

# NUMERIC ISSUES IN TEST SOFTWARE CORRECTNESS

Robert G. Hayes, Raytheon Infrared Operations, 805.562.2336, rghayes@computer.org  
Gary B. Hughes, Ph.D., Raytheon Infrared Operations 805.562.2348, gbhughes@raytheon.com  
Phillip M. Dorin, Ph.D., Loyola Marymount University, 310.338.0000, pdorin@lmu.edu  
Raymond J. Toal, Ph.D., Loyola Marymount University, 310.338.2773, rtoal@lmu.edu

## ABSTRACT

Test system designers are comfortable with the concepts of precision and accuracy with regard to measurements achieved with modern instrumentation. In a well-designed test system, great care is taken to ensure accurate measurements, with rigorous attention to instrument specifications and calibration. However, measurement values are subjected to representation and manipulation as limited precision floating-point numbers by test software. This paper investigates some of the issues related to floating point representation of measurement values, as well as the consequences of algorithm selection. To illustrate, we consider the test case of standard deviation calculations as used in the testing of Infrared Focal Plane Arrays. We consider the concept of using statistically-based techniques for selection of an appropriate algorithm based on measurement values, and offer guidelines for the proper expression and manipulation of measurement values within popular test software programming frameworks.

Keywords: measurement accuracy, precision, floating-point representation, standard deviation, statistically based algorithm selection, Monte Carlo techniques.

## 1 INTRODUCTION

Some well-known software bugs have resulted in security compromises, sensitive data corruption, bills for items never purchased, spacecraft crashing on other planets, and the loss of human life. Ensuring correctness of software is a necessity. We work toward software correctness via two approaches: testing and formal verification. In both cases we strive to demonstrate that an algorithm or device behaves according to its *specification* — usually a statement relating inputs to expected outputs.

In automated test systems, great care is taken to achieve high quality measurements with expensive, highly accurate and precise instrumentation, in order to reduce systematic errors and blunders. However the measured inputs are still algorithmically manipulated as floating-point values and in many cases precision and accuracy get thrown away. This loss will be missed

during test if the test limits are expressed with the same limited precision floating-point hardware on which we run the test program. Ideally, test system results should be compared with “true” floating point values, but generating such values requires arbitrary-precision arithmetic that is usually too resource-intensive to be practical. In its place engineers may be able to carefully manage precision by tracking error bounds in computation. Where the nature of the computation shows that the error bounds cannot be made sufficiently tight, probabilistic methods of computation, such as Monte Carlo Arithmetic, can be employed.

The rest of this paper is organized as follows. In Section 2 we provide background information on accuracy and precision, common problems encountered with floating-point code in test software, and the IEEE 754 floating-point standard. Section 3 considers the problem of how different implementations of the same function can produce differing behavior with given floating-point inputs, and looks at ways to select algorithms to minimize precision loss. This discussion focuses on standard deviation algorithms for a case study. Section 4 looks at the use of arbitrary-precision arithmetic as an alternative approach to floating-point code, and Section 5 examines another alternative, Monte Carlo arithmetic. Section 6 concludes and summarizes the work.

## 2 BACKGROUND

### 2.1 Accuracy and Precision

Testing and verification are subject to the constraints of accuracy and precision. *Accuracy* refers to the relationship between a measured quantity and its “true” value. *Precision* refers to the number of significant digits that can be represented by a machine. For example a device can have a precision of six decimal digits but allow for readings accurate to only four. Such a device may report a voltage measurement of 2.16734 volts (six digits of precision) but we only “believe” the first four digits, 2.167 (our accuracy). For all practical purposes, no physical quantity such as a voltage can be measured “exactly”: there always exists an uncertainty associated with the measurement [Vin90]. Note that integer-valued quantities can be represented exactly, but these are *counted*, not *measured*. It’s unrealistic to think that measurement uncertainty can be eliminated. Estimation of these uncertainties often involves the technique of making multiple measurements to reduce the uncertainty due to random errors, and quantifying the uncertainty using the standard deviation. For example, at Raytheon Infrared Operations, standard deviation is used to characterize the performance of Infrared Focal Plane Arrays that may contain millions of individual detectors.

