Resolving Numerical Anomalies in Scientific Computation

David H. Bailey*

February 11, 2008

Abstract

The increasing numbers of large scientific computer systems in use, together with the rapidly increasing scale of computations being attempted on such systems, means that numerical anomalies pose a growing threat to the soundness of these computations. This note presents a number of examples of common numerical anomalies that can arise, illustrated by some actual computer programs available on a website, together with a discussion of how these problems could be mitigated with improved software tools.

*Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA, dhbailey@lbl.gov. Supported in part by the Director, Office of Computational and Technology Research, Division of Mathematical, Information and Computational Sciences, U.S. Department of Energy, under contract number DE-AC02-05CH11231.

1 Introduction

In recent years, the field of large-scale scientific computation has grown very rapidly, as indicated for instance by the Top500 list of the world's most powerful computer systems [10]. At the present date, the world's most powerful system is the IBM BlueGene/L at the Lawrence Livermore National Laboratory, which features 212,992 processing cores and a sustained performance rate on the scalable Linpack benchmark of 478 trillion floating point operations per second (478 Tflop/s). This is followed by a system in Germany at 167 Tflop/s, a system in the U.S. at 126 Tflop/s, and a system in India at 117 Tflop/s. The number 500 ranked system on the current list has 1,344 processors and runs at 5.9 Tflop/s. It is a remarkable fact that as recently as eight years ago, such a system would have been the top-ranked system. The current list is the most geographically and organizationally diverse ever.

One inescapable consequence of the greatly expanding number of scientific systems, and the scale of scientific computations being performed on them, is that numerical anomalies and difficulties that heretofore have been minor nuisances are now much more likely to have significant effects. At the same time, the proliferation of these systems means that the majority of persons running these applications are not experts in numerical analysis, and thus are more likely to be unaware of the potential numerical difficulties that may already exist in their computer programs.

Fortunately, remedies are available, at least in theory if not in widespread practice. For instance, it is often possible to run an application after altering the rounding mode of the floating-point arithmetic hardware, to see if the results at the end of the computation match those of a normal run to the desired level of precision. Another possibility is to employ software that converts a program to perform some or all portions of the computation using, say, double or even quadruple the standard 64-bit precision available most systems today. A third option is to employ interval arithmetic.

It is often argued that numerical difficulties are mostly due to programmer "errors," or that they can usually be rectified by employing more sophisticated numerical algorithms and/or programming techniques. This may true in some cases, although, as will be clear below, certainly not in all cases. But even where this is true, the fact remains that making fundamental algorithmic changes is usually a relatively difficult and expensive (in programmer time) process, whereas, for instance, substituting higher precision arithmetic for a certain trouble-prone section of code can often be done rather painlessly.

Each of the three measures mentioned above has its advantages and drawbacks. Changing the rounding mode, for instance, can disclose the existence of some numerical difficulties, but not others, and it can do nothing to remedy problems when they are found. Double-double, quad-double or even arbitrary precision arithmetic software can be used both to detect and to remedy many problems, but the software interface for these libraries is not foolproof (as will be seen below), and run time may increase significantly, depending on how much of a computer program is converted to use higher precision. Interval arithmetic can be used to produce a "certificate" that the final results are correct to within a certain accuracy (in a certain sense), but it cannot find or remedy certain types of anomalies, and its run-time cost is the greatest.

In summary, the expected increase in numerical difficulties and their potential for seriously impacting the reliability of large-scale scientific computing, calls for the devel-

opment of some intelligent software tools that can analyze code, both at the source level and possibly also during execution, and can both detect and remedy a fairly wide class of numerical anomalies. First of all, however, it is important to catalogue the sorts of anomalies and numerical sensitivities that commonly occur, so as to constitute a target for these envisioned utilities.

To that end, a set of numerical case studies is presented here, illustrated with computer programs available on a website, each of which illustrates a common type of numerical difficulty in a reasonably realistic context of scientific computation. None of these examples, taken by itself, is "new" in the sense of not being previously appreciated by researchers in the numerical analysis community. But collectively they are presented as a starting point — a relatively easy-to-understand set of codes illustrating the sorts of actual numerical difficulties that arise in real-world scientific computation, which can be the target for software tools to be developed in this arena.

