ARPREC: An Arbitrary Precision Computation Package

David H. Bailey, Yozo Hida, Xiaoye S. Li and Brandon Thompson[1]
Lawrence Berkeley National Laboratory, Berkeley, CA 94720
Draft Date: 2002-10-14

Abstract

This paper describes a new software package for performing arithmetic with an arbitrarily high level of numeric precision. It is based on the earlier MPFUN package [2], enhanced with special IEEE floating-point numerical techniques and several new functions. This package is written in C++ code for high performance and broad portability and includes both C++ and Fortran-90 translation modules, so that conventional C++ and Fortran-90 programs can utilize the package with only very minor changes. This paper includes a survey of some of the interesting applications of this package and its predecessors.

1. Introduction

For a small but growing sector of the scientific computing world, the 64-bit and 80-bit IEEE floating-point arithmetic formats currently provided in most computer systems are not sufficient. As a single example, the field of climate modeling has been plagued with non-reproducibility, in that climate simulation programs give different results on different computer systems (or even on different numbers of processors), making it very difficult to diagnose bugs. Researchers in the climate modeling community have assumed that this non-reproducibility was an inevitable consequence of the chaotic nature of climate simulations. Recently, however, it has been demonstrated that for at least one of the widely used climate modeling programs, much of this variability is due to numerical inaccuracy in a certain inner loop. When some double-double arithmetic routines, developed by the present authors, were incorporated into this program, almost all of this variability disappeared [12].

The double-double (32 digits) package mentioned above and a related quad-double (64 digits) package have been used in other scientific applications as well [13]. These include a vortex roll-up simulation [10] and a bound-state calculation of Coulomb three-body systems [11]. In each case, the computational results would have been meaningless without the additional numeric precision. In fact, the Coulomb three-body calculations eventually had to rely on arbitrary precision arithmetic software, as more than 100 digit accuracy was required. It is also expected that future analysis of vortex roll-up phenomena will require significantly more precision than provided by the quad-double package.

Perhaps the most interesting applications of high precision arithmetic are in the arena of experimental mathematics. One key computation used in this research is integer relation detection. Given a vector $x = (x_1, x_2, x_3, \cdots, x_n)$ of real or complex numbers, an integer relation algorithm finds integers a_i, not all zero, such that

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

At the present time, the most effective algorithm for integer relation detection is the recently discovered "PSLQ" algorithm of mathematician-sculptor Helaman Ferguson [6]. PSLQ was

[1]This work is supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy, under contract number DE-AC03-76SF00098.

recently named one of ten "algorithms of the century" by the editors of *Computing in Science and Engineering* [3]. For almost all applications of an integer relation algorithm such as PSLQ, very high precision arithmetic must be used. Such computations are typically done using 100-500 digit precision, although some problems require thousands of digits.

The best-known result found using integer relation computations is the "BBP" formula for π, which was discovered in 1997 using a 2-level version of PSLQ implemented with the MPFUN multiple-precision arithmetic package [5]:

$$\pi \;=\; \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

This formula, after a minor manipulation, yields a very simple algorithm to directly compute hexadecimal or binary digits of π beginning at the n-th digit, without needing to compute any of the first $n-1$ digits, and requiring only a very small amount of computer memory. Since 1997, similar formulas have been found, using PSLQ computations, for numerous other mathematical constants, including logarithms of certain integers, the Riemann zeta function evaluated at certain odd integers, arctangents of certain rational numbers, and certain constants that arise in quantum field theory. In some cases, such as Catalan's constant, the new computer-discovered formulas are the best known formulas for computing high-precision numerical values (independent of the individual binary digit calculation property). A compendium of these "BBP-type" formulas is available online [1].

As an aside, recently it has been shown that these computer-discovered BBP-type formulas have deep implications for the centuries-old question of whether (and why) mathematical constants such as π and $\log 2$ are normal. Here *normal* base b means that every m-long string of digits in the base-b expansion appears with limiting frequency b^{-m}, namely the frequency one would expect if the digits had been generated at random. It is easy to show using measure theory that "almost all" real numbers are normal. Further, it is widely believed, based on computational studies, that many of the well-known constants of mathematics, including $\pi,\ e,\ \sqrt{2},\ \log 2,\ \log 10$ and many others, are normal to commonly used number bases. But it is an embarrassing fact of modern mathematics that none of these constants has ever been *proven* normal to any base.

Recently some significant progress was achieved in this arena. In particular, it has been demonstrated that the normality of the class of real constants for which BBP-type formulas are known (which includes $\pi,\ \log 2$ and others), reduces to a plausible conjecture from the field of chaotic dynamics [9]. In the latest development along this line, a certain class of real numbers, uncountably infinite in number (but not yet including π), has been proven normal base 2 [9].