Yet even highly accurate and precise measurements are subject to precision loss when manipulated by test software. Subtraction of nearly equal values is generally the most problematic as it results in *cancellation* of significant digits. Complex iterative calculations, especially those in feedback systems often introduce *rounding errors* that over time can annihilate all significant digits [PJS92].

James Gosling claimed “95% of the folks out there are completely clueless about floating-point” and Kahan agrees [KD98]. Worse, Kahan has observed “much of what is taught in school about floating-point error analysis is wrong” and goes on to point out thirteen (!) prevalent

misconceptions about floating-point arithmetic. In addition, many programming languages fail to give programmers any access to hardware traps and flags necessary to efficiently manage non-finite intermediate results of a computation that can disappear in subsequent stages of a computation. Because *testing* is inherently concerned with the relationship between “true” values and floating-point representations, test system engineers and those creating test software, require a working knowledge of the kinds of nonintuitive issues surrounding floating-point code, as well as the technical details of floating-point representations and arithmetic. They must also be familiar with techniques for managing rounding errors and cancellation, and with alternatives to floating-point code including arbitrary-precision arithmetic and Monte Carlo arithmetic.

## **2.2 Floating Point Foibles**

Let’s look at two floating point related peculiarities test software developers may encounter. While neither of these are particularly profound from a technical viewpoint, nevertheless they can be the cause of grief in test systems software.

### **2.2.1 Formatted Output Precision Is Not Floating Point Representation Precision**

When creating a test report, it’s very common to make comparisons between a measured result and a limit or limits, make a PASS/FAIL assessment, and print the result. Quite often, the report is tabular in form with limited room for including all of the significant decimal digits of the floating-point values involved. The programming language’s output formatting capability is used to create a text string with the numeric values rounded to as many digits as will fit in the table cell or column. As a consequence, the PASS/FAIL result of the numeric comparison sometimes does not agree with the rounded values displayed. For example, two floating-point values that differ only in the fourth digit, will appear to be the same if displayed to three or fewer digits. More than a few indignant quality engineers have grieved hapless programmers over this issue.

Similarly, inexperienced developers may be astonished to discover that when a single precision floating point value is assigned to a double precision variable, apparently invalid decimal digits often appear (depending on the decimal value) at positions beyond those significant to the single value. The double precision value is of course no less accurate; however, if only *significant* decimal digits were being displayed or output, when the additional significant decimal digits are shown, the apparent affect is that the value somehow became less accurate.

### **2.2.2 Comparing Floating Point Values**

When porting source code between compilers and platforms, the same source code can yield different results depending upon the manner in which the compiler evaluates expressions, regardless of the similarity of the underlying floating point hardware.

National Instruments, a vendor of both test hardware and test software development systems, notes that the following code will produce different results for different compilers even when compiled and executed on the same machine.

```

/* More precision than can be represented. Note that an unsuffixed
   floating point constant is interpreted as a double in ISO C */
#define PI 3.1415926535897932384626433832795028841971

main() {

    /* Test 1 */

    if (acos(-1) == PI)
        printf("acos(-1) == PI Succeeded\n");
    else
        printf("Huh? acos(-1) and PI should be equal!!\n");

    /* Test 2 */

    if (1/acos(-1) == 1/PI)
        printf("1/acos(-1) == 1/PI Succeeded\n");
    else
        printf("Huh? 1/acos(-1) == 1/PI should be equal!!\n");
}

```

Different results indeed occur: when compiled with Microsoft Visual C++ 6.0, the first test fails and the second passes. When using the National Instrument's LabWindows/CVI development environment, both tests pass. In both cases, the same machine was used, an Intel Pentium III Coppermine processor running Windows NT 4.0.

To insure source code portability for situations such as these, NI offers a floating-point comparison software routine that provides a tolerance factor for small differences in floating point values.

### 2.3 IEEE 754 Floating Point Standard

Many seemingly difficult and non-intuitive issues surrounding floating-point code could be more easily managed if engineers and programmers became familiar with the details of floating-point representations and arithmetic. Prior to the mid 1980's hardware vendors sold machines with distinct and incompatible floating-point representations. Today nearly every major hardware vendor supports the IEEE 754 standard. An important reason for the success of IEEE 754 is that it mandates not only the precise layout of bits in a number's representation, but also the behavior of most arithmetic operations, including the handling of exceptional situations. This ensures portability of software across conforming machines.