For brevity, this collection focuses on Fortran-90 examples, not only because Fortran remains the most widely used language for heavy-duty scientific computation, but also because there are some anomalies (including the first-mentioned one in the next section) that arise in Fortran that do not arise in most other languages. Also for brevity, except where noted otherwise the `gfortran` compiler environment is assumed, running on an Intel-based Apple Macintosh system. It should be emphasized, though, that in almost all cases the phenomena mentioned below occur with many other Fortran systems and hardware platforms.

All of these programs can be obtained in a gzipped file, available at the following URL: `http://crd.lbl.gov/~dhbailey/dhbpapers/numerical-bugs.tar.gz`.

2 Language-Based Difficulties

It often comes as a shock to programmers to find that with some Fortran systems, numeric constants appearing in Fortran programs are converted to full double precision accuracy only if they are written with the "d" notation, as in `1.25d0`; otherwise they are converted to only 32-bit (roughly 6-7 digit) precision. For example, if the code segment

```
double precision pi1, pi2
. . .
pi1 = 3.14159265358979323846
pi2 = 3.14159265358979323846d0
```

is compiled and executed using the `gfortran` compiler system on a Macintosh Intel-based system, it produces the values 3.1415927410125732 and 3.1415926535897931, respectively.

Note that even though the variables `pi1` and `pi2` are each declared double precision, and in each case the constant is specified to ample precision (20 digits), only in the second case is the constant converted to full 64-bit precision. The present author has seen this anomaly numerous times, including, embarrassingly enough, in at least one of his own codes. It is worth emphasizing here that even experienced Fortran programmers, if they have predominately worked in an environment where double-precision arithmetic and conversions are the default, may not realize that this is not the universal standard,

and may experience significant numerical inaccuracies when they port their programs to other systems.

A related problem, which is potentially more insidious (and not unique to the Fortran language), is the problem of expressions that involve both integer or 32-bit real data and 64-bit (or 80-bit) real data, in mixed-mode operations. As an example, consider this segment of code:

```
integer n
double precision t1, t2
parameter (n = 3)
. . .
t1 = 1. / n
t2 = 1.d0 / n
```

When this code is compiled and executed with `gfortran`, only the second executable line produces the correct value, 0.3333333.... This is because `gfortran`, like many other Fortran systems, computes the right-hand side of the first line only using single precision arithmetic, since it involves a constant of type real and a variable of type integer. The result of this division is then converted to double (by zero extension) before storing in the left-hand-side variable.

One could argue that this anomaly is due to a programmer error — all floating-point constants in code that is intended to be performed using double-precision (64-bit or 80-bit) arithmetic should include the "d" to unambiguously specify double precision operations and conversions. But keep in mind that on many legacy systems, notably Cray vector systems, single precision real means 64-bit floating-point (although the format was somewhat different than IEEE 64-bit format), and many codes originally written on such systems are still in use today on IEEE-based systems. Such codes are often run today with a compiler flag, such as gfortran's "-fdefault-real-8", which specifies that all computations involving single precision real data are to be performed with 64-bit (or 80-bit) arithmetic, and all single precision real constants are to be converted to 64-bit (or 80-bit) precision. But extracts of such code are often incorporated into other programs where the programmers do not use such a compiler option, and thus the potential for inaccuracy remains.

This language convention, namely that mixed-mode operations are performed strictly according to the needs of the two operand types, is also present in other languages besides Fortran. What's more, this convention presents difficulties for employing high-precision software, as shall be seen below.

3 Avoidable Numerical Anomalies

A second category of numerical anomalies that often occur in scientific computer programs are segments of code that could, in principle, be performed highly accurately using only standard 64-bit (or 80-bit) IEEE floating-point arithmetic, but because of the way they are written an excessive amount of numerical error usually results.

One of the most common examples in this category is a conditional branch with a floating-point comparison that may fail to hold, depending on numerical subtleties. In

almost all cases that the author has seen, the code once did operate correctly as intended, but subsequently either the parameters were changed, or the code was ported to some other system that has a slightly different arithmetic system (e.g., 80-bit instead of 64-bit), or even a newer version of the compiler was used, which generates hardware operations in a slightly different order.

