Finding large Poisson polynomials using four-level variable precision

David H. Bailey
Department of Computer Science, University of California, Davis
Computational Research Department, Lawrence Berkeley National Laboratory (retired)
Davis, California, USA and Berkeley, California, USA
dhbailey@ucdavis.edu

***Abstract*—The tension between "correct and reproducible" computing and "high-performance" computing is particularly acute in computational mathematics applications, some of which require very high numeric precision (hundreds or thousands of digits) to produce correct and reproducible results, resulting in very long run times. One solution is to employ a variable precision design, namely using ordinary single- or double-precision arithmetic as much as possible, and shifting to higher levels of precision only when necessary. But managing a large variable precision computation is a significant programming challenge.**

This study presents results using a new arbitrary precision package and a new four-level variable precision application code to study a class of polynomials arising from the Poisson potential function of mathematical physics. The application employs four levels of precision: IEEE double, IEEE quad, medium precision (typically 100-1,000 digits) and full precision (typically 5,000 to 50,000 digits). Using this software, we have computed the minimal polynomials associated with the Poisson potential function, not just in the special case $x = 1/s$ and $y = 1/s$ (for integers $1 \le s \le 50$), as in an earlier study, but also for many instances of the broader class of rationals $x = p/s$ and $y = q/s$, for $1 \le p, q < s/2 \le 50$. In this paper, we present some initial results of these computations, and also present some lessons learned from using extreme variable precision in a computational mathematics application.

***Index Terms*—variable precision, arbitrary precision, PSLQ**

I. Reproducibility and numerical precision

Ensuring correctness and reproducibility is a premier objective in all fields of scientific research. Unfortunately, the field of high-performance computing has lagged such efforts in other fields of research. One of the difficulties noted in a 2014 ICERM workshop was the issue of numerical reproducibility:

> Numerical reproducibility has emerged as a particularly important issue, since the scale of computations has greatly increased in recent years, particularly with computations performed on many thousands of processors and involving similarly large datasets. Large computations often greatly magnify the level of numeric error, so that numerical difficulties that were once of little import now are large enough to alter the course of the computation or to draw into question the overall validity of the results. [5]

One remedy to numerical difficulties is to employ a high level of precision throughout the code. But this is usually quite wasteful, since in most applications high precision is only required in a few places. A better solution is to employ a variable precision design, using low precision arithmetic as much as possible, and switching to higher levels only when necessary. Compared with using a fixed level of precision for the entire application, a variable precision framework usually features faster processing, better cache utilization, lower memory usage, lower offline data storage and lower energy costs (depending on details such the algorithm being used and the nature of the iterative convergence). But unless a variable precision computation is carefully managed, there is potential for loss of accuracy and reproducibility.

	Number of bits				
Name	Sign	Exp.	Mant.	Hidden	Digits
IEEE half	1	5	10	1	3
ARM half	1	5	10	1	3
bfloat16	1	8	7	1	2
IEEE single	1	8	23	1	7
IEEE double	1	11	52	1	15
IEEE extended	1	15	64	0	19
IEEE quad	1	15	112	1	34
double-double	1	11	104	2	31
quad-double	1	11	208	4	62
double-quad	1	15	224	1	68
arbitrary	varies	varies	varies	varies	varies

TABLE I
Commonly used precision formats

At the present time, a number of different floating-point formats are in use in the high-performance computing world; see Table I. The three 16-bit formats are used heavily in machine learning; IEEE 32-bit arithmetic is used in graphics and other compute-intensive applications. IEEE 64-bit is a staple of larger applications. IEEE 128-bit (quad) is not yet implemented in hardware on most processors, but is supported in software with several compilers, including the gcc, gfortran and Intel compilers, and is being used in a number of scientific applications, for example in mathematical biology [15]. Even higher levels of precision, typically ranging from 100 to 100,000 digits and requiring special software, are needed in some studies in computational mathematics and physics. We will address one such application in this paper.

II. Earlier work on Poisson polynomials

Lattice sums related to the Poisson potential function, which naturally arise in studies of gravitational and electrostatic potentials, have been studied for many years in the mathematical

physics community, for example in [1], [13], [14]. Recently interest in this topic has been rekindled in light of some intriguing applications to practical image processing. These developments have underscored the need to better understand the underlying theory.

In two earlier papers [4], [6], the present author, Jonathan Borwein (deceased 2016), Richard Crandall (deceased 2012), I. J. Zucker, Jason Kimberley and Watson Ladd analyzed the simple two-dimensional case:

$$\phi_2(x,y) = \frac{1}{\pi^2} \sum_{m,n \text{ odd}} \frac{\cos(m\pi x)\cos(n\pi y)}{m^2+n^2}. \qquad (1)$$

By employing a computational approach, these authors empirically discovered and then proved the intriguing fact that when x and y are *rational numbers*, then

$$\phi_2(x,y) = \frac{1}{\pi} \log \alpha(x,y), \qquad (2)$$

where $\alpha(x,y)$ is an *algebraic number*, namely the root of a degree-m polynomial with integer coefficients, for some m.

This result can be explored computationally as follows: Given rationals x and y, compute $\alpha = \exp(8\pi\phi_2(x,y))$ to high precision, then calculate the $(m+1)$-long vector $(1, \alpha, \alpha^2, \cdots, \alpha^m)$, which is input to a variant of the PSLQ algorithm to discover the coefficients of the polynomial of degree m, if it exists, satisfied by α (this process will be described in more detail below).