A floating-point number is represented as

$$\pm d_0.d_1d_2d_3\dots d_{p-1} \times b^e$$

where  $d_0 \dots d_{p-1}$  are the digits of the significand;  $b$  is the base and  $e$  is the exponent. If  $d_0=1$  the representation is said to be *normalized*, otherwise the represented value is a *subnormal*.

Normalized values have the full  $p$  digits of precision. IEEE 754 specifies four representations, all of which have  $b = 2$ : single, double, single-extended and double-extended.

### 2.3.1 Single Precision

The single-precision, or 32-bit, format is used for the **float** type in Java and C and the REAL\*4 type in Fortran. Here the standard specifies a precise layout of bits

```

3 32222222 222111111111110000000000
1 09876543 21098765432109876543210
+---+-----+-----+
|s|   e   |           f           |
+---+-----+-----+

```

Here bit 31 is the sign-bit, bits 30-23 are eight bits of exponent, and bits 22-0 are 23 bits of fraction. If  $e$  is neither all zeros nor all ones, the represented number is  $1^{-s} \times (1.f)_2 \times 2^{e-127}$ , so we effectively have 24 bits of precision since the leading 1 bit is assumed, not stored. Subnormals are represented with an  $e$  field of all zeros, where we have  $1^{-s} \times (0.f)_2 \times 2^{-126}$ . When the  $e$  field is all ones, we represent infinities (+8 and -8) and NaNs (“not-a-number”); full details can be found in the standard itself or in many excellent summaries.

### 2.3.2 Double Precision

The double-precision, or 64-bit, format is used for the **double** type in Java and C and the REAL\*4 type in Fortran. The standard also specifies a precise bit layout for this format; bit 63 is the sign-bit, bits 62-53 comprise 11 bits of exponent in the range -1022...1023 for normalized values, and bits 52-0 are the 53 bits of fraction (with 54 true bits of precision since the leading 1 is not stored). Subnormals, infinities and NaNs are treated as in the single-precision case.

### 2.3.3 Single-Extended Precision

The standard does not specify a precise bit layout for the single-extended precision format, stating only that there must be at least 31 bits of precision for normalized values with a minimum exponent of at most -1022 and a maximum at least 1023.

### 2.3.4 Double-Extended Precision

The standard does not specify a precise bit layout for the double-extended precision format, stating only that there must be at least 63 bits of precision for normalized values with a minimum exponent of at most -16382 and a maximum at least 16383.

The floating point units on Intel’s ubiquitous x86 microprocessor product line contain a particular implementation of this format, called the 80-bit, or ten-byte, real format. This format uses one sign bit, fifteen exponent bits and 64 fraction bits (in which the leading bit is explicit, not assumed). This 80-bit format is the native format on the x86; programmers can store values

in memory as 32-bit floats or 64-bit doubles, but when loaded into registers these are expanded into the 80-bit format, and all computations are done and intermediate results stored in it.

Another implementation of double-extended is the so-called “quadruple” or 128-bit, format, which is expected to become widely supported.

### 2.3.5 Ranges

Programmers need to be aware of the range and precision of floating-point formats. For example if measurements are done on systems with 8 decimal digits of precision, computations in single-precision are likely unwarranted. Generally, computations should be done with as much precision as is efficiently available; the deliberate use of less-precise formats must be rigorously justified as acceptable for one’s particular data distribution.

	Single	Double	Intel’s 80-bit
Size in bits	32	64	80
Bits of precision	24 (23 explicit)	53 (52 explicit)	64 (all explicit)
Decimal digits of precision	7.22	15.95	19.26
Largest magnitude Normalized	$\pm(2-2^{-23})\times 2^{127}$ $\pm 3.4028\times 10^{38}$	$\pm(2-2^{-52})\times 2^{1023}$ $\pm 1.7977\times 10^{308}$	$\pm(2-2^{-63})\times 2^{16383}$ $\pm 1.1897\times 10^{4932}$
Smallest magnitude Normalized	$\pm 2^{-126}$ $\pm 1.1755\times 10^{-38}$	$\pm 2^{-1022}$ $\pm 2.2251\times 10^{-308}$	$\pm 2^{-16382}$ $\pm 3.3621\times 10^{-4932}$
Smallest magnitude subnormal	$\pm 2^{-149}$ $\pm 1.4013\times 10^{-45}$	$\pm 2^{-1074}$ $\pm 4.9407\times 10^{-324}$	$\pm 2^{-16446}$ $\pm 1.8225\times 10^{-4951}$

