A Thread-Safe Arbitrary Precision Computation Package (Full Documentation)

David H. Bailey [*]

June 13, 2021

Abstract

Numerous research studies have arisen, particularly in the realm of mathematical physics and experimental mathematics, that require extremely high numeric precision. Such precision greatly magnifies computer run times, so software packages to support high-precision computing must be designed for thread-based parallel processing.

This paper describes a new package ("MPFUN2015") that is thread-safe, even at the language interface level, yet still permits the working precision level to be freely changed during execution. The package comes in two versions: (a) a completely self-contained, all-Fortran version that is simple to install; and (b) a version based on the MPFR package (for lower-level operations) that is more complicated to install but is approximately 3X faster. Except for a few special functions, the two versions are "plug-compatible" in the sense that applications written for one also run with the other. Both versions employ advanced algorithms, including FFT-based arithmetic, for optimal performance. They also detect, and provide means to overcome, accuracy problems rooted in the usage of inexact double-precision constants and expressions. A high-level Fortran-90 interface, supporting both multiprecision real and complex datatypes, is provided for each, so that most users need only to make minor changes to existing code.

[*]Lawrence Berkeley National Laboratory (retired), 1 Cyclotron Road, Berkeley, CA 94720, USA, and University of California, Davis, Department of Computer Science. E-mail: `david@davidhbailey.com`.

Contents

1 Applications of high-precision computation

For many scientific calculations, particularly those that employ empirical data, IEEE 32-bit floating-point arithmetic is sufficiently accurate, and is preferred since it saves memory, run time and energy usage. For other applications, 64-bit floating-point arithmetic is required to produce results of sufficient accuracy, although some users find that they can obtain satisfactory results by switching between 32-bit and 64-bit, using the latter only for certain numerically sensitive sections of code. Software tools are being developed at the University of California, Berkeley and elsewhere to help users determine which portions of a computation can be performed with lower precision and which must be performed with higher precision [26].

Other applications, particularly in the fields of mathematical physics and experimental mathematics, require even higher precision — tens, hundreds or even thousands of digits. Here is a brief summary of these applications:

1. Supernova simulations (32–64 digits).
2. Optimization problems in biology and other fields (32–64 digits).
3. Coulomb n-body atomic system simulations (32–120 digits).
4. Electromagnetic scattering theory (32–100 digits).
5. The Taylor algorithm for ODEs (100–600 digits).
6. Ising integrals from mathematical physics (100–1000 digits).
7. Problems in experimental mathematics (100–50,000 digits and higher).

These applications are described in greater detail in [1, 3], which provides detailed references. Here is a brief overview of a handful of these applications:

1.1 The PSLQ integer relation algorithm

Very high-precision floating-point arithmetic is now considered an indispensable tool in experimental mathematics and mathematical physics [1]. Many of these computations involve variants of Ferguson's PSLQ integer relation detection algorithm [17, 8]. Suppose one is given an n-long vector (x_i) of real or complex numbers (presented as a vector of high-precision values). The PSLQ algorithm finds the integer coefficients (a_i), not all zero, such that

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = 0$$

(to available precision), or else determines that there is no such relation within a certain bound on the size of the coefficients. Alternatively, one can employ the Lenstra-Lenstra-Lovasz (LLL) lattice basis reduction algorithm to find integer relations [22], or the "HJLS" algorithm, which is based on LLL. Both PSLQ and HJLS can be viewed as schemes to compute the intersection between a lattice and a vector subspace [14]. Whichever algorithm is used, integer relation detection requires very high precision—at least $(n \times d)$-digit precision, where d is the size in digits of the largest a_i and n is the vector length, or else the true relation will be unrecoverable.

1.2 High-precision numerical integration

One of the most fruitful applications of the experimental methodology and the PSLQ integer relation algorithm has been to identify classes of definite integrals, based on very high-precision numerical values, in terms of simple analytic expressions.

These studies typically employ either Gaussian quadrature or the "tanh-sinh" quadrature scheme of Takahasi and Mori [28, 2]. The tanh-sinh quadrature algorithm approximates the integral of a function on $(-1, 1)$ as

$$\int_{-1}^{1} f(x)\,\mathrm{d}x \approx h \sum_{j=-N}^{N} w_j f(x_j), \tag{1}$$

where the abscissas x_j and weights w_j are given by

$$\begin{aligned} x_j &= \tanh\left(\pi/2 \cdot \sinh(hj)\right) \\ w_j &= \pi/2 \cdot \cosh(hj) / \cosh\left(\pi/2 \cdot \sinh(hj)\right)^2, \end{aligned} \tag{2}$$

and where N is chosen large enough that summation terms in (1) beyond N (positive or negative) are smaller than the "epsilon" of the numeric precision being used. Full details are given in [2]. An overview of applications of high-precision integration in experimental mathematics is given in [4].

1.3 Ising integrals

In one study, tanh-sinh quadrature and PSLQ were employed to study the following classes of integrals [7]. The C_n are connected to quantum field theory, the D_n integrals arise in the Ising theory of mathematical physics, while the E_n integrands are derived from D_n:

$$C_n = \frac{4}{n!} \int_0^\infty \cdots \int_0^\infty \frac{1}{\left(\sum_{j=1}^n (u_j + 1/u_j)\right)^2} \frac{\mathrm{d}u_1}{u_1} \cdots \frac{\mathrm{d}u_n}{u_n}$$

$$D_n = \frac{4}{n!} \int_0^\infty \cdots \int_0^\infty \frac{\prod_{i<j} \left(\frac{u_i - u_j}{u_i + u_j}\right)^2}{\left(\sum_{j=1}^n (u_j + 1/u_j)\right)^2} \frac{\mathrm{d}u_1}{u_1} \cdots \frac{\mathrm{d}u_n}{u_n}$$

$$E_n = 2 \int_0^1 \cdots \int_0^1 \left(\prod_{1 \le j < k \le n} \frac{u_k - u_j}{u_k + u_j} \right)^2 \mathrm{d}t_2 \, \mathrm{d}t_3 \cdots dt_n.$$

In the last line $u_k = \prod_{i=1}^k t_i$.

In general, it is very difficult to compute high-precision numerical values of n-dimensional integrals such as these. But as it turn out, the C_n integrals can be converted to one-dimensional integrals, which are amenable to evaluation with the tanh-sinh scheme:

$$C_n = \frac{2^n}{n!} \int_0^\infty p K_0^n(p) \, \mathrm{d}p.$$

Here K_0 is the *modified Bessel function* [24]. 1000-digit values of these sufficed to identify the first few instances of C_n in terms of well-known constants. For example, $C_4 = 7\zeta(3)/12$, where ζ denotes the Riemann zeta function. For larger n, it quickly became clear that the C_n approach the limit

$$\lim_{n \to \infty} C_n = 0.630473503374386796122040192710 \ldots.$$

This numerical value was quickly identified, using the Inverse Symbolic Calculator 2.0 (now available at `http://carma-lx1.newcastle.edu.au:8087`), as

$$\lim_{n \to \infty} C_n = 2e^{-2\gamma},$$

where γ is Euler's constant. This identity was then proven [7].

Other specific results found in this study include the following:

$$
\begin{aligned}
D_3 &= 8 + 4\pi^2/3 - 27\,\mathrm{L}_{-3}(2) \\
D_4 &= 4\pi^2/9 - 1/6 - 7\zeta(3)/2 \\
E_2 &= 6 - 8\log 2 \\
E_3 &= 10 - 2\pi^2 - 8\log 2 + 32\log^2 2 \\
E_4 &= 22 - 82\zeta(3) - 24\log 2 + 176\log^2 2 - 256(\log^3 2)/3 \\
&\quad +16\pi^2\log 2 - 22\pi^2/3 \\
E_5 &= 42 - 1984\,\mathrm{Li}_4(1/2) + 189\pi^4/10 - 74\zeta(3) - 1272\zeta(3)\log 2 + 40\pi^2\log^2 2 \\
&\quad -62\pi^2/3 + 40(\pi^2\log 2)/3 + 88\log^4 2 + 464\log^2 2 - 40\log 2,
\end{aligned}
$$

where ζ is the Riemann zeta function and $\mathrm{Li}_n(x)$ is the polylog function.

E_5 was computed by first reducing it to a 3-D integral of a 60-line integrand, which was evaluated using tanh-sinh quadrature to 250-digit arithmetic using over 1000 CPU-hours on a highly parallel system. The PSLQ calculation required only seconds to produce the relation above. This formula remained a "numerical conjecture" for several years, but was proven in March 2014 by Erik Panzer, who mentioned that he relied on these computational results to guide his research.

1.4 Algebraic numbers in Poisson potential functions

The Poisson equation arises in contexts such as engineering applications, the analysis of crystal structures, and even the sharpening of photographic images. In two recent studies [5, 6], the present author and others explored the following class of sums:

$$\phi_n(r_1, \ldots, r_n) = \frac{1}{\pi^2} \sum_{m_1, \ldots, m_n \text{ odd}} \frac{e^{i\pi(m_1 r_1 + \cdots + m_n r_n)}}{m_1^2 + \cdots + m_n^2}. \tag{3}$$

After extensive high-precision numerical experimentation using (**??**), we discovered (then proved) the remarkable fact that when x and y are rational,

$$\phi_2(x, y) = \frac{1}{\pi} \log A, \tag{4}$$

where A is an *algebraic number*, namely the root of an algebraic equation with integer coefficients.