As an example, consider the problem of using Simpson's rule to evaluate the definite integral of a specified function on a given interval (a, b). In other words, given n, define $h = (b - a)/(2n)$. Then

$$\begin{aligned}\int_a^b f(x)\,dx \quad &\approx \quad \frac{h}{3}\left[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h)\right. \\ &\qquad \left. + \cdots + 4f(a+(2n-1)h) + f(b)\right].\end{aligned}$$

If one follows modern-day prudent programming practices, this should be coded with a DO-loop (or the equivalent in other languages) of length n that unambiguously terminates the summation at the proper pre-determined stopping point. A computer program demonstrating this scheme to evaluate $\int_0^1 \sin(\pi x/2)\,dx = 2/\pi$, using $n = 1000000$, is available at the URL above as simpsons.f90.

But older code (and even some newer codes) may implement this or similar calculations using a conditional branch, terminating the loop when $a + kh \geq b$ for some k. In some contexts, such a branch may be unavoidable. If such an operation is done in this way using an open-ended incremental loop, it is actually rather likely that the comparison test will fail because $a + kh$ is slightly less than b due to numerical round-off error. Indeed, this exact situation occurs in the sample code simpsonsbranch.f90. Here the summation loop is performed for one additional iteration, adding a significant error to the result.

It is also worth adding that, in this instance, employing higher-precision arithmetic is *not* effective in remedying or even detecting the error condition, nor is interval arithmetic, as commonly implemented, of any value in this situation.

One reasonable remedy for such errors (or potential errors) is to add some "fuzz" (a small numerical value, typically something like 10^{-10}) to the left-hand side of the $\leq$ comparison test. The actual size of the "fuzz" must be selected carefully and must be re-checked if any parameters, such as the number of subintervals, are changed. A modification like this could be performed semi-automatically with a suitable software tool. A modification of the Simpson's program along this line is shown in the sample program simpsonsfuzz.f90.

The Simpson's rule computation just mentioned is also subject to a second type of avoidable numerical anomaly, namely excessive round-off error when large numbers of small values are added. For instance, even with the "fuzz" adjustment above, the resulting double-precision Simpson's rule calculation, with $n = 1000000$, is only accurate to 10 digits, whereas it theoretically should be accurate to nearly 16 digits. The problem is that repeated additions of the increment h results in significant cumulative error in the argument x that is used as input to the function evaluation. In addition, repeated additions of the function value, multiplied by 2 or 4 as appropriate, yield cumulative numerical error in the result.

Some algorithmic approaches are known that can help in such a situation. For instance, it often helps to store all of the results to be summed in an array, then sort the array and

add from smallest to largest. However, in the author's view, the most straightforward way to remedy this type of problem is to perform the incrementing of x and the addition of function-weight products using higher-precision arithmetic. In the case of the Simpson's rule calculation under consideration (and in almost all other cases the author is aware of), twice the normal precision, namely 128-bit or "double-double" precision, suffices.

At the present time, no major vendor of widely used computer processors offers hardware support for 128-bit floating-point arithmetic. Some Fortran systems offer language support for 128-bit floating-point arithmetic (through software), but most do not. The gfortran compiler, for instance, does not. Hardly any C or C++ systems support such a datatype. However, a package is available from the author, Yozo Hida and Xiaoye Li which supports "double-double" (approximately 31 digits) and "quad-double" arithmetic (approximately 62 digits) [12]. This software allows one to convert a Fortran-90, Fortran-95 or C++ program to use double-double or quad-double arithmetic. In most cases it is only necessary to only change type statements of the variables to be treated as double-double or quad-double, and to delimit constants with apostrophes (so that the interface software will convert them to full precision). Most other operations, including arithmetic operations and even many transcendental function references, are then performed automatically. This software is available from the the URL `http://crd.lbl.gov/~dhbailey/mpdist`.

In the situation mentioned above (namely minimizing numerical error in performing a large sum), as in many other applications the author has seen, happily it is not necessary to convert the entire application to double-double arithmetic. Instead, only operations involving the argument x, the increment h and the sum need to be performed using double-double arithmetic. In particular, the relatively expensive function evaluation operations need be performed only in standard 64-bit precision. Thus the overall run time does not increase by a large factor, as it typically does if the entire computation must be performed using higher precision arithmetic. A modification of the Simpson's program along this line, named simpsonsd.f90, is available from the URL above. It is remarkable that this relatively minor change produces a result that accurately replicates the theoretical answer, namely $2/\pi$, to 15-digit accuracy.

