MPFUN2020: A new thread-safe arbitrary precision package (Full Documentation)

David H. Bailey *

December 25, 2021

Abstract

Numerous research studies have arisen, particularly in 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 arbitrary precision software package ("MPFUN2020") that features several significant improvements over an earlier package. It comes in two versions: a self-contained all-Fortran version, and a version based on the MPFR package, which is even faster. Both of these versions feature: (a) a completely thread-safe design, so user codes can be converted for parallel execution at the application level; (b) a full-featured high-level Fortran interface, so that most applications can be converted to multiprecision with relatively minor changes to source code; (c) full support for both real and complex datatypes; (d) a wide variety of transcendental functions and special functions; (e) run-time checking and other facilities to overcome problems with converting double precision constants and data; (f) a medium precision datatype, which improves performance and reduces memory cost on large variable precision applications; and (g) interoperability — with certain simple restrictions, application codes written for one version can also be run with the other.

*Lawrence Berkeley National Laboratory (retired), 1 Cyclotron Road, Berkeley, CA 94720, USA, and University of California, Davis, Department of Computer Science. E-mail: dhbailey@lbl.gov.

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 calculations, 64-bit floating-point arithmetic is required to produce results of sufficient accuracy; still others switch between 32-bit and 64-bit. Software tools are being developed to help users determine which portions of a computation can be performed with lower precision and which must be performed with higher precision [29].

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 [5, 4], 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 [5]. Many of these computations involve variants of Ferguson's PSLQ integer relation detection algorithm [20, 11]. 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 [25], 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 [17]. 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 [31, 3]. 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 [3]. An overview of applications of high-precision integration in experimental mathematics is given in [6].

1.3 Ising integrals

In one study, tanh-sinh quadrature and PSLQ were employed to study the following classes of integrals [9]. 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:

$$C_n = \frac{2^n}{n!}\int_0^\infty pK_0^n(p)\,\mathrm{d}p.$$

Here K_0 is the *modified Bessel function* [27]. 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 [9].

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 [7, 8], 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 (3), 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 [7], 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 [11]. 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. Table 1 shows some examples [7].

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 [7].

Using this data, Jason Kimberley of the University of Newcastle, Australia, conjectured a formula that gives the degree d as a function of k [7]. These computations required prodigiously high precision (up to 10,000 digits) and very long run times. Other runs were attempted, but failed.

In a subsequent study, the present author and three other researchers were able to dramatically extend these studies, confirming Kimberley's formula for almost all integers up to $s = 52$, and also for $s = 60$ and $s = 64$. These runs, which required precision levels up to 64,000 digits, were facilitated by a significantly improved PSLQ code, the much faster MPFUN2015 package described in [1], as well as parallel execution (facilitated by the thread-safe nature of this package). Examination of these results led to additional discoveries, and ultimately to a full proof of Kimberley's conjecture as well as other facts about these curious polynomials, such as the fact that the polynomials are palindromic (left-to-right symmetric) when s is even [10].

In a recent follow-up study, these results were extended, using the arbi-

trary precision package described in this paper, to examine the broader class of rationals $x = p/s$ and $y = q/s$, for $1 \le p, q < s/2 \le 50$. We found that Kimberley's formula does not give the correct degree to all of these cases. Indeed, the degree of these polynomials differs in puzzling ways that will require additional computation and analysis [2].

2 Current high-precision 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; sadly it is not yet widely implemented in hardware, although it is supported, in software, by some compilers. Another option for this level of precision is "double-double" arithmetic (approximately 31 digits), which consists of two 64-bit IEEE floats, or even quad-double arithmetic (approximately 62 digits), which consists of four IEEE 64-bit floats (see the QD package in the list below) [24].

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 contain the mantissa.

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 that make code conversions relatively painless.

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

- ARPREC: Supports arbitrary precision integer, real, complex and transcendental functions, with high-level C++ and Fortran-90 interfaces; available at `https://www.davidhbailey.com/dhbsoftware/`.

- CLN: Supports arbitrary precision integer, real, complex and transcendental functions in C++; available at `http://www.ginac.de/CLN`.

- GMP: Supports low-level high-precision integer, rational and floating-point calculations; available at `http://gmplib.org`.
- Julia: High-level programming environment incorporating GMP and MPFR; available at `http://julialang.org`.
- MPFR: Supports (at a low level) multiple-precision floating-point computations with correct rounding, based on GMP; includes numerous algebraic and transcendental functions, and a thread-safe build option; very fast timings; available at `http://www.mpfr.org`.
- MPFR++: High-level C++ interface to MPFR (although currently available version is not up-to-date with MPFR); available at `http://perso.ens-lyon.fr/nathalie.revol/software.html`.
- MPFR C++: High-level C++ interface to MPFR with a thread-safe option; available at `http://www.holoborodko.com/pavel/mpfr`.
- mpmath: Python library for arbitrary precision floating-point arithmetic, with transcendentals; available at `https://code.google.com/p/mpmath`.
- NTL: C++ library for arbitrary precision integer and floating-point arithmetic; available at `http://www.shoup.net/ntl`.
- Pari/GP: Computer algebra system including high-precision arithmetic and transcendental functions; available at `http://pari.math.u-bordeaux.fr`.
- QD: Supports "double-double" (approx. 31 digits) and "quad-double" (approx. 62 digits) arithmetic, with transcendental functions; includes high-level interfaces for C++ and Fortran-90; available at `https://www.davidhbailey.com/dhbsoftware/`.
- Sage: Open-source symbolic computing system including high-precision facilities; available at `http://www.sagemath.org`.

2.1 Thread safety and parallel implementations

The scientific computing world is moving rapidly into multicore and multi-node parallel computing [32], so that future improvements in performance

on high-precision computations will only be obtained by aggressively exploiting parallelism. Exploiting parallelism is certainly an attractive option for multiprecision applications, given the high cost of performing such calculations. It is difficult to achieve significant parallel speedup within a single high-precision arithmetic operation, but in many cases modest speedups, at least, can be obtained at the loop level, i.e., by performing in parallel individual iterations of DO or FOR loops in an application code that contains multiprecision operations.

On modern systems that feature multicore processors, parallel computing is most efficiently performed using a shared memory, multithreaded environment such as OpenMP [32] within a single node, even if a message passing system such as MPI is employed for parallelism between nodes.