### 2.3.6 Flags and Traps

An extremely important part of the IEEE 754 standard is that operations that are invalid, that overflow, that underflow, that divide by zero, or that produce inexact results are flagged. Programmers can examine these flags and even set traps for them thereby avoiding the error-prone pollution of if-then-else statements for each operation. Unfortunately most high-level programming languages fail to give much (if any) support for floating-point flags and traps — even those languages advertising support of the standard — thus requiring the use of specialized libraries, or assembly language. Again test software in particular should make use of these techniques, lest the test code *and* the units under test both compute the same incorrect results from failing to detect NaNs or infinities in intermediate computations.

## 3 ALGORITHM SELECTION

Armed with the knowledge of floating point representation an engineer can make a proper decision of an algorithm implementation when multiple implementations exist. For example, the expression  $(x^2-y^2)$  can also be computed as  $(x+y)(x-y)$ . In the former case the rounding errors from the squaring operations will become the significant digits in the result in the case where the

subtraction results in cancellation, though [Gol91] points out that the latter expression is sometimes less accurate.

### 3.1 Case Study: Problems with Variance Computation in a COTS Framegrabber

A real-world example of the consequences of algorithm selection was encountered when we were using a commercial, off the shelf (COTS) framegrabber to capture video imagery from a FLIR (Forward Looking Infrared) under test. The problem was to "find the dot" within a frame of captured video, the dot being formed by the reflection of a laser target designator. We used a pattern-matching function provided by the framegrabber vendor that "walked" an idealized image of the laser dot across the video frame, looking for minima in the statistical variance between the pixel values of the idealized image and the frame. To our consternation, the routine failed miserably, unable to find a laser dot that was clearly visible in the raw data to the unaided eye. Analysis of the algorithm used by the vendor disclosed that the calculation of the sample variance (the square of the sample standard deviation) was being implemented according to the formula:

$$s^2 = \frac{1}{n-1} \sum_{j=1}^n (x_j - \bar{x})^2$$

Note the term that squares the difference between a value  $x$  and the mean  $\bar{x}$ . If the value of  $x$  is close to the mean (and this was the case with the test method being used), the difference is small, and the square of that small difference is smaller yet. As a result, all of the precision was being "washed out" such that the variance minima were not being detected. We proposed an alternative algorithm using the following, mathematically equivalent expression for the sample variance:

$$s^2 = \frac{1}{n-1} \sum_{j=1}^n x_j^2 - \bar{x}^2$$

When the same function was re-written using this alternative expression that avoids squaring the very small differences between floating point values, the "find the dot" routine worked just fine. We should note that the DSP implementation was in a vendor-unique 32 bit DSP floating point format (not IEEE 754), and that while we had found a pathological data set for the first expression, an equally ill-formed data set might well exist for the second expression.

There are several expressions for calculating sample variance. Each implementation has advantages and disadvantages, with the usual tradeoff being between computational efficiency versus error. The following two examples illustrate this trade off.

Suppose we need to acquire  $m$  data values  $\{X_1, X_2, X_3, \dots, X_m\}$ , and we want to estimate the sample mean ( $\bar{x}$ ) and sample standard variance ( $s^2$ ) for the data set. These values can be calculated using the one-pass West's algorithm:

The calculation for the mean  $\bar{x}$  is given by the following algorithm:

$$\begin{aligned}
M_1 &= X_1 \\
M_k &= M_{k-1} + \left( \frac{X_k - M_{k-1}}{k} \right), k = 2, \dots, m \\
\bar{x} &= M_m
\end{aligned}$$

$M$  is a real-valued storage register, even if the data values are integers.  $M_k$  holds the value of the mean of the first  $k$  data values acquired. As each data value is acquired, three operations are required to calculate the mean: one addition, one subtraction and one division.

