Algorithm 910: A Portable C++ Multiple-Precision System for Special-Function Calculations

CHRISTOPHER KORMANYOS

This article presents a portable C++ system for multiple precision calculations of special functions called *e_float*. It has an extendable architecture with a uniform C++ layer which can be used with any suitably prepared MP type. The system implements many high-precision special functions and extends some of these to very large parameter ranges. It supports calculations with $30 \cdots 300$ decimal digits of precision. Interoperabilities with Microsoft's® CLR, Python, and Mathematica® are supported. The e_float system and its usage are described in detail. Implementation notes, testing results, and performance measurements are provided.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: *Object-oriented programming*; G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic; multiple precision arithmetic*; G.1.2 [**Numerical Analysis**]: Approximation—*Special function approximations; elementary function approximation*; G.4 [**Mathematical Software**]: *Algorithm design and analysis; efficiency; portability; reliability and robustness*

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Airy, Bessel, C++, elliptic integrals, gamma, generic programming, Hermite, hypergeometric, Laguerre, Legendre, multiple precision, object oriented, orthogonal polynomials, parabolic cylinder, polylogarithm, polymorphism, special functions, zeta

ACM Reference Format:
Kormanyos, C. 2011. Algorithm 910: A portable C++ multiple-precision system for special-function calculations. ACM Trans. Math. Softw. 37, 4, Article 45 (February 2011), 27 pages.
DOI = 10.1145/1916461.1916469 http://doi.acm.org/10.1145/1916461.1916469

1. INTRODUCTION

There are many multiple-precision (MP) packages available. 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. In addition, 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 and 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

Author's address: C. Kormanyos; email: `e_float@yahoo.com`.

DOI 10.1145/1916461.1916469 http://doi.acm.org/10.1145/1916461.1916469

Table I. Functions Supported by e_float

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	$\in \mathbb{Q}$	
$\sqrt{2}, \pi, e, \log(2), \gamma,$ *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	$x \in \mathbb{R}; \nu \in \mathbb{R}$	$\lvert x\rvert, \lvert\nu\rvert \lesssim 10^9$
$T_n(x), U_n(x), L_n(x), H_n(x)$	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 $\nu, \mu \ni \mathbb{N}$, 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 $x < 0$

MP system other than e_float currently offers such a high degree of portability, such a wide range of functions, and such a rich selection of powerful interoperabilities.

The functions supported by e_float including parameter ranges and known limitations are listed in Table I. 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

Table II. System Configurations of e_float

Config.	`e_float` Class	Capabilities	Dependencies
efx	`efx::e_float`	Functions, algorithm design, test, and benchmark	—
gmp	`gmp::e_float`	Functions, algorithm design, test, and benchmark	GMP library
mpfr	`mpfr::e_float`	Functions, algorithm design, test, and benchmark	GMP library, MPFR library
f90	`mpfr::e_float`	Functions, algorithm design, test, and benchmark	`REAL(KIND=16)` wrap, Fortran runtime 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

complex numbers. The special functions are implemented primarily for real parameters, but some also accept complex parameters.

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 ISO [2006b]; Microsoft [2009a, 2010]; PSF [2009]; Foord and Muirhead [2009]; Wolfram [1999]). 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 [ISO 2006a], 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 the effective distribution of computational complexity.

The e_float system supports a variety of configurations using several `e_float` classes (see Section 2) as well as other libraries and systems. These are shown in Table II. Details about the capabilities and dependencies of the configurations are also included in the table.

2. THE E_FLOAT SYSTEM

2.1 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 nonportable 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 is defined within its own unique

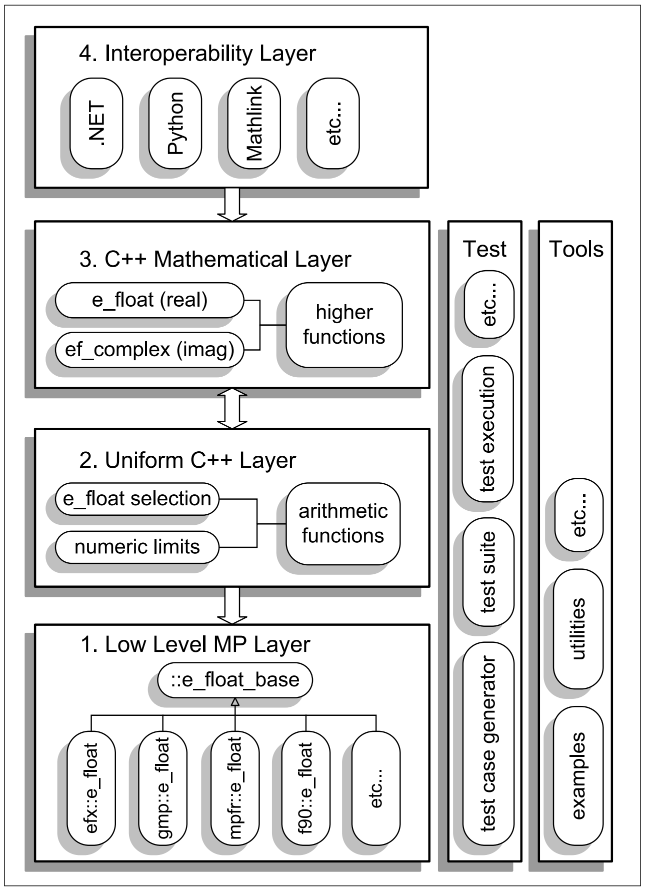

Fig. 1. The e_float system architecture.

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 several public functions and about 30 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 [Fousse et al. 2007] (see Section 2.2) 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` (see Section 4.2).

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`, that is, 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 Richter [2006] are used to create managed C++ classes and wrapper functions which embody the full functionality of layers 2 and 3. These are compiled into a single CLR assembly which can be used with all Microsoft® CLR languages including C#, managed C++/CLI, Iron-Python, etc. Compatibility with the Microsoft®.NET Framework 3.5 has been tested. A second interoperability employs the `boost.python` library [Abrahams 2008] 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 the MathLink® facility of 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.

The test block (see Section 4.1) 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 (see Section 2.4) show practical, nontrivial 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 [GMP 2008] (see Section 2.2) 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.

Table III. MP Classes in e_float

Class name	Digits (+padding)	Max./min.
`efx::e_float`	30–300 (+15%)	$\sim 10^{\pm 3.063 \ldots 10^{18}}$
`gmp::e_float`	30–300 (+15%)	$\sim 10^{\pm 6.646 \ldots 10^{8}}$
`mpfr::e_float`	30–300 (+15%)	$\sim 10^{\pm 3.232 \ldots 10^{8}}$
`f90::e_float`	30 (+4–5 digits)	$\sim 10^{\pm 4{,}930}$

2.2 MP Types

There are several MP implementations available to the scientific community. There are classic Fortran 77 and Fortran 90 implementations [Bailey 1993, 1995; Brent 1978; Smith 1991, 1998]. These packages are built upon the strength of procedural Fortran programming. There are newer MP implementations written in C and C++, such as GMP [GMP 2008], MPFR [Fousse et al. 2007], ARPREC [Bailey et al. 2002], and NTL [Shoup 2008]. These packages use somewhat portable C or C++ programming techniques in varying degrees of quality.

Four MP types have been selected for e_float. These are listed in Table III. The table also includes information about their digit ranges and approximate minimum and maximum values. The MP type and the number of decimal digits are fixed at compile time. The precision of an `e_float` object can be dynamically changed during runtime for intermediate calculation steps, but never increased to more than the fixed number of digits. 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.

GMP 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 [2008] port.

A new MP type called *EFX* (*for 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 (see Section 4.2). 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 humanly 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 range of about ± 4000. Due to its limited range, `f90::e_float` only passes about 75% of the test cases shown in Section 4.1. Nonetheless, if this numeric range is sufficient for the application, then `f90::e_float` is extremely fast because it uses a native data type.

e_float supports several MP types—and it can be extended to support others. In general, using a given MP type with e_float does not require detailed understanding of the underlying implementation. This is because users of e_float achieve a high degree

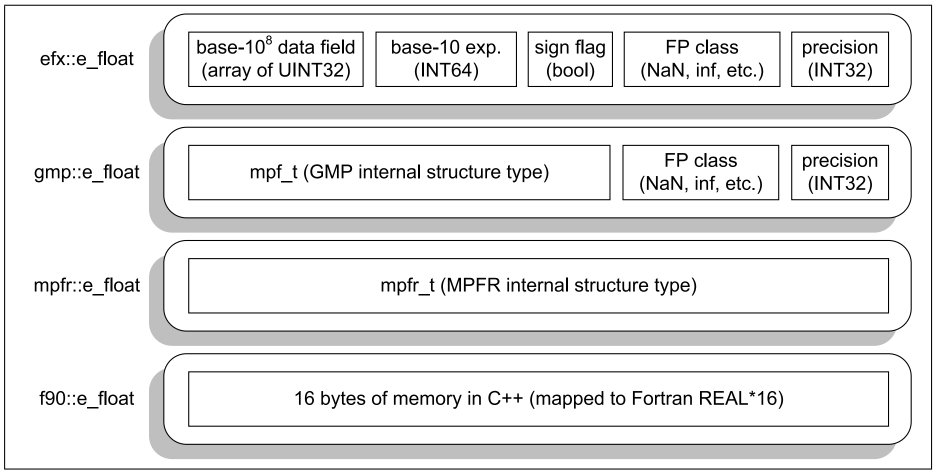
efx::e_float
base-10^8 data field (array of UINT32)
base-10 exp. (INT64)
sign flag (bool)
FP class (NaN, inf, etc.)
precision (INT32)
gmp::e_float
mpf_t (GMP internal structure type)
FP class (NaN, inf, etc.)
precision (INT32)
mpfr::e_float
mpfr_t (MPFR internal structure type)
f90::e_float
16 bytes of memory in C++ (mapped to Fortran REAL*16)

Fig. 2. Internal representations and data structures of the different MP types.

of independence from the MP implementation by employing e_float's uniform interface for high-level MP algorithm design (see Section 2.1).