In our experiments we computed $\alpha = A^8 = \exp(8\pi\phi_2(x,y))$, using some rapidly convergent formulas found in [5], for various simple rationals x and y (as it turns out, computing A^8 reduces the degree of polynomials and so computational cost). Then we generated the vector $(1, \alpha, \alpha^2, \cdots, \alpha^d)$ as input to a program implementing the three-level multipair PSLQ program [8]. When successful, the program returned the vector of integer coefficients $(a_0, a_1, a_2, \cdots, a_d)$ of a polynomial satisfied by α as output. With some experimentation on the degree d, and after symbolic verification using *Mathematica*, we were able to ensure that the resulting polynomial is in fact the minimal polynomial satisfied by α. Table 1 shows some examples [5].

s	Minimal polynomial corresponding to $x = y = 1/s$:
5	$1 + 52\alpha - 26\alpha^2 - 12\alpha^3 + \alpha^4$
6	$1 - 28\alpha + 6\alpha^2 - 28\alpha^3 + \alpha^4$
7	$-1 - 196\alpha + 1302\alpha^2 - 14756\alpha^3 + 15673\alpha^4 + 42168\alpha^5 - 111916\alpha^6 + 82264\alpha^7$ $-35231\alpha^8 + 19852\alpha^9 - 2954\alpha^{10} - 308\alpha^{11} + 7\alpha^{12}$
8	$1 - 88\alpha + 92\alpha^2 - 872\alpha^3 + 1990\alpha^4 - 872\alpha^5 + 92\alpha^6 - 88\alpha^7 + \alpha^8$
9	$-1 - 534\alpha + 10923\alpha^2 - 342864\alpha^3 + 2304684\alpha^4 - 7820712\alpha^5 + 13729068\alpha^6$ $-22321584\alpha^7 + 39775986\alpha^8 - 44431044\alpha^9 + 19899882\alpha^{10} + 3546576\alpha^{11}$ $-8458020\alpha^{12} + 4009176\alpha^{13} - 273348\alpha^{14} + 121392\alpha^{15}$ $-11385\alpha^{16} - 342\alpha^{17} + 3\alpha^{18}$
10	$1 - 216\alpha + 860\alpha^2 - 744\alpha^3 + 454\alpha^4 - 744\alpha^5 + 860\alpha^6 - 216\alpha^7 + \alpha^8$

Table 1: Sample of polynomials produced in earlier study [5].

Using this data, Jason Kimberley of the University of Newcastle, Australia, conjectured a formula that gives the degree d as a function of k [5]. These computations required prodigiously high precision and correspondingly long run times. For example, finding the degree-128 polynomial satisfied by $\alpha = \exp(8\pi\phi_2(1/32, 1/32))$ required 10,000-digit precision. Other runs were attempted, but failed.

With these prodigiously high numeric precision levels and huge run times, it is clear that a highly parallel PSLQ facility, together with a parallelizable (at the application level) arbitrary precision facility, is badly needed for further research. But attempts to exploit this parallelism for such problems to date have been largely hamstrung by the fact that most available arbitrary precision software packages are not thread-safe.

2 High-precision floating-point software

By far the most common form of extra-precision arithmetic is roughly twice the level of standard 64-bit IEEE floating-point arithmetic. One option is the IEEE standard for 128-bit binary floating-point arithmetic, with 113 mantissa bits, but sadly it is not yet widely implemented in hardware, although it is supported, in software, in some compilers.

Another software option for this level of precision is "double-double" arithmetic (approximately 31-digit accuracy). This datatype consists of a pair of 64-bit IEEE floats (s, t), where s is the 64-bit floating-point value closest to the desired value, and t is the difference (positive or negative) between the true value and s. One can extend this design to quad-double arithmetic, which operates on strings of four IEEE 64-bit floats, providing roughly 62-digit accuracy. These two datatypes are supported by the QD package, which includes high-level language interfaces for C++ and Fortran (see below) [21].

For higher-levels of precision, software packages typically represent a high-precision datum as a string of floats or integers, where the first few words contain bookkeeping information and the binary exponent, and subsequent words (except perhaps near the end) contain the mantissa. For moderate precision levels (up to roughly 1000 digits), arithmetic on such data is typically performed using adaptations of familiar schemes.

Above about 1000 or 2000 decimal digits, advanced algorithms should be employed for maximum efficiency. For example, a high-precision multiply operation can be performed by noting that the key operation is merely a linear convolution, which may be performed using fast Fourier transforms (FFTs). Efficient implementations of this scheme can dramatically accelerate multiplication, since the FFT reduces an $O(n^2)$ operation to an $O(n \log n \log\log n)$ operation [13, Section 2.3] (see also Section 4.2).

2.1 Available software packages

Software for performing high-precision arithmetic has been available for quite some time, for example in the commercial packages *Mathematica* and *Maple*. However, until 10 or 15 years ago, those with applications written in more conventional languages, such as C++ or Fortran-90, often found it necessary to rewrite their codes, replacing each arithmetic operation with a subroutine call, which was a very tedious and error-prone process. Nowadays there are several freely available high-precision software packages, together with accompanying high-level language interfaces, utilizing operator overloading, that make code conversions relatively painless.

Here are some packages for high-precision floating-point computation:

- ARPREC: supports arbitrary precision real, integer and complex, with many algebraic and transcendental functions. Includes high-level interfaces for C++ and Fortran-90. Available at `http://crd-legacy.lbl.gov/~dhbailey/mpdist`.
- CLN: C++ library supporting arbitrary precision integer, real and complex, with numerous algebraic and transcendental functions. Available at `http://www.ginac.de/CLN`.
- GMP: supports high-precision integer, rational and floating-point calculations. Distributed under the GNU license by the Free Software Foundation. Available at `http://gmplib.org`.
- Julia: high-level programming environment that incorporates GMP and MPFR. Available at `http://julialang.org`.
- MPFR: supports multiple-precision floating-point computations with correct rounding, based on GMP. Includes numerous algebraic and transcendental functions, and a thread-safe build option. Available at `http://www.mpfr.org`.
- MPFR++: a high-level C++ interface to MPFR (although the currently available version is not up-to-date with MPFR). Available at `http://perso.ens-lyon.fr/nathalie.revol/software.html`.
- MPFR C++: a high-level C++ interface to MPFR with a thread-safe option. See Section 3.3 for some additional details. Available at `http://www.holoborodko.com/pavel/mpfr`.
- MPFUN90: similar to ARPREC in user-level functionality, but written entirely in Fortran-90. Includes a Fortran-90 high-level interface. Available at `http://crd-legacy.lbl.gov/~dhbailey/mpdist`.
- mpmath: a Python library for arbitrary precision floating-point arithmetic, including numerous transcendentals. Available at `https://code.google.com/p/mpmath`.

- NTL: a C++ library for arbitrary precision integer and floating-point arithmetic. Available at `http://www.shoup.net/ntl`.

- Pari/GP: a computer algebra system that includes facilities for high-precision arithmetic, with many transcendental functions. Available at `http://pari.math.u-bordeaux.fr`.

- QD: includes routines to perform "double-double" (approx. 31 digits) and "quad-double" (approx. 62 digits) arithmetic, as well as many algebraic and transcendental functions. Includes high-level interfaces for C++ and Fortran-90. Available at `http://crd-legacy.lbl.gov/~dhbailey/mpdist`.

- Sage: an open-source symbolic computing system that includes high-precision facilities. Available at `http://www.sagemath.org`.

2.2 Thread safety

The scientific computing world is moving rapidly into multicore and multi-node parallel computing, because the frequency and performance of individual processors is no longer rapidly increasing [29]. Thus it is clear that future improvements in performance on high-precision computations will only be obtained by aggressively exploiting parallelism. It is difficult to achieve significant parallel speedup *within* a single high-precision arithmetic operation, but parallelization at the *application* level, e.g., parallelizing a DO or FOR loop in an application code, is an attractive option.

It is possible to perform some high-precision computations in parallel by utilizing message passing interface (MPI) software at the application level. MPI employs a "shared none" environment that avoids many difficulties. Indeed, several high-precision applications have been performed on highly parallel systems using MPI, including the study mentioned in Section 1.3 [7].

But on modern systems that feature multicore processors, parallel computing is more efficiently performed using a shared memory, multithreaded environment such as OpenMP [29] within a single node, even if MPI is employed for parallelism between nodes. Furthermore, algorithms such as PSLQ, for example, can only be parallelized efficiently at a rather low loop level — MPI implementations are not effective here unless the precision level is exceedingly high.

Computations that use a thread-parallel environment such as OpenMP must be entirely "thread-safe," which means, among other things, that there is no read/write global data, because otherwise there may be difficulties with processors stepping on each other during parallel execution. Employing "locks" and the like may remedy such difficulties, but this reduces parallel efficiency and is problematic for code portability and installation.

One impediment to thread safety is the design of the operator overloading feature of modern computer languages, which is the only reasonable way to program a complicated high-precision calculation. Here "operator overloading" means the feature, available in several high-level languages including C++ and Fortran-90, where algebraic operators, e.g., +, -, * and /, are extended to high-precision operands. Such facilities typically do not permit one to carry information such as the current working precision level.

Most arbitrary precision packages generate a "context" of auxiliary data, such as the current working precision level and data to support transcendental function evaluation. Such data, if not handled properly, can ruin thread safety. For most high-precision computation packages, the available documentation makes no statement one way or the other (which almost certainly means that they are *not* thread-safe).

Of the packages listed in Section 2, only one is a certified thread-safe, high-level floating-point package (i.e., uses operator overloading to interface with ordinary source code), namely the MPFR C++ package [23], which is built upon the lower-level MPFR package [19]. The MPFR package in turn is very well-designed, features correct rounding to the last bit, includes numerous transcendental and special functions, and achieves the the fastest overall timings of any floating-point package in the above list [15].

According to the documentation, the MPFR package has a "thread-safe build option." When this is invoked, the package generates a context, local to the thread, to support computation of certain transcendental functions at a particular precision level, whenever a high-precision operation is initiated in that thread. This is not ideal, since it means that if one is using thread-based parallelism to parallelize a low-level loop, this context must be generated at the start of each thread and freed at the end of the thread. However, one of the developers of MPFR has promised to the present author that in a future version, this context can be generated in a one-time initialization, after which all computation will be thread-safe.

There is, to this author's knowledge, no high-level, thread-safe arbitrary precision package to support Fortran applications, prior to the present work.

While thread safety is of paramount importance, several other lessons should be noted when designing a high-precision floating-point arithmetic facility, based on the present author's experience:

- Double precision constants and expressions pose a serious problem, since they are not automatically converted to high precision due to operator precedence rules in most programming languages. A high percentage of accuracy failures reported to the present author by users of his earlier packages (ARPREC, MPFUN90 and QD) are due to the usage of inexact double precision values.

- Complicated installation scripts are problematic. Many support inquiries for the author's earlier packages have been for installation issues, not the underlying multiprecision software. Some users prefer older, out-of-date software, simply because it is easy to install.

- Special system settings, system-dependent features and exotic language features pose serious difficulties for portability and maintenance.

- As with any software activity, long-term support is a nagging issue.

3 A thread-safe arbitrary precision computation package

With this background, the present author has developed a new software package for arbitrary-precision computation, named "MPFUN2015." It is available in two versions: (a) MPFUN-Fort, a completely self-contained, all-Fortran version that is simple to install; and (b) MPFUN-MPFR, a version based on the MPFR package for lower-level operations that is more complicated to install but is approximately 3X faster for most applications. Except for a few special functions, the two versions are "plug-compatible" in the sense that applications written for one will also run with the other without any changes (see Section 6.4 for details). These versions feature:

1. Full support for both real and complex datatypes, including all basic arithmetic operations and transcendental functions. A full-featured high-level language interface for Fortran-90 is provided, so that most users need only make minor changes to existing double precision code. A C++ interface is planned but is not yet written.