In another application of high-precision computing to mathematical research, PSLQ computations have been used to investigate sums of the form

$$S(k) \;=\; \sum_{n>0} \frac{1}{n^k \binom{2n}{n}}$$

For small k, these constants are known to satisfy simple mathematical identities, such as $S(4) = 17\pi^4/3240$. For many years researchers have sought generalizations of these formulas for $k > 4$. Recently some general formulas were found using PSLQ. In particular, the constants $\{S(k) | k = 5 \ldots 20\}$ have now been evaluated in terms of multiple zeta values, which are defined by

$$\zeta(s_1, s_2, \cdots, s_r) \;=\; \sum_{k_1 > k_2 > \cdots > k_r > 0} \frac{1}{k_1^{s_1}\, k_2^{s_2} \cdots k_r^{s_r}}$$

and multiple Clausen values of the form

$$M(a,b) \;=\; \sum_{n_1>n_2>\ldots>n_b>0} \frac{\sin(n_1\pi/3)}{n_1^a} \prod_{j=1}^{b} \frac{1}{n_j}$$

A sample result, obtained using PSLQ, is the following:

$$\begin{aligned} S(9) \;=\; & \pi\left[2M(7,1)+\frac{8}{3}M(5,3)+\frac{8}{9}\zeta(2)M(5,1)\right]-\frac{13921}{216}\zeta(9) \\ & +\frac{6211}{486}\zeta(7)\zeta(2)+\frac{8101}{648}\zeta(6)\zeta(3)+\frac{331}{18}\zeta(5)\zeta(4)-\frac{8}{9}\zeta^3(3) \end{aligned}$$

Recently $S(20)$ was evaluated with this methodology, which involved an integer relation problem of size $n = 118$. This solution required 5,000 digit arithmetic and several hours of computation on a high-end workstation. The solution is given explicitly in [6].

One other result that we shall mention here arose from the numerical discovery, using PSLQ, by British physicist David Broadhurst that

$$\alpha^{630}-1 \;=\; \frac{(\alpha^{315}-1)(\alpha^{210}-1)(\alpha^{126}-1)^2(\alpha^{90}-1)(\alpha^3-1)^3(\alpha^2-1)^5(\alpha-1)^3}{(\alpha^{35}-1)(\alpha^{15}-1)^2(\alpha^{14}-1)^2(\alpha^5-1)^6\alpha^{68}}$$

where $\alpha = 1.176280818259917\ldots$ is the larger real root of Lehmer's polynomial

$$0 \;=\; \alpha^{10}+\alpha^9-\alpha^7-\alpha^6-\alpha^5-\alpha^4-\alpha^3+\alpha+1$$

The relation led Broadhurst to believe that a valid ladder of polylogarithms exists at order $n = 17$, contrary to a suggestion in [15]. Later, he and one of the present authors were able to find 125 non-zero integers a, b_j, c_k, up to 292 digits in size, such that the relation

$$a\,\zeta(17) \;=\; \sum_{j=0}^{8} b_j\,\pi^{2j}(\log\alpha)^{17-2j} + \sum_{k\in D(\mathcal{S})} c_k\,\mathrm{Li}_{17}(\alpha^{-k})$$

holds to more than 50,000 decimal digits. This set of 125 integers was found in more than more way. One of these computations was performed on the NERSC Cray T3E, a highly parallel system at Lawrence Berkeley Laboratory. In this calculation one of the present authors employed a three-level, multi-pair PSLQ algorithm, implemented using the MPFUN multiple-precision software, and using MPI for parallel execution. This computation required 50,000 decimal digit arithmetic, and approximately 44 hours on 32 CPUs of the T3E, completing after 236,713 iterations [7].

2. Overview of the ARPREC Package

The new Arbitrary Precision (ARPREC) computation package is based in part on the Fortran-90 MPFUN package [2], which in turn is based on an earlier Fortran-77 package [4]. Both of the earlier packages were targeted to a Fortran environment. With the Fortran-77 MPFUN package, users indicate which variables are to be treated as multiple precision by means of special comments inserted in the code. A translation tool then transforms the code into a program that includes explicit calls to routines from the MPFUN library. In the Fortran-90 version of MPFUN, the object-oriented facilities built into the Fortran-90 language (notably custom datatypes and operator overloading) are exploited to provide the same functionality without the

need of a translation tool. In addition, the Fortran-90 version of MPFUN incorporates some advanced algorithms that provide better accuracy and performance.