Nonetheless, there are slight differences among the `e_float` implementations such as internal memory management or maximum and minimum range. Some types of computations, such as memory-intensive matrix calculations or those stressing extreme range, need this kind of information. In addition, system designers who want to extend e_float by adding support for another MP implementation need to understand the details of the internal MP representation in order to effectively adapt it to e_float. For example, `efx::e_float` is a low-level MP implementation built from scratch. All of its components such as data-field, sign, and floating-point class (like NaN or inf) are built from native C++ data types. On the other hand, `mpfr::e_float` does not need to implement these low-level primitives because it is based on MPFR's much higher-level data type `::mpfr_t` which already includes them.

The internal representations of the MP types used in e_float are shown in Figure 2. Class objects of type `efx::e_float` are represented internally with a non-dynamic `UINT32` data array, each element of which contains one base 10^8 word of the `e_float`'s value. The data array has enough elements to hold the digits plus the padding, given by `[(digits+padding)/8]+1`; see Table III. In addition, the `efx::e_float` class has an `INT64` base 10 exponent, a Boolean sign flag, an enumeration field representing the floating–point class, and an additional `INT32` value for the precision. Class objects of type `gmp::e_float` are represented internally with an instance of GMP's data structure `::mpf_t` [GMP 2008]. They also have the floating-point class field and an additional `INT32` value for the precision. Class objects of type `mpfr::e_float` are represented internally with an instance of MPFR's data structure `::mpfr_t` Fousse et al. [2007]. Finally, class objects of type `f90::e_float` are represented internally with 16 bytes of non-dynamic memory, mapped to a Fortran `REAL(KIND=16)`.

The data arrays of both `efx::e_float` objects as well as `f90::e_float` objects are created on the stack, without using dynamic memory allocation, whereby both `gmp::e_float` objects as well as `mpfr::e_float` objects, which actually are based on the same low-level GMP data structure, `::mpf_t`, and the memory management thereof do use dynamic memory allocation (see Fousse et al. [2007] for further details). However, allocation and corresponding deallocation for a given instance of a `gmp::e_float` object or a `mpfr::e_float` object are performed one single time for the entire lifetime of the object—in the class creator and destructor. This reduces the impact of memory fragmentation.

2.3 Using the System

Every effort has been made to ensure that building, using and adapting the e_float system is intuitive and straightforward. The default build supports execution of the test suite (see Section 4.1). 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 in the file `test/test.cpp`. Designers of advanced applications can remove the test suite entirely and create a custom build. As briefly discussed previously in Section 2.2, e_float supports several MP types. However, the system uses only one MP type at a time. The MP type used in a given build instance is defined by the value of the `MP` command line option for UNIX/Linux–GNU builds or by appropriate project selection in Microsoft® Developer Studio® 2008 for Windows® builds. Listing 1 below shows a UNIX/Linux–GNU build command line using `mpfr::e_float`.

Listing 1. This `bash` session starts the build and sets the make option `MP` to `mpfr`. If `MP` is left blank, its default is `MP=efx`.

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

Complete compatibility with the usual C++ semantics for real numbered arithmetic is available. All single-argument `e_float` class constructors are declared with the `explicit` qualifier. This prevents unwanted side effects such as automatic compile-time conversion between plain old data (POD) types and `e_float` objects. Real-numbered functions are defined within the namespace `ef`. Complex-numbered functions are defined within the namespace `efz`. A specialization of the standard numeric limits class has been defined. Its name is `std::numeric_limits<e_float>`. Support for formatted string output is defined for `std::ostream` objects using the usual C++ semantics. The precision of the output stream must be appropriately set in order to display the full precision of an `e_float`. Listing 2 below shows function calculations, numeric limits, output precision and printing.

Listing 2. This C++ code calculates some real-valued function results and prints them to the standard output stream. The `std::numeric_limits` template is used for setting output stream precision.

```
void test::spot::test_spot(void)
{
  // Calculate some real 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);

  // ...set the output precision...
  std::cout.precision(std::numeric_limits<e_float>::digits10);

  // ...and print the values.
  std::cout << s << '\n' << g << '\n' << b << '\n';
}
```

Complex numbers (ef_complex) are compatible with the semantics of the template class std::complex<*typename* T> from the C++ STL. Complex numbers can be mixed with real numbers. Support for formatted string output using std::ostream objects is defined. Listing 3 below shows complex-numbered calculations combined with STL containers and algorithms.

Listing 3. This C++ code calculates some complex–numbered function values and stores them in a container, a std::deque. The values are printed with the std::copy algorithm in combination with an std::ostream_iterator.

```
void test::spot::test_spot(void)
{
  // Calculate complex values in an STL deque...
  std::deque<ef_complex> values;
  for(INT32 k = 0; k < 10; k++)
  {
    const ef_complex z(ef::third() + k, ef::catalan());
    values.push_back(efz::gamma(z));
  }

  // ...set the output precision...
  std::cout.precision(std::numeric_limits<e_float>::digits10);

  // ...and print the values using STL's copy algorithm.
  std::copy(values.begin(), values.end(),
            std::ostream_iterator<ef_complex>
            (std::cout,"\n"));
}
```

2.4 Examples

Additional practical applications of e_float are provided by seven handwritten example files. The example files are stored in examples/example∗.cpp, and they are part of the tools block (Section 2.1).

Example 1 shows real-numbered usage in combination with a timing measurement. It calculates 21 nontrivial values of $P^{\mu}_{\nu}(x)$, with $\nu, \mu, x \in \mathbb{R}$. Example 2 shows complex-numbered usage in combination with a timing measurement. It calculates 21 nontrivial 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, $\Lambda_\nu(x) = \left\{\Gamma(\nu+1)\, J_\nu(x)\right\} / \left(\frac{1}{2}x\right)^\nu$ [Jahnke and Emden 1945]. 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 (see Section 2.1). Example 4 performs a numerical differentiation, $\frac{\partial}{\partial\nu} J_\nu(151+\gamma)\big|_{(\nu=123+G)}$, where γ is Euler's constant and G 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. Luke [1977] 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 I 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 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 argument $x \in \mathbb{R}$ and positive integer order $n \in \mathbb{N}^{+}$.

2.5 Interoperability in Detail

2.5.1 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 [MPMATH 2009] can be achieved with Python strings. Listing 4 shows e_float's interoperability with Python.

Listing 4. This Python code loads the shared library and generates 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)
>>> q = ef.quarter()
>>> lst=[ef.cyl_bessel_j(ef.third(), k + q) for k in
    range(10)]
>>> for y in lst:  print y
```

2.5.2 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. Listing 5 illustrates the syntax of e_float in the C# language.

Listing 5. This C# code uses e_float's CLR export. Unlike traditional C++, CLR languages create `e_float` objects with `operator new` (or `operator gcnew` for managed C++/CLI), relying on so-called "garbage collection" for automatic `delete`.