2. A 100% thread safe design, even at the user language interface level, since no “context” or auxiliary data needs to be generated (unless exceedingly high precision is used). The working precision can be freely changed (up to a flexible limit) during execution, even in low-level loops, without ruining thread safety. See however Section 3.3.

3. Numerous transcendental functions and special functions, including most of the intrinsic funtions specified in the Fortran-2008 standard. See Section 6.1 for a complete listing.

4. There is no need to manually allocate or deallocate temporary multiprecision variables or arrays in subroutines — all such data works with the standard automatic array feature of Fortran, and is thread-safe.

5. No system-dependent features or settings. Proper operation does *not* depend on the IEEE rounding mode of the user application (although see item 11 below).

6. Straightforward installation, using scripts provided for various environments. MPFUN-Fort is supported with the GNU, IBM, Intel and Portland Group Fortran compilers, and MPFUN-MPFR is supported on the GNU and Intel Fortran compilers, provided that the gcc compiler is also available for installation of GMP and MPFR.

7. Precision is theoretically scalable to millions of decimal digits, although at present the MPFUN-Fort version is limited to 1.2 million digits.

8. Advanced algorithms, including FFT-based arithmetic, are employed in both versions for top performance even at very high precision levels.

9. A highly effective solution is provided to the double precision accuracy problem mentioned in Section 2.2.

10. The overall runtime performance of the MPFUN-Fort version is somewhat faster than the present author’s ARPREC package; the MPFUN-MPFR version is roughly 3X faster than either on large problems.

11. With the MPFUN-MPFR version, computations are guaranteed accurate to the last bit. However both versions perform computations to one more word (14–19 digits) of precision than requested by the user, minimizing roundoff error.

3.1 Data structure

For the MPFUN-Fort version, the structure is a $(N+6)$-long vector of 64-bit IEEE floating-point words, where N is the number of manttisa words:

- Word 0: Total space allocated for this array, in 64-bit words.
- Word 1: The working precision level (in words) associated with this data.
- Word 2: The number of mantissa words N; the sign of word 2 is the sign of the value.
- Word 3: The multiprecision exponent, base 2^{48}.
- Word 4 to $N+3$: N mantissa words (whole numbers between 0 and $2^{48}-1$).
- Word $N+4$ and $N+5$: Scratch words for internal usage.

For the MPFUN-MPFR version, the structure is a $(N+6)$-long vector of 64-bit integers:

- Word 0: Total space allocated for this array, in 64-bit words.
- Word 1: The working precision level (in bits) associated with this data.
- Word 2: The sign of the value.
- Word 3: The exponent, base 2.
- Word 4: A pointer to the first word of the mantissa, which in MPFUN-MPFR always points to Word 5.
- Word 5 to $N+4$: Mantissa words (unsigned integers between 0 and $2^{64}-1$).
- $N+5$: Not used at present.

Note that in the MPFUN-MPFR version, words 1 through $N+4$ correspond exactly to the data structure of the MPFR package.

For both versions, a complex multiprecision datatype is a contiguous pair of real multiprecision data. Note that the imaginary member of the real-imaginary pair starts at an offset in the array equal to the value of word 0. Note also that this offset is independent of the working precision.

3.2 Modules

The MPFUN-Fort software includes the following separate modules, each in its own source file:

1. MPFUNA: Contains compile-time global data. In the MPFUN-Fort version, this module includes data to support FFT-based arithmetic and binary values of $\log 2$, π and $1/\sqrt{2}$ (up to 19,500-digit precision).

2. MPFUNB (present only in MPFUN-Fort): Handles basic arithmetic functions, rounding, normalization, square roots and n-th roots. The FFT-based arithmetic facility to support very high precision computation is included in this module.

3. MPFUNC (present only in MPFUN-Fort): Handles binary-to-decimal conversion, decimal-to-binary conversion and input/output operations.

4. MPFUND (present only in MPFUN-Fort): Includes routines for all common transcendental constants and functions, as well as special routines, implementing advanced algorithms, for very high precision levels.

5. MPFUNE (present only in MPFUN-Fort): Includes routines for special functions, such as the BesselJ, gamma, incomplete gamma and zeta functions.

6. MPFUNF: Defines the default (maximum) precision. This is the only module that needs to be modified by the user.

7. MPFUNG: A high-level user interface that connects to user code via operator overloading and function extension.

8. MPMODULE: The main module that references the others and is referenced by the user.

3.3 The MPFUN2015 solution to thread safety

All of the software modules above are 100% thread safe. There are no global parameters or arrays, except for static, compile-time data, and no initialization is required unless extremely high precision is required. For both the MPFUN-Fort and MPFUN-MPFR versions, working precision level is passed as a subroutine argument, ensuring thread safety. For the MPFUN-MPFR version, for the time being thread safety cannot be guaranteed unless the user's code does not involve transcendental functions in multi-threaded sections of code. This limitation will be removed in a future release.

Thread safety at the language interface or user level in both versions is achieved by assigning a working precision level to *each multiprecision datum*, which then is passed through the multiprecision software. Note, in the data structure for both MPFUN-Fort and MPFUN-MPFR given in Section 3.1 above, that word 1 (the second word of the array) is the working precision level associated with that datum. This solves the thread safety problem when precision is dynamically changed in the application, although it requires a somewhat different programming style, as we shall briefly explain here (see Section 6.4 for additional details).

To use either version from a Fortran program, the user first sets the parameter `mpipl`, the "default" precision level in digits, which is the maximum precision level to be used for subsequent computation, and is used to set the amount of storage required for multiprecision data. `mpipl` is set in a parameter statement at the start of module MPFUNF, which is in file mpfunf.f90. In the code as distributed, `mpipl` is set to 1200 digits (sufficient to run the six test problems of Section 7), but it can be set to any level greater than or equal to 30 digits. `mpipl` is converted to mantissa words (parameter `mpwds`), which is the internal default precision level.

All computations are performed to `mpwds` words precision unless the user, in an application code, specifies a lower value. During execution, the user can change the working precision assigned to a multiprecision variable or array by using the built-in functions `mpreal` and `mpcmplx` (see Section 6.4 for details). The working precision level assigned to a particular multiprecision variable or array element can be monitored using the built-in function `mpwprec`.

During execution, the result of any operation involving multiprecision variables or array elements "inherits" the working precision level of the input operands (if the operands have different working precision levels, the higher precision level is chosen for the result). When assigning a multiprecision variable or array element to double precision constant or variable, or when reading multiprecision data from a file, the result is assigned the default precision unless a lower precision level is specified.

4 Numerical algorithms used in MPFUN-Fort

As mentioned above, MPFUN-MPFR relies on MPFR for all lower-level arithmetic operations and transcendental functions, whereas MPFUN-Fort is completely self-contained. The algorithms employed in the MPFR package are described in [19]. This section presents a brief overview of the algorithms used in MPFUN-Fort. Those readers primarily interested in MPFUN-MPFR may skip to Section 5.

4.1 Algorithms for basic arithmetic

Multiplication. For modest levels of precision, MPFUN-Fort employs adaptations of the usual schemes we all learned in grade school, where the number base is $2^{48} = 281474976710656$ instead of ten. In the inner loop of the multiplication routine division routines, note that two such numbers a and b in the range $[0, 2^{48})$ must be multiplied, obtaining the exact 96-bit result.

This is performed in MPFUN-Fort by splitting both a and b into high- and low-order 24-bit parts, using the sequence of operations $a_1 = 2^{24}\,\mathrm{int}\,(2^{-24}a)$, $a_2 = a - a_1$. Note that $a = a_1 + a_2$. If the four half-sized values are a_1, a_2, b_1, b_2, then calculate $c = a_1 \cdot b_2 + a_2 \cdot b_1$, then $c_1 = 2^{48}\,\mathrm{int}\,2^{-48}(c)$, $c_2 = c - c_1$. Finally calculate $d_1 = 2^{-48}(a_1 \cdot b_1 + c_1)$, $d_2 = a_2 \cdot b_2 + c_2$. Then d_1 and d_2 are the high- and low-order 48-bit mantissa words of the product, with the proviso that although $d = d_1 + d_2 = a \cdot b$ is mathematically correct, d_1 and d_2 might not be the precisely correct split into two words in the range $[0, 2^{48})$. However, since a double precision datatype can exactly hold whole numbers up to 2^{53} in size, no accuracy is lost. In fact, this operation can be repeatedly done in a loop, provided that this data is periodically corrected in a normalization operation. Note also that the splitting of data in at least one of the two arguments at the beginning can be done outside the inner loop.

The resulting scheme is very efficient yet totally reliable — in effect, it performs quad precision with only a handful of register-level operations. Note also that if two n-word arguments are multiplied, and the working precision is also n words, then since only an n-word result is returned, only slightly more than half of the "multiplication pyramid" need be calculated.

Division. A similar approach can be taken to division. Again, the key step is to inexpensively simulate quad precision in the inner loop, using the scheme outlined above.

Square roots. Square roots are calculated by the following Newton-Raphson iteration, which converges to $1/\sqrt{a}$ [10, pg. 227]:

$$x_{k+1} = x_k + 1/2 \cdot (1 - x_k^2 \cdot a) \cdot x_k, \tag{5}$$

where the multiplication $() \cdot x_k$ is performed with only half of the normal level of precision. These iterations are performed with a working precision level that approximately doubles with each iteration, except that at three iterations before the final iteration, the iteration is repeated without doubling the precision, in order to enhance accuracy. The final iteration is performed as follows (due to A. Karp):

$$\sqrt{a} \approx (a \cdot x_n) + 1/2 \cdot [a - (a \cdot x_n)^2] \cdot x_n, \tag{6}$$

where the multiplications $(a \cdot x_n)$ and $[] \cdot x_n$ are performed with only half the final level of precision. If this is done properly, the total cost of the calculation is only about three times the cost of a single full-precision multiplication.

n-th roots. A similar scheme is used to compute n-th roots for any integer n. Computing x_k^n, which is required here, can be efficiently performed using the binary algorithm for exponentiation. This is merely the observation that exponentiations can be accelerated based on the binary expansion of the exponent: for example, 3^{17} can be computed as $((((3)^2)^2)^2)^2 \cdot 3 = 129140163$.