The calculations for the sample variance by West's algorithm are given by:

$$\begin{aligned}
T_1 &= 0 \\
T_k &= T_{k-1} + (k-1) \cdot (X_k - M_{k-1}) \cdot \left( \frac{X_k - M_{k-1}}{k} \right), k = 2, \dots, m
\end{aligned}$$

The variance  $s^2$  can then be found for the set of  $m$  data values as

$$s^2 = \frac{T_m}{m-1}$$

$T$  is a real-valued storage register, even if the data values are integers.  $T_k$  holds the value of the variance of the first  $k$  data values acquired times the constant  $k-1$ . As each data value is acquired, five operations are required: one addition, two subtractions, and two multiplications (one subtraction and one division were already performed while calculating the mean).

The total operations count for calculating the sample mean and sample variance for West's Algorithm is:

$$8 + 3/m \text{ Floating Point Operations per data point acquired.}$$

This algorithm gives a mathematically equivalent expression for sample mean. But, the expression for variance is not exact. According to [Chan79], an upper bound for the relative error in the estimate of sample variance for a set of  $m$  data values is:

$$2\kappa\eta + \left( \frac{m}{2} + 1 \right)^\eta$$

where  $\kappa$  is the condition number of the data, and  $\eta$  is the relative precision of the computer.

West's algorithm is optimized for speed of computation, with some trade off in error. In contrast, the corrected two-pass algorithm [Press92] attempts to dynamically correct for machine round-off error, at some expense in computationally efficiency. The algorithm is given by:



$$s^2 = \frac{1}{m-1} \left\{ \sum_{j=1}^m (x_j - \bar{x})^2 - \frac{1}{m} \left[ \sum_{j=1}^m (x_j - \bar{x}) \right]^2 \right\}$$

If the calculation of the mean were exact, and there were no round-off errors in the subtraction, the second sum would evaluate to zero. Since the arithmetic is not exact, the second sum accumulates an approximation of the total round-off error, and applies the correction to the standard formula for sample variance.

Since there are several unique algorithms, each with inherent advantages and disadvantages, standard deviation therefore makes an interesting case study for algorithm selection.

### 3.2 Convergence

Validation of algorithms that are numerical in nature often presents unique challenges. A piece of numerical code could be in error because the programmer implemented the algorithm incorrectly. These types of errors can be investigated formally. But, even if the code correctly implements the desired algorithm, numerical routines can still produce erroneous results. For example, in the case of iterative procedures, one issue is *convergence*: some algorithms converge to the correct solution under most circumstances, but fail for certain “pathological” cases. The classic example is Newton’s method for root finding, when applied to a function that has a horizontal slope at some point near the root. For such a function and some initial guesses, a standard implementation of Newton’s method may fail to converge to the root. Determining the conditions under which an iterative algorithm will converge to the correct solution is usually possible, and fixes are often available to make the algorithm more robust (as is the case with Newton’s method).

### 3.3 Roundoff Error

Another problem encountered by some numerical routines is *round-off error*. Floating-point representation of real numbers introduces round-off error, and arithmetic operations can accumulate or magnify these errors to the point where the final result of a numerical calculation is meaningless. An oft-cited example is solving a linear system by Gauss-Jordan elimination where the coefficient matrix is nearly singular. A basic implementation of Gauss-Jordan elimination can give grossly erroneous results, and the errors result entirely from magnification of round-off errors. Fortunately, in this case, the problem can be overcome by modifying the algorithm, *e.g.*, to include maximum pivot scaling [Press92]. But the key point here is that the basic algorithm for Gauss-Jordan elimination, correctly implemented, can accurately solve one linear system, but not another. Thus, one problem with round off is that errors can be significant or not, *depending on the data* that is being analyzed.

### 3.4 Data Dependent Algorithm Selection

Suppose a routine is required to calculate the standard deviation of a set of input values. Does the code return an acceptable value for standard deviation? This question is far less benign than

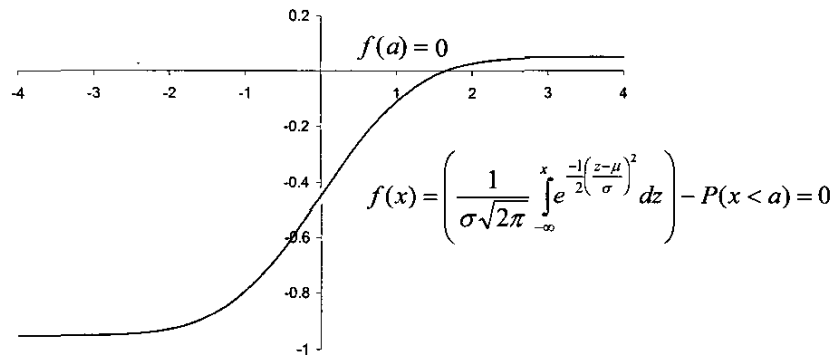
it might seem, even for the seemingly simple case of calculating standard deviation. As previously noted, the error can be unacceptably large even if the standard deviation algorithm is implemented correctly. Furthermore, the same algorithm might produce an acceptable result for one data set and an unacceptable one for another data set. The acceptability of the result may also depend on its intended application.