Some results from one earlier study [4] are shown in Table II. Among other things, note that when s is an even integer, the minimal polynomial corresponding to the case $x = y = 1/s$ is always palindromic (i.e., coefficient $a_k = a_{m-k}$, where m is the degree). For instance, when $s = 8$ note that the coefficients of the corresponding polynomial $1 - 88\alpha + 92\alpha^2 - 872\alpha^3 + 1990\alpha^4 - 872\alpha^5 + 92\alpha^6 - 88\alpha^7 + \alpha^8$ read the same backwards and forwards. Does this extend to larger even integers s?

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 II
SAMPLE OF POLYNOMIALS PRODUCED IN EARLIER STUDY [4].

Furthermore, Jason Kimberley of the University of Newcastle, Australia observed that the degree $m(s)$ of the minimal polynomial associated with the case $x = y = 1/s$ appears to be given by the following rule: Set $m(2) = 1/2$. Otherwise for primes p congruent to 1 modulo 4, set $m(p) = (p-1)^2/4$, and for primes p congruent to 3 modulo 4, set $m(p) = (p^2-1)/4$. Then for any other positive integer s whose prime factorization is $s = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$,

$$m(s) = 4^{r-1} \prod_{i=1}^{r} p_i^{2(e_i-1)}\, m(p_i). \qquad (3)$$

Does Kimberley's formula hold for larger integers s?

In a second earlier study [6], we explored this with more capable tools: (a) a new thread-safe, high-level arbitrary precision package, 3X faster than the package used in the original study; (b) a new three-level multipair PSLQ integer relation algorithm, 4X faster than the scheme used earlier; and (c) a parallel implementation on a 16-core system. These enhancements resulted in a combined speedup of up to 156X.

Using this improved software, we confirmed that Kimberley's formula holds for all integers s up to 52 (except for $s = 41, 43, 47, 49, 51$, which were too expensive to test) and also for $s = 60$ and $s = 64$. These computations employed up to 64,000-digit precision, and produced polynomials with degrees up to 512 and integer coefficients up to 10^{229}.

By doing Google searches on the coefficients of the resulting polynomials, we found a connection to a 2010 paper by Savin and Quarfoot [17]. These investigations subsequently led to a proof, given in [6], that Kimberley's formula (3) is valid, and also a proof of the fact that when s is even, the minimal polynomial corresponding to $x = y = 1/s$ is palindromic (i.e., coefficient $a_k = a_{m-k}$, where m is the degree).

As gratifying as these results are, they still leave many questions unanswered. In particular, note that the above analyses focused on the special case $x = y = 1/s$ for integer $1 \le s \le 50$. What happens for more general rational arguments, say for $x = p/s$, $y = q/s$ for integers $1 \le p, q < s \le 50$, or for even higher bounds than 50? Is there an analogue to Kimberley's formula for this more general class? Computationally exploring this larger class is clearly much more expensive than just the special case $x = y = 1/s$. Can the computational tools used above be further accelerated?

In this paper, we describe a significantly enhanced version of the computational software. We first developed a new arbitrary precision software package that supports a dynamically varying level of precision, with roughly 4X faster performance (in the all-Fortran version) than the version used earlier. Secondly, we developed a new four-level multipair PSLQ code, employing double precision, quad precision, medium precision (typically 100-1,000 digits) and full precision (typically 5,000 to 50,000 digits), which produces some additional speedup over the three-level code used before. In this paper, we briefly describe this new software and present some initial results.

III. THREAD-SAFE ARBITRARY PRECISION SOFTWARE

Arbitrary precision computations such as the Poisson polynomial problem discussed in this paper are both very compute-intensive and very unforgiving of software bugs or numerical

inaccuracy. For example, a computation with 100-digit precision arithmetic is typically 100X more expensive than one that requires only ordinary double precision, and a computation with 1,000-digit precision is typically 1,000X to 10,000X more expensive than double, depending on whether transcendental functions are involved. Because the performance rates of individual processor cores are no longer rapidly increasing, state-of-the-art computing applications such as this must employ multicore and multi-node parallelism [8].

In arbitrary precision applications, it is difficult to achieve significant parallel speedup *within* a single arbitrary precision arithmetic operation, but significant speedups can be achieved at the *application* level (e.g., parallelizing a loop containing arbitrary precision operations), using a shared-memory, multithreaded parallel model such as OpenMP. However, such computations must be entirely "thread-safe."

Unfortunately, most of the available software packages for arbitrary precision computation are *not* thread-safe, particularly in a dynamically variable precision application. Some packages, for example, employ global read/write arrays to control the current working precision and to support transcendental functions, but these features destroy thread safety. In some cases, these difficulties might be rectified by employing parallel programming pragmas, but such changes would need to be made *within* the package itself, which is typically not an option, and may reduce overall performance.

Until recently, only one arbitrary precision floating-point package was certified thread-safe, namely the GNU MPFR package [11]. The MPFR package features correct rounding, includes numerous transcendental and special functions, and achieves very fast timings. However, MPFR is only a low-level library; it does not include high-level language bindings, which, practically speaking, are required to support full-scale application codes. Also, MPFR requires a lengthy installation procedure with administrator permission (to install MPFR and GMP on the user's computer), which discourages many users. Further, if a thread-safe build option is not properly invoked during installation, the resulting software is not thread-safe.