Note that these algorithms are trivially thread-safe, since no auxiliary data is involved.

4.2 FFT-based multiplication

Although the multiplication algorithm described above is very efficient, for higher levels of precision (above approximately 2500 digits, based on the present author's implementation), significantly faster performance can be achieved by employing an FFT-convolution approach [13][10, pg. 223–224].

Suppose one wishes to multiply two n-precision values whose mantissa words are given by $a = (a_0, a_1, a_2, \cdots, a_{n-1})$ and $b = (b_0, b_1, b_2, \cdots, b_{n-1})$. It is easy to see that the desired result, except for releasing carries, is an acyclic convolution. In particular, assume that a and b are extended to $2n$ words each by padding with zeroes. Then the product $c = (c_k)$ is given by

$$c_k = \sum_{j=0}^{2n-1} a_j b_{k-j}, \qquad 0 \le k < 2n, \tag{7}$$

where b_{k-j} is read as b_{k-j+2n} when $k-j$ is negative. This convolution can be calculated as $(c) = F^{-1}[F(a) \cdot F(b)]$, where $F(a)$ and $F(b)$ denote a real-to-complex discrete Fourier transform (computed using an FFT algorithm), the dot means element-by-element complex multiplication, and $F^{-1}[]$ means an inverse complex-to-real FFT. The c_k results from this process are floating-point numbers. Rounding these values to the nearest integer, and then releasing carries beginning at c_{2n-1} gives the desired multiplication result.

The FFT-based multiplication facility of the present MPFUN-Fort software supports a precision level up to approximately 1.2 million digits. Beyond this level, numerical round-off error in the FFT is too great to reliably round the c_k results to the nearest integer. If the maximum rounding error exceeds 0.375 (beyond which is deemed unsafe), an error message is output. A planned future enhancement to MPFUN-Fort will extend the usable precision level to at least 100 million decimal digits.

In contrast to the basic arithmetic algorithms, FFT-based multiplication requires precomputed FFT root-of-unity data. However, all the requisite FFT data to support any precision level up to 19,500 digits is stored as static data in module MPFUNA. Thus up to this level, MPFUN-Fort requires no initialization, and is completely thread-safe (since no "context" must be created or freed). If even higher precision is required, the requisite FFT data is generated by calling `mpinit` — see Table 4 and Section 6.4 for details — after which all computations are completely thread-safe.

4.3 Advanced algorithm for division

With an FFT-based multiplication facility in hand, division of two extra-high-precision arguments a and b can be performed by the following scheme. This Newton-Raphson algorithm iteration converges to $1/b$ [10, pg. 226]:

$$x_{k+1} = x_k + (1 - x_k \cdot b) \cdot x_k, \tag{8}$$

where the multiplication $() \cdot x_k$ is performed with only half of the normal level of precision. These iterations are performed with a working precision level that is approximately doubles with each iteration, except that at three iterations before the final iteration, the iteration is repeated without doubling the precision, in order to enhance accuracy. The final iteration is performed as follows (due to A. Karp):

$$a/b \approx (a \cdot x_n) + [a - (a \cdot x_n) \cdot b] \cdot x_n, \tag{9}$$

where the multiplications $a \cdot x_n$ and $[] \cdot x_n$ are performed with only half of the final level of precision. The total cost of this procedure is only about three times the cost of a single full-precision multiplication.

4.4 Basic algorithms for transcendental functions

Most arbitrary precision packages require a significant "context" of data to support transcendental function evaluation at a particular precision level, and this data is often problematic for both thread safety and efficiency. For example, if this context data must be created and freed within each running thread, this limits the efficiency in a multithreaded environment. With this in mind, the transcendental function routines in MPFUN-Fort were designed to require only a minimum of context, which context is provided in static data statements, except when extremely high precision is required.

Exponential and logarithm. In the current implementation, the exponential function routine in MPFUN-Fort first reduces the input argument to within the interval $(-\log(2)/2, \log(2)/2]$. Then it divides this value by 2^q, producing a very small value, which is then input to the Taylor series for $\exp(x)$. The working precision used to calculate the terms of the Taylor series is reduced as the terms get smaller, thus saving approximately one-half of the total run time. When complete, the result is squared q times, and then corrected for the initial reduction. In the current implementation, q is set to the nearest integer to $(48n)^{2/5}$, where n is the number of words of precision.

Since the Taylor series for the logarithm function converges much more slowly than that of the exponential function, the Taylor series is not used for logarithms unless the argument is extremely close to one. Instead, logarithms are computed based on the exponential function, by employing the following Newton iteration with a level of precision that approximately doubles with each iteration:

$$x_k = x_k - \frac{e^x - a}{e^x}. \qquad (10)$$

Trigonometric functions. The sine/cosine routine first reduces the input argument to within the interval $(-\pi, \pi]$. This value is then divided by 2^q and then input to the Taylor series for $\sin(x)$, with a linearly varying precision level as above. Then the double-angle formulas

$$\cos(2x) = 1 - 2\sin^2(x) \tag{11}$$
$$\cos(2x) = 2\cos^2(x) - 1, \tag{12}$$

are applied q times (formula (11) is used once, and (12) thereafter). In the current implementation, q is set to the greatest integer in $\sqrt{2N}$, where N is the precision in bits, unless the reduced argument is very close to one, in which case $q = 0$. When complete, $\sin(x)$ is computed as $\sqrt{1 - \cos^2(x)}$, with corrected sign, except for the case $q = 0$, when $\cos(x)$ is computed as $\sqrt{1 - \sin^2(x)}$.

The inverse cos/sin function is based on the sine routine, by employing a Newton iteration with a level of numeric precision that roughly doubles with each iteration.

Power function. The power function, namely a^b for real $a > 0$ and b, can be computed as $e^{b \log a}$. To further accelerate this operation, the MPFUN-Fort software first examines the value of b to see if it is a rational number with numerator and denominator up to 10^7 size, using the extended Euclidean algorithm performed in double precision. If it is, a^b is performed using a combination of the binary algorithm for exponentiation for the numerator, and the n-th root function for the denominator.

Note that for all of the above algorithms, the only context required is the numerical values of $\log(2), \pi$ and, if FFT-based multiplication is employed, the requisite FFT root-of-unity data. For precision levels up to approximately 19,500 digits, these values are stored as static data in module MPFUNA. If higher precision is required, this data is generated by calling subroutine `mpinit` — see Table 4 and Section 6.4 for details.

4.5 Advanced algorithms for transcendental functions

The above transcendental function schemes are adequate for modest levels of precision. For higher levels of precision, advanced, quadratically convergent algorithms are employed.

Logarithm and exponential. To compute logarithms for higher levels of precision, MPFUN-Fort employs an algorithm due to Richard Brent [12]: Inputs t that are extremely close to 1 are handled using a Taylor series. Otherwise, let n be the number of bits of precision required in the result. If t is exactly two, select $m > n/2$. Then the following formula gives log(2) to the required precision:

$$\log(2) = \frac{\pi}{2mA(1, 4/2^m)}. \tag{13}$$

Here $A(a, b)$ is the limit of the arithmetic-geometric mean (AGM): Let $a_0 = a$ and $b_0 = b$; then iterate

$$a_{k+1} = (a_k + b_k)/2, \qquad b_{k+1} \; = \; \sqrt{a_k b_k} \tag{14}$$

until convergence. For other t, select m such that $s = t2^m > 2^{n/2}$. Then the following formula gives $\log t$ to the required precision:

$$\log(t) = \frac{\pi}{2A(1, 4/s)} - m \log(2). \tag{15}$$

Given this algorithm for logarithms, high-precision exponentials can be calculated using the following Newton iteration, using a precision level that approximately doubles with each iteration as before:

$$x_{k+1} = x_k - x_k(\log x_k - a). \tag{16}$$

Trigonometric functions. Trigonometric functions and their inverses can be calculated by applying the above algorithms for a complex argument and recalling Euler's formula

$$e^{r+i\theta} = e^r(\cos(\theta) + i\sin(\theta)). \tag{17}$$

The complex logarithm thus gives the inverse trigonometric functions. In particular, given the complex argument $x + iy$, the complex logarithm gives r and θ (with θ in the interval $(-\pi, \pi]$) such that $r\cos(\theta) = x$ and $r\sin(\theta) = y$. The complex exponential function, which gives cos and sin, can be computed from the complex logarithm function using Newton iterations as before.

Along this line, π itself can be computed using an AGM-based scheme due to Brent and Salamin [12]. Set $a_0 = 1$, $b_0 = 1/\sqrt{2}$, and $d_0 = \sqrt{2} - 1/2$. Then iterate the following starting with $k = 1$ until convergence: with $k = 1$:

$$\begin{aligned} a_k &= (a_{k-1} + b_{k-1})/2 \\ b_k &= \sqrt{a_{k-1} b_{k-1}} \\ d_k &= d_{k-1} - 2^k (a_k - b_k)^2. \end{aligned} \tag{18}$$

Then $p_k = (a_k + b_k)^2/d_k$ converges to π.

Each of these advanced algorithms, which are based on the AGM, is "quadratically convergent" — successive iterations approximately doubles the number of correct digits.

Based on the present author's implementation, the advanced exponential function algorithm is faster than the conventional routine beginning at about 5,800 digits. However, the advanced logarithm scheme is faster than the conventional logarithm after only 430 digits. Sadly, although the advanced trigonometric function algorithm is not faster until above approximately 1,000,000 digits, the advanced inverse trigonometric routine is faster after only 1,500 digits.

Euler's gamma constant. A function is also provided to compute Euler's constant $\gamma =$. Euler's constant γ was calculated using the following formula, which is an improvement of a technique previously used by Sweeney [27]. If a result accurate to at least B bits is desired, first select the integer $N = \lceil \log_2(B \log 2) \rceil$. Then

$$\gamma \approx \frac{2^N}{e^{2^N}} \sum_{m=0}^{\infty} \frac{2^{mN}}{(m+1)!} \sum_{t=0}^{m} \frac{1}{t+1} \; - \; N \log 2. \tag{19}$$

The error in this approximation is less than $1/(2^N e^{2^N})$. This algorithm is *not* quadratically convergent, but appears to converge reasonably rapidly. Unlike π and $\log 2$, the value of γ is not stored — it must be calculated explicitly by the user, using the function `egamma` (see Table 4).

Note that none of the above algorithms requires any context, except for the numerical values of $\log 2$ and π, which, as noted above, are stored in the program code itself for precision levels up to 19,500 digits.