Problems with round-off error can sometimes be foreseen. For example, round-off error can occur when two numbers that are nearly equal are subtracted. In this case, the true difference between the numbers may lie at or beyond the last floating-point digit represented in either number. Avoiding or correcting for this case is usually sufficient to make the code more robust to round-off error. But in most cases, coding for robustness to counter propagation of round-off errors includes performance trade-offs.

To test the performance of numerical algorithms, there is a need to create data sets with known characteristics. For example, a set of values that is approximately normally distributed with a given mean and standard deviation can be generated by repeatedly inverting the Gaussian probability distribution function. Starting with a random deviate  $z \in [0, 1]$ , seek the value of  $a$  where  $P(x < a) = z$  for a standard normal distribution. Define the function

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2} dy - z$$

The root of  $f$  will give a normally distributed deviate from the random deviate  $z$ , and can be found using bisection, a method which is guaranteed to find the root for any probability distribution function.



Alternatively, an approximately normally distributed data set can be generated from a uniform deviate using the central limit theorem. If the original uniform distribution has mean  $\bar{x}$  and standard deviation  $\sigma_x$ , then define a random variable  $X$  consisting of the mean of  $n$  samples drawn at random from the uniform distribution. By the central limit theorem,  $X$  will be normally

distributed with mean  $\bar{x}$  and standard deviation  $\frac{\sigma_x}{\sqrt{n}}$  [Rice95]. So, to create a data set that is approximately normally distributed, start with a uniform distribution and sample it; the sample averages will give the desired data set.

The variance of a continuous random variable  $X$  with expected value  $E(x) = \mu$  and density function  $f(x)$  is [Rice95]:

$$\text{Var}(X) = \int_{-\infty}^{\infty} (x - \mu)^2 f(x) dx$$

For a uniform distribution over the interval  $[\mu - c, \mu + c]$ , the density function is given by  $f(x) \equiv \frac{1}{2c}$  on  $[\mu - c, \mu + c]$ , and zero elsewhere. So, the variance for a uniform distribution on  $[\mu - c, \mu + c]$  is

$$\begin{aligned} \text{Var}(X) &= \int_{\mu-c}^{\mu+c} (x - \mu)^2 \frac{1}{2c} dx \\ &= \frac{c^2}{3} \end{aligned}$$

The second method for generating a Gaussian deviate has the advantage that it does not depend on the convergence of a root-finding algorithm. The following function implements the second method:

```
#define MAX_14_BIT 16383
#define CLT_MAX 500

/-- Generate a Gaussian deviate as an unsigned, 14-bit value, with
/-- given mean and standard deviation (sigma). Successive calls to
/-- the function will return a set of values that is approximately
/-- normally distributed with sample mean and sample standard
/-- deviation that are input to the function.
unsigned short Gaussian (double mean, double sigma)
{
    int i;
    double deviate = 0.0;
    double range = sigma * sqrt (3.0 * CLT_MAX);

    /-- here is the central limit theorem
    for (i = 1; i <= CLT_MAX; i++) {
        deviate += ((2.0*((double) rand() / (double) RAND_MAX )) - 1.0)
            * range + mean;
    }

    /-- the normally distributed deviate will be the average of the
    /-- CLT_MAX values generated in the loop
    deviate /= CLT_MAX;

    /-- Clip the deviate in the range 0 to MAX_14_BIT
    if (deviate > MAX_14_BIT)
        deviate = MAX_14_BIT;
    if (deviate < 0.0)
```

```

    deviate = 0.0;
return (unsigned short) deviate;
}

```

Now, a normally distributed data set can be generated whose mean and variance are pre-determined. Such a data set can be used to test the accuracy of algorithms used to compute sample mean and variance. But, in order to do this, it is essential that the 'exact' value of the mean and variance are known. But, numerical routines inherently have round-off errors whose effects on the final calculation are indeterminable. How can the exact values be calculated?