Computations that use a thread-parallel environment such as OpenMP must be entirely "thread-safe." One impediment to thread safety for multiprecision applications is the design of the operator overloading feature (i.e., extending $+$, $-$, $\times$, $\div$ to multiprecision data) of modern computer languages, which typically does not permit one to carry information such as the current working precision level. Also, most high-precision packages generate a "context" of auxiliary data, such as the current working precision level and data to support transcendental function evaluation, which often ruin thread safety unless special care is taken.

Of the packages listed in Section 2, only one is both thread-safe and supports arbitrary precision floating-point calculations with a high-level interface, namely the MPFR C++ package [26]. This package is built upon the lower-level MPFR package [22], which in turn is 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 [18].

There was, to this author's knowledge, no high-level, thread-safe arbitrary precision package to support Fortran applications, prior to MPFUN2015.

2.2 MPFUN2015

In response to these challenges, in 2015 the present author developed a new software package for arbitrary-precision computation, named MPFUN2015, which is documented in an earlier paper [1]. It is offered in two versions: (a) MPFUN-Fort: an all-Fortran version based on floating-point arithmetic; and (b) MPFUN-MPFR: a version that calls the MPFR package for all low-level

functions and operations. The run-time performance of MPFUN20-MPFR is several times faster than MPFUN-Fort, but installation is significantly more complicated (because the GMP and MPFR packages must first be installed).

Both versions feature full support for both real and complex multiprecision datatypes, including transcendental functions. In most cases, only minor changes need be made to convert an existing double precision code. Both versions are thread-safe, so that user applications can be converted to parallel execution at the application level.

However, this software has some shortcomings, as have become evident in recent work by the present author, among others: (a) as mentioned above, the run-time performance of the MPFUN-Fort package is typically 3X-5X slower than the MPFUN-MPFR package; (b) with both versions of the package, significant memory is wasted for applications where part of multiprecision arrays require only a medium level of precision; and (c) for such applications, significant CPU time is wasted in data movement.

3 MPFUN2020: A new thread-safe package

To that end, the present author has written a new thread-safe arbitrary precision package. As before, it is offered in two versions: an all-Fortran version (MPFUN20-Fort), which runs several times faster than the earlier MPFUN-Fort, and also MPFUN20-MPFR, which is even faster and more complete than the earlier MPFUN-MPFR.

Both of these versions feature: (a) a completely thread-safe design, so user codes can be converted for parallel execution at the application level; (b) a full-featured high-level Fortran interface, so that most applications can be converted to multiprecision with relatively minor changes to source code; (c) full support for both real and complex datatypes; (d) a wide variety of transcendental functions and special functions; (e) run-time checking and other facilities to overcome problems with converting double precision constants and data; (f) a medium precision datatype, which improves performance and reduces memory cost on large variable precision applications; and (g) interoperability — with certain simple restrictions, application codes written for one version can also be run with the other.

The sharply improved performance of the all-Fortran MPFUN20-Fort version is due primarily to changing the underlying design of the package to be based on 64-bit integer operations, rather than on 64-bit floating-point oper-

ations as in the earlier MPFUN-Fort. Additionally, the MPFUN20-Fort version employs FFT-based arithmetic for significantly faster execution at very high precision levels — see Sections 7.4 and 7.5. The new MPFUN20-MPFR version is faster than the MPFUN20-Fort version on most applications, although it is more complicated to install, because the GMP and MPFR packages must first be installed, usually requiring administrator privilege.

What follows is a brief description of the MPFUN2020 software, followed by instructions for installation and usage.

3.1 Data structure

For the MPFUN20-Fort version, the structure is a $(N + 6)$-long vector of 64-bit integers, where N is the number of mantissa words:

- Word 0: Total space allocated for this array, in 64-bit integer 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^{60}.
- Word 4 to $N + 3$: N mantissa words (whole numbers between 0 and $2^{60} - 1$).
- Word $N + 4$ and $N + 5$: Scratch words for internal usage.

For the MPFUN20-MPFR version, the structure is a $(N + 6)$-long vector of 64-bit integers, where N is the number of mantissa words:

- 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 MPFUN20-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 MPFUN20-MPFR version, words 1 through $N + 4$ correspond exactly to the data structure of the MPFR package.

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

3.2 Modules

The MPFUN20-Fort and MPFUN20-MPFR versions both include the following separate modules, each in its own source file:

1. MPFUNA: Contains compile-time global data. In the MPFUN20-Fort version, this includes the binary values of $\log 2$, π and γ (up to 20,000-digit precision).

2. MPFUNB (only in MPFUN20-Fort): Handles basic arithmetic functions, rounding, normalization, square roots and n-th roots.

3. MPFUNC (only in MPFUN20-Fort): Handles binary-to-decimal conversion, decimal-to-binary conversion and input/output operations.

4. MPFUND (only in MPFUN20-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 (only in MPFUN20-Fort): Includes routines for special functions, such as the BesselJ, gamma, incomplete gamma and zeta functions.

6. MPFUNF: Defines the default standard precision and medium precision levels, in digits, and also the equivalent levels in words. This is the only module that needs to be modified by the user in normal usage.

7. MPFUNG: A high-level user interface that provides support for the standard high-precision datatype.

8. MPFUNH: A high-level user interface that provides support for the medium precision datatype.

9. MPMODULE: The main module that references the others; in normal usage it is the only module that is directly referenced by the user.

3.3 The MPFUN2020 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. The working precision level is passed as a subroutine argument, ensuring thread safety. The MPFUN20-MPFR version is thread safe provided that the MPFR package is compiled with the thread-safe option.

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 structures 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 below (see Section 5.5).

All computations are performed to a fixed precision (set by the user in module MPFUNF) unless the user specifies a lower value for certain calculations (see Section 5.5 for details). 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 double precision constant or variable to a multiprecision variable or array element, or when reading multiprecision data from a file, the result is assigned the default precision unless a lower precision level is specified.

4 Installation, compilation and linking

The two versions of the MPFUN2020 package, together with installation instructions, are available at `https://www.davidhbhailey.com/dhbsoftware`.

Installation, compilation and linking is relatively straightforward, provided that one has a Unix-based system, such as Linux or Apple OS X, and

a Fortran-2008-compliant compiler. See the `README` file in the distribution package for details.