4.6 Special functions

Modern mathematical and scientific computing frequency often involves other, more sophisticated functions, which collectively are termed "special functions" [16]. A number of these functions are planned for future inclusion in MPFUN-Fort, and will be added to the version as they are developed. Here is a brief description of the functions that have been implemented and the algorithms employed. In each case, care is taken to preserve thread safety, and to avoid, as far as possible, any need to precalculate auxiliary data.

BesselJ function. The BesselJ function, or, more formally, the Bessel function of the first kind, is defined as [16, 10.2.2]:

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z^2}{4}\right)^k}{k!\Gamma(\nu + k + 1)}. \tag{20}$$

For modest-sized values of z (the present author uses the condition $z < 1.73d$, where d is the precision level in digits), the MPFUN-Fort BesselJ function implements formula (20). Note that although (20) involves the gamma function (see below), this is only called once to compute $\Gamma(\nu + 1)$, after which the recursion $\Gamma(t+1) = t\Gamma(t)$ can be applied for the other terms.

For larger values of z, the following asymptotic formula is employed [16, 10.17.3]. Define $\omega = z - \nu\pi/2 - \pi/4$, and

$$a_k(\nu) = \frac{(4\nu^2 - 1^2)(4\nu^2 - 3^2) \cdots (4\nu^2 - (2k-1)^2)}{k!8^k}. \tag{21}$$

Then

$$J_\nu(z) = \left(\frac{2}{\pi z}\right)^{1/2} \left(\cos\omega \sum_{k=0}^{\infty} \frac{(-1)^k a_{2k}(\nu)}{z^2 k} - \sin\omega \sum_{k=0}^{\infty} \frac{(-1)^k a_{2k+1}(\nu)}{z^{2k+1}}\right). \tag{22}$$

One important detail omitted from the above discussion is that large amounts of cancellation occurs in these formulas. Thus when evaluating these formulas, a working precision of 1.5 times the normal working precision is employed.

No precalculated auxiliary data is needed for either of these algorithms, so they are thread safe.

Gamma function. The gamma function employs a very efficient but little-known formula due to Ronald W. Potter [25], as follows. If the input t is a positive integer, then $\Gamma(t) = (t-1)!$. If not, use the recursion $\Gamma(t+1) = t\Gamma(t)$ to reduce the argument (positive or negative) to the interval $(0, 1)$. Then define $\alpha = \text{nint}\,(n/2 \cdot \log 2)$, where n is the number of bits of precision and nint means nearest integer, and set $z = \alpha^2/4$. Define the Pochhammer function as

$$(\nu)_k = \nu(\nu + 1)(\nu + 2) \cdots (\nu + k - 1). \tag{23}$$

Then define the functions

$$A(\nu, z) = \left(\frac{z}{2}\right)^{\nu} \nu \sum_{k=0}^{\infty} \frac{(z^2/4)^k}{k!(\nu)_{k+1}}$$

$$B(\nu, z) = \left(\frac{z}{2}\right)^{-\nu} (-\nu) \sum_{k=0}^{\infty} \frac{(z^2/4)^k}{k!(-\nu)_{k+1}}. \tag{24}$$

With these definitions, the gamma function can then be computed as

$$\Gamma(\nu) = \sqrt{\frac{A(\nu, z)}{B(\nu, z)} \frac{\pi}{\nu \sin(\pi\nu)}}. \tag{25}$$

No auxiliary data is needed for this algorithm, so it is thread-safe.

Incomplete gamma function. For modest-sized positive arguments (the author uses the condition $z < 2.768d$, where d is the precision level in digits), the MPFUN-Fort incomplete gamma function is evaluated using the following formula [16, 8.7.3]:

$$\Gamma(a, z) = \Gamma(a) \left(1 - \frac{z^a}{e^z} \sum_{k=0}^{\infty} \frac{z^k}{\Gamma(a + k + 1)}\right). \tag{26}$$

Note, as with the BesselJ function, that although formula (26) involves the gamma function, this is only called once to compute $\Gamma(a + 1)$, after which the recursion $\Gamma(t + 1) = t\Gamma(t)$ can be applied for all other terms.

For large values of z, the following asymptotic formula is used [16, 8.11.2]:

$$\Gamma(a, z) \approx \frac{z^{a-1}}{e^z} \sum_{k=0}^{\infty} \frac{(-1)^k (1 - a)_k}{z^k} \tag{27}$$

No auxiliary data is needed for this algorithm, so it is thread-safe.

Riemann zeta function. For large positive arguments s (the present author uses the condition $s > 2.303d/\log(2.215d)$, where d is the precision in digits), it suffices to use the definition of zeta, namely

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}. \tag{28}$$

For modest-sized arguments, the zeta function can be evaluated by means of this formula, due to Peter Borwein [11]. Select n to be the number of digits of precision required for the result. Define

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} - 2^n \right), \tag{29}$$

where the empty sum is zero. Then

$$\zeta(s) \approx \frac{-1}{2^n(1-2^{1-s})} \sum_{j=0}^{2n-1} \frac{e_j}{(j+1)^s}. \tag{30}$$

The above formulas are used for positive real arguments (except $s = 1$, for which the zeta function is undefined). For negative s, the following "reflection formula" is used to convert the calculation to a positive argument:

$$\zeta(s) = \frac{2\cos(\pi(1-s)/2)\Gamma(1-s)\zeta(1-s)}{(2\pi)^{1-s}}. \tag{31}$$

Formulas (29), (30) and (31) are implemented as the zeta function. No auxiliary data for this algorithm required, so it is thread-safe.

A even faster algorithm, based on the Euler-Maclaurin summation formula, can be derived from the following [16, 25.2.9]: Select an integer parameter $N > 0$ (the present author uses $N = 0.6d$, where d is the number of digits of precision). Then

$$\zeta(s) \approx \sum_{k=1}^{N} \frac{1}{k^s} + \frac{1}{(s-1)N^{s-1}} - \frac{1}{2N^s} + \sum_{k=1}^{\infty} \binom{s+2k-2}{2k-1} \frac{B_{2k}}{2kN^{s-1+2k}}, \tag{32}$$

where B_{2k} are the even Bernoulli numbers

$$B_{2k} = \frac{(-1)^{k-1}2(2k)!\zeta(2k)}{(2\pi)^{2k}}. \tag{33}$$

Since the zeta function evaluations in (33) are for positive even integer arguments, they can be calculated quickly using (30). Once the requisite even Bernoulli numbers B_{2k} (up to index k approximately matching the precision level in digits) are computed, the function that implements formula (32) is thread-safe.

5 Installation, compilation and linking

Installation, compilation and linking is relatively straightforward, provided that you have a Unix-based system, such as Linux or Apple OSX, with a command-line interface (such as the Terminal application of Apple OSX). For Apple OSX systems, you first must install the "Command Line Tools" package, which is available (for free) from the Apple Developer website. Instructions for doing this are provided in the README.txt file in the distribution package for either MPFUN-Fort or MPFUN-MPFR.

To install the MPFUN-MPFR version, you must first install the GMP and MPFR packages. The latest versions are available from `https://gmplib.org` and `http://www.mpfr.org/mpfr-current/`, respectively. Instructions for installing GMP and MPFR are included in the README.txt file for MPFUN-MPFR.

The gfortran compiler, which is highly recommended for either version of MPFUN2015, is available (for free) for a variety of systems at the website `https://gcc.gnu.org/wiki/GFortranBinaries`.

The MPFUN-Fort version also works with IBM's xlf compiler, Intel's ifort and Portland Group's pgf90. The MPFUN-MPFR version also works with Intel's ifort (it has not yet been tested on the other two). Compile-link scripts for each supported compiler are provided in the distribution software directory.

Each version of the software comes in two variants:

- Variant 1: This is recommended for basic applications that do not dynamically change the precision level (or do so only rarely).
- Variant 2: This is recommended for more sophisticated applications that dynamically change the precision level (see Section 6.4 below).

As an example, if one has installed the GNU gfortran compiler, then variant 1 can be compiled by typing

```
./gnu-complib1.scr
```

Then the application program progx.f90 can be compiled and linked with variant 1 of the library, producing the executable progx, by typing

```
./gnu-complink1.scr progx
```

These scripts assume that the user program is in the same directory as the library files; this can easily be changed by editing the script files.

6 Fortran coding instructions

A high-level Fortran interface is provided for both versions of the package. A C++ interface is planned but not complete.

As mentioned above, to use either version of the MPFUN2015 package, the user first sets the parameter `mpipl`, the "default" precision level in digits, which is the maximum precision level to be used for subsequent computation, and is used to specify the amount of storage required for multiprecision data. `mpipl` is set in a parameter statement at the start of module MPFUNF, which is in file mpfunf.f90. In the code as distributed, `mpipl` is set to 1200 digits (sufficient to run the six test programs of Section 7), but it can be set to any level greater than or equal to 32 digits. `mpipl` is automatically converted to mantissa words by the formula `mpwds = int (mpipl / mpdpw + 2)`, where `mpdpw` is a system parameter, and where `int ()` means truncate to integer. For MPFUN-Fort, `mpdpw` is $\log_{10}(2^{48}) = 14.44943979187\ldots$, whereas for MPFUN-MPFR it is $\log_{10}(2^{64}) = 19.26591972249\ldots$ (both values are double precision approximations). The resulting parameter `mpwds` is the internal default precision level, in words. All computations are performed to `mpwds` words precision unless the user, in an application code, specifies a lower value.

After setting the value of `mpipl` in module MPFUNF, compile either variant 1 or variant 2 of the library, using one of the scripts mentioned above.

Next, place the following line in every subprogram of the user's application code that contains a multiprecision variable or array, at the beginning of the declaration section, before any implicit or type statements:

```
use mpmodule
```

To designate a variable or array as multiprecision real (MPR) in an application program, use a Fortran-90 type statement with the type `mp_real`, as in this example:

```
type (mp_real) a, b(m), c(m,n)
```

Similarly, to designate a variable or array as multiprecision complex (MPC), use a type statement with the type `mp_complex`. Thereafter when one of these variables or arrays appears, as in the code

```
d = a + b(i) * sqrt(3.d0 - c(i,j))
```

the proper underlying multiprecision routines are automatically called.

Most common mixed-mode combinations (arithmetic operations, comparisons and assignments) involving MPR, MPC, double precision (DP), double complex (DC), and integer operands are supported. A complete list of supported mixed-mode operations is given in Table 2. See Section 6.3 below about DP and DC constants and expressions.

Input/output of MP variables or array elements is done using special subroutines. See Table 6.1 and Section 6.2 for details.