IV. The MPFUN2020 arbitrary precision package

In the previous study [6], we utilized a package for arbitrary precision floating-point computation, developed by the present author, named MPFUN2015 [2]. It is available in two versions: (a) a self-contained, all-Fortran version; and (b) a version that calls the MPFR package for low-level operations, which is approximately 5X faster. Both versions are thread-safe, by avoiding global read/write data, and by incorporating the current working precision into the data structure of every arbitrary precision datum. Both versions also include a high-level Fortran language interface, implemented using custom datatypes and operator overloading, which supports arbitrary precision real and complex data, numerous transcendentals and special functions. In most cases, an application using ordinary IEEE double precision can be converted to use the MPFUN2015 package simply by changing a few type statements and a few other minor modifications.

However, this software has some limitations. The all-Fortran version of MPFUN2015 is certainly easy to install; the library can be compiled in just a few seconds with a short script. But the fact that it is 5X slower than the MPFR-based version is a significant handicap. On the other hand, the MPFR-based version based is much more complicated to install, as noted above. Can the best features of the two versions be combined?

To that end, the present author has written a new arbitrary precision package, named MPFUN2020 [3]. It features:

- An all-Fortran design, based on eight-byte integer arithmetic, that can be compiled in a few seconds with any Fortran-2008 compliant compiler, including the gfortran, Intel and NAG Fortran compilers, on a variety of systems, including Mac OS X and various Linux platforms.
- A 100% thread-safe design as before.
- A full-featured high-level Fortran language interface, so that only type statements and a few other minor changes are required to convert most double precision codes.
- Support for common transcendental functions (sin, cos, exp, etc.) and several special functions.
- Support for arbitrary precision real and complex datatypes, plus interfaces for IEEE double and quad.
- Support for both a full and a medium precision datatype, which in some applications (such as Poisson problems) results in lower memory usage and memory traffic.
- FFT-based multiplication for faster performance at very high precision.
- Overall performance rates that are several times faster than the earlier all-Fortran package, and nearly as fast as the MPFR-based version.

The all-Fortran version of MPFUN2020 is approximately 26,000 lines of Fortran. Details and code are available at [3].

In addition, the present author has also prepared an updated MPFR-based version for the MPFUN2020 package. While it still requires a lengthy initial installation procedure (to install MPFR and GMP), it runs somewhat faster than the all-Fortran version of MPFUN2020. The two versions are "plug-compatible," in the sense that if a simple guideline is followed, application codes written to work with one version will also work with the other, without any changes whatsoever. The MPFR-based version of MPFUN2020 is approximately 20,000 lines of Fortran; it calls the MPFR library (93,000 lines of C), which in turn calls the GMP library (83,000 lines of C).

Both the all-Fortran and the MPFR-based versions have been carefully tested, using rigorous test programs included in the package and other tests. Further, as we will see in Sections V and VIII, the Poisson polynomial calculations described in this paper are very rigorous tests of computational integrity.

V. A four-level multipair PSLQ code

Given an $(m+1)$-long input vector $X = (x_i)$ of real numbers (given as very high-precision floating-point values), an integer relation algorithm attempts to find a nontrivial $(m+1)$-long vector of integers (a_i) such that

$$a_0 x_0 + a_1 x_1 + a_2 x_2 + \cdots + a_m x_m = 0, \qquad (4)$$

to within the tolerance of the numeric precision being used.

In the application discussed in this paper, where we are given a high-precision floating-point value α that is suspected to be an algebraic number of degree m, one computes the $(m+1)$-long vector $X = (1, \alpha, \alpha^2, \cdots, \alpha^m)$ and inputs the vector to an integer relation-finding algorithm. If an integer relation (a_i) is found for X that holds to the level of precision being used, then the resulting vector of integers may be the coefficients of an integer polynomial of degree m satisfied by α (subject to further verification).

As a simple illustration, suppose one suspects that the real constant α, whose numerical value to 80 digits is 2.11959126982917513132984833493468711062807145783249543921443000325120624764609847..., is an algebraic number of degree eight. After computing the vector $(1, \alpha, \alpha^2, \cdots, \alpha^8)$ and applying the multipair PSLQ integer relation algorithm, the relation $(1, -216, 860, -744, 454, -744, 860, -216, 1)$ is produced, so that α appears to satisfy the polynomial $1 - 216\alpha + 860\alpha^2 - 744\alpha^3 + 454\alpha^4 - 744\alpha^5 + 860\alpha^6 - 216\alpha^7 + \alpha^8 = 0$. This is the fourth of the polynomials listed in Table II.

The multipair PSLQ algorithm [7] is a more efficient and moderately parallelizable variant of PSLQ, the most widely used integer relation algorithm (although some use a variant of LLL [10]). Iterations of the multipair PSLQ algorithm develop a sequence of invertible integer matrices A_n, their inverses B_n and real matrices H_n (in lower quadrature form), so that the reduced vector $y = B_n \cdot X$ has steadily smaller entries, until one entry of y is smaller than the epsilon specified for detection, or precision has been exhausted.

The size of the drop in $\min(|y_i|)$ when the relation is detected can be viewed as a confidence level that the relation so discovered is a real mathematical relation and not merely a numerical artifact — see Figure 1. A drop of 20 or more orders of magnitude almost always indicates a real relation, and, by implication, that all hardware and software performed impeccably. In the calculations described in Section VIII, all listed results exhibited a drop of at least 300 orders.

Integer relation detection (by any algorithm) requires very high precision: at least $(m+1)\max_k \log_{10}|a_k|$ digits, or there is no chance of finding the underlying relation. Multipair PSLQ is very efficient with precision, compared with other integer relation algorithms, in the sense that it can typically detect a relation when the numeric precision is only a few percent higher than this minimum bound [7].