```
using e_float_clr.cli;
namespace test
{
  class Program
  {
```

Table IV. Compilers and Build Systems Supported by e_float

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

```
      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());
      }
    }
  }
```

2.5.3 Computer Algebra Systems. Using a computer algebra system from within e_float, in particular extending e_float with Mathematica®, has been discussed in Example 7, Section 2.3. 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. This, as well as support for other computer algebra systems, are developments in progress for future publications.

3. IMPLEMENTATION DETAILS

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 [ISO 2003, 2007]. 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 [MISRA 2004, 2008]. A great effort has been invested in advanced C++ optimization techniques in order to improve system performance (see Section 4.2). Generic and object-oriented programming methods have been used to create the efficient and flexible numerical software architecture which was shown in Section 2.1. In addition, consistent use of standard containers and algorithms of the STL and TR1 [Josuttis 1999; Becker 2006] has significantly reduced and evenly distributed the computational complexities of the entire program.

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 IV have been used to develop, build, and test e_float. Tools from Microsoft®, Intel® and GNU are supported (see Microsoft [2008, 2009b]; Intel [2008]; GCC [2009]; GNU [2006]). 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 (see Section 4.1).

There are hundreds of carefully crafted algorithms implemented in e_float. Many of these are at least partly based on previous algorithms for conventional double-precision or quadruple-precision calculations in procedural languages such as Fortran or C. However, it has been necessary to redesign these algorithms for C++ and significantly extend them, at times using non-trivial programming techniques, for precision reaching hundreds of digits. Of the vast number of resources used to investigate these algorithms, some of the more important include Abramowitz and Stegun [1972], Amos [1986], Andrews et al. [2000], Arndt and Haenel [2000], Bailey [1993], Borwein [1995], Cody [1993], Cody and Waite [1980], Erdélyi et al. [1981], Espinosa and Moll [2004], Galassi et al. [2009], Gil et al. [2006; 2007], Gourdon and Sebah [2008], Knuth [1998], Luke [1977], Oldham et al. [2009], Olver [1997], Press et al. [2002], Temme [2007], Vepštas [2008], Watson [1995], Weisstein [2010], Wikipedia [2009], Wimp [1984], Wolfram [1999], Wolfram Research [2010], and Zhang and Jin [1996].

3.1 Basic Arithmetic

Real numbers, basic mathematical operations, and complex numbers are implemented as described in Section 2.1. Rational numbers such as 1, 2, $\frac{1}{3}$, or $\texttt{INT32}_{\texttt{max}}$ are set once and returned from interface functions as references to static constant `e_float` objects using a single instance of the number. This technique, known as a *singleton instance*, is safe and efficient. Mathematical constants Finch [2003] such as $\sqrt{2}$, π, e, and γ have been precomputed to 1100 digits. They are managed with the same functional semantics as the rational numbers, but created from STL strings.

3.2 Integer Functions

Prime numbers and prime factorization are supported only as utilities for other calculations such as the Riemann zeta function. The implementations use trivial sieves and divide-and-conquer and are not meant for high-performance use.

Integer functions such as factorial, Bernoulli numbers, and Euler numbers use precomputed tables containing 1001 nonzero entries. Asymptotic expansions are used for $n \gtrsim 10^3$. Stirling numbers of the second kind $S_n^{(m)}$ are only supported as a utility. $S_n^{(m)}$ uses precomputed values of the first 101 rows of the flattened triangle and slow recursion for higher indexes.

3.3 Elementary Functions

Elementary functions use Taylor series, argument scaling, recursion, and Newton iteration. The algorithms are similar to those in the articles describing previous MP packages.

3.4 Airy Functions

Airy functions use Taylor series for small arguments. Large positive arguments use divergent asymptotic hypergeometric expansions in the nonoscillatory region. Large negative arguments use divergent asymptotic hypergeometric expansions in the oscillatory region. Intermediate positive arguments use the representation in terms of Bessel functions $K_\nu(x)$ in the nonoscillatory region. Intermediate negative arguments use the representation in terms of Bessel functions $J_\nu(x)$ in the oscillatory region.

Airy functions are the first "higher transcendental" functions described in this manuscript. Therefore, their computations will be described in detail as an example for other similar calculations. Consider the sketch in Figure 3, which depicts the different convergence zones and types of calculations used for computing real-valued Airy

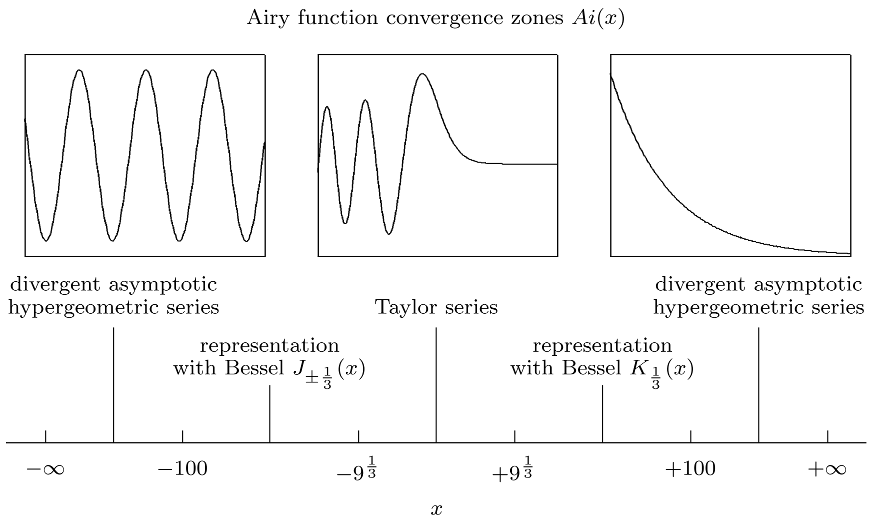
Airy function convergence zones $Ai(x)$
divergent asymptotic hypergeometric series
representation with Bessel $J_{\pm\frac{1}{3}}(x)$
Taylor series
representation with Bessel $K_{\frac{1}{3}}(x)$
divergent asymptotic hypergeometric series
$-\infty$
-100
$-9\frac{1}{3}$
$+9\frac{1}{3}$
$+100$
$+\infty$
x

Fig. 3. The five different convergence zones and types of calculations used for computing the real-valued Airy function $Ai(x)$ are depicted in this sketch. The symbols $\pm\infty$ represent the minimum and maximum supported ranges for Airy functions, as shown in Table I.

functions $Ai(x)$. Disregarding slight differences among expansions for positive and negative arguments, there are three convergence zones shown in Figure 3: small argument, large argument and intermediate argument. The algorithmic scheme used for handling calculations in these convergence zones is shown in Listing 6. This general scheme is also used for several other special function calculations—in particular for those which can be represented in terms of generalized hypergeometric functions such as Bessel and Legendre functions.

Listing 6. Sample code is shown for handling calculations in different convergence zones.

```
e_float ef::my_function(const e_float& x)
{
  const bool b_neg = ef::isneg(x);
  const e_float xx = ef::fabs(x);

