Monte Carlo Arithmetic: a framework for the statistical analysis of roundoff error

D. Stott Parker (stott@cs.ucla.edu)*
Brad Pierce (pierce@cs.ucla.edu)*
Computer Science Department
University of California
Los Angeles, CA 90095-1596

Paul R. Eggert (eggert@twinsun.com)
Twin Sun, Inc.
360 N. Sepulveda Blvd., Suite 2055
El Segundo, CA 90245-4462

March 28, 2000

Abstract

Monte Carlo Arithmetic (MCA) is an extension of standard floating-point arithmetic that exploits randomness in basic floating-point operations. MCA uses randomization to implement random rounding — which makes rounding errors behave like random variables — and random unrounding — which randomly extends the inputs of arithmetic operations to arbitrary precision. In computation with MCA, one can use either random rounding or random unrounding, or both. Using neither gives ordinary floating-point arithmetic.

Randomization has practical applications. Random unrounding detects catastrophic cancellation, which is the primary way that significant digits are lost in numerical computation. Running a program multiple times gives different results each time, and the differences between these results track loss of their significant digits, and also measure their sensitivity to perturbations in the computation. In fact, randomization also can be used to represent inexact values (values known to only a few significant digits), and to implement virtual precision, a scheme that lets us vary the precision of floating-point computation dynamically.

Randomization also has theoretical applications. It can transform a floating-point computation into a Monte Carlo computation, and transform roundoff analysis into statistical analysis. Because they behave like random variables, the rounding errors produced by random rounding can be analyzed statistically. This gives a way to avoid some anomalies of floating-point arithmetic. For example, while floating-point summation is not associative, Monte Carlo summation is 'statistically associative' up to the standard error of the sum. Generally, MCA gives a different perspective on the study of error.

Keywords: floating-point arithmetic, floating-point rounding, roundoff error, random rounding, ANSI/IEEE floating-point standards, significance arithmetic, Monte Carlo methods

*This material is based upon work supported by the National Science Foundation under Grant No. CCR97-32309. Any opinions, findings and conclusions or recomendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF).

1 Introduction

This is a short review of an extension to floating-point arithmetic that we have investigated over the past five years.

1.1 Monte Carlo Arithmetic

Monte Carlo Arithmetic (MCA) is a model of floating-point arithmetic in which arithmetic operators and their operands are *randomized* (randomly perturbed). For example, rather than insist that floating-point addition obey

$$x \oplus y = fl[\, x + y \,] = (x + y)\,(1 + \delta)$$

where δ is some deterministically-defined value (such as the relative error incurred by rounding to the nearest floating-point number, as in IEEE floating-point arithmetic [1, 3]), we allow δ to be a random variable (in implementations δ is a pseudorandom generator.)

MCA is an extension of floating-point arithmetic that permits randomization to be used both in rounding and in processing inexact operands. The result of every arithmetic operation is randomized in a predefined way. As a result, the addition $x \oplus y$ can yield different values if evaluated multiple times.

With MCA, then, if a program is run on the same input n times, then each recalculation yields a slightly different answer. These answers constitute a sample distribution to which the whole array of standard statistical methods can be applied. If (as is often the case) the sample mean $\widehat{\mu}$ estimates the exact solution, then the sample standard deviation $\widehat{\sigma}$ estimates the error in the result of any single run, and the sample standard error $\widehat{\sigma}/\sqrt{n}$ estimates the error in $\widehat{\mu}$ after n such runs. Each recalculation is an experiment in a Monte Carlo simulation — a simulation of the sensitivity to rounding of this particular combination of input and program.

1.2 Why Monte Carlo Arithmetic is useful

Two general arguments can be made about the usefulness of MCA:

- MCA has practical applications. It gives an empirical way for end users to gauge the number of significant digits in computed values. It also gives a way to gauge the 'stability' of a program (i.e., gauge its sensitivity to computation errors). Also, it gives a way to represent inexact values (values known only to a few digits).

- MCA has theoretical applications. It permits Monte Carlo methods to be used in estimating the accumulation of roundoff error. By treating individual rounding errors as random variables, it permits statistical methods to be used in analyzing error. As a result, well-known anomalies of floating-point [9, p.196] can be circumvented by MCA.

Probably the strongest argument for considering MCA is that it gives a way to make numerical computing *empirical*, and answer the following very practical questions:

> How much sensitivity to rounding errors is there in the results that were generated by a particular code running on a particular machine applied to a particular input? How can one determine the degree to which roundoff error clouds these particular results?

MCA illustrates the potential for tools that support new kinds of *a posteriori* roundoff error analysis.

Of course, Monte Carlo error analysis can give no iron-clad guarantees that serious roundoff error is absent in a computation. As Kahan [8, p.24] points out, computations can be "virulently unstable but in a way that almost always diverges to the same wrong destination". The practitioner must decide whether the level of sensitivity suggested by this simulation faithfully reflects reality. MCA is not a panacea; it is a tool.

Furthermore, there are many other established approaches for analyzing roundoff error in numerical computations. A few are: running in higher precision, computing condition numbers, using interval arithmetic, and performing a formal roundoff analysis. Each has strengths and weaknesses. Monte Carlo error analysis is just an alternative; it will not replace these methods.

We are not disputing that formal roundoff error analysis is indispensable in the design of high-quality numerical algorithms. Still: after design, algorithms must still be implemented as routines in a high-level computer language, then combined with other routines into a complete program. Moreover, before a program can be used, it must be translated by a compiler into the code of the particular machine on which it will be run. These issues are very difficult to treat formally.

> Competent engineers rightly distrust all numerical computations and seek corroboration from alternative numerical methods, from scale models, from prototypes, from experience, ...
> — W. Kahan [8, p.34]

There are many ways to look at error analysis, and new ones can definitely help. We believe that Monte Carlo Arithmetic will be a useful tool in the analysis of roundoff error.