Our most recent study [6] of the Poisson polynomial problem employed a three-level variable precision code, based on a scheme sketched in [7]: (a) double precision; (b) medium precision, typically 100-1000 digits; and (c) full precision, typically 5,000 to 50,000 digits. With this scheme, almost all iterations of the multipair PSLQ algorithm are performed in double precision. When an entry of the double precision y vector is smaller than 10^{-14}, or when an entry of the double precision A or B array contains a value exceeding 10^{13}, the medium precision arrays are updated from the double precision

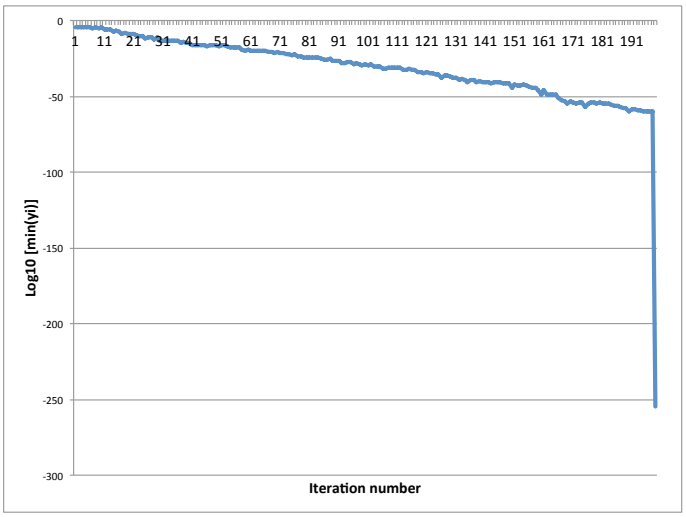

Fig. 1. Plot of $\log_{10}(\min|y_i|)$ versus iteration number in a typical multipair PSLQ run. Note the sudden drop at iteration 199 to the multiprecision "epsilon" (10^{-250} in this run), a drop of 200 orders of magnitude.

arrays using matrix multiplication via the formulas

$$y := \hat{B} \cdot y,\ B := \hat{B} \cdot B,\ A := \hat{A} \cdot A,\ H := \hat{A} \cdot H, \qquad (5)$$

where the hat notation indicates the double precision arrays.

When an entry of the medium precision y vector is smaller than the medium precision epsilon, or when the medium precision A or B array contains a value nearly as large as the medium precision maximum, then the full precision arrays are updated from the medium precision arrays using similar formulas. Full details of this process will not be given here, but suffice it to say that considerable care must be taken in this implementation to correctly handle all precision handoffs and scenarios that may arise.

For the study described in this paper, we developed and deployed a new *four-level* implementation of the multipair PSLQ algorithm, employing: (a) IEEE double precision; (b) IEEE quad precision; (c) medium precision, typically 100-1000 digits; and (d) full precision, typically 5,000 to 50,000 digits. IEEE quad precision was handled via the software support for 16-byte reals that is now included in the gfortran and Intel Fortran compilers, among others.

The principal savings of including IEEE quad precision in the precision hierarchy occurs when the double precision iterations encounter a value that too large to be represented exactly (i.e. a value exceeding 2^{53}). In such instances, the current iteration must be abandoned and the calculation must retreat to double precision arrays saved in an earlier iteration; ten iterations are then repeated using quad precision. Using quad precision instead of medium precision for these repeated iterations yields significant run-time savings, compared with the three-level code used in the previous study. Using quad precision also yields run-time savings in the initial iterations, typically several hundred in number, which must be performed in higher precision.

On the other hand, this four-level code is correspondingly more complicated than the three-level code, increasing the number of lines of code in the PSLQ section by nearly 50%. *As before, considerable care must be taken to correctly handle all precision handoffs and other scenarios that may arise*.

VI. High-level computational algorithm

Here is the high-level algorithm we employed to discover the Poisson polynomials, condensed and updated from [6]:

1) Given rationals $x = p/s$ and $y = q/s$ (typically satisfying $1 \le p, q < s/2 \le 50$), select a conjectured minimal polynomial degree m, a medium precision level P_1 digits, a full precision level P_2 digits and other parameters for the run.
2) Calculate $\phi_2(x, y)$ to P_2-digit precision using the following formula from [4]:

$$\phi_2(x, y) = \frac{1}{2\pi} \log \left| \frac{\theta_2(z, q)\theta_4(z, q)}{\theta_1(z, q)\theta_3(z, q)} \right|, \tag{6}$$

where $q = e^{-\pi}$ and $z = \frac{\pi}{2}(y + ix)$. Compute the four theta functions using the following rapidly convergent formulas from [9, p. 52]:

$$\begin{aligned}
\theta_1(z, q) &= 2 \sum_{k=1}^{\infty} (-1)^{k-1} q^{(2k-1)^2/4} \sin((2k-1)z), \\
\theta_2(z, q) &= 2 \sum_{k=1}^{\infty} q^{(2k-1)^2/4} \cos((2k-1)z), \\
\theta_3(z, q) &= 1 + 2 \sum_{k=1}^{\infty} q^{k^2} \cos(2kz), \\
\theta_4(z, q) &= 1 + 2 \sum_{k=1}^{\infty} (-1)^k q^{k^2} \cos(2kz).
\end{aligned} \tag{7}$$

3) Calculate $\alpha = \exp(8\pi\phi_2(x, y))$ and generate the $(m + 1)$-long vector $X = (1, \alpha, \alpha^2, \cdots, \alpha^m)$ to P_2-digit precision (the 8 is inserted here to accelerate solution).
4) Apply the four-level multipair PSLQ algorithm to find an integer relation for X, if one exists, to the precision being used (P_2 digits). For large problems, employ a parallel version of the four-level multipair PSLQ code, using the OpenMP `DO PARALLEL` construct to perform certain time-intensive loops in parallel.
5) If a numerically significant relation is not found, try again with a larger degree m or a higher full precision level P_2. If a relation is found, employ the polynomial factorization facilities in *Mathematica* or *Maple* to ensure that the resulting polynomial is irreducible.