  if(xx <= my_limit_small_arg)
  {
    return MyFunctionSeries::AtZero(x);
  }
  else if(xx <= my_limit_intermediate_arg)
  {
    return b_neg ?  MyFunctionSeries::AtTransitionMinus(xx)
                 :  MyFunctionSeries::AtTransitionPlus (xx);
  }
  else
  {
    return b_neg ?  MyFunctionSeries::AtInfinityMinus(xx)
                 :  MyFunctionSeries::AtInfinityPlus (xx);
  }
}
```

In order to complete this example, all five expansions for the computation of $Ai(x)$, which are used in both the five convergence zones of Figure 3 as well as the five paths of Listing 6, are shown in Equations (1)–(5).

$$Ai(-x) \approx \frac{1}{\sqrt{\pi}\sqrt[4]{x}}\left\{\sin\left(\frac{2}{3}x^{3/2}+\frac{\pi}{4}\right){}_4F_1\left(\frac{1}{12},\frac{5}{12},\frac{7}{12},\frac{11}{12};\frac{1}{2};-\frac{9}{4x^3}\right) - \right.$$
$$\left.\frac{5}{48x^{3/2}}\cos\left(\frac{2}{3}x^{3/2}+\frac{\pi}{4}\right){}_4F_1\left(\frac{7}{12},\frac{11}{12},\frac{13}{12},\frac{17}{12};\frac{3}{2};-\frac{9}{4x^3}\right)\right\}, \quad \text{if } x < -100. \tag{1}$$

$$Ai(-x) = \frac{1}{3}\sqrt{x}\left\{J_{\frac{1}{3}}\left(\frac{2}{3}x^{3/2}\right)+J_{-\frac{1}{3}}\left(\frac{2}{3}x^{3/2}\right)\right\}, \quad \text{if } -100 \le x < -9^{\frac{1}{3}}. \tag{2}$$

$$Ai(x) = \frac{1}{3^{2/3}\,\Gamma\left(\frac{2}{3}\right)}\,{}_0F_1\left(;\frac{2}{3};\frac{x^3}{9}\right) - \frac{x}{\sqrt[3]{3}\,\Gamma\left(\frac{1}{3}\right)}\,{}_0F_1\left(;\frac{4}{3};\frac{x^3}{9}\right), \quad \text{if } -9^{\frac{1}{3}} \le x \le 9^{\frac{1}{3}}. \tag{3}$$

$$Ai(x) = \frac{1}{\pi}\sqrt{\frac{x}{3}}\,K_{\frac{1}{3}}\left(\frac{2}{3}x^{3/2}\right), \quad \text{if } -9^{\frac{1}{3}} < x \le 100. \tag{4}$$

$$Ai(x) \approx \frac{1}{2\sqrt{\pi}\sqrt[4]{z}}\,e^{-\frac{2}{3}x^{3/2}}\,{}_2F_0\left(\frac{1}{6},\frac{5}{6};;-\frac{3}{4z^{3/2}}\right), \quad \text{if } x > 100. \tag{5}$$

Inserting implementations of Equations (1)–(5) into Listing 6 results in an algorithm for $Ai(x)$ which is essentially identical to the one implemented in e_float. $Ai(x)$, as well as the remaining Airy functions have been calculated according to this scheme. Equations (1)–(5) as well as the other relevant expansions for $Bi(x)$, $Ai'(x)$, and $Bi'(x)$ have been taken from subsections of Wolfram Research [2010] "Bessel-Type Functions—Airy Functions—AiryAi[z], etc."

Zeros of $Ai(x)$ and $Bi(x)$ on the negative real axis use Newton iteration. The well-known quadratic convergence of this iteration ensures high performance for these calculations. Two dedicated classes, one each for zeros of $Ai(x)$ and $Bi(x)$, have been implemented for performing Newton iteration. These classes are derived from e_float's generic Newton iteration template utility, `Util::FindRootNewton-Raphson<`*typename* `T>`, with `T=e_float`. Initial estimates of the zeros come from conventional double-precision algorithms in Abramowitz and Stegun [1972], in which Table 10.13 on page 478 lists the first 10 zeros of $Ai(x)$ and $Bi(x)$ with better than six-digit precision and zeros with index higher than 10 have been estimated using Equations 10.4.94, 10.4.98, and 10.4.105, all on page 450. The test suite (see Section 4.1) only exercises the zeros of $Ai(x)$ and $Bi(x)$ up to the $\sim$ 20th zero. However, additional detailed tests have verified e_float's calculations of the zeros of $Ai(x)$ and $Bi(x)$ to full precision up to the $\sim$ 100,000th zero.

3.5 Bessel Functions

Fractional Bessel functions are computed on the positive real axis. Integer Bessel functions are computed on the real axis. Small arguments use Taylor series. Large positive arguments use divergent asymptotic hypergeometric expansions. Intermediate positive arguments use recursion and scaling with sums. There are various convergence zones and types of calculations used for computing Bessel functions. Unlike the convergence zones for real-valued Airy functions which depend only on the argument x, the convergence zones for real-valued Bessel functions depend on both the argument x as well as the order ν and they must be characterized in two-dimensional x-ν space. For example, the convergence zones used for computations of $J_\nu(x)$ are sketched in Figure 4. The other kinds of Bessel functions have similar convergence zones.

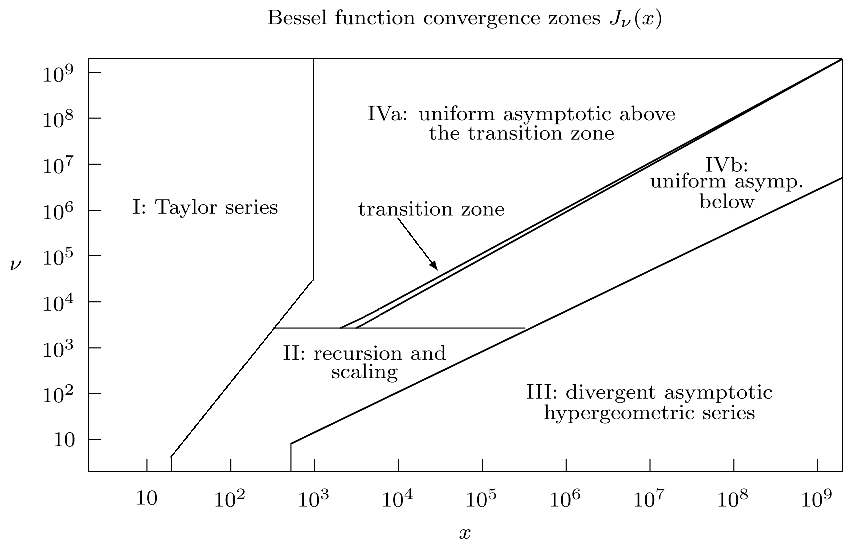
Bessel function convergence zones $J_\nu(x)$
I: Taylor series
IVa: uniform asymptotic above the transition zone
transition zone
IVb: uniform asymp. below
II: recursion and scaling
III: divergent asymptotic hypergeometric series
ν
x
10
10^2
10^3
10^4
10^5
10^6
10^7
10^8
10^9

Fig. 4. The various convergence zones and types of calculations used for computing the cylindrical Bessel function $J_\nu(x)$ are depicted in this sketch.

In particular, $J_\nu(x)$, $Y_\nu(x)$, $I_\nu(x)$, and $K_\nu(x)$ are calculated in the small-argument region with Taylor series, conveniently represented in terms of generalized hypergeometric functions taken from subsections of Wolfram Research [2010] "Bessel-Type Functions—Bessel Functions—BesselJ[ν, z], etc." These are also shown in Equations 7.2.1(2), 7.2.1(4), 7.2.2(12), and 7.2.2(13) on pages 4–5 of Erdélyi et al. [1981], Volume II. The divergent large-argument series for $J_\nu(x)$, $Y_\nu(x)$, $I_\nu(x)$, and $K_\nu(x)$ can also be represented in terms of generalized hypergeometric functions. These have also been taken from Wolfram Research [2010].

Calculations for intermediate arguments are more complicated. The direction of stable recursions in this parameter region is downward for $I_\nu(x)$ and $J_\nu(x)$, but upward for $K_\nu(x)$ and $Y_\nu(x)$. Bessel functions $I_\nu(x)$ and $J_\nu(x)$ with intermediate arguments are calculated with downward recursion beginning with a suitably high order, such as $\nu_{\texttt{frac}} + N_{\texttt{offset}}$ with $N_{\texttt{offset}} \in \mathbb{N}^+$. Scaling is done according to Equation 11.5(3) with $\phi = 0$ for $I_\nu(x)$ and Equation 11.5(7) for $J_\nu(x)$, taken from page 369 of Watson [1995]. A reusable and extendable class hierarchy is used to simplify the non-linear equation solving techniques for calculating of the starting orders for downward recursions. $Y_\nu(x)$ uses a Miller algorithm for intermediate arguments, which is described in Equations 5.48–5.53 on page 78 of Wimp [1984]. $K_\nu(x)$ uses another Miller-like algorithm for intermediate arguments, which is described in Section III of Temme [1975].

Uniform Airy-type asymptotic expansions are used for efficient and fast calculations of Bessel functions $J_\nu(x)$ and $Y_\nu(x)$ with very large order, $|\nu| \gtrsim 2{,}000 - 10{,}000$. $J_\nu(x)$ and $Y_\nu(x)$ use Equations 9.3.35–9.3.42, excluding 9.3.37, all on page 368 of Abramowitz and Stegun [1972]. The so-called *transition region* (see Figure 4), in which the order is near the argument, causes a singularity in Equations 9.3.35 and 9.3.36. The singularity in the transition region has been avoided by using stable downward recursion for $J_\nu(x)$ or stable upward recursion for $Y_\nu(x)$, starting with two function values which are far enough away from the transition region to maintain precision. The number of orders required to avoid the transition region varies as a function of ν and has been empirically characterized up to $\nu \sim 10^9$—ranging from a few hundred orders near $\nu \sim 10^4$ to about 30,000 orders near $\nu \sim 10^9$. $I_\nu(x)$ and $K_\nu(x)$ use the Debye-type uniform asymptotic expansions given in Equations 9.7.7 and 9.7.8 on page 378 of Abramowitz and Stegun [1972]. These converge less quickly than the uniform Airy-type expansions, so they are used with slightly higher order $|\nu| \gtrsim$ 10,000–50,000, for which the convergence properties are sufficient for maintaining full precision.