One additional example of an avoidable numerical anomaly is given in the sample program funarc.f90. This code attempts to calculate the arc length of a somewhat irregular function, namely $g(x) = x + \sum_{0 \le n \le 5} 2^{-k} \sin(2^k x)$, over the interval $(0, \pi)$. The plot of $g(x)$ is shown in Figure 1.

The task here is to sum $\sqrt{h^2 + (f(x_k + h) - f(h))^2}$ for $x_k \in (0, \pi)$ divided into n subintervals, where $n = 1000000$, so that $h = \pi/1000000$ and $x_k = kh$). When this is coded in a straightforward fashion using ordinary double precision arithmetic, on an Intel-based Macintosh system (see program funarc.f90), one obtains 5.795776322413031, which differs somewhat from the "correct" value, namely 5.795776322412856, which is produced by the double-double version of the program (funarcdd.f90). The double-double version runs 20 times more slowly than the double-precision version. However, if we modify the double-precision program, declaring the summation variable to be of type double-double, so that the summation is performed using double-double arithmetic, one obtains 5.795776322412856 (program funarcdpdd.f90), which is identical to the correct value. The cost of employing double-double arithmetic in this partial manner is negligible, since almost all the CPU time in this program is spent in the function evaluation, which is still done entirely with ordinary double-precision arithmetic.

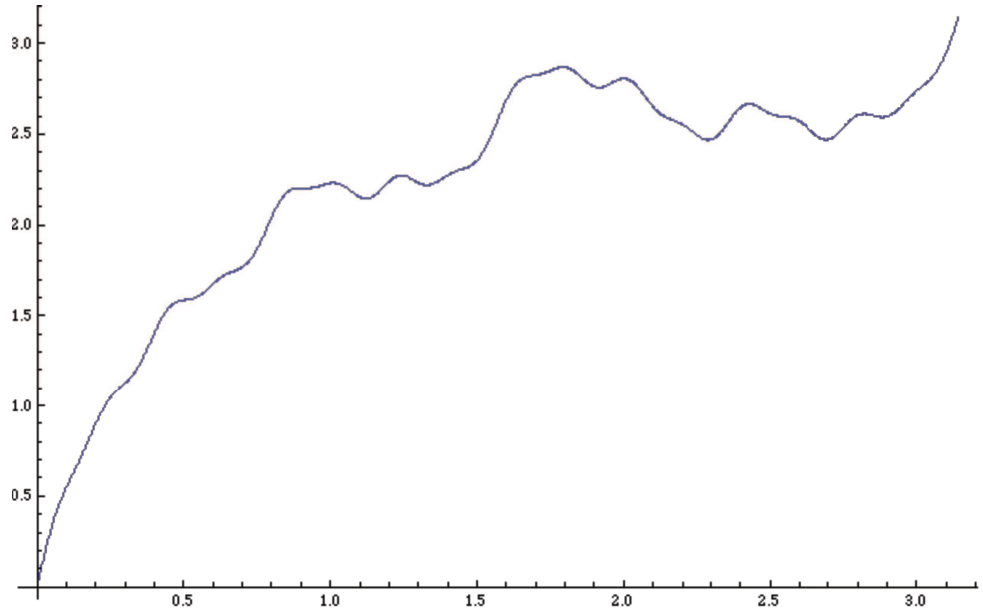

Figure 1: Plot of function $g(x)$.

These last two examples illustrate the author's experience in using double-double and/or quad-double arithmetic to remedy numerical difficulties in various scientific computation programs — if used judiciously, namely only in the most numerically sensitive areas of the program, these facilities can produce numerically satisfactory results with only a minor increase in total run time. However, proper usage requires some measure of numerical expertise to identify exactly which sections of the program require higher-precision arithmetic. Clearly a software tool that could assist the programmer to make this determination would be a great benefit in such efforts.

4 Unavoidable Numerical Anomalies

The final category of numerical anomalies we will consider are those that cannot be remedied by means of a minor modification to the program, short of performing the majority of operations in high-precision arithmetic. One might think that only highly exotic types of calculations really require double-double or higher precision for all or most arithmetic. In fact, numerous such applications exist. A collection of some that the present author is familiar with is given in [1].