Note that formulas (6) and (7) above involve sines and cosines of complex arguments (since z is complex). The MPFUN2020 software includes full support for transcendental functions and the complex datatype, so these formulas were implemented simply as stated in (6) and (7).

The high-level program that includes the computation of $\phi_2(x, y)$ and the four-level multipair PSLQ scheme, as specified in the steps above, is approximately 3,500 lines of Fortran.

Arbitrary precision package	Multipair PSLQ	Run time (CPU seconds) $x = y = 1/29$	$x = y = 1/35$
Old all-Fortran	3-level	247819.68	220202.68
New all-Fortran	3-level	55691.66	49221.37
	4-level	52920.67	47093.43
New MPFR-based	3-level	45038.87	41442.57
	4-level	45411.23	39762.23

TABLE III
Comparative performance of code versions

Recall from Table II, covering the case $x = y = 1/s$, that when s is even, the corresponding minimal polynomial is palindromic, or in other words that coefficient $a_k = a_{m-k}$, where m is the degree of the polynomial. In [6] this observation was proved to hold for all positive integers s. In the more general case $x = p/s$ and $y = q/s$ discussed in this paper, we have observed, but have not yet proved, that the resulting minimal polynomial is always palindromic when s is even.

After completion of the previous study [6], it was pointed out to the present author by Nitya Mani, a student at Stanford University [16], that if α satisfies a palindromic polynomial of degree m, then $\alpha + 1/\alpha$ satisfies a polynomial of degree $m/2$, and the degree-m polynomial satisfied by α can then be easily reconstructed from the degree-$m/2$ polynomial satisfied by $\alpha + 1/\alpha$. This fact was exploited in the present study to greatly reduce the run time for the cases when s is even.

VII. Performance

Table III shows some performance results comparing the old and new arbitrary precision packages and the three-level and four-level PSLQ codes. The two application cases displayed are: (a) $x = y = 1/29$, which requires 19,700-digit arithmetic (full precision), producing a polynomial of degree 196 with coefficients ranging from 1 up to roughly 10^{87}; and (b) $x = y = 1/35$, which requires 18,900-digit arithmetic (full precision), producing a polynomial of degree 192 with coefficients ranging from 1 up to roughly 10^{85}. These runs were performed on an Apple MacPro with a 3 GHz 8-core Intel Xeon E5 processor and 32 GByte main memory. The codes were compiled using gfortran/gcc version 10.2.0, with MPFR 4.0.2 and GMP 6.2.0.

Over 99% of the runtime of these calculations was in the multpair PSLQ code. Within the multipair PSLQ code, the breakdown of runtime by precision level is as follows (based on the all-Fortran, 4-level run of $x = y = 1/35$): double precision 5.9%, quad precision 0.1%, medium precision (950 digits) 50.0%, and full precision (18,900 digits) 44.0%.

The results in Table III show that the code using the new all-Fortran version of MPFUN2020 is only 15% slower than the MPFR-based version. This ratio is a dramatic improvement over the earlier all-Fortran package, which is 5.5X slower than the code using the MPFR-based version. The 4-level PSLQ package resulted in an improvement of typically 5% over the 3-level PSLQ package used in the earlier study, although, oddly, in one case (lower left corner of the table) it made no significant difference.

Each version of these codes is moderately parallelizable using OpenMP. For some parallel performance results of 3-level codes, see [6]. Because of the iterative nature of the multipair PSLQ algorithm, there does not appear to be any hope at the present time of employing massively parallel processing on a single case run. On the other hand, in studies of Poisson polynomials such as this, where many different cases need to be catalogued, each case can be performed independently, so there is substantial natural parallelism that can be exploited (and was exploited in this study).

VIII. Results and analysis

For this study, numerous cases were run, as summarized in Table IV, using the software described in the previous sections. These cases are: $x = 1/s$, $y = q/s$, for s from 10 to 40 (except 37, which was too expensive to complete for this study), and, for a given s, for q in the range $1 \le q < s/2$. Only the degrees of the resulting polynomials are shown in Table IV; the full polynomials and other details of runs, together with the source code for the runs, are available from the author. These runs were performed on an Apple MacPro with a 3 GHz 8-core Intel Xeon E5 processor and 32 GByte memory. The codes were compiled using gfortran/gcc version 10.2.0.

It can be shown from formula (1) that $\phi_2(m+x, n+y) = \phi_2(x, y)$, for any integers m, n, so there is no need to run cases $x = p/s$, $y = q/s$, where $p, q \le 0$ or $p, q > s$. In fact, by similar reasoning, one only need examine cases where $1 \le p, q < s/2$. These are the cases shown in Table IV.

Run times vary dramatically in these cases, from less than 0.01 seconds for the case $x = y = 1/10$, to 1,048,000 CPU-seconds (291 CPU-hours) for the case $x = y = 1/39$. Each of the $s = 39$ cases employed 42,000-digit arithmetic (full precision), producing polynomials of degree 288, with coefficients ranging from 1 up to approximately 10^{129} (except for $q = 14$, which produced a polynomial of degree 144).