The ARPREC library improves upon the Fortran-90 MPFUN package in several ways, and extends its functionality to the realm of C and C++ programs. In particular, the ARPREC package features:

- Code written in C++ for high performance and broad portability.
- C++ and Fortran-90 translation modules that permit conventional C++ and Fortran-90 programs to utilize the package with only very minor changes to source code.
- Arbitrary precision integer, floating and complex datatypes.
- Support for datatypes with differing precision levels.
- Inter-operability with conventional integer and floating-point datatypes.
- Common transcendental functions (sqrt, exp, sin, etc).
- Quadrature routines (for numerical integration).
- PSLQ routines (for integer relation detection).
- Special routines for extra-high precision (> 1000 digits) computation.

The ARPREC library uses arrays of 64-bit floating point numbers to represent high precision floating point numbers. The first word contains the number of words allocated for the array. The absolute value of the second word of the array is the number of words in the mantissa of the arbitrary precision number, and the sign of the second word represents the sign of the arbitrary precision number. The third word represents the exponent of the arbitrary precision number (powers of 2^{48}). Mantissa words begin with word four. Each mantissa word contains an integer value in $[0, 2^{48})$, or in other words successive 48-bit sections of the arbitrary precision number. Thus the arbitrary precision array $(a_k,\ 1 \le k \le n+5)$ represents the value

$$A \quad = \quad \pm\, 2^{48 a_3} \sum_{k=0}^{n-1} a_{k+4} 2^{-48k}$$

where $\pm = \text{sign}(a_2)$ and $n = |a_2|$. The last two words in the array are for convenience during arithmetic operations. The earlier MPFUN package uses a similar format, except that the initial word containing the number of words allocated is absent, and 32-bit floating point arrays are used instead of 64-bit floating-point arrays.

3. Arithmetic Operations

The most important algorithmic difference between ARPREC and MPFUN is in the low-level arithmetic algorithms. In both packages, multiplication relies on a variation of the common long multiplication algorithm. In ARPREC, each mantissa word contains an integer value in the range $[0, 2^{48} - 1)$, so each row of the multiplication pyramid contains values in the range $[0, 2^{96} - 1)$. Since IEEE 64-bit floating-point arithmetic is only accurate to 53 bits, it is clear that advanced techniques must be used here to resolve all 96 bits. In the ARPREC package this is done by utilizing the following algorithms from the authors' earlier double-double and quad-double package (which in turn is based on earlier work by Priest, Kahan and Knuth) [13]:

Double + double. This computes the high- and low-order words of the sum of two IEEE 64-bit values a and b.

1. $s \leftarrow a \oplus b$
2. $v \leftarrow s \ominus a$
3. $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
4. return (s, e).

Split. This splits an IEEE 64-bit value a into $a_{\rm hi}$ and $a_{\rm lo}$, each with 26 bits of significand and one hidden bit, such that $a = a_{\rm hi} + a_{\rm lo}$.

1. $t \leftarrow (2^{27} + 1) \otimes a$
2. $a_{\rm hi} \leftarrow t \ominus (t \ominus a)$
3. $a_{\rm lo} \leftarrow a \ominus a_{\rm hi}$
4. return $(a_{\rm hi}, a_{\rm lo})$.

Double × double. This computes the high- and low-order words of the product of two IEEE 64-bit values a and b.

1. $p \leftarrow a \otimes b$
2. $(a_{\rm hi}, a_{\rm lo}) \leftarrow$ SPLIT(a)
3. $(b_{\rm hi}, b_{\rm lo}) \leftarrow$ SPLIT(b)
4. $e \leftarrow ((a_{\rm hi} \otimes b_{\rm hi} \ominus p) \oplus a_{\rm hi} \otimes b_{\rm lo} \oplus a_{\rm lo} \otimes b_{\rm hi}) \oplus a_{\rm lo} \otimes b_{\rm lo}$
5. return (p, e).

With regards to the double × double algorithm, we should note that some processors, notably IBM PowerPC and RS6000 processors and Intel IA-64 processors, have a "fused multiply-add" (FMA) instruction that greatly simplifies double × double operations. In this case one can simply write $p \leftarrow a \otimes b$ and $e \leftarrow fl(a \otimes b - p)$. However, it is often necessary to specify a special compiler option (such as -qstrict on IBM systems) to insure that this code is performed as written.