1.3 Roadmap

The remaining sections of this paper go as follows:

Section 2. Some Examples
We present several examples that both expose some inherent weaknesses inherent in floating-point computation, and show some practical applications of Monte Carlo Arithmetic. These examples are small and mostly familiar, so that the MCA concept is completely grounded, and readers can appreciate what it offers.

Section 3. Monte Carlo Arithmetic
We formalize MCA and explain why it succeeds in exposing error in the preceding suite of examples. We also give some perspective on why MCA has been formalized as it is, and contrast it with an interesting alternative.

Section 4. Other Strengths of Monte Carlo Arithmetic
We give several other illustrative examples that highlight several theoretical and practical strengths of MCA. Specifically, we show that MCA avoids floating-point anomalies, gives an implementation of variable precision (if we randomize bits in computed results below a requested precision), and produces roundoff errors with good statistical properties.

Section 5. Conclusion
Finally, we try to put the contributions here in perspective. The primary message to draw from our results is that, although MCA is only a straightforward extension of floating-point arithmetic, it is a useful tool.

Again, this paper is only a short review of several years' investigation. A much longer and more thorough presentation is available in [10]. This paper, and a C demonstration program by Parker that runs all the examples in this paper, are available at `http://www.cs.ucla.edu/~stott/mca/`.

2 Some Examples

2.1 A simple example

Kahan (e.g., [7]) has stressed that even computations as simple as solving $ax^2 - bx + c = 0$ present interesting problems for floating-point arithmetic. Consider finding the second root for

$$7\,x^2 \quad - \quad 8686\,x \quad + \quad 2 \quad = \quad 0.$$

With $a = 7$, $b = -8686$ and $c = 2$, the C statement

```
r  =  (-b - sqrt(b*b-4*a*c))/(2*a);
```

yielded Table 1, using IEEE floating-point with the default rounding (round to nearest).

precision	r
IEEE single precision	.000279018
exact solution (rounded)	.00023025562642454231

Table 1: Root of $7\,x^2 \ - \ 8686\,x \ + \ 2 \ = \ 0$, computed with IEEE floating-point.

When we instead randomize the inputs and outputs of floating-point operations, the results vary each time we execute the program. (The coefficients a, b, c and the constants 2 and 4 were not randomized as they are exact, i.e., representable precisely in floating-point format.) Using gcc version 2.7.2 with no options (except '-lm') on a Sun SPARCstation 20 model 501-2324 running SunOS 5.5, running the program 5 times with single precision MCA yielded Table 2. For simplicity, we used IEEE single precision floating-point representation in our implementation of MCA. Similar results can be produced in any precision.

run	r
1	.000198747
2	.000248582
3	.000251806
4	.000177380
5	.000203571
sample mean:	.000216017
sample standard deviation:	.000032739
sample standard error:	.000014641

Table 2: Root of $7\,x^2 \ - \ 8686\,x \ + \ 2 \ = \ 0$, computed with single precision MCA.

Running a program n times with MCA ultimately gives, for each value x being computed, n samples $x_1, \ldots, x_n$ that disagree on the random digits of their errors. These samples have an underlying distribution with **mean** μ and **standard deviation** σ, which are respectively estimated by the computed **sample mean** (average) $\widehat{\mu}$ and **(unbiased) sample standard deviation** $\widehat{\sigma}$:

$$\widehat{\mu} \quad = \quad \frac{1}{n} \sum_{i=1}^{n} x_i \qquad\qquad \widehat{\sigma} \quad = \quad \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (\, x_i \ - \ \widehat{\mu}\,)^2}.$$

The standard deviation in Table 2 gives a rough estimate of the error in any one run *and* in the result computed using IEEE floating-point.

An estimate of the error in $\widehat{\mu}$ is the standard error:

$$\textbf{sample standard error} \quad = \quad \sigma/\sqrt{n}, \qquad \text{which we approximate as } \widehat{\sigma}/\sqrt{n}.$$

Just as the standard deviation of the samples $\widehat{\sigma}$ gives an estimate of the error in any one sample, the standard error is the standard deviation of $\widehat{\mu}$, and gives an estimate of the error in $\widehat{\mu}$.

Use of standard error is common in scientific work [12]. A widely-used notation expressing the order of magnitude of error is

$$\text{“}\mu \quad = \quad \textbf{sample mean} \ \pm \ \textbf{sample standard error}\text{”} \qquad (\text{“}\ \mu \quad = \quad \widehat{\mu} \ \pm \ \widehat{\sigma}/\sqrt{n}\ \text{”}).$$

This notation gives "one-standard-deviation" estimates on the error in $\widehat{\mu}$, so it does not give any kind of hard bound. It merely says that the sample standard error is an order-of-magnitude estimate of the error in $\widehat{\mu}$. This is sufficient for our purposes here.

2.2 A startling example

Consider the following marvelous iteration adapted from [2]. Define the sequence (x_k)

$$\begin{aligned} x_0 &= 1.5100050721319 \\ x_{k+1} &= (3x_k^4 - 20x_k^3 + 35x_k^2 - 24) \ / \ (4x_k^3 - 30x_k^2 + 70x_k - 50). \end{aligned}$$

As demonstrated in Table 3, *depending on the precision of one's machine, the sequence converges to either 1, 2, 3, or 4.* Actually IEEE double precision converges to 3, and IEEE single precision converges to 2.

precision	x_{30} *(computed with decimal arithmetic)*
30 digits	`3.00000000000000000000000000000`
24 digits	`3.00000000000000000000000`
20 digits	`3.0000000000000000000`
16 digits	`3.000000000000000`
12 digits	`1.99999999990`
10 digits	`4.000000000`
8 digits	`1.9999980`
6 digits	`1.99990`
4 digits	`2.097`
2 digits	`1.0`

Table 3: Values of x_{30} computed with rounded decimal arithmetic of different precisions.

With MCA, we obtained the values in Table 4. With 10 samples, again using uniform input and output randomization, the extremely unstable nature of the iteration is discernible from the large standard deviation of x_{30}. Single-precision computation for this iteration converges to 2, but the large standard deviations in this table show that, with MCA, results other than 2 are obtained with high probability. Many standard deviations in this table are as large as the average values, in this case reflecting a complete loss of significant digits.