Each case in Table IV exhibited a drop of at least 300 orders of magnitude at detection (in most cases over 1000 orders of magnitude), so that the polynomials produced by the runs are very unlikely to be numerical artifacts (see Section V). *Mathematica* 12.1.0 and *Maple* 2020 were employed to confirm that the polynomials are irreducible.

Table V, shown on the last page in a small font, presents one representative minimal polynomial, namely the degree-100 polynomial found by the program for the case $x = y = 1/25$. It is typical of Poisson polynomials, in that the initial coefficient is 1 or -1, then coefficients ascend to a maximum size (here roughly 10^{45}), and then descend back down.

This crescent-shaped pattern, starting with 1 or -1, is additional strong numerical evidence that the polynomial produced by the computer program is the true minimal polynomial associated with this case, and that all hardware, software and application code performed flawlessly, since otherwise it is *exceedingly unlikely* that the resulting coefficients would have this distinctive and highly improbable pattern. By contrast, in cases where the program failed to find a numerically significant relation, say due to a coding bug, insufficient degree m or insufficient precision level P_2, the resulting erroneous integer coefficients typically were all roughly the same size, within one or two orders of magnitude.

	q: first row 1–10; second row, 11–20									
	1	2	3	4	5	6	7	8	9	10
s	11	12	13	14	15	16	17	18	19	20
10	8	16	4	16						
11	30	30	30	30	30					
12	16	32	16	32	8					
13	36	36	36	36	18	36				
14	24	48	24	48	24	48				
15	32	32	32	16	32	32	32			
16	32	64	32	64	32	64	16			
17	64	64	64	32	64	64	64	64		
18	36	72	36	72	36	72	36	72		
19	90	90	90	90	90	90	90	90	90	
20	32	64	32	64	32	64	32	64	16	
21	96	96	96	96	96	96	96	48	96	96
22	60	120	60	120	60	120	60	120	60	120
23	132	132	132	132	132	132	132	132	132	132
	132									
24	64	128	64	128	32	128	32	128	64	128
	32									
25	100	100	100	100	100	100	50	100	100	100
	100	100								
26	72	144	72	144	36	144	72	144	72	144
	72	144								
27	162	162	162	162	162	162	162	162	162	162
	162	162	162							
28	96	192	96	192	96	192	96	192	96	192
	96	192	48							
29	196	196	196	196	196	196	196	196	196	196
	196	98	196	196						
30	64	128	64	128	64	128	64	128	64	128
	32	128	64	128						
31	240	240	240	240	240	240	240	240	240	240
	240	240	240	240	240					
32	128	256	128	256	128	256	128	256	128	256
	128	256	128	256	64					
33	240	240	240	240	240	240	240	240	240	120
	240	240	240	240	240	240				
34	128	256	128	256	128	256	128	256	128	256
	128	256	64	256	128	256				
35	192	192	192	192	192	96	192	192	192	192
	192	192	192	192	192	192	192			
36	144	288	144	288	144	288	144	288	144	288
	144	288	144	288	144	288	72			
38	180	360	180	360	180	360	180	360	180	360
	180	360	180	360	180	360	180	360		
39	288	288	288	288	288	288	288	288	288	288
	288	288	288	144	288	288	288	288	288	
40	128	256	128	256	128	256	128	256	64	256
	64	256	128	256	128	256	128	256	64	

TABLE IV
MINIMAL POLYNOMIAL DEGREES FOR THE CASES $x = 1/s, y = q/s$.

The results in Table IV are encouraging, but they raise more questions than answers. To begin with, note that these results, except the results in the first column, are *not* all consistent with Kimberley's formula, which has been proven only for the cases $p = q = 1$. For example, note that when s is even, the degrees for even q are, in most cases, double the degrees for $q = 1$. But there are exceptions, when the degrees are only half as large. For instance when $s = 36$, degrees alternate between 144 and 288, yet when $q = 17$, the degree is 72. There are also anomalies when s is odd. Note, for instance, that when $s = 35$, all degrees are 192, as given by Kimberley's formula,

except the degree is 96 when $q = 6$. Is there a generalization of Kimberley's formula for the larger class $1 \le p, q < s/2$?

IX. Conclusions and future research

While these results are a useful start, understanding the puzzling behavior of the Poisson polynomials will require substantial additional computation. In particular, more cases $x = p/s$, $y = q/s$, for integers $1 \le p, q < s/2$, need to be computed, instead of just the cases with $p = 1$, as analyzed in this paper, or just $p = q = 1$, as in the previous study [6]. Including a range of both p and q will greatly multiply the computational cost. Cases with $s > 40$, which are very expensive to compute, also need additional study.

In light of challenges such as this, research is needed in how to more rapidly perform PSLQ-type integer relation computations, and how to efficiently implement these computations on a highly parallel platform. While, say, a 12X parallel speedup on a 16-core system is certainly welcome, a scheme to efficiently employ hundreds or thousands of cores on a single case would be better, even given the fact that separate cases can be run independently in parallel. A fundamentally different integer relation algorithm may be required [6].

Note, by the way, that simply parallelizing a full-precision implementation of an algorithm such as multipair PSLQ, which possibly would achieve large speedups, is not helpful, since this would violate the principle, enunciated for example in [8], that any timing comparisons in parallel computing must be made to the most efficient practical serial algorithm. In this case, the most efficient practical serial algorithm is a three- or four-level multipair PSLQ algorithm, so this must be the starting point for any efficient parallel implementation (and parallel versions to date have yielded only modest speedups).

