right) \approx \texttt{-5.980883901705929125018083995261080589601569670160466225963954055923614936288067418735478313503042667}\times 10^{42814}$$

$Q^{-2099}_{2347}(\gamma) \approx$ 5.134474212409628343860552869233848621335039670532952891598215234544503149892985668770122300019211348$\times 10^{-6860}$

$J_{\pi+10^8}(G \cdot 10^8) \approx$ 0.00007101587864721510600609002545136459450330588284140175201908135062824517495900855118876479980481012041

$K_{690}(\pi + 310) \approx$ 2.473093134580761699282091225752444991289365302904873098822780430687162258158192556310616234787924259$\times 10^{128}$

$D_{\frac{1201}{12}}(40) \approx$ 1.524674031836673520032956906376264551740269827916850281507352611567663272230385469253930112335260440$\times 10^{-15}$

6.2 Extending the Test System

The test system can be extended by adding a dedicated test case.

- Each individual test case must be implemented in its own individual automatically generated source file.
- The test case sources are generated with a standalone project called “TestCaseGenerator” located in the directory `__TestCaseGenerator`.
- Test case generation is only supported for Windows® platforms.
- Design the parameters of the test case and add the relevant source code the the test case generator project. Use the many examples of this in the source code in order to add the new case. Note that it takes a long time to generate all of the test case sources. Therefore, a judicious use of commenting can be used to comment–out the test cases which do not need to be generated.
- When the desired new test case file has been generated, add it to the e_float project in the appropriate directory and add it to the solution either in Developer Studio or the the relevant file list for GNUmake.
- The new test case can now be compiled and the call of the test case function can be added to the list of test cases, for example in `test_real.cpp` or `test_imag.cpp`.

References

[1] David Abrahams. *boost.python Index*. boost, `http://www.boost.org/doc/libs/1_42_0/libs/python/`, 2008.

[2] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions, 9th Printing*. Dover Publications, New York, 1972.

[3] Pete Becker. *The C++ Standard Library Extensions: A Tutorial and Reference*. Addison Wesley, Reading Massachusetts, 2006.

[4] boost. *Boost Software License Version 1.0*. Boost, `http://www.boost.org/users/license/`, 2003.

[5] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison Wesley, Reading Massachusetts, 1992.

[6] Michael J. Foord and Christian Muirhead. *IronPython in Action*. Manning Publications, Greenwich, CT, 2009.

[7] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple–precision binary floating–point library with correct rounding. *ACM Trans. Math. Soft.*, 33(2):1–15, 2007.

[8] GCC. *The GNU Compiler Collection Version 4.3.3*. Free Software Foundation, `http://gcc.gnu.org/`, 2008.

[9] Brian Gladman. *Building GMP and MPFR with Microsoft Visual Studio 2008 and YASM*. Brian Gladman, `http://gladman.plushost.co.uk/oldsite/computing/gmp4win.php`, 2008.

[10] GMP. *GNU Multiple Precision Arithmetic Library Version 4.2.4*. Free Software Foundation, `http://gmplib.org/`, 2008.

[11] GNU. *GNU General Public License*. Free Software Foundation, `http://www.gnu.org/copyleft/gpl.html`, 2007.

[12] GNUmake. *GNUmake Version 3.81*. Free Software Foundation, `http://www.gnu.org/software/make/`, 2006.

[13] Intel. *Intel® C++ Compiler Professional Edition*. Intel, `http://software.intel.com/en-us/intel-compilers/`, 2008.

[14] ISO. *ISO/IEC 14882:2003 - Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, 2003.

[15] ISO. *ISO/IEC 19768:2005 C++ Library Extensions TR1*. International Organization for Standardization, Geneva, Switzerland, 2005. (Draft).

[16] ISO. *ISO/IEC 23270:2006 : Information Technology Programming Languages – C#*. International Organization for Standardization, Geneva, Switzerland, 2006.

[17] ISO. *ISO/IEC 23271:2006 : Information Technology Programming Languages – Common Language Infrastructure (CLI) Partitions I to VI*. International Organization for Standardization, Geneva, Switzerland, 2006.

[18] ISO. *ISO/IEC 29123:draft Extensions to the C++ Library to Support Mathematical Special Functions*. International Organization for Standardization, Geneva, Switzerland, 2009. (Draft).

[19] E. Jahnke and F. Emden. *Tables of Functions with Formulae and Curves*. Dover, New York, NY, 4 edition, 1945.

[20] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley, Reading Massachusetts, 1999.

[21] Yudell L. Luke. *Algorithms for the Computation of Mathematical Functions*. Academic Press, New York, NY, 1977.

[22] Microsoft. *Microsoft® Visual Studio Professional 2008 with Service Pack 1*. Microsoft, `http://www.microsoft.com/visualstudio/`, 2008.

[23] Microsoft. *IronPython*. Microsoft Corporation, `http://www.ironpython.net/`, 2009.

[24] Microsoft. *NMAKE Reference*. Microsoft, `http://msdn.microsoft.com/en-us/library/dd9y37ha(VS.71).aspx`, 2009.

[25] Microsoft. *Common Language Runtime Overview*. Microsoft, `http://msdn.microsoft.com/en-us/library/ddk909ch.aspx`, 2010.

[26] MISRA. *MISRA–C 2004: Guidelines for the Use of the C Language in Critical Systems*. MISRA Consortium, `http://www.misra.org.uk/`, 2004.

[27] MISRA. *MISRA–C++ 2008: Guidelines for the Use of the C++ Language in Critical Systems*. MISRA Consortium, `http://www.misra-cpp.org/`, 2008.

[28] MPMATH. *Python library for arbitrary–precision floating–point arithmetic*. mpmath Project, `http://code.google.com/p/mpmath/`, 2009.

[29] PSF. *Python Programming Language — Official Website*. Python Software Foundation, `http://www.python.org/`, 2009.

[30] Jeffrey Richter. *CLR Via C#*. Microsoft Press, Redmond, WA, 2 edition, 2006.

[31] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison Wesley, Boston, 2003.

[32] Eric Weisstein. *Wolfram Mathworld® Website*. Wolfram Research, `http://mathworld.wolfram.com/`, 2009.

[33] Wikipedia. *Wikipedia — The Free Encyclopedia*. Wikipedia, `http://en.wikipedia.org/wiki/`, 2009.

[34] Stephen Wolfram. *The Mathematica® Book*. Cambridge University Press, Cambridge, UK, 4 edition, 1999.

[35] Wolfram Research. *The Wolfram Functions Site*. Wolfram Research, `http://functions.wolfram.com/`, 2009.

[36] Daoqi Yang. *C++ and Object Oriented Numeric Computing for Scientists and Engineers*. Springer, New York, 2001.