The gfortran compiler (highly recommended for both versions of the package) is available for a variety of systems at `https://gcc.gnu.org/wiki/GFortranBinaries`; for Mac OS X systems, see `https://github.com/fxcoudert/gfortran-for-macOS/releases`. The MPFUN2020 software has been tested on:

1. gfortran compiler, version 11.2.0, on an Apple Mac OS X system, version 12.0.1, with an Intel Core i5 processor.

2. gfortran compiler, version 11.2.0, on an Apple Mac OS X system, version 12.0.1, with an Apple Silicon (ARM) M1 Pro processor.

3. gfortran compiler, version 10.2.1, on a Debian Linux system, version 4.19.152-1, with an Intel processor (MPFUN20-Fort only).

4. NAG nagfor compiler, version 7.0, on a Debian Linux system, version 4.19.152-1, with an Intel processor (MPFUN20-Fort only).

5. Intel ifort compiler, version 2021.4.0, on a Debian Linux system, version 4.19.152-1, with an Intel processor (MPFUN20-Fort only).

4.1 The two variants

Each version of the software comes in two variants:

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

- Variant 2: This is recommended for more sophisticated applications that dynamically change the working precision level. It does not allow some mixed-mode combinations, and requires one to explicitly specify a working precision parameter for some functions. However, in the present author's experience, these restrictions result in less overall effort to produce a debugged, efficient application code. (See Section 5.5 below.)

The Fortran source files and scripts required for each of these variants are in the respective directories fortran-var1 and fortran-var2.

Compile/link scripts are available in the fortran-var1 and fortran-var2 directories for the gfortran and Intel ifort compilers, and, with MPFUN20-Fort, for the NAG nagfor compiler. These scripts automatically select the proper source files from the package for compilation. For example, to compile variant 1 of either the MPFUN20-Fort or MPFUN20-MPFR library using the GNU gfortran compiler, go to the fortran-var1 directory and type

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

To compile and link the application program `tpslq1.f90` for variant 1, using the GNU gfortran compiler, producing the executable file `tpslq1`, type

```
./gnu-complink1.scr tpslq1
```

To execute the program, with output to `tpslq1.txt`, type

```
./tpslq1 > tpslq1.txt
```

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.

Several sample test programs, together with reference output files, are included in the fortran-var1 and fortran-var2 directories — see Section 6.

5 Coding Fortran applications

A high-level Fortran interface is provided for each version of the package (MPFUN20-Fort and MPFUN20-MPFR).

To use the software in a user program, first set the parameter `mpipl`, the default standard 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 in file mpfunf.f90 in the fortran-var1 or fortran-var2 directory of the software. In the code as distributed, `mpipl` is set to 2500 digits (sufficient to run the six test programs of Section 6), but it can be set to any level greater than 50 digits. `mpipl` is automatically converted to mantissa words by the formula

```
mpwds = int (mpipl / mpdpw + 2),
```

where `mpdpw` is a system parameter set in file mpfuna.f90 (`mpdpw` $= \log_{10} 2^{60} \approx 18.0617997\ldots$ for MPFUN20-Fort and $\log_{10} 2^{64} \approx 19.2659197\ldots$ for MPFUN20-MPFR). The resulting parameter `mpwds` is the internal default precision level,

in words. All subsequent computations are performed to `mpwds` words precision unless the user, in an application code, specifies a lower precision.

After setting the value of `mpipl`, if needed, compile the appropriate version of the library, using one of the scripts mentioned in the previous section.

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 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, 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 multiprecision real (MPR), multiprecision complex (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 5.3 below about DP and DC constants and expressions.

Input and output of MP variables or array elements are done by using the special subroutines `mpread` and `mpwrite`. See Table 4 and Section 5.2.

In MPFUN20-Fort, the above instructions apply if the precision level, namely `mpipl`, is 20,000 digits or less. If one requires a precision level greater than 20,000 digits, then in addition to changing the value of `mpipl` (and, thus, `mpwds`) in module MPFUNF in file mpfunf.f90, one must include a call to `mpinit` at the start of execution, in a single-threaded section of code, before any multiprecision operations are performed:

```
call mpinit (mpwds)
```

See Table 4 and Section 5.5 for additional details.

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 5.5.

5.1 Functions and subroutines

Most Fortran-2008 intrinsic functions [21] are supported 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 6 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 (if supported by the compiler). The variable names `r1,r2,r3` are MPR; `z1` is MPC; `d1` is DP; `dc1` is DC; `i1,i2,i3,n` are integers; `s1` is `character(1)`; `sn` is `character(n)` for any `n`; and `rr` is a vector of MPR of length `n`.

5.2 Input and output of multiprecision data

Binary-decimal conversion and input or output of multiprecision data are handled by special subroutines, as briefly mentioned in Table 4:

1. `subroutine mpeform (r1,i1,i2,s1)`. This converts the MPR number `r1` into decimal 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 >= i2 + 20` must hold.

2. `subroutine mpfform (r1,i1,i2,s1)`. This converts the MPR number `r1` into decimal 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 >= i2 + 20` must hold.

3. `subroutine mpread (i1,r1)`. This reads the MPR number `r1`, presumed in decimal format, 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

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(z1)`	Imaginary part of complex arg
MPR	`aint(r1)`	Truncate 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 [1]
MPR	`bessel_j1(r1)`	Bessel function of the first kind, order 1 [1]
MPR	`bessel_jn(n,r1)`	Besel function of the first kind, order `n` [1]
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(n,r1)`	Besel function of the second kind, order `n` [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)`	Mod function = `r1 - r2*aint(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	`sinh(r1)`	Hyperbolic sine
MPR	`sqrt(r1)`	Square root of real arg
MPC	`sqrt(z1)`	Square root of complex arg
MPR	`tan(r1)`	Tangent function
MPR	`tanh(r1)`	Hyperbolic tangent function

Table 3: Fortran-2008 intrinsic functions extended to multiprecision. Notes: [1]: For MPFUN20-Fort, `r1` must not exceed 2000.