Dynamic polymorphism and STL containers have been used to greatly reduce the program complexity of the uniform asymptotic expansions. The coefficients and denominators of the polynomials used for these expansions have been recently added by the author to Sloane's [2009] *Encyclopedia*. The new sequences A144617, A144618, and A144622 contain coefficients and denominators of the first 121 polynomials.

The zeros of $J_\nu(x)$ and $J_n(x)$ use Newton iteration. As described for the calculations of the zeros of Airy functions, the calculations of the zeros of $J_\nu(x)$ and $J_n(x)$ also use classes derived from e_float's generic Newton iteration mechanism. Initial estimates of the zeros come from conventional double-precision algorithms described on page 371 of Abramowitz and Stegun [1972]. For small $\nu \lesssim 1.2$, an estimate of the first zero uses interpolation from tabulated values, themselves computed with Mathematica® to five digits in steps of $\delta\nu = 0.2$. Subsequent zeros are obtained from Equation 9.5.12. For $\nu \gtrsim 1.2$, an estimate of the first zero is obtained from Equation 9.5.14, with estimates of subsequent zeros calculated from Equation 9.5.22 (with $k = 1$) and Equation 9.5.26. The very terse text following Equation 9.5.25 describes the nonlinear inversion of z as a function of ζ required by Equations 9.5.22 and 9.5.26. For e_float, this has been accomplished by performing an order 2 Taylor expansion of Equation 9.3.39 for large z and solving the resulting quadratic equation for z, taking the positive root.

3.6 Orthogonal Polynomials

Chebyshev, Laguerre, and Hermite polynomials are calculated with recursion. Upward recursion has been implemented for $T_n(x)$, $U_n(x)$, $L_n(x)$, and $H_n(x)$ with $x \in \mathbb{R}, x \in \mathbb{Z}$ using an optimized C++ template subroutine which has been adapted from the Fortran subroutine `OTHPL` on page 24 of Zhang and Jin [1996]. See section 3.10 of this article for generalized Hermite functions $H_\nu(x)$ and generalized Laguerre functions $L_\nu^\lambda(x)$. Legendre polynomials $P_n(x)$ and $Q_n(x)$ are predominantly calculated with upward recursion using Equation 10.10(9) shown on page 179 of Erdélyi et al. [1981], Volume II. Some calculations of Legendre polynomials $P_n(x)$ with very high degree $n \gtrsim 10^4$ and $|x| < 1$ use a nonuniform asymptotic expression involving trigonometric functions, from Equation 10.05 on page 313 of Olver [1997]. Legendre polynomials $P_n(x)$ with large argument use asymptotic series expansion. See Section 3.9 of this article for the generalized Legendre functions, $P_\nu^\mu(x)$ and $Q_\nu^\mu(x)$.

3.7 Gamma and Related Functions

The gamma function uses a Taylor series expansion for small arguments. The first 20 coefficients in the series expansion $1/\Gamma(z) \sim z + \gamma z^2 - 0.655878\cdots z^3 \cdots$ have been numerically computed to 400 digits with Mathematica®. These are accumulated and inverted to obtain $\Gamma(z)$.

Intermediate and large arguments $z \in \mathbb{R}$ with $\mathfrak{Re}(z) > 0$ use Stirling's asymptotic expansion with Bernoulli numbers (Abramowitz and Stegun [1972] Equation 6.1.40, page 257),

$$\ln \Gamma(z) \approx \left(z - \frac{1}{2}\right) \ln z - z + \frac{1}{2} \ln(2\pi) + \sum_{k=1}^{\infty} \frac{B_{2k}}{2k(2k-1)z^{2k-1}}\,. \tag{6}$$

Stirling's asymptotic expansion is divergent. Therefore, in order to compute $\Gamma(z)$ for intermediate arguments, e_float increases the argument and computes the value of $\Gamma(z + N_{\texttt{offset}})$ with $N_{\texttt{offset}} \in \mathbb{N}^{+}$. To obtain the final result, downward recursion is subsequently used after exponentiation of Stirling's expansion in Equation (6). The value of $|z + N_{\texttt{offset}}|$ needed for efficient convergence has been determined empirically. It ranges from 60 to 240 for precision ranging from 30 to 300 digits. The implementation of Stirling's expansion carefully checks the base 10 order of its result and avoids unnecessary divergence by stopping as soon as the desired precision (including its digit padding) has been reached. Arguments with $\mathfrak{Re}(z) < 0$ use reflection (Abramowitz and Stegun [1972] Equation 6.1.17, page 256).

Gamma function calculations have employed static polymorphism in order to use the same template algorithm for both real as well as complex calculations, thus reducing program complexity.

The incomplete gamma function $\Gamma(x, a)$ is computed using $\Gamma(x)$ in combination with the lower incomplete gamma function $\gamma(x, a)$, itself computed with the identity ${}_1F_1(a; a+1; -x) = a x^{-a} \gamma(x, a)$. Additional functions such as Pochhammer symbols, binomial coefficients, and some beta functions are supported. However, these are only used as utilities.

Polygamma functions use Taylor series expansion for small arguments. Large arguments use asymptotic expansion with Bernoulli numbers. Both are taken from subsections of Wolfram Research [2010] "Gamma, Beta, Erf—Differentiated Gamma Functions—PolyGamma[ν, z],"

$$\psi^{n}(x) \approx \frac{(-1)^{n-1} n!}{x^{n+1}} + \sum_{k=0}^{\infty} \frac{\psi^{k+n}(1) z^{k}}{k!}, \qquad \text{if } |x| \lesssim 1, \tag{7}$$

$$\psi^{n}(x) \approx \frac{(-1)^{n-1}(n-1)!\,(n+2x)}{2x^{n+1}} - (-1)^{n} \sum_{k=1}^{\infty} \frac{(2k+n+1)!\,B_{2k}}{(2k)!\,z^{2k+n}}, \qquad \text{if } x \gtrsim 500. \tag{8}$$

Polygamma functions for intermediate argument and order $m > 1$ use Euler-MacLaurin summation,

$$\frac{(-1)^{m+1}}{m!}\psi^{m}(x) \approx \left(\sum_{k=0}^{N-1} \frac{1}{(x+k)^{m+1}} \right) + \frac{1}{2(x+N)^{m+1}} + \frac{1}{(x+N)^{m}} + \sum_{k=1}^{\infty} \frac{B_{2k}}{(2k)!} \frac{(m+1)_{2k-1}}{(x+N)^{m+2k}}, \tag{9}$$

with $N = 0.4\,\texttt{digits} + 4m$, where $\texttt{digits}$ is the number of decimal digits of precision, and the infinite series stops when the desired precision (including its digit padding) has been reached. Argument reflection uses expansion with Stirling's numbers of the second kind.

3.8 Hypergeometric Series

Calculations of ${}_1F_1(a; b; x)$ use Taylor series for small arguments. Intermediate arguments use expansion in terms of Bessel functions (Abramowitz and Stegun [1972]

Equation 13.3.8, page 506), with $h = -\pi/10$. Large arguments use asymptotic expansion from Abramowitz and Stegun [1972] Equation 13.5.1, page 508. Downward recursion is used for $a > 0$, $b < -10$. The function ${}_1F_1(a; b; x)$ is subsequently used for other functions such as generalized Laguerre and incomplete gamma functions.

As shown in Table I, there is only limited support for generalized hypergeometric functions, ${}_pF_q(a_1, \ldots a_p; b_1, \ldots b_q; x)$. These are primarily implemented as utilities for other calculations, such as those shown in Equations (1), (3), and (5). As such, they are not intended to cover the complete parameter ranges. Taylor series expansion are used for these.

3.9 Generalized Legendre Functions

Calculations of generalized Legendre functions of the first and second kind, both of type 1, use Taylor series at +1, from Equations 3.4(6) and 3.4(10) in Section 3.4, "Legendre Functions on the Cut," on pages 143–144 of Erdélyi et al. [1981], Volume I,

$$P_\nu^\mu(x) = \frac{1}{\Gamma(1-\mu)} \left(\frac{1+x}{1-x}\right)^{\mu/2} {}_2F_1\left(-\nu, \nu+1; 1-\mu; \frac{1-x}{2}\right), \tag{10}$$

$$\begin{aligned} Q_\nu^\mu(x) = \frac{\pi \csc(\mu\pi)}{2} \Bigg\{ & \frac{\cos(\mu\pi)}{\Gamma(1-\mu)} \left(\frac{1+x}{1-x}\right)^{\mu/2} {}_2F_1\left(-\nu, \nu+1; 1-\mu; \frac{1-x}{2}\right) \\ & - \frac{(\nu-\mu+1)_{2\mu}}{\Gamma(\mu+1)} \left(\frac{1-x}{1+x}\right)^{\mu/2} {}_2F_1\left(-\nu, \nu+1; \mu+1; \frac{1-x}{2}\right) \Bigg\}. \end{aligned} \tag{11}$$