Generally speaking, one cannot identify unstable iterations with a fixed procedure. However, huge variances among intermediate iterates should raise warnings about the iteration, and in this example do reflect instability.

k	x_k *(run 1)*	x_k *(run 2)*	x_k *(run 3)*	x_k *(run 4)*	x_k *std. dev.*
1	1.510005	1.510005	1.510005	1.510005	.00000
2	2.377459	2.377445	2.377435	2.377399	.00003
3	1.509883	1.509991	1.510115	1.510208	.00014
4	2.378099	2.377534	2.376805	2.376342	.00078
5	1.505354	1.509288	1.514308	1.517498	.00537
6	2.404056	2.381407	2.354659	2.338808	.02886
7	1.270905	1.481644	1.640637	1.707481	.19418
8	0.543637	2.582219	2.060582	2.018529	.87673
9	0.823412	3.916467	1.997540	1.999803	1.28080
10	0.960642	4.016273	2.000010	1.999986	1.27894
⋮	⋮	⋮	⋮	⋮	⋮
30	1.000001	4.000012	1.999982	1.999992	1.25831

Table 4: Four sample runs of x_k and their standard deviation, using single precision MCA.

2.3 Mild Instability

For another example, consider the Tchebycheff polynomial studied by Wilkinson

$$\begin{aligned} T_{20}(z) &= \cos(20\ \cos^{-1}(z)) \\ &= 524288\, z^{20} - 2621440\, z^{18} + 5570560\, z^{16} - 6553600\, z^{14} + 4659200\, z^{12} \\ &\quad -2050048\, z^{10} + 549120\, z^{8} - 84480\, z^{6} + 6600\, z^{4} - 200\, z^{2} + 1 \end{aligned}$$

The polynomial is moderately ill-conditioned near 1, because of cancellation among the coefficients. The coefficients are all representable exactly within IEEE 24-bit single precision (just: the binary representation of 6553600 is 23 digits long). The increase in ill-conditioning as z nears 1 can be visualized with MCA and a scatterplot, as shown in Figure 1.

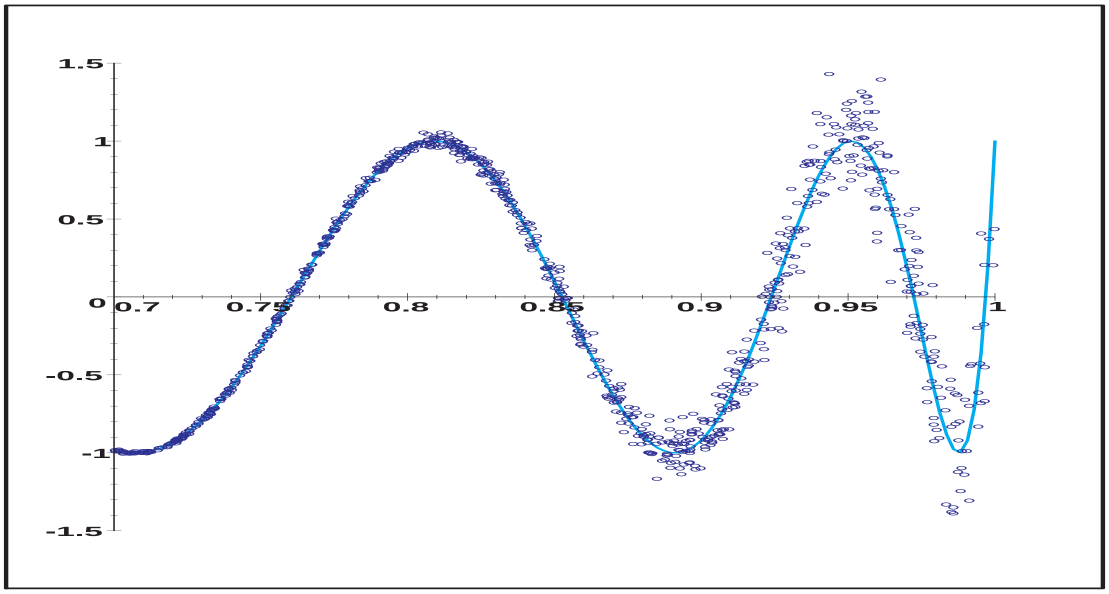

Figure 1: MCA single-precision results performed at random points, exhibiting instability of the Horner $T_{20}(z)$ computation.

2.4 Gaussian Elimination, ill-conditioning, and significant digits

Next, consider solving the linear system $A\mathbf{x} = \mathbf{b}$ using Gaussian elimination with partial pivoting. A widely used rule of thumb is that the relative error $\|\widehat{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\|$ in the computed solution $\widehat{\mathbf{x}}$ is on the order of $\kappa(A) \cdot \mathbf{u}$, where $\kappa(A)$ is the condition number of A and $\mathbf{u}$ is the unit roundoff (bound on relative error due to rounding) [6]. When $\kappa(A)$ is very large, A is called *ill-conditioned.* This problem becomes a classic example of ill-conditioning when we use the nearly singular $n \times n$ Hilbert matrix $A = (1/(i + j - 1))$ (or more precisely its floating-point approximation).

Using a double-precision implementation of MCA, we ran Gaussian elimination with partial pivoting on the 10×10 inexact Hilbert matrix and a specific $\mathbf{b}$ vector. With 5 samples we obtained the solution in Table 5.