Finally, the challenges of implementations such as those described in this paper underscore the need for more research on improved software facilities to handle variable precision computing, including half, single, double and quad, as well as extreme precision. Some suggestions include:

1) Seamlessly incorporating an arbitrary precision facility, say based on the MPFR package, in future Fortran and C language standards.
2) Checking all available software supporting variable and arbitrary precision for thread safety, and fixing or deprecating libraries that are not strictly thread-safe.
3) Rethinking automatic type conversions and mixed-mode operations, which, although part of most languages and compilers, hinder the development of correct and reproducible variable precision computing, in the present author's opinion. At the least, could some of these features be optionally disabled or modified?
4) Investigating novel numerical algorithms, error control and error analysis techniques, appropriate not only for single and double precision on a single processor system, but also for the realm of very high precision and very highly parallel computing.
5) Developing fundamentally new paradigms for variable precision computing, perhaps along the lines of [12].

References

[1] J. Ablinger, J. Blumlein and C. Schneider, "Harmonic sums and polylogarithms generated by cyclotomic polynomials," *Journal of Mathematical Physics*, vol. 52 (2011), 102301.

[2] D. H. Bailey, "MPFUN2015: A thread-safe arbitrary precision package," manuscript, 16 Feb 2021, https://www.davidhbailey.com/dhbpapers/mpfun2015.pdf.

[3] D. H. Bailey, "MPFUN2020: A new thread-safe arbitrary precision package," manuscript, 10 Jul 2021, https://www.davidhbailey.com/dhbpapers/mpfun2020.pdf.

[4] D. H. Bailey, J. M. Borwein, R. E. Crandall and I. J. Zucker, "Lattice sums arising from the Poisson equation," *Journal of Physics A: Mathematical and Theoretical*, vol. 46 (2013), 115201.

[5] D. H. Bailey, J. M. Borwein, O. Caprotti, U. Martin, B. Salvy and M. Taufer, "Opportunities and challenges in 21st century mathematical computation: ICERM workshop report," 31 Jul 2014, https://www.davidhbailey.com/dhbpapers/ICERM-2014.pdf.

[6] D. H. Bailey, J. M. Borwein, J. Kimberley and W. Ladd, "Computer discovery and analysis of large Poisson polynomials," *Experimental Mathematics*, vol. 26 (27 Aug 2016), 349–363.

[7] D. H. Bailey and D. J. Broadhurst, "Parallel integer relation detection: Techniques and applications," *Mathematics of Computation*, vol. 70, 236 (2000), 1719–1736.

[8] D. H. Bailey, R. F. Lucas and S. W. Williams, ed., *Performance Tuning of Scientific Applications*, CRC Press, Boca Raton, FL, 2011.

[9] J. M. Borwein and P. B. Borwein, *Pi and the AGM*, John Wiley and Sons, New York, 1987.

[10] J. Chen, Y. Feng and W. Wu, "Two variants of HJLS-PSLQ with applications," *Proceedings of SNC'14*, 88–96, Shanghai, China, 2014.

[11] L. Fousse, G. Hanrot, V. Lefevre, P. Pelissier and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transactions on Mathematical Software*, vol. 33 (2 Jun 2007), 13–es; code available at https://www.mpfr.org.

[12] J. L. Gustafson, *The End of Error: Unum Computing*, Chapman and Hall / CRC Computational Science, 2nd corrected printing, 1st ed., 2016.

[13] S. Kanemitsu, Y. Tanigawa, H. Tsukada and M. Yoshimoto, "Crystal symmetry viewed as zeta symmetry," p. 91–130 in T. Aoki, S. Kanemitsu, M. Nakahara and Y. Ohno, eds., *Zeta Functions, Topology and Quantum Physics*, Springer, New York, 2005 (rep. 2010).

[14] S. Kanemitsu and H. Tsukada, "Crystal symmetry viewed as zeta symmetry II," 275–292 in K. Alladi, J. R. Klauder and C. R. Rao, eds., *The Legacy of Alladi Ramakrishnan in the Mathematical Sciences*, Springer, New York, 2010.

[15] D. Ma and M. Saunders, "Solving multiscale linear programs using the simplex method in quadruple precision," in M. Al-Baali, L. Grandinetti and A. Purnama, ed., *Recent Developments in Numerical Analysis and Optimization*, Springer, NY, 2017, https://web.stanford.edu/group/SOL/reports/quadLP3.pdf.

[16] N. Mani, "Palindrome degree reduction," manuscript, 19 Dec 2015, https://www.davidhbailey.com/dhbpapers/mani-palindrome.pdf.

[17] G. Savin and D. Quarfoot, "On attaching coordinates of Gaussian prime torsion points of $y^2 = x^3 + x$ to $Q(i)$," March 2010, https://www.math.utah.edu/~savin/EllipticCurvesPaper.pdf.