The addition and multiplication algorithms above produce a high-order word (s or p) and a low-order word e, where s or p is the closest 53-bit floating-point approximation to the result, and where e is the difference between the exact value and s or p. In the ARPREC package, when two 48-bit integer values are multiplied, what we really require are the high-48 bits and the low-48 bits of the result. These can be obtained as $p' = 2^{48}\ {\rm nint}(p/2^{48})$ and $e' = (p - p') \oplus e$. For arguments in the range $[0, 2^{48})$, the nearest integer function nint can be efficiently calculated as ${\rm nint}(a) = (a \oplus 2^{52}) \ominus 2^{52}$.

In order to significantly accelerate long multiply operations, 32 rows of the multiplication pyramid are added before releasing carries. This is safe, because 32 times a 48-bit integer is a 53-bit integer, which can be represented exactly in a 64-bit IEEE word. Additionally, if n words is the working precision level, then only the first $n+1$ words of the $2n$-long multiplication pyramid are computed.

The double × double multiply algorithm above requires 17 flops (three flops on systems with an FMA). The scheme to reduce results to 48-bit form adds another six flops. The resulting cost figure is significantly higher than in MPFUN. However, because the word size in ARPREC is twice as large (48 bits versus 24), the inner loop of the long multiply routine is executed only one-fourth as often. Furthermore, the ARPREC operations indicated above can be done entirely in registers, without memory references, whereas the inner loops in MPFUN involve vector memory fetch and store operations that are much more expensive. Main memory latency

on most present-day microprocessors exceeds 100 clock periods, and is headed higher. Even cache latency is more than 10 clock periods (for level two cache) on most new systems. Thus the ARPREC design that involves register arithmetic operations is likely to result in significant performance advantages in the years ahead.

4. Taylor Series Routines

For moderate precision inputs, the ARPREC exponential, sine and cosine routines employ Taylor series schemes, but with some improvements from the corresponding MPFUN routines. One of these improvements is to dynamically adjust the working precision as the computation progresses. After the first several terms of a Taylor's series have been computed and summed, the order of magnitude of the result is known. Since the exponent of the result is known, and the number of words of precision to be computed is known, the exponent that relates to the last word of the result is known. With each succesive term, the exponent in words of the new term is estimated before computing the term. This value is used to determine the precision to which the term will need to be computed. In this way, the precision required for successive terms decreases, resulting in an overall savings of approximately one-half in these routines.

The ARPREC sine-cosine routine employs this scheme, and in addition employs a finer reduction of the input argument than in MPFUN. The MPFUN routine finds the nearest multiple of $\pi/16$ to the argument, and reduces the argument to the difference between this multiple and the argument. This reduced argument is used in the Taylor series, and then trigonometric addition identities are used to calculate the sine and cosine of the original argument. The new version finds the nearest multiple of $\pi/256$ rather than $\pi/16$, and stores the sines and cosines of $k\pi/256$ in a table. Subsequent calls to the routine thus do not have to recompute these values. This requires some storage space, but provides significant savings in computation time for applications where many sine and/or cosine evaluations are required.

5. Extra-High Precision Multiplication

For very high precision (above about 1000 decimal digits), the ARPREC package, like the MPFUN package, uses an FFT routine to perform multiplications (since multiplication of two fixed-radix numbers is merely the linear convolution of the arrays, followed by release of carries). In order to perform the convolution accurately, each 64-bit word in the two input arrays is first split into four words, each with 12 bits. The four-step FFT algorithm is then applied to this data [14]. The two resulting arrays of complex-valued results, one array for each argument of the product, are multiplied componentwise, followed by an inverse FFT using the FFT. The result of the inverse transform is the convolution of the 12-bit integer arrays. This is converted into a 48-bit integer array, then carries are released to yield the final product result. In addition to algorithmic changes that stem from input arrays having 48 bits per word instead of 24, the ARPREC code employs a radix-4 variant of the four-step FFT algorithm, which results in slightly faster execution. An additional improvement is to eliminate some unnecessary data copy operations.

6. Newton Iteration Routines

Several routines in the ARPREC library utilize Newton iteration algorithms, with a dynamic level of precision that approximately doubles with each iteration. Some of these routines have been improved in the new ARPREC package. For example, the square root routine now employs

the iteration

$$x_{k+1} \quad = \quad x_k + 1/2 \cdot (A - x_k^2)/x_k,$$

where the division is performed in half precision. The MPFUN routine requires three multi-precision multiplications per iteration, plus one half-precision multiplication. The ARPREC routine requires just one multiplication and one half-precision division. This change results in a speedup of about 10-20% at modest precision levels. Because division is significantly slower than multiplication at very high precision levels, the advanced high precision square root routine has not been changed.