In the MPFUN-Fort version, the above instructions apply if the precision level, namely `mpipl`, is 19,500 digits or less. For higher precision, in addition to changing `mpipl` to this higher level, one must call `mpinit` at the start of execution, before any multiprecision computation is done. If this is a multithreaded application, this initialization must be done in single-threaded mode. With variant 1, subroutine `mpinit` has an optional argument, which is the maximum precision level, in words; if not present, the default precision, namely `mpwds` words (which corresponds to `mpipl` digits), is assumed. In variant 2, this argument is required. See Section 6.4 for details. When the initialization feature of MPFR is available, this same call will apply to the MPFUN-MPFR version.

Note in particular that for the time being, computations performed using the MPFUN-MPFR version that involve transcendental functions are *not* thread-safe, unless one has built the MPFR library with the thread-safe build option. This limitation will be removed in a future release of the package.

6.1 Functions and subroutines

Most Fortran-2008 intrinsic functions [18] are supported in MPFUN2015 with MPR and MPC arguments, as appropriate. A full listing of these functions is shown in Table 3. In each case, these functions represent a straightforward extension to MPR or MPC arguments, as indicated. Tables 4 and 5 present a list of additional functions and subroutines provided in this package. In these tables, “F” denotes function, “S” denotes subroutine, “MPR” denotes multiprecision real, “MPC” denotes multiprecision complex, “DP” denotes double precision, “DC” denotes double complex, “Int” denotes integer and “Q” denotes IEEE quad precision (i.e., real*16), if supported. The variable names `r1,r2,r3` are MPR, `z1` is MPC, `d1` is DP, `dc1` is DC, `i1,i2,i3` are integers, `s1` is character*1, `sn` is character*n for any n, and `rr` is MPR of length `i1`.

Operator	Arg 1	Arg 2	Operator	Arg 1	Arg 2
`a = b`	MPR	MPR	`+, -, *, /`	MPR	MPR
(assignment)	DP	MPR	$(+,-,\times,\div)$	DP	MPR
	Int	MPR		MPR	DP
	MPR	MPC		Int	MPR
	MPC	MPR		MPR	Int
	MPC	MPC		MPC	MPC
	DP	MPC		DP	MPC
	DC	MPC		MPC	DP
	MPR	DP [1]		DC	MPC
	MPR	Int [1]		MPC	DC
	MPR	Char [1]		MPR	MPC
	MPC	DP [1]		MPC	MPR
	MPC	DC [1]			
`a**b`	MPR	Int	`==, /=`	MPR	MPR
(a^b)	MPR	MPR	($=, \neq$ tests)	DP	MPR
	MPC	Int		MPR	DP
	MPC	MPC		Int	MPR
	MPR	MPC		MPR	Int
	MPC	MPR		MPC	MPC
`<=, >=, <, >`	MPR	MPR		DP	MPC
($\leq, \geq, <, >$ tests)	DP	MPR		MPC	DP
	MPR	DP		DC	MPC
	Int	MPR		MPC	DC
	MPR	Int		MPR	MPC
				MPC	MPR

Table 2: Supported mixed-mode operator combinations. MPR denotes multiprecision real, MPC denotes multiprecision complex, DP denotes double precision, DC denotes double complex, Int denotes integer and Char denotes arbitrary-length character string. Note:
[1] These operations are not allowed in variant 2 — see Section 6.4.

Type	Name	Description
MPR	`abs(r1)`	Absolute value
MPR	`abs(z1)`	Absolute value of complex arg
MPR	`acos(r1)`	Inverse cosine
MPR	`acosh(r1)`	Inverse hyperbolic cosine
MPR	`aimag(r1)`	Imaginary part of complex arg
MPR	`aint(r1)`	Truncate real to integer
MPR	`anint(r1)`	Round to closest integer
MPR	`asin(r1)`	Inverse sine
MPR	`asinh(r1)`	Inverse hyperbolic sine
MPR	`atan(r1)`	Inverse tangent
MPR	`atan2(r1,r2)`	Arctangent with two args
MPR	`atanh(r1)`	Inverse hyperbolic tangent
MPR	`bessel_j0(r1)`	Bessel function of the first kind, order 0
MPR	`bessel_j1(r1)`	Bessel function of the first kind, order 1
MPR	`bessel_jn(i1,r1)`	Besel function of the first kind, order `i1`
MPR	`bessel_y0(r1)`	Bessel function of the second kind, order 0 [1]
MPR	`bessel_y1(r1)`	Bessel function of the second kind, order 1 [1]
MPR	`bessel_yn(i1,r1)`	Besel function of the second kind, order `i1` [1]
MPC	`conjg(z1)`	Complex conjugate
MPR	`cos(r1)`	Cosine of real arg
MPC	`cos(z1)`	Cosine of complex arg
MPR	`cosh(r1)`	Hyperbolic cosine
DP	`dble(r1)`	Convert MPR argument to DP
DC	`dcmplx(z1)`	Convert MPC argument to DC
MPR	`erf(r1)`	Error function
MPR	`erfc(r1)`	Complementary error function
MPR	`exp(r1)`	Exponential function of real arg
MPR	`exp(z1)`	Exponential function of complex arg
MPR	`gamma(r1)`	Gamma function
MPR	`hypot(r1,r2)`	Hypotenuse of two args
MPR	`log(r1)`	Natural logarithm of real arg
MPR	`log(z1)`	Natural logarithm of complex arg
MPR	`log_gamma(r1)`	Log gamma function [1]
MPR	`max(r1,r2)`	Maximum of two (or three) args
MPR	`min(r1,r2)`	Minimum of two (or three) args
MPR	`mod(r1,r2)`	Fortran mod function = `r1 - r2 * int (r1 / r2 )`
MPR	`sign(r1,r2)`	Transfer of sign from `r2` to `r1`
MPR	`sin(r1)`	Sine function of real arg
MPC	`sin(z1)`	Sine function of complex arg
MPR	`sqrt(r1)`	Square root of real arg
MPC	`sqrt(z1)`	Square root of complex arg
MPR	`sinh(r1)`	Hyperbolic sine
MPR	`tan(r1)`	Tangent function
MPR	`tanh(r1)`	Hyperbolic tangent function

Table 3: Fortran-2008 intrinsic functions extended to multiprecision. Notes: [1]: Available in MPFUN-MPFR.

Type	Name	Description
F(MPC)	`mpcmplx(r1,r2)`	Converts (`r1,r2`) to MPC [1]
F(MPC)	`mpcmplx(dc1)`	Converts DC arg to MPC [1]
F(MPC)	`mpcmplx(z1)`	Converts MPC arg to MPC [1]
F(MPC)	`mpcmplxdc(dc1)`	Converts DC to MPC, without checking [1, 2]
S	`mpcssh(r1,r2,r3)`	Returns both cosh and sinh of `r1`, in the same time as calling just cosh or just sinh
S	`mpcssn(r1,r2,r3)`	Returns both cos and sin of `r1`, in the same time as calling just cos or just sin
S	`mpeform(r1,i1,i2,s1)`	Converts `r1` to char*1 string in Ei1.i2 format, suitable for output (Sec. 6.2)
S	`mpfform(r1,i1,i2,s1)`	Converts `r1` to char*1 string in Fi1.i2 format, suitable for output (Sec. 6.2)
F(MPR)	`mpegamma()`	Returns Euler's γ constant [1]
S	`mpinit()`	Initializes for extra-high precision (Sec. 6) [1]
F(MPR)	`mplog2()`	Returns log(2) [1]
F(MPR)	`mpnrt(r1,i1)`	Returns the `i1`-th root of `r1`
F(MPR)	`mppi()`	Returns π [1]
F(MPR)	`mpprod(r1,d1)`	Returns `r1*d1`, without checking [2]
F(MPR)	`mpquot(r1, d1)`	Returns `r1/d1`, without checking [2]
S	`mpread(i1,r1)`	Inputs `r1` from Fortran unit `i1`; up to five MPR args may be listed (Sec. 6.2) [1]
S	`mpread(i1,z1)`	Inputs `z1` from Fortran unit `i1`; up to five MPC args may be listed (Sec. 6.2) [1]
F(MPR)	`mpreal(r1)`	Converts MPR arg to MPR [1]
F(MPR)	`mpreal(z1)`	Converts MPC arg to MPR [1]
F(MPR)	`mpreal(d1)`	Converts DP arg to MPR [1, 2]
F(MPR)	`mpreal(q1)`	Converts real*16 to MPR [1, 2]
F(MPR)	`mpreal(s1,i1)`	Converts char*1 string to MPR (Sec. 6.2) [1]
F(MPR)	`mpreal(sn)`	Converts char*n string to MPR (Sec. 6.2) [1]
F(MPR)	`mpreald(d1)`	Converts DP to MPR, without checking [1, 2]
F(Int)	`mpwprec(r1)`	Returns precision in words assigned to r1
F(Int)	`mpwprec(z1)`	Returns precision in words assigned to z1
S	`mpwrite(i1,i2,i3,r1)`	Outputs `r1` in Ei2.i3 format to unit `i1`; up to five MPR args may be listed (Sec. 6.2)
S	`mpwrite(i1,i2,i3,z1)`	Outputs `z1` in Ei2.i3 format to unit `i1`; up to five MPC args may be listed (Sec. 6.2)
F(Q)	`qreal(r1)`	Converts MPR to real*16

Table 4: Additional general routines (F: function, S: subroutine). Notes:
[1]: In variant 1, an integer precision level argument (mantissa words) may optionally be added as the final argument; this argument is required in variant 2. See Section 6.4.
[2]: These do not check DP or DC values. See Section 6.3.

Type	Name	Description
F(MPR)	`agm(r1,r2)`	Arithmetic-geometric mean
F(MPR)	`airy r1)`	Airy function [3]
S	`berne(i1,rr)`	Array of first `i1` even Bernoulli numbers [1, 2]
F(MPR)	`besselj(r1,r2)`	BesselJ function with two MPR arguments [2]
F(MPR)	`digamma(r1)`	Digamma function [3]
F(MPR)	`expint(r1)`	Exponential integral function [3]
F(MPR)	`gammainc(r1,r2)`	Incomplete gamma function [2]
F(MPR)	`polylog(n1,r1)`	Polylogarithm function [3, 4]
F(MPR)	`zeta(r1)`	Zeta function
F(MPR)	`zetaem(n1,rr,r1)`	Zeta function with precomputed even Bernoulli numbers [2]

Table 5: Additional special functions. Notes:
[1]: In variant 1, an integer precision level argument (mantissa words) may optionally be added as the final argument; this argument is required in variant 2. See Section 6.4.
[2]: Only available with MPFUN-Fort.
[3]: Only available with MPFUN-MPFR.
[4]: Currently restricted to `n1` = 2.