Type	Name	Description
S	`mpbinmd(r1,d1,i1)`	Converts `r1` to the form `d1 * 2^i1`
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	`mpdecmd(r1,d1,i1)`	Converts `r1` to the form `d1 * 10^i1`
S	`mpeform(r1,i1,i2,s1)`	Converts `r1` to char(1) string in Ei1.i2 format, suitable for output (Sec. 5.2)
S	`mpfform(r1,i1,i2,s1)`	Converts `r1` to char(1) string in Fi1.i2 format, suitable for output (Sec. 5.2)
F(MPR)	`mpegamma()`	Returns Euler's γ constant [1]
S	`mpinit()`	Initializes for extra-high precision (Sec. 5) [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]
F(MPR)	`mprand(r1)`	Returns pseudorandom number, based on `r1` Start with an irrational, say `r1 = mplog2()` Typical iterated usage: `r1 = mprand(r1)`
S	`mpread(i1,r1)`	Inputs `r1` from Fortran unit `i1`; up to five MPR args may be listed (Sec. 5.2) [1]
S	`mpread(i1,z1)`	Inputs `z1` from Fortran unit `i1`; up to five MPC args may be listed (Sec. 5.2) [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(s1,i1)`	Converts char(1) string to MPR (Sec. 5.2) [1]
F(MPR)	`mpreal(sn)`	Converts char(n) string to MPR (Sec. 5.2) [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. 5.2)
S	`mpwrite(i1,i2,i3,z1)`	Outputs `z1` in Ei2.i3 format to unit `i1`; up to five MPC args may be listed (Sec. 5.2)

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 Sec. 5.5.
[2]: These do not check DP or DC values. See Sec. 5.3.

Type	Name	Description
F(MPR)	`mpprod(r1,q1)`	Returns `r1*q1`, without checking [2]
F(MPR)	`mpquot(r1,q1)`	Returns `r1/q1`, without checking [2]
F(MPR)	`mpreal(q1)`	Converts quad real to MPR [1]
F(MPR)	`mprealq(d1)`	Converts quad to MPR, without checking [1, 2]
F(Q)	`qreal(r1)`	Converts MPR to quad.

Table 5: Support for IEEE quad precision (if available) (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 Sec. 5.5.
[2]: These do not check quad values. See Sec. 5.3.

Type	Name	Description
F(MPR)	`agm(r1,r2)`	Arithmetic-geometric mean
F(MPR)	`airy(r1)`	Airy function [3]
S	`mpberne(n,rr)`	Initialize array `rr`, of length `n`, with first `n` even Bernoulli numbers [1, 2]
F(MPR)	`bessel_in(n,r1)`	BesselI function with order n [2]
F(MPR)	`bessel_kn(n,r1)`	BesselK function with order n [2]
F(MPR)	`digamma(r1)`	Digamma function [3]
F(MPR)	`expint(r1)`	Exponential integral function [3]
F(MPR)	`incgamma(r1,r2)`	Incomplete gamma function [2]
S	`polylog_ini(n,rr)`	Initialize array `rr`, of size `|n|`, for computing polylogarithms when `n < 0` [1, 2, 4]
F(MPR)	`polylog_neg(n,rr,r1)`	Polylogarithm function for `n < 0`, using precomputed data in `rr` (see `polylog_ini`) [2, 4]
F(MPR)	`polylog_pos(n,r1)`	Polylogarithm function for `n >= 0` [2]
F(MPR)	`zeta(r1)`	Zeta function with MPR argument
F(MPR)	`zetaem(n,rr,r1)`	Zeta function, using precomputed even Bernoulli numbers in `rr` (see `berne` above); `n` must be greater than precision in decimal digits [2]
F(MPR)	`zeta_int(n)`	Zeta function with integer argument [1, 2]

Table 6: 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 Sec. 5.5.
[2]: Only available with MPFUN20-Fort.
[3]: Only available with MPFUN20-MPFR.
[4]: For `polylog_ini` and `polylog_neg`, the integer `n` is limited to the range $(-1000, -1)$.

a leading digit (possibly zero), must have a period somewhere; may include an `e, d, E` or `D` followed by an integer exponent; but must *not* have embedded blanks. 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 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`, in a format analogous to Fortran E format, left-justified in the field. The argument `i2` (input) is the length of the output field, and `i3` (input) is the number of digits after the decimal point; `i3` must exceed `i2` by 20. 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 working precision level argument (mantissa words) may optionally be added as the final argument; this argument is required in variant 2. See Section 5.5.

5.3 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 full multiplication routine. Certainly the usage of double precision constants such as modest-sized whole numbers and exact binary fractions (e.g., 3., 12345., 2.5, 6.125), which are entirely safe in a multiprecision application, should be allowed.

Users should be aware, however, that there are some hazards in this type of programming, inherent in conventions adopted by all Fortran compilers. For example, the code

```
r1 = 3.14159d0
```

where `r1` is MPR, does NOT produce the true multiprecision equivalent of 3.14159. In fact, the software will flag such usage with a run-time error. To obtain the full MPR converted value, write this as

```
r1 = '3.14159d0'
```

or, if using variant 2, as

```
r1 = mpreal ('3.14159d0', nwds)
```

where `nwds` is the level of working precision (in words) to be assigned to `r1`. Similarly, the code

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

where `r1` and `r2` are MPR, does NOT produce the true multiprecision value one might expect, since the expression `3.d0 * sqrt (2.d0)` will be performed in double precision, according to Fortran-90 precedence rules. In fact, the above line of code will result in a run-time error. To obtain the fully accurate result, write this as

```
r2 = r1 + 3.d0 * sqrt (mpreal (2.q0))
```

or, if using variant 2, as

```
r2 = r1 + 3.d0 * sqrt (mpreal (2.q0, nwds))
```

where `nwds` is the level of working precision in words.

To help avoid such problems, both versions of the MPFUN2020 software check *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 (including several of the sample test

codes mentioned in Section 6) contain legitimate double precision constants that are trapped by this test. 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 5.5); and the fourth converts a DC value to MPC (with an optional precision level parameter — see Section 5.5). These routines do *not* check the double precision argument to see if it has more than 40 significant bits.

5.4 Support for quad precision

Several processor/compiler systems now support IEEE quad (128-bit) floating-point arithmetic. If one's platform does support IEEE quad, limited support is provided in the MPFUN2020 package. In particular, the `mpreal` function accepts a quad argument for conversion to multiprecision, and the `qreal` function accepts a multiprecision argument for conversion to quad. Other routines available are `mpprod, mpquot` and `mprealq`. Note that the `MPREAL` function checks the quad argument to see if it has more than 90 significant bits; if so, an error is flagged, as mentioned above for double precision conversions. To convert without checking, use `mprealq`.

As with the double precision conversion routines, when using variant 1, an integer working precision level argument (mantissa words) may optionally be added as the final argument for the functions `mpreal` and `mprealq`; this argument is required in variant 2. See Section 5.5.