$\mathbf{b}$	*exact solution to* $A\mathbf{x} = \mathbf{b}$	*IEEE double solution*	*MCA double solution*
1	2.4609375	$2.460\mathit{8240}$	$2.460\mathit{8\underline{335}}$
2^{-1}	−216.5625	$-216.5\mathit{526178}$	$-216.5\mathit{5\underline{34088}}$
2^{-2}	4439.53125	$4439.\mathit{3195461}$	$4439.\mathit{\underline{3359787}}$
2^{-3}	−36599.0625	$-3659\mathit{7.1294209}$	$-3659\mathit{7.\underline{2756677}}$
2^{-4}	148507.734375	$14849\mathit{8.4820907}$	$14849\mathit{\underline{9.1670631}}$
2^{-5}	−323760.9375	$-3237\mathit{35.4324287}$	$-3237\mathit{3\underline{7.2859033}}$
2^{-6}	387105.46875	$3870\mathit{63.5276615}$	$3870\mathit{\underline{66.5268827}}$
2^{-7}	−239301.5625	$-2392\mathit{60.9547177}$	$-2392\mathit{\underline{63.8179388}}$
2^{-8}	59825.390625	$598\mathit{04.0379124}$	$598\mathit{0\underline{5.5248358}}$
2^{-9}	0.	$\mathit{4.7021712}$	$\mathit{\underline{4.3783411}}$

Table 5: Gaussian Elimination solutions. The 5-sample average MCA solution is shown. Non-significant digits are italicized; underlined digits are within the magnitude of the standard error.

IEEE double precision gives about 16 decimal digits' precision, so $\mathbf{u} \approx 10^{-16}$. When $n = 10$ the condition number $\kappa(A) \approx 10^{13}$, so the rule of thumb predicts a relative error in the computed solution of $\|\widehat{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\| \approx 10^{-3}$. This is often interpreted as predicting that $\widehat{\mathbf{x}}$ will be accurate to about 3 digits. This prediction is borne out by Table 5 — except for the last entry, which has zero significant digits. The rule of thumb makes no guarantees about specific entries of $\mathbf{x}$.

Table 5 also shows that the standard errors in the computed MCA solution reflect the problem ill-conditioning. For this problem, randomization appears to give a good empirical estimate of the actual number of significant digits in each entry of $\widehat{\mathbf{x}}$. Moreover, the relative size of the standard error measures algorithm stability (sensitivity to perturbation) and problem ill-conditioning.

Often it is pointed out that $\widehat{\mathbf{x}}$ is the exact solution of a perturbed problem $(A + E)\mathbf{x} = \mathbf{b}$, where E is a matrix of small errors ($\|E\| \approx \mathbf{u}\|A\|$). While backward error results like this give reassurance, and show what floating-point arithmetic does guarantee, it is important to recognize that they do not give guarantees about the number of significant digits in any specific result.

2.5 Standard Deviations: caveat error

The **textbook algorithm for standard deviation** shown in Figure 2 is known to everyone. It is probably executed many billions of times every day. Yet it is unstable. When the standard deviation is orders of magnitude smaller than the mean, its results may be completely inaccurate. For example, when `N` is 127, `X(I) = I` $+ 10^5$, and IEEE single-precision computation is used, this algorithm computes `SIGMA` as 45.6126, although the correct value is about 36.8058.

```
C __ Textbook standard deviation algorithm __
      SUM = 0
      SUMSQ = 0
      DO 10 I=1,N
      SUM = SUM + X(I)
 10   SUMSQ = SUMSQ + X(I)**2
      XBAR = SUM/N
      SIGMASQ = (SUMSQ - SUM*XBAR) / (N-1)
      SIGMA = SQRT(SIGMASQ)
```

```
C __ Two-pass algorithm __
      SUM = 0
      DO 10 I=1,N
 10   SUM = SUM + X(I)
      XBAR = SUM/N
      T = 0
      DO 20 I=1,N
 20   T = T + (X(I) - XBAR)**2
      SIGMA = SQRT( T / (N-1) )
```

Figure 2: The textbook algorithm and the two-pass algorithms for computing standard deviations. Despite its enormous popularity, the textbook algorithm is unstable.

Ironically, the 'less clever' **two-pass standard deviation algorithm** is much more accurate. We can show this by computing, for different values of c,

$$\widehat{\sigma} \quad = \quad \text{standard deviation}(\ \{\ 1+c,\ \ldots,\ 127+c\ \}\ )$$

using both algorithms. Despite the fact that, for any c, we should have

$$\widehat{\sigma} \quad = \quad \text{standard deviation}(\ \{\ 1,\ \ldots,\ 127\ \}\ ) \quad = \quad \sqrt{170688/126} \quad \approx \quad 36.8058$$

the two algorithms give different results. Figure 3 shows how that the textbook algorithm's computed values vary significantly from 36.8058 as c increases, while the two-pass algorithm's computed results remain accurate. MCA gives a way to visualize the instability and appreciate the importance of choice of algorithm.