6.2 Input and output of multiprecision data

Binary-decimal conversion and input or output of multiprecision data is *not* handled by standard Fortran read/write commands, but instead is handled by special subroutines, as briefly mentioned in Table 4. Here are the details:

1. `subroutine mpeform (r1,i1,i2,s1)`. This converts the MPR number `r1` into character form in the character*1 array `s1`. The argument `i1` (input) is the length of the output string, and `i2` (input) is the number of digits after the decimal point. The format is analogous to Fortran E format. The result is left-justified among the `i1` cells of `s1`. The condition `i1` $\geq$ `i2` +20 must hold.

2. `subroutine mpfform (r1,i1,i2,s1)`. This converts the MPR number `r1` into character form in the character*1 array `s1`. The argument `i1` (input) is the length of the output string, and `i2` (input) is the number of digits after the decimal point. The format is analogous to Fortran F format. The result is right-justified among the `i1` cells of `s1`. The condition `i1` $\geq$ `i2` +10 must hold.

3. `subroutine mpread (i1,r1)`. This reads the MPR number `r1` from Fortran logical unit `i1`. The digits of `r1` may span more than one line, provided that a backslash (\) appears at the end of a line to be continued. Individual input lines may not exceed 2048 characters in length. Format: The input number must have a leading digit (possibly zero), and must have a period somewhere, but must *not* have embedded blanks; an exponent with `e, d, E` or `D` may optionally follow the numeric value. Up to five MPR arguments may be included in argument list. See item 9 below on an additional precision argument.

4. `subroutine mpread (i1,z1)`. This is the same as the previous item (3), except that the input argument `z1` is of type MPC (a pair of MPR). Up to five MPC arguments may be included in argument list. See item 9 below on an additional precision argument.

5. `function mpreal (s1,i1)`. This converts the string `s1`, which is of type `character*1` and length `i1`, to MPR. See item 3 for format. See item 9 below on an additional precision argument.

6. `function mpreal (sn)`. This converts the string `sn`, which may be of type `character*`n for any n, to MPR. See item 3 for format. On some systems, n may be limited, say to 2048; if this is a problem, use previous item (5). See item 9 below on an additional precision argument.

7. `subroutine mpwrite (i1,i2,i3,r1)`. This writes the MPR number `r1` to Fortran logical unit `i1`. The argument `i2` (input) is the length of the output field, and `i3` (input) is the number of digits after the decimal point. The format is analogous to Fortran E format and is left-justified in the field. Up to five MPR arguments may be included in argument list.

8. `subroutine mpwrite (i1,i2,i3,z1)`. This is the same as the previous item (7), except that the argument `z1` is of type MPC (a pair of MPR). Up to five MPC arguments may be included in argument list.

9. **Note**: For `mpread` (items 3 and 4) and `mpreal` (items 5 and 6), when using variant 1, an integer precision level argument (mantissa words) may optionally be added as the final argument; this argument is required in variant 2. See Section 6.4.

6.3 Handling double precision values

Double precision constants and expressions are indispensable in high-precision applications. For one thing, the product, say, of a multiprecision value times a double precision value is more rapidly computed with a routine dedicated to this task than converting the double precision value to multiprecision and then calling the multi-multi multiplication routine. Certainly the usage of double precision constants such as modest-sized whole numbers and exact binary fractions (e.g., 0., 12345., 3.125), which are entirely safe in a multiprecision application, should be allowed.

However, problems can arise, which are inherent in how modern computer languages handle double precision constants and expressions. For example, the code

```
r1 = 3.14159d0
```

where the left-hand side is multiprecision, does *not* produce the full-precision equivalent of 3.14159, since by standard rules of precedence, the right-hand side is evaluated in double precision, then converted (by zero extension) to the left-hand side. When using the package, one can avoid this problem by writing this as

```
r1 = '3.14159'
```

By enclosing the constant in apostrophes (and changing it to a literal), this indicates to the MPFUN2015 software that the constant is to be evaluated to full precision.

A closely related problem is the following: The code

```
r2 = r1 + 3.d0 * sqrt (2.d0)
```

does *not* produce a fully accurate result, since the subexpression `3.d0 * sqrt (2.d0)` is performed in double precision (again, according to standard rules of operator precedence in all major programming languages). The solution here is to write this as

```
r2 = r1 + 3.d0 * sqrt (mpreal (2.d0))
```

or, if using variant 2, as

```
r2 = r1 + 3.d0 * sqrt (mpreal (2.d0, nwds))
```

where `nwds` is the precision level, in words, to be assigned to the constant 2 (see Section 6.4). This forces all operations to be done using MP routines.

To help avoid such problems, the MPFUN2015 low-level software checks *every* double precision value (constants, variables and expression values) in a multiprecision statement at *execution time* to see if it has more than 40 significant bits. If so, it is flagged as an error, since very likely such usage represents an unintended loss of precision in the application program. This feature catches 99.99% of accuracy loss problems due to the usage of inexact double precision values.

On the other hand, some applications contain legitimate double precision constants that are trapped by this test. For example, in the tpslqm2 and tpslqm3 programs listed in Section 7, exact double precision values can arise that are greater than 40 bits in size. In order to permit such usage, four special functions have been provided: `mpprod, mpquot, mpreald, mpcmplxdc` (see Table 4). The first and second return the product and quotient, respectively, of a MPR argument and a DP argument; the third converts a DP value to MPR (with an optional precision level parameter — see Section 6.4); and the fourth converts a DC value to MPC (with an optional precision level parameter — see Section 6.4). These routines do *not* check the double precision argument to see if it has more than 40 significant bits.

6.4 Dynamically changing the working precision

Different applications have different requirements for language support. One distinction that immediately arises is between applications that do not need to change the working precision from the initially-defined default level (or change it only rarely) and those which, usually for performance reasons, require that the working precision be changed frequently.

Accordingly, for both MPFUN-Fort and MPFUN-MPFR, there are two variants of the language interface module MPFUNG (see Section 3.2):

1. Variant 1: This is recommended for basic applications that do not dynamically change the precision level (or do so only rarely).

2. Variant 2: This is recommended for more sophisticated applications that dynamically change the precision level.

In particular, with variant 1:

1. Assignments of the form R = X, where R is MPR and X is DP, integer or literal are permitted. Assignments of the form Z = Y, where Z is MPC and Y is DP or DC, are permitted.

2. The routines `mpcmplx, mpcmplxdc, mpegamma, mpinit, mplog2, mppi, mpread, mpreal` and `mpreald` each have an (optional) integer argument as the final argument in the list. This argument is the precision level, in words, to be assigned to the result(s). If this argument is not present, the default precision level (`mpwds` words, corresponding to `mpipl` digits) is assumed.

In contrast, with variant 2:

1. The assignments mentioned in item 1 above are *not permitted.* If any of these appears in code, compile-time errors will result. Instead, one must use `mpreal` and `mpcmplx`, as appropriate, with the precision level (in mantissa words) as the final argument, to perform these conversions.

2. The optional precision level arguments mentioned in item 2 above are *required* in all cases.

Note that the `mpreal` function, with the precision level (in words) as the second argument, can be used to assign an MPR argument with one precision level to an MPR variable or array element with a different working precision level. The same is true of `mpcmplx`. The working precision currently assigned to any MP variable or array element may be obtained by using the function `mpwprec` — see Table 4.

Along this line, when one uses the precision level arguments, a precision level of `ndig` digits can be converted to words by the formula `nwds = int (ndig / mpdpw + 2)`. By using the global built-in variable `mpdpw` (which is different between MPFUN-Fort and MPFUN-MPFR) in this way, the user code remains portable between the two versions.

As it turns out, in most applications, even those that frequently require the working precision to be changed, only a few changes need to be made to the source code. Consider, for example, the following user code, where the default precision is set in module MPFUNF as 1200 digits:

```
integer k, nx
parameter (nx = 128)
type (mp_real) x(nx)
...
x(1) = 1.d0
do k = 2, nx
   x(k) = 2.d0 * x(k-1) + 1.d0
```

```
enddo
```

This code, as written, is permissible with variant 1, but not with variant 2, because the assignment `x(k) = 1.d0` is not allowed. Furthermore, all operations are performed with the default (maximum) precision level of 1200 digits. So with variant 2, where one wishes to perform this loop with a precision level of approximately 200 digits, this should be written as:

```
integer k, ndig, nwds, nx
parameter (nx = 128, ndig = 200, nwds = int (ndig / mpdpw + 2))
type (mp_real) x(nx)
...
x(1) = mpreal (1.d0, nwds)
do k = 2, nx
   x(k) = 2.d0 * x(k-1) + 1.d0
enddo
```

Note that by changing `x(1) = 1.d0` to `x(1) = mpreal (1.d0, nwds)`, the array element `x(1)` is assigned the value 1.0, with a working precision of `nwds` words (i.e., 200 digits). In the loop, when `k` is 2, `x(2)` also inherits the working precision level `nwds` words, since it is computed with an expression that involves `x(1)`. By induction, all elements of the array `x` inherit the working precision level `nwds` words (i.e., 200 digits).

This scenario is entirely typical of other types of algorithms and applications — in most cases, it is only necessary to make a few code changes, such as in assignments to double precision values before a loop, to completely control dynamic precision. It is recommended, though, that the user employ the system function `mpwprec`, which returns the working precision level currently assigned to an input multiprecision variable or array element (see Table 4), to ensure that the precision level the user thinks is assigned to a variable is indeed the level being used by the program.

Using variant 2, with its stricter coding standards, requires a bit more programming effort, but in the present author's experience, when dealing with applications that dynamically change the precision level, this additional effort is more than repaid by fewer debugging and performance problems in actual usage. A code written for variant 2 also works with variant 1, but not vice versa. See the sample test codes mentioned in the next section, all of which are written to conform to the standards of variant 2.

7 Performance of sample applications

Numerous full-scale multiprecision applications have been implemented using the MPFUN2015 software, including some that dynamically vary the working precision level. In most cases, only very minor modifications needed to be made to existing double precision source code. Some examples include the following:

1. tpslq1: A one-level standard PSLQ program; finds the coefficients of the degree-30 polynomial satisfied by $3^{1/5} - 2^{1/6}$. Size of code: 792 lines. Precision level: 240 digits.