5.5 Dynamically changing the working precision

Some multiprecision applications run fine with a fixed precision level, but others are more efficiently implemented with a working precision level that is changed frequently. Accordingly, for each version of the package, there are two variants of the package, as mentioned above (see Section 4):

- Variant 1: This is recommended for beginning users and for basic applications that do not dynamically change the working precision level (or do so only rarely).
- Variant 2: This is recommended for more sophisticated applications that dynamically change the working precision level. It does not allow

some mixed-mode combinations, and requires one to explicitly specify a working precision parameter for some functions. However, in the present author's experience, these restrictions result in less overall effort to produce a debugged, efficient application code.

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, mpreald, mprealq` and `zeta_int` each have an (optional) integer argument as the final argument in the list. This argument is the working 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 working precision level arguments mentioned in item 2 above are *required* in all cases. For example, if the full default precision level (`mpwds`, corresponding to `mpipl` digits) is required, then a call to one of the functions or subroutines in item 2 above must have `mpwds` as the final argument.

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 MPFUN20-Fort and MPFUN20-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 to implement variable precision. Consider, for example, the following user code, where the default precision is set in file mpfunf.f90 to 2500 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 2500 digits. So with variant 2, where one wishes to perform this loop with a precision level of, say, 500 digits, this should be written as:

```
integer k, ndig, nwds, nx
parameter (nx = 128, ndig = 500, 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., 500 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., 500 digits).

This scenario is entirely typical of using variable precision — 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 pre-

cision. *However, it is highly recommended that the user frequently employ the system function* `mpwprec`*, which returns the working precision level currently assigned to an multiprecision variable or array element (see Table 4), to ensure that the working 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 Section 6, all of which are written to conform to the stricter standards of variant 2.

5.6 Medium precision datatype

In many high-precision applications, only part of the multiprecision variables and arrays contain full precision data; others contain data with only modest precision, typically only a fraction as high (although still higher than double precision). Since all multiprecision data are allocated sufficient space to accommodate full precision values, much of the storage and data movement costs for modest precision data are wasted.

To reduce memory usage and improve performance in such applications, a medium precision real and a medium precision complex datatype have been defined, and are available in both the MPFUN20-Fort and MPFUN20-MPFR versions of the software. To use these datatypes, do the following:

First, set the medium precision level `mpiplm` in file mpfunf.f90; by default it is set to 250 digits. Then in each user subprogram that will include either full precision or medium precision data, insert the following line:

```
use mpmodule
```

as with standard full precision. To designate a variable or array as medium precision real, use a Fortran-90 type statement with the type `mp_realm`, as in this example:

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

Similarly, use the type `mp_cmplxm` for medium precision complex data.

Direct assignments between full precision and medium precision variables, as well as mixed-mode arithmetic operations between full precision and medium precision variables, are *not* allowed. If one needs to convert a regular full precision value to a medium precision value, use the function

mprealm, as in this example

```
type mp_real a; mp_realm b
b = mprealm (a)
```

Similarly, to convert a medium precision value to a full precision value, use the function mpreal, as in

```
type mp_real a; mp_realm b
a = mpreal (b)
```

Note however, that with either mpreal or mprealm, in variant 1, an integer working precision level argument (mantissa words) may optionally be added as the final argument; this argument is required in variant 2. See Section 5.5 for details.

Each of the intrinsic functions listed in tables 2, 3, 4, 5 and 6 is also supported with the medium precision datatype, except for the following: mpreald, mprealq, mpcmplx, mpcmplxd, mpegamma, mppi, mplog2 and zeta_int. The medium precision equivalents of these functions are: mprealdm, mprealqm, mpcmplxm, mpcmplxdm, mpegammam, mppim, mplog2m and zeta_intm, respectively. Also, the comment regarding the working precision argument in the previous paragraph applies to each of these functions. For example, if one has set the medium precision level mpiplm to 250 digits in file mpfunf.f90, and if one is using variant 2, then one can retrieve the precomputed value of π to a medium precision variable b by typing