#### 4 ARBITRARY-PRECISION ARITHMETIC

To calculate the exact value of sample mean and variance for a given data set, it is possible to use a computer algebra program such as Maple or Mathematica. The original data set is read in, and converted to integers. The computer algebra system operates on the data values as integers. The calculations between integers are accomplished with integer arithmetic, so there is no round-off error in the calculations. The exact rational number can be approximated to any degree of precision. The final answers can then be calculated to any level of precision, accurate to the last decimal place represented. The following Maple code implements this idea:

```

> sumdata:=0:
> for i from 1 by 1 to 128 do
    sumdata:=sumdata+data[i]:
end do:
> datamean:=sumdata/128;
datamean :=  $\frac{1048599}{128}$ 
> evalf(datamean,30);
8192.1796875000000000000000000000
> sumdata:=0:
> for i from 1 by 1 to 128 do
    sumdata:=sumdata+(data[i]-datamean)^2;
end do:
> variance:=sumdata/(127);
variance :=  $\frac{4696943}{16256}$ 
> evalf (variance,30);
288.935962106299212598425196850
> standard_deviation:=sqrt(variance);
standard_deviation :=  $\frac{1}{2032}\sqrt{1193023522}$ 
> evalf (standard_deviation,30);
16.9981164281898955523023417348

```

Libraries for arbitrary-precision arithmetic can be obtained for most popular programming languages. Java features the class `BigDecimal` in its Core API (standard library). Programmers utilizing such libraries do need to be aware of performance implications of trying to achieve the maximum precision possible. For example, computing the sequence

$$\begin{aligned}p(0) &= 0.01 \\ p(n+1) &= 4p(n) - 3p(n)^2\end{aligned}$$

takes several hours to compute the 20th iteration with the Java `BigDecimal` class.

## 5 MONTE CARLO METHODS

Another approach to dealing with precision loss and round-off error is to introduce randomness into floating-point algorithms. An algorithm that, in some essential way, employs the tossing of a fair coin is variously called a probabilistic, or randomized, or Monte Carlo program. Typically the algorithm, one or more times during execution, will need to make a random selection from among  $n$  alternatives; it then proceeds to toss an  $n$ -sided coin, which, operationally, amounts to getting a random integer in the range  $1..n$ . Some of the better known uses of randomization are found in primality testing (the Solovay-Strassen and Miller-Rabin algorithms); in randomized string-matching (Karp-Rabin); in Monte Carlo integration; in finding nearest-neighbors (Rabin); and in randomized Quicksort (Knuth). An excellent survey of the uses of randomization in sequential and distributed algorithms is [Gup94].

Monte Carlo Arithmetic (MCA) [Par97] employs randomization in floating-point operations so as to effect (1) random rounding, which produces round-off errors that are both random and uncorrelated; and (2) precision bounding, which can be used to detect the “catastrophic cancellation” that results from subtraction of substantially similar floating-point values (and, not incidentally, to implement dynamically varying precision). In MCA, arithmetic operators and operands are randomized, or perturbed with random values in some predefined way; thus, using MCA, multiple evaluations of the sum of two operands will generally yield different results each time. The following illustrative example is given in [Par97]:

Consider the equation

$$a x^2 - b x + c = 0,$$

with  $a = 7169$ ,  $b = 8686$ , and  $c = 2631$ ; exact (rounded) solutions are

$$\begin{aligned}r1 &= .60624386632168620, \\ r2 &= .60536165746126819.\end{aligned}$$

The C language statements,

$$\begin{aligned}r1 &= (-b + \text{sqrt}(b*b - 4*a*c)) / (2 * a); \\ r2 &= (-b - \text{sqrt}(b*b - 4*a*c)) / (2 * a);\end{aligned}$$

using IEEE floating-point, with round-to-nearest (the default), yielded the values

$$r1 = .606197, \quad r2 = .605408.$$

It was also run five times with (single-precision) MCA, yielding five distinct values for each of  $r1$  and  $r2$ , which averaged out to

$$r1 = .606213, \quad r2 = .605338,$$

with standard deviations of .000037 and .000033, and standard errors of .000016 and .000015, respectively. The standard deviations provide an estimate of the absolute errors in the computed results, and a large standard deviation would indicate a large error. The standard error estimates the absolute error in the computed average, tends to increase inversely with the square root of  $n$ , and, it turns out, estimates the number of significant digits in the result. So, taking a sample of (in this case, five) computations provided good average values, as well as statistics that reflect deviations among the samples and estimates of the absolute error.