7. Performance

The performance of the ARPREC package was tested by running the following simple program using both ARPREC and MPFUN++ (a C++ translation of MPFUN, available at `http://www.cs.unc.edu/Research/HARPOON/mpfun++`):

```
a = 1.0;
b = 9.03;
c = 6.01;
for(i=1;i<80000;i++) {
a = ((a*a) + b) / c;
}
```

The value of A converges to 3.0 properly using both libraries. At 400 requested digits, the ARPREC library requires 29 64-bit mantissa words, while MPFUN++ requires 56 32-bit mantissa words. Operator overloading was avoided in the implementation, and the call to the multiply routine was timed independently of the other operations. These runs were performed on a Sun Ultra 1, with a 167 MHz processor. The same compiler optimization flags were used in both runs. The average multiplication time for 80,000 multiplications was 10.80 seconds for ARPREC versus 14.78 seconds for MPFUN.

In an additional test of the new ARPREC package, we recently (July 2002) repeated the PSLQ computation, mentioned in the introduction, of the 125 integers in the relation involving the root of Lehmer's polynomial. This run was made on the IBM SP (Seaborg) system in the NERSC computer center at the Lawrence Berkeley National Laboratory. For this computation, 64 CPUs (four nodes) were used. The computation completed in 236,555 iterations (slightly different than before, due to minor numerical differences), requiring only 16.6 hours. 50,000 digit arithmetic (using the ARPREC software) was used here, as in the previous computation.

In the latest run, the minimum and maximum y vector entries at the point of relation detection were approximately $1.6326 \times 10^{-49772}$ and $1.3489 \times 10^{-36401}$, respectively. The ratio of these two values, which can be thought of as a "confidence ratio" of the results, is approximately $8.2623 \times 10^{-13372}$. This very small ratio, together with the fact that the first value is nearly 10^{-50000}, means that it is exceedingly unlikely that the results produced by this computation are merely a spurious artifact of numerical round-off error (although this cannot be taken as rigorous proof of these relations). It is also very unlikely that any significant error occurred in the ARPREC package during this extended computation. We believe this to be the largest single integer relation problem ever solved.

References

[1] David H. Bailey, "A Compendium of BBP-Type Formulas," manuscript, available at `http://www.nersc.gov/~dhbailey/dhbpapers`.

[2] David H. Bailey, "A Fortran-90 Based Multiprecision System", *ACM Transactions on Mathematical Software*, vol. 21, no. 4, 1995, pg. 379–387.

[3] David H. Bailey, "Integer Relation Detection," *Computing in Science and Engineering*, Jan-Feb 2000.

[4] David H. Bailey, "Multiprecision Translation and Execution of Fortran Programs," *ACM Transactions on Mathematical Software*, vol. 19, no. 3 (Sept. 1993), pg. 288–319.

[5] David H. Bailey, Peter B. Borwein and Simon Plouffe, "On The Rapid Computation of Various Polylogarithmic Constants", *Mathematics of Computation*, vol. 66, no. 218, 1997, pg. 903–913.

[6] David H. Bailey and David Broadhurst, "Parallel Integer Relation Detection: Techniques and Applications," to appear in *Mathematics of Computation*.

[7] David H. Bailey and David J. Broadhurst, "A Seventeenth-Order Polylogarithm Ladder," manuscript, 2001, available at `http://www.nersc.gov/~dhbailey/dhbpapers`.

[8] David H. Bailey and Richard E. Crandall, "On the Random Character of Fundamental Constant Expansions," *Experimental Mathematics*, vol. 10, no. 2 (June 2001), pg. 175-190.

[9] David H. Bailey and Richard E. Crandall, "Random Generators and Normal Numbers," manuscript, Mar. 2002, available at `http://www.nersc.gov/~dhbailey/dhbpapers`.

[10] David H. Bailey, R. Krasny and R. Pelz, "Multiple Precision, Multiple Processor Vortex Sheet Roll-Up Computation," *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 1993, SIAM, Philadelphia, pg. 52–56.

[11] David H. Bailey and Alexei M. Frolov, "Universal Variational Expansion for High-Precision Bound-State Calculations in Three-Body Systems," *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 35 (2002), pg. 1–12.

[12] Yun He and Chris Ding, "Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications," *Journal of Supercomputing*, vol. 18 (2001), pg. 259–277.

[13] Yozo Hida, Xiaoye S. Li and David H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic," *Proceedings of ARITH-15*, IEEE Computer Society, 2001.

[14] Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelplhia, 1992.

[15] Don Zagier, "Special Values and Functional Equations of Polylogarithms", Appendix A in Leonard Lewin, editor, *Structural Properties of Polylogarithms*, Mathematical Surveys and Monographs, vol. 37, American Mathematical Society, Providence, RI, 1991.