```
b = mppim (mpiplm)
```

Usage of the medium precision datatype is illustrated in the programs tpslqm3.f90 and tpphix3.f90 in the set of sample application programs. See Section 6 below for details.

6 Sample applications and performance

Numerous full-scale multiprecision applications have been implemented using the MPFUN2020 software, including some that dynamically vary the working precision level. In most cases, only relatively minor modifications needed to be made to existing double precision source code.

The current release of the software includes a set of sample application programs in the fortran-var1 and fortran-var2 directories (the files are identical between directories):

1. `testmpfun.f90`: This briefly tests most individual MPFUN2020 operations and functions (including mixed mode arithmetic, comparison operations, transcendental functions and special functions), by comparing each result with benchmark results in the file testmpfun.ref.txt, which must be present in the same directory. This is not an exhaustive test of all possible scenarios, but it often detects bugs and compiler issues. Size of code: 788 lines. Precision: 500 digits.

2. `tpslq1.f90`: 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: 788 lines. Precision level: 240 digits.

3. `tpslqm1.f90`: 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: 936 lines. Precision level: 240 digits.

4. `tpslqm2.f90`: 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: 1765 lines. Precision level: 640 digits; switches frequently between multiprecision and double precision.

5. `tpslqm3.f90`: 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: 2318 lines. Precision level: 1100 digits; switches frequently between full precision (1100 digits), medium precision (varies from 50 to 110 digits) and double precision.

6. `tpphix3.f90`: 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, $k = 28$, $m = 96$, and a palindromic option is employed so that the multipair PSLQ routines (which are part of this application) search for a relation of size 49 instead of 97. This computation involves transcendental functions and both real and complex multiprecision arithmetic. Size of code: 2669 lines. Precision level: 2500 digits; switches frequently between full precision (2500 digits), medium precision (varies from 50 to 250 digits) and double precision.

7. `tquad.f90`: 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: 1559 lines. Precision level: 1000 digits, but most computation is done to 500 digits; switches frequently between 500 and 1000 digits.

8. `tquadgs.f90`: 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 runs much longer than tquad, due to the expense of computing weights and abscissas. Once the weights and abscissas are computed, they are written to a file, so they can be reused in future runs. Size of code: 757 lines. Precision level: 1000 digits, but most computation is done to 500 digits; switches frequently between 500 and 1000 digits.

In addition, the fortran-var1 and fortran-var2 directories include test scripts that compile the library and run each of the above sample programs above (except tquadgs.f90, which takes considerably more run time). In directory fortran-var1, these scripts are:

- `gnu-mpfun-tests1.scr`
- `intel-mpfun-tests1.scr`
- `nag-mpfun-tests1.scr` (MPFUN20-Fort only)

and the same scripts in directory fortran-var2, except for 2 instead of 1 in the filenames. For each test program, the script outputs either TEST PASSED or TEST FAILED. If all tests pass, then one can be fairly confident that the MPFUN2020 software and underlying compilers are working properly.

These programs are provided, in part, as examples of programming techniques when using the MPFUN2020 package. Users may feel free to adapt these codes, although the present author asks to be 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.

Code name	Precision (digits)	param. k	param. m	MPFUN20-Fort	MPFUN-MPFR
`tpslq1`	240			8.09	4.25
`tpslqm1`	240			3.76	3.11
`tpslqm2`	650			5.12	6.12
`tpslqm3`	1100			20.19	24.95
`tquad`	1000			31.75	7.08
`tpphix3`	2500	28	96	18.56	17.88
`tpphix3`	5500	25	100	548.54	475.48

Table 7: Timings on a suite of test programs (seconds).

6.1 Timings

Table 7 presents some performance timings comparing the two versions of the package for the first six test programs listed above, plus an additional run using the `tpphix3.f90` code, with different parameters k and m, and without the palindromic option, which is not available when k is odd.

These runs were performed using the GNU gfortran compiler (version 10.2.0). For the MPFUN20-MPFR version, the GNU C compiler (version 10.2.0) was used to build the GMP and MPFR libraries. These runs were made on an Apple MacPro system with a 2.4 GHz Quad-Core Intel Core i5 processor and 16 Gbyte of main memory. For uniformity, the timings are listed in the Table 7 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.

7 Appendix: Numerical algorithms

This section presents a brief overview of the algorithms used in MPFUN20-Fort. The algorithms used in MPFUN20-MPFR (except for pseudorandom number generation, which is given below) are described in [22].

7.1 Algorithms for basic arithmetic

Multiplication. For modest levels of precision, MPFUN20-Fort employ adaptations of the usual schemes we all learned in grade school, where the number base is $2^{60} = 1152921504606846976$. Note 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, which involves obtaining an accurate trial divisor, is employed for division.

Square roots. Square roots are calculated by the following Newton-Raphson iteration, which converges to $1/\sqrt{a}$ [13, 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 $((((3)^2)^2)^2)^2 \cdot 3 = 129140163$.

Pseudorandom number generation. The pseudorandom number included in the package (both in the MPFUN20-Fort and MPFUN20-MPFR) versions is the following:

$$x_{n+1} = \text{frac}\,(5501758857861179 \cdot x_n), \tag{7}$$

where the calculation is performed with slightly more than the standard precision for full accuracy. Here the large prime 5501758857861179 is specified as the double precision constant `mprandx`, set in module MPFUNA in file mpfuna.f90. Its value may be changed if desired, but must not exceed 2^{53}. To use the generator, first set a MPR variable `r1` to some irrational value between 0 and 1, such as $\sqrt{1/2}$, $\log(2)$ or $\pi/4$. Then successive iterates can be generated by typing