-1
$-33300\,\alpha^{1}$
$-8029750\,\alpha^{2}$
$+1542379500\,\alpha^{3}$
$-293348778825\,\alpha^{4}$
$-28080295402080\,\alpha^{5}$
$-6629781350330800\,\alpha^{6}$
$+58722280292148000\,\alpha^{7}$
$-7197243181862115100\,\alpha^{8}$
$+182328184490475806640\,\alpha^{9}$
$+1047464861734017883208\,\alpha^{10}$
$+1806185067754657491760\,\alpha^{11}$
$-27945881324717771782562700\,\alpha^{12}$
$+43132729144820226386860000\,\alpha^{13}$
$-256411580898396643830482000\,\alpha^{14}$
$+1396700356872030954211266400\,\alpha^{15}$
$+9088870493426588389401095750\,\alpha^{16}$
$-733599064553317639364961825000\,\alpha^{17}$
$+13704149574920386191809571588500\,\alpha^{18}$
$-157819237362090093545769457641320\,\alpha^{19}$
$+1321853578066306850140386747623398\,\alpha^{20}$
$-8347129396578224228862993897237280\,\alpha^{21}$
$+38212880291083166937296196441746800\,\alpha^{22}$
$-104022238521780343721711917072676000\,\alpha^{23}$
$-74032065945345957753107951351295900\,\alpha^{24}$
$+2695579175102470750839898367021130608\,\alpha^{25}$
$-17580580889577528003991569683501290200\,\alpha^{26}$
$+74952710236773360685253102982788566000\,\alpha^{27}$
$-239080432366992082679933177546734267500\,\alpha^{28}$
$+594175511524312220492120757330324484000\,\alpha^{29}$
$-1200481201725892359379030176076009068400\,\alpha^{30}$
$+2370791216795342465482695956290874196000\,\alpha^{31}$
$-6958365090143220739303965994873839612375\,\alpha^{32}$
$+29340961737653182271816694472006500014100\,\alpha^{33}$
$-118465202788489603844178657796002545834090\,\alpha^{34}$
$+394388553221080980775811796973709891661972\,\alpha^{35}$
$-1059773119301328191261276484738640092975935\,\alpha^{36}$
$+2274309666161593753722296548558864618267200\,\alpha^{37}$
$-3729748087865447793480560952748424761349600\,\alpha^{38}$
$+3877725636887659946158475367090512632719680\,\alpha^{39}$
$+846723867393661736729146827990176999234824\,\alpha^{40}$
$-15960428173973401824575411855716955373304480\,\alpha^{41}$
$+47118520726980412118527981001595834084077200\,\alpha^{42}$
$-96560391964393397280402148487438432608004000\,\alpha^{43}$
$+159442357501276027171661799158377030358819240\,\alpha^{44}$
$-222707090658310894211715578612740014870494016\,\alpha^{45}$
$+268336339651868593336977017479411555696276960\,\alpha^{46}$
$-280387518545606966350071856817231380200513600\,\alpha^{47}$
$+252610498127566038453865516310244367924136500\,\alpha^{48}$
$-192275255815619622137470846245628244410759600\,\alpha^{49}$
$+117222736648159046124836188054855685036349976\,\alpha^{50}$
$-47209171248166363599083489199392291307559600\,\alpha^{51}$
$-4942922820170238254884354265559936285783500\,\alpha^{52}$
$+36400498321158716446352704351511072445758400\,\alpha^{53}$
$-50714975271028177138621429774290438585502240\,\alpha^{54}$
$+52508039392118222466312798018961036691290304\,\alpha^{55}$
$-45426960236840080028112470330711413632425560\,\alpha^{56}$
$+33189561099972574095810629259221045183420000\,\alpha^{57}$
$-20184727758357355309386195853692979100127600\,\alpha^{58}$
$+10078674543222169818704320601337376637695840\,\alpha^{59}$
$-4216672803412456875435463721793522689401592\,\alpha^{60}$
$+1722566374653535529797841998864588575882560\,\alpha^{61}$
$-940122938034032272236892859372976755103200\,\alpha^{62}$
$+672500761678739947489218957049737685556800\,\alpha^{63}$
$-461584164001204318877039495371276336862015\,\alpha^{64}$
$+265582157437208924810452188771019969018068\,\alpha^{65}$
$-127220613278091048562581392274127224367210\,\alpha^{66}$
$+52420552698515377367407956646216340562900\,\alpha^{67}$
$-19432168360605247012293398573854487548375\,\alpha^{68}$
$+6715976041225158674019447203640258687520\,\alpha^{69}$
$-2160843064282037086551775761827735549808\,\alpha^{70}$
$+617282737944009515075102146542202738080\,\alpha^{71}$
$-145142540389934636970079472194894728300\,\alpha^{72}$
$+25729074826332880513711838275787958000\,\alpha^{73}$
$-3264527861366792560817221584368055000\,\alpha^{74}$
$+473260992168563618404463109451174000\,\alpha^{75}$
$-178858482702935977606858126791577500\,\alpha^{76}$
$+66028976498454634546379541782684000\,\alpha^{77}$
$-15761469487294858226633454090400400\,\alpha^{78}$
$+2480913858831352983037735837103840\,\alpha^{79}$
$-283939423573690705553213818042394\,\alpha^{80}$
$+27924016290268330781557752871960\,\alpha^{81}$
$-1890264935738308757140819195500\,\alpha^{82}$
$-197646050954205989496871249000\,\alpha^{83}$
$+92058025400867206020845473350\,\alpha^{84}$
$-15898833972285544673912261280\,\alpha^{85}$
$+1776153714893883182147476400\,\alpha^{86}$
$-134561774900933516621588000\,\alpha^{87}$
$+5713502521319481241590900\,\alpha^{88}$
$-28013036278808426259920\,\alpha^{89}$
$-10740617849680182084536\,\alpha^{90}$
$+416466536564927187120\,\alpha^{91}$
$-5335454032157718300\,\alpha^{92}$
$-61275002107276000\,\alpha^{93}$
$-662049081315760\,\alpha^{94}$
$+123800803998624\,\alpha^{95}$
$-330633806665\,\alpha^{96}$
$-8699478100\,\alpha^{97}$
$+13986250\,\alpha^{98}$
$+6700\,\alpha^{99}$
$-1\,\alpha^{100}$

TABLE V
DEGREE-100 MINIMAL POLYNOMIAL FOUND FOR THE CASE $x = y = 1/25$