There are other special series for integer degree or integer order which have been taken from subsections of Wolfram Research [2010] "Hypergeometric Functions—Legendre Functions—LegendreP[ν, μ, 2, z]—LegendreQ[ν, μ, 2, z], etc." Higher-degree $|\nu| \gtrsim 10$ and higher-order $|\mu| \gtrsim 10$ use upward recursion. Integer degree combined with integer order uses upward recursion. Care has been taken to employ recursion solely in parameter regions for which it is stable over thousands of iterations. If both degree ν and order μ need recursion, then it is "up, and to the right," beginning with upward recursion of degree ν followed by upward (rightward) recursion of order μ. Actually, two recursions of ν are needed because recursion of μ requires two starting points, $P_\nu^{\mu\_\mathtt{frac}}(x)$ and $P_\nu^{1+\mu\_\mathtt{frac}}(x)$, where μ`_frac` is the fractional part of μ. Negative arguments use reflection of argument. Negative degree ν uses reflection of degree. Negative order μ uses reflection of order, whereby reflection of order is avoided in some parameter regions because of precision loss (see the virtual Boolean function `NeedsReflectNegativeOrder`). The various reflection equations are shown in Erdélyi et al. [1981], Volume I, Equations 3.4(14) and (15); Equations 3.4(7) and (16); and Equations 3.4(17) and (18).

Generalized Legendre functions have employed dynamic polymorphism to reduce program complexity. This is possible because the basic algorithms for calculating $P_\nu^\mu(x)$ and $Q_\nu^\mu(x)$ are essentially identical. Algorithmic differences, parameterization of the regions of stable recursion, are handled with the C++ virtual function mechanism.

3.10 Laguerre, Parabolic Cylinder, and Hermite Functions

Generalized Laguerre functions $L_\nu^\lambda(x)$ are calculated using the hypergeometric identity ${}_1F_1(-\nu; \lambda+1; x) = \left\{\Gamma(\lambda+\nu+1) / \Gamma(\nu+1)\right\} L_\lambda^\nu(x)$. Mixed-integer parameter functions $L_m^\lambda(x)$ rely on the hypergeometric identity combined with recursion. Pure-integer parameter functions $L_m^n(x)$ rely on recursion.

Parabolic cylinder functions or Weber functions use Taylor series for small x, asymptotic series expansion for large x and recursion, with algorithms adapted from those in two recent publications by Gil et al. [2006; 2007]. Uniform asymptotic expansion in terms of elementary functions is used for certain ranges of very large ν, as described in Chapter 12.5(a) of Gil et al. [2007]. The coefficients of the polynomials of the uniform asymptotic expansions have been extended to high precision. These have been recently added by the author to Sloane's [2009] *Encyclopedia*. The new sequence A158503 contains the numerators of the coefficients of the first 121 polynomials, with denominators in A001164.

The generalized Hermite function $H_\nu(x)$ uses a relation with the parabolic cylinder function $D_\nu(x) = 2^{-\nu/2}\, e^{-x^2/4}\, H_\nu(x/\sqrt{2})$.

3.11 Elliptic Integrals

Elliptic integrals use the arithmetic geometric mean (AGM) originally from King [1924], and adapted from the Fortran subroutine `ELIT` on page 664 of Zhang and Jin [1996]. The argument convention with $k^2 = m$ is used.

3.12 Zeta Functions

Riemann zeta functions use an expansion of the product of primes for large arguments (Abramowitz and Stegun [1972] Equation 23.2.2 on page 807). Small and intermediate arguments use an accelerated alternating converging series from Gourdon and Sebah [2008], originally from Borwein [1995].

Hurwitz zeta functions $\zeta(s, a)$ use the primary power series,

$$\zeta(s, a) \approx \sum_{k=0}^{N} \frac{1}{(k+a)^s}, \tag{12}$$

for large arguments with the convergence test $s \log_{10}(a + N) \gtrsim$ `digits` with $N \lesssim 167$–500 for `digits` ranging from 30 to 300. Certain calculations with large, negative s, meaning $s < (-20)$–(-60), and $0 < a < 200$ use an exponential Fourier series in combination with argument scaling. Intermediate arguments use Euler-Maclaurin summation,

$$\begin{aligned} \zeta(s, a) \;\approx\; & \left(\sum_{k=0}^{N-1} \frac{1}{(a+k)^s} \right) + \frac{1}{2(a+N)^s} \\ & + \frac{(a+N)^{1-s}}{s-1} + \sum_{k=1}^{\infty} \frac{B_{2k}}{(2k)!} \frac{(s)_{2k}}{(a+N)^{s+2k+1}}, \end{aligned} \tag{13}$$

with $N = 10 + 0.5\,$`digits`$\,+ 1.1\,|\mathfrak{Im}(s)|$, where `digits` is the number of decimal digits of precision, and the infinite series stops when the desired precision (including its digit padding) has been reached.

Zeta function calculations have employed static polymorphism in order to use the same template algorithms for both real as well as complex calculations, thus reducing program complexity.

3.13 Polylogarithms

Polylogarithms $\mathrm{Li}_n(x)$ are calculated for $x \in \mathbb{R}$, $x \leq 1$ with $n \in \mathbb{N}$ for $0 \leq x \leq 1$, and $n \in \mathbb{N}^+$ for $x < 0$. The computations use a Taylor series for argument near 1, a different

Taylor series for argument near zero, and another asymptotic series for large negative argument, all taken from subsections of Wolfram Research [2010] "Zeta Functions and Polylogarithms—Polylogarithms—PolyLog[ν, z]."

4. RESULTS

4.1 Testing and Precision

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 the 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, and 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 precomputed 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 ~1–100 individual numerical values, resulting in a total of ~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 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 IV. The aforementioned test result was also briefly given in Section 3. It is one the most significant results of this article. All testing evidence indicates that e_float correctly calculates all of the function values for the parameter ranges listed in Table I 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 the 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 nonconfirmed values which have been computed with e_float include the following.

$$P_{\pi+20{,}000}^{\gamma+10{,}000}\left(\tfrac{2}{3}\right) \approx \texttt{-5.980883901705929125018083995261080589601569670160}$$
$$\texttt{46622596395405592361493628806741873547831350304266 7}\times 10^{42814}$$

$$Q_{2347}^{-2099}(\gamma) \approx \texttt{5.134474212409628343860552869233848621335039670532 95}$$
$$\texttt{28915982152345445031498929856687701223000192113 48}\times 10^{-6860}$$

$$J_{\pi+10^8}\left(A\cdot 10^8\right) \approx \texttt{0.00007101587864721510600609002545136459450330588}$$
$$\texttt{28414017520190813506282451749590085511887647998048 1012041}$$

$$K_{690}(\pi+310) \approx \texttt{2.4730931345807616992820912257524449912893653029 0}$$
$$\texttt{487309882278043068716225815819255631061623478792425 9}\times 10^{128}$$

$$D_{\frac{1201}{12}}(40) \approx \texttt{1.524674031836673520032956906376264551740269827916 8}$$
$$\texttt{502815073526115676632722303854692539301123352604 40}\times 10^{-15}$$

4.2 Performance

Extensive timing analyses and performance measurements have been carried out with e_float. Timing measurements have been done using a desktop PC with an Intel® Core™2 Quad CPU Q9400 at 2.66 GHz with 4 GB RAM. Both Windows® 7 Enterprise x64 as well as Ubuntu 9.10 amd64 Linux operating systems have been used for timing measurements. Codes for timing tests have been compiled with compilers from GNU and Microsoft® (see Table IV) with the highest speed optimization settings for each compiler.

Care has been taken to use code sequences which accurately measure function performance without extraneous I/O operations or other overhead. Function timing measurements use loops which calculate many function values. In this way, accurate single value function timing can be calculated by dividing the total loop time by the number of loop iterations. When devising timing loops, constants like π, A, or γ have been strategically combined with the loop index in order to simultaneously sweep a function's parameter range and also ensure that all intermediate operations are performed with noninteger operations requiring full precision. Code samples for timing measurements are shown in Listing 7.

Listing 7. This C++ code measures the performance of, for example, the Bessel function of integer order $J_n(x)$ with small to medium parameters $x \in \mathbb{R}$, $n \in \mathbb{N}^+$.