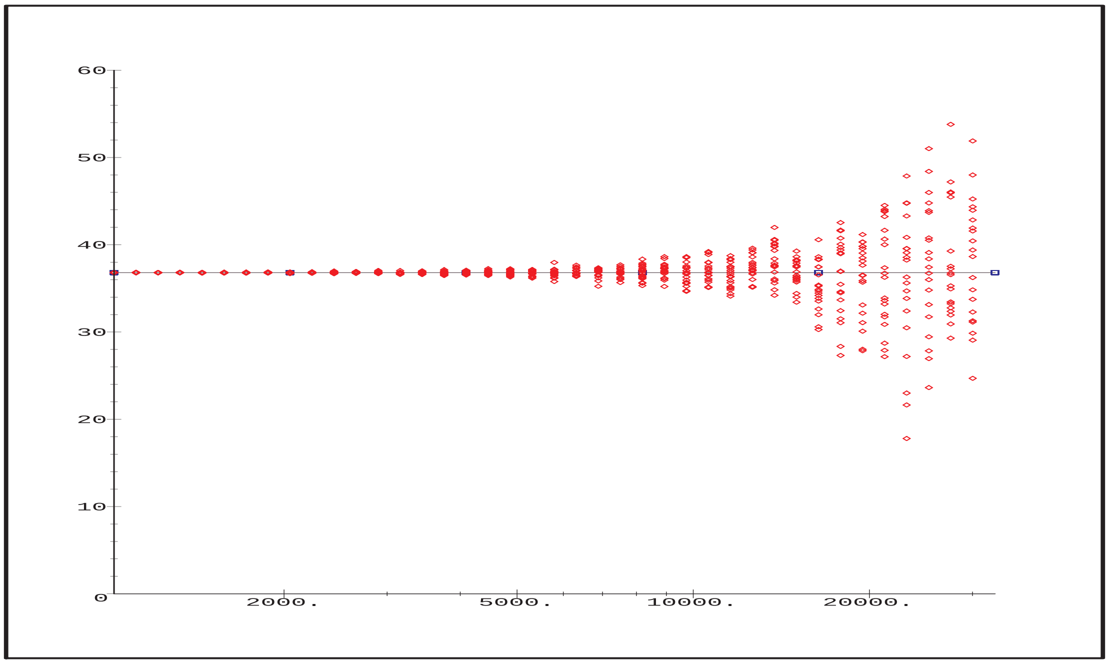

Figure 3: Computed `SIGMA` values for the standard deviation of $\{1+c, 2+c, \ldots, 127+c\}$ using the textbook algorithm (scattered points) and the two-pass algorithm (the line at 36.8058), which gives the correct result. The horizontal axis gives the value of c, and for each c value chosen, twenty single precision MCA samples are plotted.

2.6 Chaos: your results will vary

Chaotic computations are often defined as those that are "highly sensitive to initial conditions". Formally, this can be equivalent to requiring that the computations be numerically unstable. Nevertheless, tutorials about chaos often start with this definition, and then proceed by reporting results of straightforward numeric computations.

For example, consider the two iterations of an instance of the **logistic equation**

$$x_{n+1} \quad = \quad 4 \; x_n \; (1 - x_n)$$

plotted in Figure 4. Single-precision MCA was used, so different iterations obtain different pseudo-random rounding errors. The two iterations completely diverge from one another around iteration 20. This divergence also occurs reliably if this computation is performed multiple times, or with different random seeds. Performing the computation in double precision does not help: with double precision MCA, results diverge from one another at around 50 iterations.

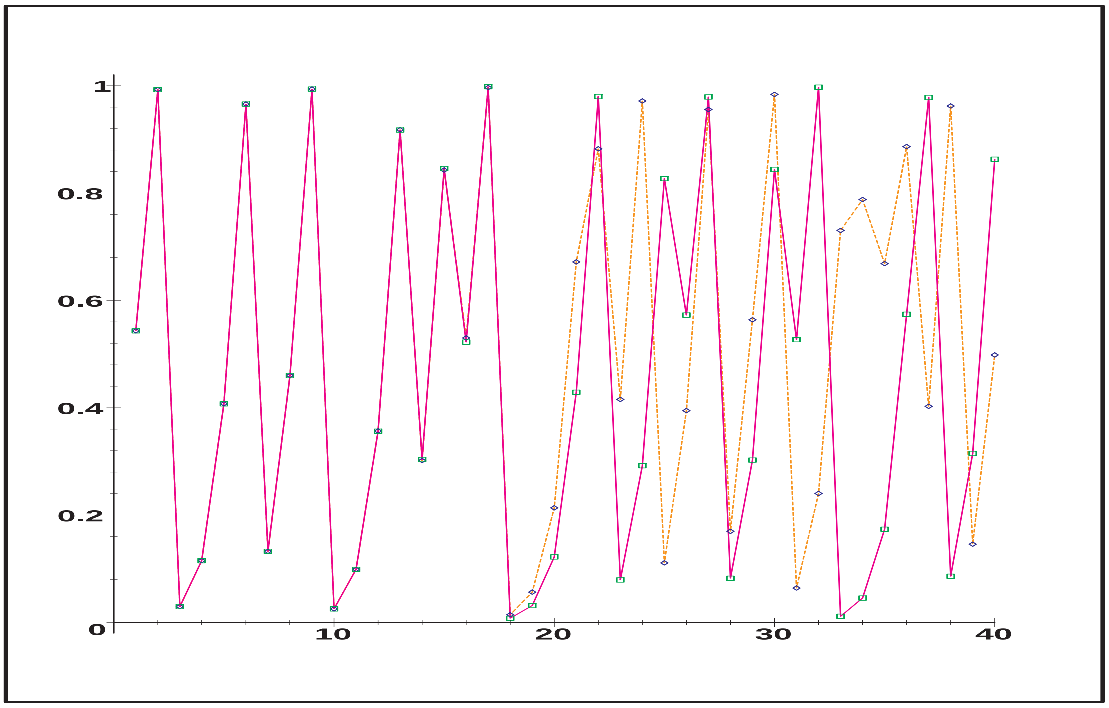

Figure 4: Two independent iterations of the logistic equation $x_{n+1} = 4x_n(1 - x_n)$ using single precision MCA ($x_1 = 0.54321$). Rounding error differences eventually lead to very different results.

While the logistic equation gives one of the simplest examples of chaos, it is closely related to other frequently-studied examples. Also, more complex naturally arising chaotic phenomena can be equally unstable. For example, divergence of results also occurs in integration of chaotic differential equations [10].

Summarizing, MCA gives a way to detect high sensitivity to roundoff in, and apply statistical analysis to, results of chaotic computations. It seems many people misunderstand the basic instability of chaotic computations, apparently believing that double precision is 'more than enough' for accurate results. It is not.

3 Monte Carlo Arithmetic

A more complete presentation of most of the material in this section is in [10]. We assume the reader is familiar with basic issues in floating-point arithmetic. Goldberg's tutorial [5] and Higham's encyclopedia [6, chs.1–5] are great references.

The relationship between randomness and error has been exploited for centuries. Highlights include Gauss' development of a theory of errors identifying them with random variables in the early 1800s; Ulam and von Neumann's development of the Monte Carlo method in 1947; use of statistical methods in numerical analysis by von Neumann and Goldstine in 1947; Forsythe's proposal for random rounding in 1950; the implementation of random rounding by Kahan in Hull and Swenson's 1966 work on roundoff error in differential equations; and the CESTAC and CADNA environments developed by Vignes since 1972. See [10] for a literature survey.