It is remarkable how innocuous-looking calculations in this category can be. For starters, suppose we wish to solve this 2×2 system of linear equations:

$$\begin{aligned} 0.25510582x + 0.52746197y &= 0.79981812 \\ 0.80143857x + 1.65707065y &= 2.51270273 \end{aligned}$$

In fact, this system has the solution $(-1, 2)$. But a double-precision implementation of this scheme (program singmat.f90) utterly fails — the result is far from correct. Double-double precision arithmetic must be used (program singmatdd.f90) in order to produce a

reliable result, because the matrix corresponding to the left-hand side is nearly singular — its determinant is roughly 10^{-16}.

A second example is equally innocuous-looking. Suppose we wish to fit the following data to a polynomial: $5, 2304, 118101, 1838336, 14855109, 79514880, 321537749, 1062287616, 3014530821$, for integer arguments $(0, 1, 2, \cdots, 8)$. The usual way to do this is to employ polynomial least squares curve fitting [14], which amounts to solving a $(n+1 \times n+1)$ linear system of equations (written in matrix form):

$$\begin{bmatrix} n & \sum_{k=1}^{n} x_k & \cdots & \sum_{k=1}^{n} x_k^n \\ \sum_{k=1}^{n} x_k & \sum_{k=1}^{n} x_k^2 & \cdots & \sum_{k=1}^{n} x_k^{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} x_k^n & \sum_{k=1}^{n} x_k^{n+1} & \cdots & \sum_{k=1}^{n} x_k^{2n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{n} y_k \\ \sum_{k=1}^{n} x_k y_k \\ \vdots \\ \sum_{k=1}^{n} x_k^n y_k \end{bmatrix}.$$

In a computer implementation of this algorithm, the linear equation solution is most commonly (and most wisely) done with library software, such as the Linpack [9] or LAPACK [8]. A double-precision implementation of this algorithm (available as polyreg.f90) incorporates the Fortran Linpack program, with a few modifications. This program fails to find the correct underlying polynomial coefficients that produce this sequence. The correct polynomial, which is produced by the double-double program polyregdd.f90, deduces that the original data sequence is given by the polynomial function

$$f(k) = 5 + 220k^2 + 990k^4 + 924k^6 + 165k^8.$$

Here is one more example of a very innocuous-looking problem: Suppose that in one's calculation, a computed floating-point value is known (or suspected) to be a rational number, and that one wishes to present this result as output in the rational form, i.e., as the quotient of two relatively prime values. This can be done by the well-known Euclidean algorithm, where one accumulates the multipliers in a 2×2 integer matrix. This algorithm can be stated as follows:

1. Initialize: Given an input real $x > 0$, set the vector $a = (1, x)$, and set the 2×2 matrix b to be the identity. Set ϵ to be some value appropriate for the precision being used, tyipcally 10^{-9} for double-precision computations, and 10^{-18} for double-double computations.
2. Iterate: On odd-numbered iterations, compute $t := \lfloor a_2/a_1 \rfloor$. Then set $a_2 := a_2 - ta_1,\ b_{1,1} := b_{1,1} + tb_{1,2},\ b_{2,1} := b_{2,1} + tb_{2,2}$. On even-numbered iterations, compute $t = \lfloor a_1/a_2 \rfloor$. Then set $a_1 := a_1 - ta_2,\ b_{1,2} := b_{1,2} + tb_{1,1},\ b_{2,2} := b_{2,2} + tb_{2,1}$.
3. Terminate when $a_1 < \epsilon$ or $a_2 < \epsilon$. If termination was on an odd-numbered iteration, then output, as the numerator and denominator of the result fraction, $b_{2,1}$ and $b_{1,1}$. If termination was on an even-numbered iteration, then output $b_{2,2}$ and $b_{1,2}$.

It is often not well-appreciated that this algorithm is very demanding of numerical precision — if one is using ordinary double precision, then it can only reliably recover fractions where the numerator and denominator are both less than about six or seven digits long. For instance, a double precision version of this scheme (confrac.f90) successfully recovers, from their double-precision numerical values, the fractions 4/7, 21/53, 12323/54311,

123419/654323, 1234577/7654337 and even 12345623/87654319. But it fails for the fraction 123456719/987654319. Larger fractions can be recovered by using double-double arithmetic, as is illustrated in the program confracdd.f90, but again there are limits to the size of fractions that can be recovered reliably.