From [Par97]:

“Our key idea is that the information lost by using finite-precision computation can also be modeled as inexactness, and can also be implemented as random error. Floating-point arithmetic differs from real arithmetic only in that additional random errors are needed to model the loss of significance caused by the restriction of values to limited precision... [F]or a bounded arithmetic expression (possibly involving inexact values), the expected value of its computed Monte Carlo average will be its expected real value. Furthermore, the accuracy of this average is estimated by its standard error.” (pp.9-10)

Again, from [Par97]:

“the distinction between exact and inexact values can be realized by modeling inexact values as random variables... our goal of MCA becomes the implementation of arithmetic on real numbers (exact values) and random variables (inexact values), and somehow this is to be done with floating-point hardware.” (op. cit., p.31)

In making the case for MCA, these are among the cited benefits:

--Unlike conventional floating-point arithmetic, MCA maintains information about the number of significant digits of floating-point values; consequently,

- It (probabilistically) detects occurrences of catastrophic cancellation;
- Variable precision is supported;
- Round-off errors are, indeed, random;
- Addition becomes (statistically) associative.

Of course, all this does not come without a price: because more logic is required than for conventional floating-point arithmetic, MCA is more expensive to implement in hardware; expressions have to be evaluated multiple times, adding computational overhead.

## 6 SUMMARY

We have seen that when manipulating measurements as floating point values, developers must take care so as to not throw away hard-earned accuracy. Seemingly simple operations can result in subtle errors that can be difficult to analyze.

Much if not most of today's test software is executed on IEEE Std. 754 implementations. It behooves us as test system developers to understand this popular standard, and the facilities it provides for detecting exceptional conditions. It's not enough to simply perform intermediate calculations in double precision: algorithm selection must be carefully considered to preserve measurement accuracy. We saw that even for the apparently simple case of calculating standard deviation, careful consideration must be given to data dependencies when selecting an algorithm.

Arbitrary precision and Monte Carlo Arithmetic offer alternative approaches for managing precision loss and round-off error, though these may be impractical for many test system implementations.

## 7 REFERENCES

[Gol91] David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic, *ACM Computing Surveys* **23**(1): 1991.

[Gup94] Rajiv Gupta, Scott A. Smolka, and Shaji Bhaskar, On Randomization in Sequential and Distributed Algorithms, *ACM Computing Surveys* **26**(1): 1994.

[Par97] D. Stott Parker, Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic, UCLA Computer Science Department Technical Report CSD-970002, 1997. Available at <http://www.cs.ucla.edu/~stott/mca/>.

[KD98] W. Kahan and Joseph D. Darcy, How Java's Floating-Point Hurts Everyone Everywhere, ACM 1998 Workshop on Java for High-Performance Network Computing, Invited Talk, <http://www.cs.ucsb.edu/conferences/java98/papers/javahurt.pdf>.

[PJS92] Heinz-Otto Peitgen, Dietmar Saupe, H. Jurgens, Chaos and Fractals: New Frontiers of Science, Springer-Verlag, 1992.

[Vin90] J. David Vincent, Fundamentals of Infrared Detector Operation and Testing, Wiley Interscience, 1990. ISBN 0-471-50272-3. pp 275-290.

[Chan79] Chan, T.F., and J.G. Lewis, 1979. Computing Standard Deviation: Accuracy. Communications of the ACM, Volume 22, Number 9, pp. 526-531.

[Press92] Press, W.H., S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, 1992. *Numerical Recipes in C: The Art Of Scientific Computing, 2d Edition*. Cambridge University Press, New York, NY, ISBN 0-521-43108-5, 992 pages.

[Rice95] Rice, J.A., 1995. *Mathematical Statistics and Data Analysis, 2d Edition*. Duxbury Press, Belmont, CA, ISBN 0-534-20934-3, 602 pages.