3.1 Exact and inexact values

Exact values are real numbers that can be exactly represented in a given floating-point format. **Inexact values**, by contrast, are either real values that must be rounded (to an approximation) in order to fit this format, or real values that are not completely known. Rounding discards information, hence usually increases inexactness.[1]

This is an important distinction. In scientific computation most quantities are inexact; constants like the speed of light in a vacuum ($\sim 2.99792458 \times 10^8 m/sec$) are known only to a few digits. The inexactness can be due to ignorance, uncertainty, estimation, measurement or computational error, but the upshot is that we have only a few significant digits.

Inexact values can arise in several ways in floating-point arithmetic:

1. An inexact value is a value known to only a few significant digits (like the speed of light).
2. An inexact value is the result of rounding of a higher-precision computation (or of rounding any inexact value).

Because an inexact value could represent a whole range of possibilities, one can only guess which of these real numbers it 'actually' represents. Conventional floating-point arithmetic, which adopts the assumptions of forward error analysis, makes the simplest possible unbiased guess — it treats all operands as if they were exact. For example, it extends single-precision numbers to double-precision format by padding the significands with zeros. In the IEEE floating-point standard [1], the 'inexact' flag bit in the status word indicates whether the result would be exact if the operands were exact.

3.2 Modeling inexactness: the Monte Carlo method

In essence, the **Monte Carlo method** is to *model any inexact value with a random variable.* For our purposes, a random variable $\widetilde{x}$ that agrees with a value x to s digits is the **randomization**

$$\widetilde{x} \quad = \quad \mathsf{inexact}\,(x,\, s,\, \xi) \quad = \quad x \;+\; 2^{e+1-s}\,\xi$$

if we are using binary floating-point arithmetic, where e is the base 2 exponent of x. Here s is a real value (typically a positive integer), and ξ is a random variable (scaled random error). Typically ξ is uniform over the interval $(-\frac{1}{2}, \frac{1}{2})$, but other distributions may be useful. Depending on the context, we can also let ξ be discrete or continuous, or depend on x.

[1]This inexactness is often compounded during the course of a computation as inexact floating-point numbers take part in arithmetic operations that yield results that are even more inexact.

The Monte Carlo method is a natural way to model inexactness in floating-point arithmetic. Specifically, the inexactness caused by rounding to the limited precision of floating-point can be modeled by random rounding errors. Given a value x and a desired precision t, the randomization of x to t digits is implemented as

$$\mathsf{randomize}\,(x) \quad = \quad \begin{cases} x & \text{if } x \text{ is exact (within } t \text{ digits)} \\ \mathsf{inexact}\,(x,\,t,\,\xi) & \text{otherwise.} \end{cases}$$

If x is not exact within t digits, this superimposes a random perturbation so that the resulting significance is bounded by t digits.

3.3 Random Rounding

Random rounding is rounding of a randomized value:

$$\mathsf{random\_round}\,(x) \quad = \quad \mathsf{round}\,(\ \mathsf{randomize}\,(x)\ ).$$

Assume henceforth in the definition of $\mathsf{randomize}\,(x)$ we take the random variable ξ to be uniformly distributed over $(-\frac{1}{2}, \frac{1}{2})$.

When t equals the machine precision, $\mathsf{round}\,(\mathsf{randomize}\,(z))$ implements the same random rounding method proposed by other analysts, including Forsythe, Hull & Swensen, and others — see [10]. In Section 3.5 below we show that one can also randomly round to any 'virtual precision' $t < p$.

Theoretically, the advantage of this choice for the distribution is the absence of bias: the resulting rounding method has zero expected error. For a real value z, the expected value (average value) of $\mathsf{random\_round}\,(z)$ is $\mathsf{E}[\ \mathsf{random\_round}\,(z)\ ] \ = \ z$. Furthermore, if z is a random variable, then $\mathsf{E}[\ \mathsf{randomize}\,(z)\ ] \ = \ \mathsf{E}[\ z\ ]$ and $\mathsf{E}[\ \mathsf{round}\,(\mathsf{randomize}\,(z))\ ] \ = \ \mathsf{E}[\ z\ ]$. These equations are proven in [10] by Parker for ξ uniform over $(-\frac{1}{2}, \frac{1}{2})$, by showing equality of the corresponding integrals using the definitions above. We have also verified them repeatedly in practice, with programs like the one in Section 4.1.

In practice, of course, $\mathsf{randomize}\,(z)$ is implemented with a pseudorandom number generator [9]. Hardware implementations of these generators (based, for example, on shift-registers) are not difficult to incorporate into a floating-point unit.

3.4 Random Unrounding

A natural randomized arithmetic comes from using input randomization: if '$\circledbullet$' is the floating-point approximation to the '$\bullet$' operation with input randomization, then

$$x \ \circledbullet \ y \quad = \quad \mathsf{round}\,(\ \ \mathsf{randomize}\,(x) \ \bullet \ \mathsf{randomize}\,(y)\ \ ).$$

Input randomization can be viewed naturally as **random unrounding** [11], which is the random conversion of a (rounded-off, inexact) floating-point value to a real value. In [11], Pierce develops a detailed implementation that maintains 'real' values in registers with higher precision than in memory, and uses rounding and random unrounding — storing to and loading from memory — only when necessary. Effectively, Pierce proposes implementing $p \gg t$, where p is the machine floating-point precision in registers and t is the precision in memory. This aspect of his scheme is similar to the IEEE 754 standard's Double-Extended scheme, and he in fact proposed an implementation using IEEE Double-Extended precision as a basis.

3.5 Monte Carlo Arithmetic

Monte Carlo Arithmetic (MCA) is a family of floating-point systems that result when we use either random rounding or random unrounding (or both, or neither). Using neither is ordinary floating-point arithmetic. Using both is called **full MCA**.

If '$\odot$' is the floating-point approximation to the '$\bullet$' operation, then in full MCA

$$x \odot y \quad = \quad \mathsf{round}(\ \mathsf{randomize}(\ \mathsf{randomize}(x)\ \bullet\ \mathsf{randomize}(y)\ )\ ).$$

Although this definition requires real arithmetic in theory, it can be implemented efficiently with finite precision, of course. Random digits can be generated incrementally, as needed.