2. tpslqm1: A one-level multipair PSLQ program; finds the coefficients of the degree-30 polynomial satisfied by $3^{1/5} - 2^{1/6}$. Size of code: 944 lines. Precision level: 240 digits.

3. tpslqm2: A two-level multipair PSLQ program; finds the coefficients of the degree-56 polynomial satisfied by $3^{1/7} - 2^{1/8}$. Size of code: 1767 lines. Precision level: 640 digits; switches frequently between multiprecision and double precision.

4. tpslqm3: A three-level multipair PSLQ program; finds the coefficients of the degree-72 polynomial satisfied by $3^{1/8} - 2^{1/9}$. Size of code: 2228 lines. Precision level: 1100 digits; switches frequently between full precision, medium precision (120 digits) and double precision.

5. tpphix3: A Poisson phi program; computes the value of $\phi_2(x, y)$ and then employs a three-level multipair PSLQ algorithm to find the minimal polynomial of degree m satisfied by $\exp(8\pi\phi_2(1/k, 1/k))$ for a given k (see Section 1.4). In the code as distributed, $k = 30$, $m = 64$, and a palindromic option is employed so that the multipair PSLQ routines (which are part of this application) searches for a relation of size 33 instead of 65. This computation involves transcendental functions and both real and complex multiprecision arithmetic. Size of code: 2506 lines. Precision level: 1100 digits; switches frequently between full precision, medium precision (160 digits) and double precision.

6. tquad: A quadrature program; performs the tanh-sinh, the exp-sinh or the sinh-sinh quadrature algorithm, as appropriate, on a suite of 18

problems involving numerous transcendental function references, producing results correct to 500-digit accuracy. Size of code: 1448 lines. Precision level: 1000 digits, but most computation is done to 500 digits; switches frequently between 500 and 1000 digits.

7. tquadgs: A quadrature program; performs the Gaussian quadrature algorithm on many of the same suite of 18 problems as in tquad, producing results correct to 500-digit accuracy. This code run much longer than tquad, due to the expense of computing weights and abscissas, but once the weights and abscissas are computed, they can be written to a file and reused in future runs. Size of code: 716 lines. Precision level: 1000 digits, but most computation is done to 500 digits; switches frequently between 500 and 1000 digits.

Each of the above seven programs, with a reference output file for comparison, is included in the software's distribution package. These codes work identically with both the MPFUN-Fort and the MPFUN-MPFR versions — no changes are required to switch from one version to the other. The script mpfun-tests.scr, which is included in the distribution package for each version, compiles variant 2 of the library, then compiles, links and runs a brief function check program and each of the above programs (except for tquadgs, which runs quite long).

If, after running the test script, the results reproduce the results in the reference output files (except for, say, the CPU run times and PSLQ iteration counts), then one can be fairly confident that the software is working properly. Note that the final output relation of the PSLQ runs might have all signs reversed from the reference output (which is mathematically equivalent).

These programs are provided, in part, as examples of programming techniques when using the MPFUN2015 package. Users may feel free to adapt these codes, although the present author would appreciate being notified and credited when this is done. All application programs and library codes are publicly available but are subject to copyright and other legal conditions. For details, see the file disclaimer.txt in the distribution package.

7.1 Timings

Table 6 presents some performance timings comparing the two versions of the package, and for the GNU gfortran and Intel ifort compilers. Note that with the MPFUN-MPFR version, the GNU C compiler was used to build the GMP and MPFR libraries, no matter what compiler was employed to compile the MPFUN-MPFR libraries and the application code. These runs were made on a 2.9 GHz MacBook Pro with an Intel Core i5 processor and 8 Gbyte of main memory. For uniformity, the timings are listed in the Table 6 to two decimal place accuracy, but, as with all computer run time measurements, they should not be considered repeatable beyond about two or three significant digits.

The tested codes include the first six applications listed in the previous section. The Poisson phi program tpphix3 was performed for a range of problem sizes without the palindromic option.

Examining these results, and focusing on the longer-running programs in the suite, we observe a roughly 3X performance advantage for the MPFUN-MPFR version over the MPFUN-Fort version, and a roughly 25% performance advantage for the Intel ifort compiler over the GNU gfortran compiler. Note that for the program tquad, which perform numerical integration, MPFUN-MPFR is roughly nine times faster, with either compiler, than MPFUN-Fort. This reflects the very fast performance of transcendental functions in MPFR. For the other applications, which are dominated by algebraic operations, the ratio is lower.

8 Conclusion

The current version of the software and documentation is available at:

`http://www.davidhbailey.com/dhbsoftware`

		tpphix3		GNU gfortran		Intel ifort	
	Precision	param.		MPFUN-	MPFUN-	MPFUN-	MPFUN-
Code name	(digits)	k	m	Fort	MPFR	Fort	MPFR
tpslq1	240			16.45	4.33	11.22	4.03
tpslqm1	240			13.91	2.94	4.63	2.61
tpslqm2	650			15.96	7.97	11.94	5.78
tpslqm3	1100			69.52	32.47	39.76	26.96
tquad	1000			76.07	7.59	76.83	7.48
tpphix3	700	18	36	7.34	2.63	3.72	2.18
tpphix3	2000	22	60	93.39	35.93	65.82	30.75
tpphix3	2100	17	64	144.91	52.05	89.94	43.91
tpphix3	2700	26	72	261.50	94.55	185.31	80.18
tpphix3	5000	25	100	2023.77	663.69	1423.03	501.51
tpphix3	8200	32	128	9704.11	3239.61	7318.59	2221.86

Table 6: Timings on a suite of test programs (seconds).

References

[1] D. H. Bailey and J. M. Borwein, "High-precision arithmetic in mathematical physics," *Mathematics*, vol. 3 (2015), pg. 337–367. `http://www.mdpi.com/2227-7390/3/2/337/pdf`.

[2] D. H. Bailey, X. S. Li and K. Jeyabalan, "A comparison of three high-precision quadrature schemes," *Experimental Mathematics*, vol. 14 (2005), no. 3, pg. 317–329.

[3] D. H. Bailey, R. Barrio, and J. M. Borwein, "High precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218 (2012), pg. 10106-10121.

[4] D. H. Bailey and J. M. Borwein, "Hand-to-hand combat with thousand-digit integrals," *Journal of Computational Science*, vol. 3 (2012), pg. 77-86.

[5] D. H. Bailey, J. M. Borwein, R. E. Crandall and J. Zucker, "Lattice sums arising from the Poisson equation," *Journal of Physics A: Mathematical and Theoretical*, vol. 46 (2013), pg. 115201, `http://www.davidhbailey.com/dhbpapers/PoissonLattice.pdf`.

[6] D. H. Bailey and J. M. Borwein, "Compressed lattice sums arising from the Poisson equation: Dedicated to Professor Hari Sirvastava," *Boundary Value Problems*, vol. 75 (2013), DOI: 10.1186/1687-2770-2013-75, `http://www.boundaryvalueproblems.com/content/2013/1/75`.

[7] D. H. Bailey, J. M. Borwein and R. E. Crandall, "Integrals of the Ising class," *Journal of Physics A: Mathematical and General*, vol. 39 (2006), pg. 12271–12302.

[8] D. H. Bailey and D. J. Broadhurst, "Parallel integer relation detection: Techniques and applications," *Mathematics of Computation*, vol. 70, no. 236 (Oct 2000), pg. 1719–1736.

[9] D. H. Bailey, X. S. Li and B. Thompson, "ARPREC: An arbitrary precision computation package," Sep 2002, `http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf`.

[10] J. M. Borwein and D. H. Bailey, *Mathematics by Experiment: Plausible Reasoning in the 21st Century*, 2nd ed., A. K. Peters, Natick, MA, 2008.

[11] P. Borwein, "An efficient algorithm for the Riemann zeta function," 1995, `http://www.cecm.sfu.ca/~pborwein/PAPERS/P155.pdf`.

[12] R. P. Brent, "Fast multiple-precision evaluation of elementary functions," *J. of the ACM*, vol. 23 (1976), 242–251.

[13] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge Univ. Press, 2010.

[14] Jingwei Chen, Damien Stehle and Gilles Villard, "A new view on HJLS and PSLQ: Sums and projections of lattices," *Proc. of ISSAC'13*, 149–156.

[15] "Comparison of multiple-precision floating-point software," `http://www.mpfr.org/mpfr-3.1.0/timings.html`.

[16] "Digital library of mathematical functions," National Institute of Standards and Technology, 2015, `http://dlmf.nist.gov`.

[17] H. R. P. Ferguson, D. H. Bailey and S. Arno, "Analysis of PSLQ, an integer relation finding algorithm," *Mathematics of Computation*, vol. 68, no. 225 (Jan 1999), pg. 351–369.

[18] "Fortran 2008," `http://fortranwiki.org/fortran/show/Fortran+2008`.

[19] "The GNU MPFR library," `http://www.mpfr.org`.

[20] "GNU MPFR library: Comparison of multiple-precision floating-point software," `http://www.mpfr.org/mpfr-current/timings.html`.

[21] Y. Hida, X. S. Li and D. H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic," *Proc. of the 15th IEEE Symposium on Computer Arithmetic* (ARITH-15), 2001.

[22] A. K. Lenstra, H. W. Lenstra, and L. Lovasz, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261 (1982), pg. 515-534.

[23] "MPFR C++," `http://www.holoborodko.com/pavel/mpfr`.

[24] *NIST Digital Library of Mathematical Functions*, version 1.0.6 (May 2013), `http://dlmf.nist.gov`.

[25] R. W. Potter, *Arbitrary Precision Calculation of Selected Higher Functions*, Lulu.com, San Bernardino, CA, 2014.

[26] C. Rubio-Gonzalez, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu and D. Hough, "Precimonious: Tuning assistant for floating-point precision," *Proceedings of SC13*, 26 Apr 2013, `http://www.davidhbailey.com/dhbpapers/precimonious.pdf`.

[27] D. W. Sweeney, "On the computation of Euler's constant", *Mathematics of Computation*, vol. 17 (1963), pg. 170–178.

[28] H. Takahasi and M. Mori, "Double exponential formulas for numerical integration," *Publications of RIMS*, Kyoto University, vol. 9 (1974), pg. 721-741.

[29] S. W. Williams and D. H. Bailey, "Parallel computer architecture," in David H. Bailey, Robert F. Lucas and Samuel W. Williams, ed., *Performance Tuning of Scientific Applications*, CRC Press, Boca Raton, FL, 2011, 11–33.