The generalization of the Euclidean algorithm to n dimensions is known as the integer relation problem: Given the n real numbers $x_1, x_2, \cdots, x_n$, find integers a_i, not all zero, if they exist, such that

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

At the present time, the best-known integer relation algorithm is the "PSLQ" algorithm of mathematician-sculptor Helaman Ferguson [11, 2, 4]. For PSLQ (or any other integer relation finding scheme) to be able to recover an underlying relation of length n with integers of size d digits, then each of the input x_i must be specified to at least nd-digit precision, and nd-digit precision must be used in the computation (otherwise the true relation will be lost in a sea of meaningless numerical artifacts).

In the past few years, PSLQ has been used to discover numerous new identities and relations in mathematics and mathematical physics, and more results are being produced every day [6, 7, 3]. Most of these calculations are done with several hundred digit-precision, but a few require much more. One result required 50,000-digit precision. An implementation of the "multi-pair" variant of PSLQ [4] is available as the sample program tpslqmqd.f90, which utilizes quad-double arithmetic. Based only on the numerical value of the input (to 60-digit accuracy), this program deduces that $\alpha = \sqrt[3]{3} - \sqrt[4]{2}$ satisfies the polynomial.

$$0 \quad = \quad -73 + 144\alpha + 540\alpha^2 + 108\alpha^3 - 12\alpha^4 + 288\alpha^5 - 54\alpha^6 + 6\alpha^8 + 12\alpha^9 - \alpha^{12}.$$

This can be done by first computing the vector of values $(1, \alpha, \alpha^2, \cdots, \alpha^{12})$, then applying the PSLQ algorithm.

Larger examples of this type require an arbitrary-precision computation software package, such as the ARPREC package due to Yozo Hida, Brandon Thompson Xiaoye Li and the present author [5]. Just as with the double-double and quad-double packages, the ARPREC package supports high-level programs in either Fortran-90, Fortran-95 or C++. In most cases, only minor modifications (mostly changing type statements) are required to convert an ordinary double-precision program to utilize the high-precision libraries.

PSLQ applications represent a new and rapidly growing application for very high precision arithmetic, particularly in math and mathematical physics research. However, great care must be taken to ensure that the results are numerically meaningful. Among other things, one must carefully monitor the minimize size of the reduced x vector as the computation progresses, and ensure that a large drop occurs on the iteration where the program detects a relation. Other details of practical implementations are given in [4].

5 Conclusions

This note has highlighted a number of common numerical difficulties, and has emphasized the fact that such difficulties are likely to become more widespread as scientists attempt

much larger computations on highly parallel computing systems. It has also been seen that while software remedies such as higher-precision arithmetic are in many cases possible, nonetheless using these software packages at present requires considerable experience and expertise. Clearly if these techniques are to enjoy wider usage, improved software tools are required.

Below are some desired features, ranging from fairly basic facilities to more advanced tools that would require significant effort to implement. Many of these features could be implemented by a tool that modifies user source code, but ultimately such a tool would have more universal applicability if it can work directly with binary code. Note, for instance, that when user codes reference library routines, source code might not even be available.

It should also be noted that a "mindless" implementation of remedies such as higher-precision arithmetic is not likely to be effective, for a variety of reasons [13]. Thus the ultimate effectiveness of such a tool depends much on a careful design that incorporates the lessons of several decades of academic research in this area.

1. Perform the entire computation with the IEEE rounding mode changed, to diagnose whether the user program is beset with numerical sensitivities.
2. Perform the computation with the rounding mode changed only in certain subroutines (and ultimately only in certain sections of code), in an attempt to further identify where numerical difficulties lie.
3. After running the user's code, say with the rounding mode changed, display the selected output data in a color-coded scheme, indicating which digits are "good" and which are unreliable.
4. Scan a user's program to see if there are instances of comparison of floating-point entities, and automatically add an adjustable "fuzz" to such comparisons, in order to see if this makes a differences in branching; if so, advise user.
5. Scan a user's program to see if there are floating-point constants that, due to language "features," are not being converted to full precision. By the way, this is also valuable if one is using double-double, quad-double or arbitrary precision software, since such facilities, by relying on standard language syntax, often do not automatically convert constants to full precision. An efficient scheme along this line should avoid conversions for constants, such as 3.625 and 0.0625, which are exact binary rational values.
6. Scan a user's program to see if there are instances of subexpressions that are being performed with less than the full precision desired for the computation. As in the previous item, this is also valuable if one is using double-double, quad-double or arbitrary precision software, since these software systems often do not detect or handle such expressions in the desired fashion.
7. When numerical sensitivities are identified, provide a semi-automatic means to attempt remedies such as performing some subroutines (or even certain sections of a single subroutine) using higher-precision arithmetic. An efficient scheme here should allow the user to specify that certain variables may remain as ordinary double-precision, because they only contain exact binary rational values. It is also important