```
r1 = mprand (r1)
```

The same scheme is used for both MPFUN20-Fort and MPFUN20-MPFR. However, the specific specific sequence is *not* the same between the two versions, nor is it same if the precision level is varied.

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

7.2 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 MPFUN20-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 MPFUN20-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 $B^{2/5}$, where B is the number of bits 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:

$$x_k = x_k - \frac{e^x - a}{e^x}. \tag{8}$$

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

$$\cos(2x) = 1 - 2\sin^2(x) \tag{9}$$

$$\cos(2x) = 2\cos^2(x) - 1, \tag{10}$$

are applied q times (formula (9) is used once, and (10) 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 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.

Euler's gamma constant. A function is also provided to turn Euler's constant $\gamma = 0.57721566490153286061\ldots$. This can be calculated using the following

formula, which is an improvement of a technique previously used by Sweeney [30]. If a result accurate to at least B bits is desired, first select the integer $N = \lceil \log_2(B \log 2) \rceil$. Then

$$\gamma \approx \frac{2^N}{e^{2^N}} \sum_{m=0}^{\infty} \frac{2^{mN}}{(m+1)!} \sum_{t=0}^{m} \frac{1}{t+1} \; - \; N \log 2. \tag{11}$$

The error in this approximation is less than $1/(2^N e^{2^N})$.

The binary values of $\log(2)$, π and γ are stored to 20,000 digit precisioon (by default) in data statements in module MPFUNA. If higher precision is required, `mpinit` must be called at the start of execution in a single-threaded section of code (see Table 4).

7.3 Special functions

Modern mathematical and scientific computing frequency often involves other, more sophisticated functions, which collectively are termed "special functions" [19]. A number of these functions have been implemented in the MPFUN20-Fort package, and others will be added 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.

BesselI function. The BesselI function, or, more formally, the modified Bessel function of the first kind, is defined as [19, 10.25.2]:

$$I_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{\left(\frac{z^2}{4}\right)^k}{k! \Gamma(\nu + k + 1)} \tag{12}$$

This function can be evaluated directly according to this definition, terminating the summation when terms are smaller than the current MP epsilon. The subroutine implementing this function does not rely on any stored data, and so is completely thread-safe.

BesselJ function. The BesselJ function, or, more formally, the Bessel function of the first kind, is defined as [19, 10.2.2]:

$$J_\nu(z) = \left(\frac{z}{2}\right)^\nu \sum_{k=0}^{\infty} \frac{(-1)^k \left(\frac{z^2}{4}\right)^k}{k! \Gamma(\nu + k + 1)}. \tag{13}$$

This function can be evaluated directly according to this definition, terminating the summation when terms are smaller than the current MP epsilon. Large amounts of cancellation occurs in this formula. Thus when evaluating these formula, a working precision of $(1+|z|/2000)$ times the normal working precision is employed. The subroutine implementing this function does not rely on any stored data, and so is completely thread-safe.

BesselK function. The BesselK function, or, more formally, the modified Bessel function of the second kind, is defined as [19, 10.31.1]:

$$K_n(z) = \frac{1}{2}\left(\frac{z}{2}\right)^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} \left(-\frac{z^2}{4}\right)^k + (-1)^{n+1} \log\left(\frac{z}{2}\right) I_n(z) \quad (14)$$

$$+ (-1)^n \frac{1}{2}\left(\frac{z}{2}\right)^n \sum_{k=0}^{\infty} \left(\psi(k+1) + \psi(n+k+1)\right) \frac{\left(\frac{z^2}{4}\right)^k}{k!(n+k)!} \quad (15)$$

This function can be evaluated directly according to this definition, terminating the summation when terms are smaller than the current MP epsilon. The $\psi(n)$ (digamma) function can be evaluated from the formula

$$\psi(n) = -\gamma + \sum_{k=1}^{n-1} \frac{1}{k}, \quad (16)$$

where γ is Euler's constant. Aside from requiring Euler's constant (which is stored in module MPFUNA), the subroutine implementing this function does not require any saved data and thus is completely thread-safe.

BesselY function. The BesselY function, or, more formally, the Bessel function of the second kind, is defined as [19, 10.8.1]:

$$Y_n(z) = \frac{1}{\pi}\left[\left(\frac{z}{2}\right)^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!} \left(\frac{z^2}{4}\right)^k + 2\log\left(\frac{z}{2}\right) J_n(z) \right. \quad (17)$$

$$\left. + \left(\frac{z}{2}\right)^n \sum_{k=0}^{\infty} \left(\psi(k+1) + \psi(n+k+1)\right) \frac{\left(-\frac{z^2}{4}\right)^k}{k!(n+k)!}\right] \quad (18)$$

This function can be evaluated directly according to this definition, terminating the summation when terms are smaller than the current MP epsilon.

Large amounts of cancellation occurs in this formula. Thus when evaluating these formula, a working precision of $(1+|z|/2000)$ times the normal working precision is employed. Aside from requiring π and Euler's constant (which are stored in module MPFUNA), the subroutine implementing this function does not require any saved data and thus is completely thread-safe.

Error function. The error function is defined as [19, 7.6.2]:

$$\operatorname{erf}(z) = \frac{2}{\sqrt{\pi}e^{z^2}} \sum_{n=0}^{\infty} \frac{2^n z^{2n+1}}{1 \cdot 3 \cdots (2n+1)}. \tag{19}$$

This formula is satisfactory for computation unless the argument z is large, in which case the following asymptotic expression works better:

$$\operatorname{erfc}(z) \approx \frac{1}{\sqrt{\pi}e^{z^2}} \sum_{n=0}^{\infty} \frac{(-1)^n 1 \cdot 3 \cdots (2n-1)}{2^n z^{2n+1}}. \tag{20}$$

where $\operatorname{erfc}(z) = 1 - \operatorname{erf}(z)$, and where the product in the numerator is 1 when $n = 0$. In the author's code, (19) is used when $|z| < B/100$, and (20) is used when $|z| \geq b/100$, where B is the number of bits of precision.

Gamma function. The gamma function employs a very efficient but little-known formula due to Ronald W. Potter [28], 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 $\alpha = \operatorname{nint}(n/2 \cdot \log 2)$, 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

$$(\nu)_k = \nu(\nu+1)(\nu+2)\cdots(\nu+k-1). \tag{21}$$

Then define the functions

$$\begin{aligned} A(\nu, z) &= \left(\frac{z}{2}\right)^{\nu} \nu \sum_{k=0}^{\infty} \frac{(z^2/4)^k}{k!(\nu)_{k+1}} \\ B(\nu, z) &= \left(\frac{z}{2}\right)^{-\nu} (-\nu) \sum_{k=0}^{\infty} \frac{(z^2/4)^k}{k!(-\nu)_{k+1}}. \end{aligned} \tag{22}$$

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

$$\Gamma(\nu) = \sqrt{\frac{A(\nu, z)}{B(\nu, z)} \frac{\pi}{\nu \sin(\pi\nu)}}. \tag{23}$$

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 MPFUN20-Fort incomplete gamma function is evaluated using the following formula [19, 8.7.3]:

$$\Gamma(a,z) = \Gamma(a)\left(1 - \frac{z^a}{e^z}\sum_{k=0}^{\infty}\frac{z^k}{\Gamma(a+k+1)}\right). \tag{24}$$

Note, as with the BesselJ function, that although formula (24) 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 [19, 8.11.2]:

$$\Gamma(a,z) \approx \frac{z^{a-1}}{e^z}\sum_{k=0}^{\infty}\frac{(-1)^k(1-a)_k}{z^k}. \tag{25}$$

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

Polylogarithm function. The polylogarithm function for integer order n is defined as [19, 25.12.10]:

$$\mathrm{Li}_n(z) = \sum_{k=1}^{\infty}\frac{z^k}{k^n} \tag{26}$$

This is satisfactory for computation for $n \geq 1$ and $|z| < 1$. When $n = 0$, one uses the formula $\mathrm{Li}_0(z) = z/(1-z)$. For negative order n, one can use the formula [33]:

$$\mathrm{Li}_{-n}(z) = \frac{1}{(1-z)^{n+1}}\sum_{k=1}^{n} a_{n,k}z^k, \tag{27}$$

where the coefficients $a_{n,k}$ are given by the recurrence

$$a_{n,k} = (n+1-k)a_{n-1,k-1} + ka_{n-1,k}, \tag{28}$$

with $a_{n,1} = 1$. Because of the different algorithms and data requirements, MPFUN2020 includes three routines for polylogarithms, namely `polylog_ini`, `polylog_neg` and `polylog_pos`. Before calling `polylog_neg`, the subroutine