Full MCA achieves two important properties. First, random unrounding detects catastrophic cancellation, which is the primary way that significant digits are lost in numerical computation. Second, random rounding produces rounding errors that are independent and random, with zero expected bias. All errors are modeled with random variables, and the virtual precision t can be changed as needed.

An additional use of randomization is in varying the effective precision of computation (even dynamically, if that is desired). The **virtual precision** t is the precision to which arithmetic values are represented (in memory). If implemented in floating-point with register precision p, we require $t \le p$. By varying t in the definition of $\mathsf{randomize}(x)$ we implement arithmetic of any desired precision $t \le p$. The ability to vary the virtual precision can be quite useful (such as in evaluating the hardware precision requirements of a particular computation), so we allow t to differ from p.

3.6 Why Monte Carlo Arithmetic worked in the examples shown earlier

In full MCA, randomization is used in all arithmetic operations. All errors are modeled with random variables, and the virtual precision t can be changed as needed. Output randomization gives random rounding, producing rounding errors that are random and have zero expected bias.

In all the examples shown earlier, however, MCA works because input randomization detects catastrophic cancellation.

Catastrophic cancellation is a major source of inaccuracy in floating-point computation. This cancellation is the loss of leading significant digits caused by subtracting two approximately equal operands (i.e., two operands whose difference has a smaller exponent than either operand). This is shown in Figure 5; boxed values denote floating-point values.

operand 1	$\boxed{\texttt{+3.495683} \times 10^{0}}$
operand 2	$\boxed{\texttt{+3.495681} \times 10^{0}}$
difference	$\texttt{+0.000002} \times 10^{0}$
normalized difference	$\boxed{\texttt{+2.000000} \times 10^{-6}}$

Figure 5: Catastrophic cancellation

If one of the operands is inexact, then the difference computed in Figure 5 may have just one significant digit, yet there is no way to detect this in modern computers. Furthermore, if either one of the operands in the subtraction is inexact, the trailing zeroes introduced by normalization are no more significant than any other sequence of digits. Yet floating-point arithmetic lacks a mechanism for recording the loss of significance.

Catastrophic cancellation occurs in all of the examples discussed above, and also is often the primary problem in horrific examples of numeric inaccuracy found in the numerical analysis literature. For example, in the quadratic equation example earlier, catastrophic cancellation arises in the difference $(-b) - (\sqrt{b^2 - 4ac})$, since the two operands agree up to the seventh decimal digit.

Randomizing the trailing zero digits detects catastrophic cancellation. If subtraction loses ℓ leading digits, then ℓ trailing random digits will be in the result. This is illustrated in Figure 6. When computed multiple times, these randomized results will disagree on the trailing ℓ digits.

operand x (inexact)	$\boxed{\texttt{+3.495683} \times 10^0}$	
$x' =$ randomize (x)		+3.495683 *20391695941600884...*
operand y (inexact)	$\boxed{\texttt{+3.495681} \times 10^0}$	
$y' =$ randomize (y)		+3.495680 *91870795420835463...*
unnormalized difference $(x' - y')$		+0.000002 *28520900520765421...*
normalized result round $(x' - y')$	$\boxed{\texttt{+2.}\,285209 \times 10^{-6}}$	

Figure 6: Example of input randomization (in a difference with catastrophic cancellation) with eight-digit decimal arithmetic ($t = p = 8$, $\beta = 10$). Boxed values are inexact floating-point values.

3.7 An alternative

Random padding [11] is an alternative method, inspired by MCA, that focuses exclusively on the detection of catastrophic cancellation. Wherever conventional floating-point arithmetic would pad on the right with zeros, random padding would instead add an unbiased random real. The most important case is prenormal randomization, illustrated in Figure 7.

operand x	$\boxed{\texttt{+3.495683} \times 10^0}$
operand y	$\boxed{\texttt{+3.495681} \times 10^0}$
unnormalized difference $(x - y)$	+0.000002
random padded unnormalized difference $(x - y)'$	+0.000002 *31083217525704126...*
normalized result round $((x - y)')$	$\boxed{\texttt{+2.}\,310832 \times 10^{-6}}$

Figure 7: Random padding in a difference with catastrophic cancellation, with eight-digit decimal arithmetic ($t = p = 8$, $\beta = 10$). Boxed values are inexact floating-point values.

Both random padding and MCA with random unrounding (input randomization) detect cancellation equally well. However, random padding is much more difficult to analyze mathematically than MCA. Also, random padding is not a full implementation of the Monte Carlo method, and it does not transform floating-point into a Monte Carlo calculus (a statistical extension of floating-point that obeys additional laws of arithmetic). It also lacks the general strengths discussed in the next section.

4 Other Strengths of Monte Carlo Arithmetic

MCA is an extension of standard floating-point arithmetic. It is not a panacea. It is not a replacement for error analysis. It is a tool with certain strengths and weaknesses.

We have seen in the examples earlier that MCA has strength in allowing us to assess the number of significant digits in computed values. However, there are many methods that will do this: run in higher precision, compute condition numbers, use interval arithmetic, perform a forward roundoff analysis, etc. MCA does not replace any of these methods, but rather complements them.

Other strengths of MCA center around its making computer arithmetic more like real arithmetic. Randomization transforms roundoff from systemic error to random error, and random errors are much easier to deal with, both formally and informally. By transforming floating-point arithmetic into a Monte Carlo discipline we obtain many useful statistical properties.

4.1 Some floating-point anomalies are avoided

It is well-known that floating-point arithmetic is not associative. Knuth gives the following example for eight-digit decimal arithmetic [9, p.196]:

$$\begin{array}{rclcl}
(\ 11111113. \oplus -11111111.\ ) \oplus 7.5111111 & = & 2.0000000 \oplus 7.5111111 & = & 9.5111111; \\
11111113. \oplus (\ -11111111. \oplus 7.5111111\ ) & = & 11111113. \oplus -11111103. & = & 10.000000.
\end{array}$$

This anomaly is a direct manifestation of catastrophic cancellation. Associativity fails after the cancellation removes all but the least significant digit, which is affected by rounding errors.