```
const Util::timer tm;
for(INT32 k = 0; k < 1000; k++)
{
  const e_float y = ef::cyl_bessel_j(k, ef::glaisher() * k);
  const double elapsed = tm.elapsed();
  std::cout << elapsed << std::endl;
  // 1.419
}
```

Listing 8. This listing shows the corresponding performance measurement in a Mathematica® session, where it is assumed that the C++ sequence had also used 100 digits.

```
In[1]:= Timing[Table[N[BesselJ[k, Glaisher k], 100],
        {k, 0, 1000 - 1, 1}]][[1]]
Out[1]= 5.219
```

Table V. Function Timings

Function	Operation	Implementation	30	100	300
times	$\pi \times \gamma$	`f90::e_float`	0.05 μs	—	—
		`mpfr::e_float`	0.34 μs	$\underline{0.40\,\mu\text{s}}$	$\underline{0.95\,\mu\text{s}}$
		`efx::e_float`	$\underline{0.14\,\mu\text{s}}$	0.41 μs	1.5 μs
		`gmp::e_float`	0.44 μs	0.52 μs	2.1 μs
		Mathematica® 7.1	0.56 μs	0.91 μs	2.9 μs
sqrt	$\sqrt{x}$, with $1 \lesssim x \lesssim 10^5$	`f90::e_float`	0.3 μs	—	—
		`mpfr::e_float`	$\underline{1.1\,\mu\text{s}}$	$\underline{1.7\,\mu\text{s}}$	$\underline{4.1\,\mu\text{s}}$
		`efx::e_float`	2.5 μs	8.4 μs	21 μs
		`gmp::e_float`	1.4 μs	2.1 μs	5.8 μs
		Mathematica® 7.1	42 μs	56 μs	61 μs
sin	$\sin(x)$, with $1 \lesssim x \lesssim 10^4$	`f90::e_float`	0.5 μs	—	—
		`mpfr::e_float`	$\underline{11\,\mu\text{s}}$	$\underline{17\,\mu\text{s}}$	$\underline{48\,\mu\text{s}}$
		`efx::e_float`	45 μs	120 μs	410 μs
		`gmp::e_float`	82 μs	95 μs	350 μs
		Mathematica® 7.1	50 μs	78 μs	200 μs
exp	e^x, with $1 \lesssim x \lesssim 10^4$	`f90::e_float`	0.4 μs	—	—
		`mpfr::e_float`	21μs	$\underline{33\,\mu\text{s}}$	$\underline{120\,\mu\text{s}}$
		`efx::e_float`	$\underline{16\,\mu\text{s}}$	36 μs	210 μs
		`gmp::e_float`	42 μs	55 μs	250 μs
		Mathematica® 7.1	40 μs	78 μs	260 μs
gamma	$\Gamma(x)$, with $1 \lesssim x \lesssim 10^3$	`f90::e_float`	0.007 ms	—	—
		`mpfr::e_float`	$\underline{0.064\,ms}$	$\underline{0.14\,ms}$	$\underline{0.7\,ms}$
		`efx::e_float`	0.064 ms	0.26 ms	1.8 ms
		`gmp::e_float`	0.17 ms	0.32 ms	2.1 ms
		MPFR's own	0.14 ms	0.27 ms	2.2 ms
		Mathematica® 7.1	0.38 ms	0.81 ms	3.8 ms
Bessel	$J_n(x)$, with $1 \lesssim x, n \lesssim 10^3$	`f90::e_float`	—	—	—
		`mpfr::e_float`	1.1 ms	$\underline{1.4\,ms}$	$\underline{2.8\,ms}$
		`efx::e_float`	$\underline{0.5\,ms}$	1.1 ms	3.8 ms
		`gmp::e_float`	1.4 ms	1.7 ms	7.8 ms
		MPFR's own	1.7 ms	2.2 ms	5.6 ms
		Mathematica® 7.1	3.7 ms	5.2 ms	40 ms
zeta	$\zeta(x)$, with $1 \lesssim x \lesssim 50$	`f90::e_float`	0.04ms	—	—
		`mpfr::e_float`	0.61 ms	$\underline{2.7\,ms}$	$\underline{12\,ms}$
		`efx::e_float`	$\underline{0.42\,ms}$	3.4 ms	31 ms
		`gmp::e_float`	1.0 ms	4.2 ms	31 ms
		MPFR's own	1.1 ms	5.6 ms	59 ms
		Mathematica® 7.1	1.2 ms	13 ms	240 ms

Detailed timing measurements using 30, 100, and 300 digits are shown in Table V. The three main `e_float` classes, `efx::e_float`, `mpfr::e_float`, and `gmp::e_float`, have roughly comparable performance. At 30 digits (excluding `f90::e_float`), `efx::e_float` is faster for four out of seven of the functions, while `mpfr::e_float` is faster for the other three. At 100 and 300 digits, `mpfr::e_float` has the highest performance for all of the functions shown in Table V. At 30 digits, `f90::e_float` is very fast. But its limited exponent size prevents it from being used with some of the algorithms for the parameter ranges shown in Table I.

The timings in the first few rows of Table V show that `mpfr::e_float` has very fast implementations of elementary functions. Recall from Section 2.1 that these functions from MPFR have been incorporated into `mpfr::e_float` using the "has-its-own" mechanism. However, beginning with the row containing the gamma function, it can

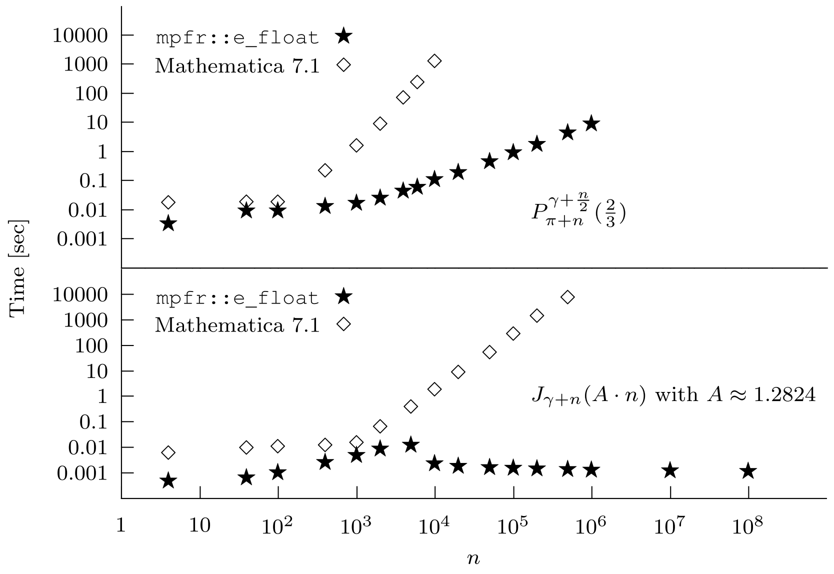

Fig. 5. Timing measurements for calculations of Legendre functions and Bessel functions with very large parameters. Measurements using `mpfr::e_float` and Mathematica® 7.1, each with 100 digits, are compared.

be seen that MPFR's own implementations of higher transcendental functions, insofar as these are even supported, are not particularly fast in comparison with the corresponding algorithms in e_float. Therefore, `mpfr::e_float` uses e_float's faster algorithms for gamma, Bessel, and zeta functions instead of MPFR's implementations of these. Thus `mpfr::e_float` is a hybrid MP type which uses the high-performance, low-level functions of MPFR combined with the fast high-level functions of e_float.

For the most part, e_float has higher performance than Mathematica®. In particular, `mpfr::e_float` is significantly faster than Mathematica® for all of the measured functions. Keep in mind, however, that Mathematica® is designed for very-high-level operation with multitudinous applications in widespread branches of analysis and visualization, whereby e_float uses highly optimized, hand-coded algorithms implemented in compiled code which specifically target nothing other than its high-performance function calculations. Comparing the performance of e_float with that of Mathematica® makes only limited sense because the two systems are designed for entirely different ranges of application.

Another perspective on timing comes from calculations of functions with very large parameters. Figure 5 shows large parameter calculations of Legendre functions and Bessel functions. Mathematica® requires seconds to minutes to hours as the parameter ranges successively increase in orders of magnitude. Mathematica® is impractical for these kinds of calculations. On the other hand, e_float remains relatively efficient as the degree and order increase. Its computation time for Legendre functions grows only linearly with increasing degree and order. This is because stable recursion is used for Legendre functions and the number of necessary recursions increases linearly with increasing degree and order. For Bessel functions, e_float's computation time rises linearly with increasing order up to $\sim 10^4$ and then, in fact, decreases for even higher orders. This is because Bessel functions with very high order use uniform asymptotic expansions which are very fast and which have coefficients inversely proportional to the squared powers of the order.

Table VI. Timing Measurements in Seconds for the Entire Test Suite at Various Digit Settings

Digits	`mpfr::e_float`		`gmp::e_float`		`efx::e_float`		Mathematica®
	GCC	VC9	GCC	VC9	GCC	VC9	
30	8	13	11	21	11	11	—
50	9	16	14	25	19	19	—
100	18	25	22	39	44	42	1800
200	42	56	61	86	152	140	—
300	94	120	137	170	354	300	2300

The final timing comparison involves the computation time for all of the function values in the entire test suite using five different digit settings. The results are shown in Table VI. This measures the overall performance of e_float in a wide variety of calculational situations. Again, the three main `e_float` classes have roughly comparable performance, with `mpfr::e_float` faster than the others. Mathematica® does not perform well in this timing comparison. In fact, `mpfr::e_float` compiled with GCC is 100 times faster than Mathematica® in this timing comparison with 100 digits.

It can be seen that the computations of e_float are quite efficient at all supported digit ranges for all supported functions, including some large parameter ranges. In particular, `mpfr::e_float` has very high performance.

5. CONCLUSION AND OUTLOOK

The portable C++ e_float system has been presented. The system can be used to calculate many high-precision special function values, some of which have very large parameters. For the first time, a system with a robust, layered architecture uses a uniform C++ interface to bridge the gap between low-level MP implementations and high-level algorithm design. This allows any suitably prepared MP type to be used with e_float. The system is well tested and has high performance. In addition, e_float includes interfaces for interoperability with very-high-level languages including Microsoft® CLR, traditional Python, IronPython, and Mathematica®. Very-high-level hybrid systems can be created with e_float's interoperabilities.

There are unlimited algorithms, functions, parameter ranges, precisions, and interoperabilities available for future investigations. Accordingly, extensions of e_float are planned. In particular, work to add more complex function parameters and extend parameter ranges by removing some of the limitations shown in Table I is already under way. It would be of interest to develop generic algorithms for continued fractions and investigate their performance for MP special-function calculations. In addition, preliminary investigations are being carried out for the extension from 30 to 300 digits to the milestone of 1000 digits. This will involve the additional use of powerful asymptotic algorithms requiring the implementation of some rudimentary symbolic methods such as polynomial integration and differentiation needed for the runtime calculation of asymptotic coefficients, as well as some slight architectural changes such as dynamic digit extension.