8. Perform some or all of the computation using software-based interval arithmetic.
9. More advanced software tools could insert real-time checks into code, which advise the user when a floating-point comparison involves a suspiciously small difference, or when a significant amount of numerical precision is being lost. But for the present, the proper focus should be on a more modest agenda that can be implemented and tested in a real-world scientific computing environment.

Along this line, it is unfortunate that computer processor vendors do not yet see a need for hardware support for the IEEE 128-bit floating-point standard. Even if 128-bit operations ran, say, four or five times more slowly than 64-bit operations, this would still greatly increase the number of users who would be willing to experiment with this feature in their codes. At the least, it would enable users to occasionally run their codes using 128-bit arithmetic, to ensure that, with recent upgrades and extensions of their applications, their numerical results are still sound. In the meantime, we will need to rely on software for this capability.

References

[1] David H. Bailey, "High-Precision Arithmetic in Scientific Computation," *Computing in Science and Engineering*, May-Jun, 2005, pg. 54-61, http://crd.lbl.gov/ dhbailey/dhbpapers/high-prec-arith.pdf.

[2] David H. Bailey, "Integer relation detection," *Computing in Science and Engineering*, Jan-Feb, 2000, 24–28, http://crd.lbl.gov/~dhbailey/dhbpapers/pslq-cse.pdf.

[3] D. Bailey, J. Borwein, N. Calkin, R. Girgensohn, R. Luke, and V. Moll, *Experimental Mathematics in Action*, AK Peters, 2007.

[4] David H. Bailey and David J. Broadhurst, "Parallel integer relation detection: techniques and applications," *Mathematics of Computation*, **70** 2001, 1719–1736, http://crd.lbl.gov/~dhbailey/dhbpapers/ppslq.pdf.

[5] David H. Bailey, Yozo Hida, Xiaoye S. Li and Brandon Thompson, "ARPREC: An Arbitrary Precision Computation Package," 2002, http://crd.lbl.gov/~dhbailey/dhbpapers/arprec.pdf. The corresponding software is available at http://crd.lbl.gov/~dhbailey/mpdist.

[6] Jonathan M. Borwein and David H. Bailey, *Mathematics by Experiment: Plausible Reasoning in the 21st Century*, AK Peters, Natick, MA, 2004. Second expanded edition, 2008.

[7] Jonathan M. Borwein, David H. Bailey and Roland Girgensohn, *Experimentation in Mathematics: Computational Paths to Discovery*, AK Peters, Natick, MA, 2004.

[8] Jack Dongarra, "LAPACK – Linear Algebra Package," http://www.netlib.org/lapack.

[9] Jack Dongarra, "LINPACK," http://www.netlib.org/linpack.

[10] Jack Dongarra, Hans Meuer, Erich Strohmaier and Horst Simon, "The Top 500 List," http://www.top500.org.

[11] Helaman R. P. Ferguson, David H. Bailey and Stephen Arno, "Analysis of PSLQ, An Integer Relation Finding Algorithm," *Mathematics of Computation*, vol. 68, no. 225 (Jan 1999), pg. 351-369, http://crd.lbl.gov/~dhbailey/dhbpapers/cpslq.pdf.

[12] Yozo Hida, Xiaoye S. Li and David H. Bailey, "Algorithms for Quad-Double Precision Floating Point Arithmetic," *15th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society, 2001, pg. 155-162, http://crd.lbl.gov/~dhbailey/dhbpapers/arith15.pdf.

[13] William Kahan, "How Futile Are Mindless Assessments of Roundoff in Floating-Point Computation?", available at www.cs.berkeley.edu/ wkahan/Mind1ess.pdf.

[14] Eric Weisstein, "Least Squares Fitting–Polynomial," http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html.