`polylog_ini` must be called to compute the coefficients $a_{n,k}$ for the given value of n, and the resulting array must be included in subsequent calls to `polylog_neg`. See Table 6 for details. Since the requisite auxiliary data is precomputed and passed via argument, `polylog_neg` is thread-safe. No data is required for `polylog_pos`, so it is also 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

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}. \tag{29}$$

For modest-sized arguments, the zeta function can be evaluated by means of this formula, due to Peter Borwein [14]. Select n to be the number of digits of precision required for the result. Define

$$e_j = (-1)^j \left(\sum_{k=0}^{j-n} \frac{n!}{k!(n-k)!} - 2^n \right), \tag{30}$$

where the empty sum is zero. Then

$$\zeta(s) \approx \frac{-1}{2^n(1-2^{1-s})} \sum_{j=0}^{2n-1} \frac{e_j}{(j+1)^s}. \tag{31}$$

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

$$\zeta(s) = \frac{2\cos(\pi(1-s)/2)\Gamma(1-s)\zeta(1-s)}{(2\pi)^{1-s}}. \tag{32}$$

Formulas (30), (31) and (32) are implemented as the zeta function `zeta`. No auxiliary data for this algorithm required, so it is thread-safe. In the case where the argument has an integer value, a separate function, `zeta_int` can be used instead, which is faster for such values.

An even faster algorithm for the zeta function, based on the Euler-Maclaurin summation formula, can be derived from the following [19, 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

$$\zeta(s) \approx \sum_{k=1}^{N} \frac{1}{k^s} + \frac{1}{(s-1)N^{s-1}} - \frac{1}{2N^s} + \sum_{k=1}^{\infty} \binom{s+2k-2}{2k-1} \frac{B_{2k}}{2kN^{s-1+2k}}, \quad (33)$$

where B_{2k} are the even Bernoulli numbers

$$B_{2k} = \frac{(-1)^{k-1} 2(2k)! \zeta(2k)}{(2\pi)^{2k}}. \quad (34)$$

In MPFUN2020, this scheme is implemented as `zetaem`.

Before calling `zetaem`, the subroutine `mpberne` must be called to compute the Bernoulli numbers, and the resulting array must be included in subsequent calls to `zetaem`. See Table 6 for details. Since the requisite auxiliary data is precomputed and passed via argument, `zetaem` is thread-safe.

In MPFUN2020, an advanced algorithm is employed for `mpberne`, which returns an array containing Bernoulli numbers B_{2k} for $k = 1$ to n, to P words precision. This is done by first computing $\{\zeta(2k),\, 0 \le k \le n\}$ based on the following known formulas:

$$\begin{aligned} \coth(\pi x) &= \cosh(\pi x)/\sinh(\pi x) \\ &= \frac{1}{\pi x} \cdot \frac{1 + (\pi x)^2/2! + (\pi x)^4/4! + \cdots}{1 + (\pi x)^2/3! + (\pi x)^4/5! + \cdots} \\ &= \frac{1}{\pi x} \cdot \left(1 + (\pi x)^2/3 - (\pi x)^4/45 + 2(\pi x)^6/945 - \cdots\right) \\ &= \frac{2}{\pi x} \sum_{k=1}^{\infty} (-1)^{k+1} \zeta(2k) x^{2k}. \end{aligned} \quad (35)$$

The strategy is to calculate the coefficients of the series by polynomial operations. Polynomial division is performed by computing the reciprocal of the denominator polynomial, by a polynomial Newton iteration, as follows. Let $N(x)$ be the polynomial approximation to the numerator series; let $D(x)$ be a polynomial approximation to the numerator numerator series; and let $Q_k(x)$ be polynomial approximations to $R(x) = 1/D(x)$. Then iterate:

$$Q_{k+1} = Q_k(x) + [1 - D(x)Q_k(x)]Q_k(x). \quad (36)$$

In these iterations, both the degree of the polynomial $Q_k(x)$ and the precision level in words are initially set to 4. When convergence is achieved at this

precision level, the degree is doubled, and iterations are continued, etc., until the final desired degree is achieved. Then the precision level is doubled and iterations are performed in a similar way, until the final desired precision level P is achieved. The reciprocal polynomial $R(x)$ produced by this process is then multiplied by the numerator polynomial $N(x)$ to yield an approximation to the quotient series. The even zeta values are then the coefficients of this series, scaled according to the formula above.

Once the even integer zeta values have been computed in this way, the Bernoulli numbers are computed via the formula (for $n > 0$):

$$B_{2n} = \frac{(-1)^{n-1}\ 2(2n)!\zeta(2n)}{(2\pi)^{2n}}. \tag{37}$$

7.4 FFT-based multiplication

Although the multiplication algorithm described above is very efficient, for higher levels of precision (above 700 words, or approximately 12,600 digits, based on the present author's implementation), significantly faster performance can be achieved by employing an FFT-convolution approach [16][13, 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

$$c_k = \sum_{j=0}^{2n-1} a_j b_{k-j}, \qquad 0 \le k < 2n, \tag{38}$$

where b_{k-j} is read as b_{k-j+2n} when $k - j$ is negative. This convolution can be calculated as

$$(c) = F^{-1}[F(a) \cdot F(b)], \tag{39}$$

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.

One important detail was omitted above: to avoid loss of numerical significance, the 60-bit mantissa words of the input multiprecision values are first divided into 15-bit sections. Then the FFT-convolution procedure is performed, and the result is suitably reconstituted to an output multiprecision value of 60-bit words at the end.

In contrast to the basic arithmetic algorithms, FFT-based multiplication requires precomputed FFT root-of-unity data. Thus, if one requires a precision level greater than 12,600 digits, one must call `mpinit` at the start of the user's program, in a single-threaded initialization section, before any parallel execution — see Table 4 and Section 5.5 for details.

7.5 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 $1/b$ [13, pg. 226]:

$$x_{k+1} = x_k + (1 - x_k \cdot b) \cdot x_k, \tag{40}$$

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):

$$a/b \approx (a \cdot x_n) + [a - (a \cdot x_n) \cdot b] \cdot x_n, \tag{41}$$

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. It is invoked if the working precision exceeds 12,600 digits, as with the FFT-based multiplication routine.

References

[1] D. H. Bailey, "MPFUN2015: A thread-safe arbitrary precision package (full documentation)," manuscript, 18 Nov 2020, https://www.davidhbailey.com/dhbpapers/mpfun2015.pdf

[2] D. H. Bailey, "Finding large Poisson polynomials using four-level variable precision," *Correctness 2021: 5th International Workshop on Software Correctness for HPC Applications*, 20 Nov 2021. https://www.davidhbailey.com/dhbpapers/correct-2021.pdf.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] D. H. Bailey, J. M. Borwein, J. Kimberley and W. Ladd, "Computer discovery and analysis of large Poisson polynomials," *Experimental Mathematics*, 27 Aug 2016, vol. 26 (2016), pg. 349-363, preprint at `https://www.davidhbailey.com/dhbpapers/poisson-res.pdf`.

[11] 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.

[12] 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`.

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

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

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

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

[17] 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.

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

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

[20] 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.

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

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

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

[24] 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.

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

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

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

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

[29] 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.

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

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

[32] 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.

[33] D. C. Wood, "The computation of polylogarithms," undated manuscript, https://www.cs.kent.ac.uk/pubs/1992/110/content.pdf.