After extension to 1000 digits, it will be of great interest to use e_float's uniform interface to MP algorithm design to implement an optimized, multithreaded, portable "PSLQ" algorithm Bailey and Broadhurst [2000] for recognizing an integer relation involving fundamental constants based on a given numerical result—such as might be obtained from a MP calculation involving special functions, integrals, products, and/or sums. PSLQ algorithms can be used to numerically verify identities and fundamental relationships in the field of experimental mathematics [Bailey et al. 2007].

The author encourages others to use e_float for high-precision calculations and also to extend it in order to create powerful computational systems useful in many branches of science and engineering. Developers can adapt their MP implementations for use with e_float, add additional functions, extend existing functions to wider parameter ranges, use e_float's interface to Mathematica®, or even add interoperabilities with other computer algebra systems, do rapid algorithm prototyping with Python and use the power of e_float in the Microsoft®.NET Framework. The e_float system provides a new platform for MP development.

ACKNOWLEDGMENTS

I would like to express my gratitude to Jörg Arndt for his valuable help with my first big–number investigations. T. Mark Dunster provided useful advice on algorithms for Legendre functions. I would also like to especially thank Nico M. Temme for his assistance on many topics of special functions.

REFERENCES

ABRAHAMS, D. 2008. *boost.python Index*. http://www.boost.org/doc/libs/1_42_0/libs/python/.

ABRAMOWITZ, M. AND STEGUN, I. A. 1972. *Handbook of Mathematical Functions* 9th Ed. Dover Publications, New York, NY.

AMOS, D. E. 1986. A portable package for Bessel functions of a complex argument and nonnegative order. *ACM Trans. Math. Softw. 12*, 3, 265–273.

ANDREWS, G., ASKEY, R., AND ROY, R. 2000. *Special Functions*. Cambridge University Press, Cambridge, UK.

ARNDT, J. AND HAENEL, C. 2000. *π Unleashed*. Springer, New York, NY.

BAILEY, D., BORWEIN, J. M., CALKIN, N. J., GIRGENSOHN, R., LUKE, D., AND MOLL, V. H. 2007. *Experimental Mathematics in Action*. A.K. Peters Press, Natick, MA.

BAILEY, D. H. 1993. Multiprecision translation and execution of Fortran programs. *ACM Trans. Math. Softw. 19*, 3, 288–319.

BAILEY, D. H. 1995. A Fortran 90–based multiprecision system. *ACM Trans. Math. Softw. 21*, 4, 379–387.

BAILEY, D. H. AND BROADHURST, D. J. 2000. Parallel integer relation detection: Techniques and applications. *Math. Comp. 70*, 1719–1736.

BAILEY, D. H., HIDA, Y., LI, X. S., AND THOMPSON, B. 2002. ARPREC: An arbitrary precision computation package. Tech. rep. Lawrence Berkeley National Laboratory, Berkeley, CA.

BECKER, P. 2006. *The C++ Standard Library Extensions: A Tutorial and Reference*. Addison Wesley, Reading, MA.

BORWEIN, P. 1995. An efficient algorithm for the Riemann zeta function. *Can. Math. Soc. Conf. Proc. 27*, 29–34.

BRENT, R. P. 1978. A Fortran multiple-precision arithmetic package. *ACM Trans. Math. Softw. 4*, 1, 57–70.

CODY, W. J. 1993. SPECFUN—A portable Fortran package of special function routines and test drivers. *ACM Trans. Math. Softw. 19*, 1, 22–30.

CODY, W. J. AND WAITE, W. 1980. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, New Jersey.

ERDÉLYI, A., MAGNUS, W., OBERHETTINGER, F., AND TRICOMI, F. G. 1981. *Higher Transcendental Functions*. Vols. 1–2. Krieger, New York, NY.

ESPINOSA, O. AND MOLL, V. H. 2004. A generalized polygamma function. *Integral Trans. Special Func. 15*, 101–115.

FINCH, S. R. 2003. *Mathematical Constants*. Cambridge University Press, Cambridge, UK.

FOORD, M. J. AND MUIRHEAD, C. 2009. *IronPython in Action*. Manning Publications, Greenwich, CT.

FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P., AND ZIMMERMANN, P. 2007. MPFR: A multiple–precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw. 33*, 2, 1–15.

GALASSI, M., DAVIES, J., THEILER, J., GOUGH, B., JUNGMAN, G., ALKEN, P., BOOTH, M., AND ROSSI, F. 2009. *GNU Scientific Library Reference Manual, for Version 1.12* 3rd Ed. Network Theory Ltd., Bristol, UK.

GCC. 2009. *The GNU Compiler Collection Version 4.4.2*. Free Software Foundation. http://gcc.gnu.org/.

GIL, A., SEGURA, J., AND TEMME, N. M. 2006. Computing the real parabolic cylinder functions $U(a, x)$, $V(a, x)$. *ACM Trans. Math. Softw. 32*, 1, 70–101.

GIL, A., SEGURA, J., AND TEMME, N. M. 2007. *Numerical Methods for Special Functions*. SIAM, Philadelphia, PA.

GLADMAN, B. 2008. *Building GMP and MPFR with Microsoft® Visual Studio 2008 and YASM*. http://gladman.plushost.co.uk/oldsite/computing/gmp4win.php.

GMP. 2008. *GNU Multiple Precision Arithmetic Library Version 4.2.4*. Free Software Foundation. http://gmplib.org/.

GNU. 2006. *GNUmake Version 3.81*. Free Software Foundation. http://www.gnu.org/software/make/.

GOURDON, X. AND SEBAH, P. 2008. *Numbers, Constants and Computation*. http://numbers.computation.free.fr/.

INTEL. 2008. *Intel® C++ Compiler Professional Edition*. http://software.intel.com/en-us/intel-compilers/.

ISO. 2003. *ISO/IEC 14882:2003: Information Technology—Programming Languages—C++*. International Organization for Standardization, Geneva, Switzerland.

ISO. 2006a. *ISO/IEC 23270:2006: Information Technology Programming Languages—C#*. International Organization for Standardization, Geneva, Switzerland.

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

ISO. 2007. *ISO/IEC 19768:2007: Information Technology—Programming Languages—Technical Report on C++ Library Extensions*. International Organization for Standardization, Geneva, Switzerland.

JAHNKE, E. AND EMDEN, F. 1945. *Tables of Functions with Formulae and Curves* 4th Ed. Dover, New York, NY.

JOSUTTIS, N. M. 1999. *The C++ Standard Library: A Tutorial and Reference*. Addison Wesley, Reading, MA.

KING, L. V. 1924. *On the Direct Numerical Calculation of Elliptic Functions and Integrals*. Cambridge University Press, Cambridge, UK.

KNUTH, D. E. 1998. *The Art of Computer Programming Vols. 1–3* 2nd Ed. Addison Wesley, Reading, MA.

LUKE, Y. L. 1977. *Algorithms for the Computation of Mathematical Functions*. Academic Press, New York, NY.

MICROSOFT. 2008. *Microsoft® Visual Studio Professional 2008*. http://www.microsoft.com/visualstudio/.

MICROSOFT. 2009a. *IronPython*. http://www.ironpython.net/.

MICROSOFT. 2009b. *NMAKE Reference*. http://msdn.microsoft.com/en-us/library/dd9y37ha(VS.71).aspx.

MICROSOFT. 2010. *Common Language Runtime Overview*. http://msdn.microsoft.com/en-us/library/ddk909ch.aspx.

MISRA. 2004. *MISRA–C 2004: Guidelines for the Use of the C Language in Critical Systems*. http://www.misra.org.uk/.

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

MPMATH. 2009. *Python Library for Arbitrary-Precision Floating-Point Arithmetic*. http://code.google.com/p/mpmath/.

OLDHAM, K., MYLAND, J., AND SPANIER, J. 2009. *An Atlas of Functions* 2nd Ed. Springer, New York.

OLVER, F. W. J. 1997. *Asymptotics and Special Functions*. A.K. Peters Press, Natick MA.

PRESS, W. H., TEUKOLSKY, S. A., VETTERLING, W. T., AND FLANNERY, B. P. 2002. *Numerical Recipes in C++* 2nd Ed. Cambridge University Press, Cambridge, UK.

PSF. 2009. *Python Programming Language—Official Website*. http://www.python.org/.

RICHTER, J. 2006. *CLR Via C#* 2nd Ed. Microsoft Press, Redmond, WA.

SHOUP, V. 2008. *NTL: A Library for Doing Number Theory*. http://www.shoup.net/ntl/.

SLOANE, N. J. A. 2009. *Online Encyclopedia of Integer Sequences*. http://www.research.att.com/~njas/sequences/.

SMITH, D. 1991. A Fortran package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw. 17*, 2, 273–283.

SMITH, D. 1998. Multiple-precision complex arithmetic and functions. *ACM Trans. Math. Softw. 24*, 4, 359–367.

TEMME, N. M. 1975. On the numerical evaluation of the modified Bessel function of the third kind. *J. Comp. Phys. 19*, 324.

TEMME, N. M. 2007. Personal communications. 2007–2009.

VEPŠTAS, L. 2008. An efficient algorithm for accelerating the convergence of oscillatory series, useful for computing the polylogarithm and Hurwitz zeta functions. *Numer. Algor. 47*, 3, 211–252.

WATSON, G. N. 1995. *A Treatise on the Theory of Bessel Functions*. Cambridge University Press, Cambridge, UK.

WEISSTEIN, E. W. 2010. *MathWorld®–A Wolfram Web Resource*. http://mathworld.wolfram.com/.

WIKIPEDIA. 2009. *Wikipedia—The Free Encyclopedia*. http://en.wikipedia.org/wiki/.

WIMP, J. 1984. *Computation with Recurrence Relations*. Pitman Publishing, London, UK.

WOLFRAM, S. 1999. *The Mathematica® Book* 4th Ed. Cambridge University Press, Cambridge, UK.

WOLFRAM RESEARCH. 2010. *The Wolfram Functions Site*. http://functions.wolfram.com/.

ZHANG, S. AND JIN, J. 1996. *Computation of Special Functions*. Wiley, New York, NY.

Received July 2009; revised February 2010, July 2010; accepted October 2010