	(11111113. ⊕ −11111111.) ⊕ 7.5111111			11111113. ⊕ (−11111111. ⊕ 7.5111111)		
n	$\widehat{\mu}$	$\pm$	$\widehat{\sigma}/\sqrt{n}$	$\widehat{\mu}$	$\pm$	$\widehat{\sigma}/\sqrt{n}$
10	9.62506	±	0.11484	9.40092	±	0.27888
100	9.49476	±	0.04241	9.42260	±	0.06533
1000	9.51095	±	0.01295	9.49816	±	0.02042
10000	9.50977	±	0.00411	9.51206	±	0.00645
100000	9.51014	±	0.00129	9.51396	±	0.00204
1000000	9.51093	±	0.00041	9.51159	±	0.00065
10000000	9.51112	±	0.00013	9.51111	±	0.00020

Table 6: Result of performing the indicated sums in single precision MCA with uniform random rounding and unrounding. Note the convergence to the exact sum value 9.5111111.

With MCA, these anomalies are avoided because random rounding has zero expected bias, as mentioned in Section 3.3. In particular, addition becomes associative. Table 6 gives a computational demonstration, showing that standard errors decrease with increasing numbers of samples. Different computed sums agree up to the standard error, and the average converges to the exact sum in the limit where the number of samples goes to infinity. Even for small values of n, the error in the average is bounded by a constant times the standard error, so we can say 'addition is associative up to the standard error of the sum'. With the right statistical caveats, we can thus formally prove that Monte Carlo arithmetic avoids certain floating-point anomalies. See [10] for more details.

Although it is possible to look at MCA as a way to get greater accuracy in some situations, it is an extremely inefficient way to do this. Assume the number of significant digits is measured by the negative log of the relative error, and relative error is estimated by the ratio of the standard error to the sample mean. This measure grows like $O(\log(n))$, where n is the number of samples. So, to get twice as many significant digits requires squaring the number of samples.

4.2 Variable precision is supported

Figure 8 gives the result of varying the virtual precision while computing the Tchebycheff polynomial $T_{20}(0.75)$, using the factored representation

$$T_{20}(z) \;=\; 1 \;+\; 8\,z^2\;(z-1)\;(z+1)\;\left(4\,z^2+2\,z-1\right)^2\;\left(4\,z^2-2\,z-1\right)^2\;\left(16\,z^4-20\,z^2+5\right)^2.$$

This computation involves only very mild cancellation and small constants.

As Figure 8 shows, increasing the virtual precision t to the single precision maximum of 24 has the desired effect of gradually increasing the accuracy of the result, modulo peculiarities in the binary representation of the intermediate results. Thus it is possible to get a qualitative sense of the accuracy of 15-bit or 10-bit computation, for example. Being able to explore accuracy/precision tradeoffs can be important in signal processing applications.

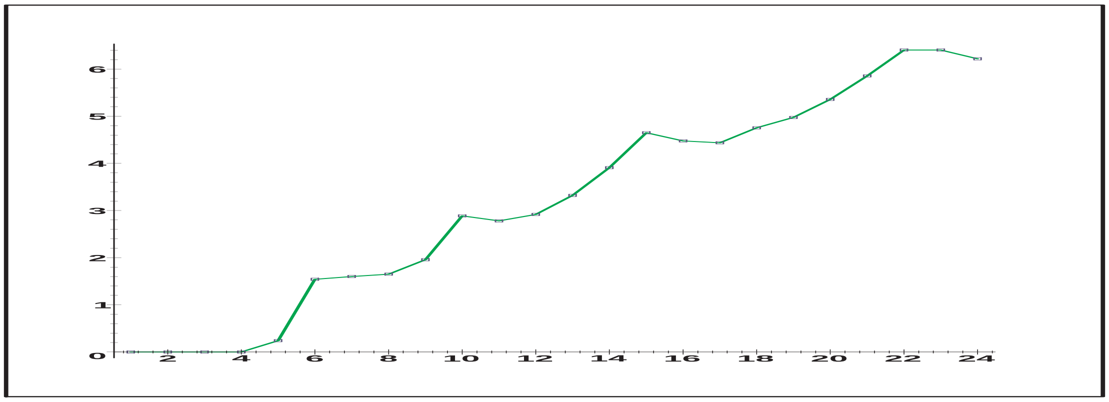

Figure 8: Number of significant decimal digits (negative base-10 log of the relative error) in the value $T_{20}(0.75)$ computed with full MCA (100 samples), at the indicated binary virtual precision t.

4.3 Roundoff errors actually do behave like random variables

Kahan [8] and others (e.g., Higham [6, §1.17,§2.6]) argue that statistical analyses of roundoff error are improperly founded because they assume rounding errors are random. With a plot resembling Figure 9, they observe that for certain values of x, a function like

$$rp(x) \;=\; \frac{622 \;-\; x\cdot(751-x\cdot(324-x\cdot(59-4\cdot x)))}{112 \;-\; x\cdot(151-x\cdot(72-x\cdot(14-x)))}$$

is sensitive to perturbation in ways that suggest its roundoff error is *not* randomly distributed. Figure 9 depicts some computed values of $rp(u+\delta)-rp(u)$, where u is the point at which $rp(x)$ achieves its maximum value. In the region depicted, this difference is approximately $-19\delta^2$. The computed values, however, do not even hint at this fact, and for some choices of δ are thousands of times greater in magnitude.

With MCA, randomization forces the rounding errors to be (pseudo)random. Randomization gives results like the equivalent plot, Figure 10, produced with full MCA (random unrounding and random rounding). Forsythe predicted random rounding could have this effect [4], but because this problem has considerable cancellation, random unrounding may give better results. Figure 11 also shows the distribution of these values. The computation of rp involves 16 arithmetic operations, and the resulting distribution is quite close to normal.

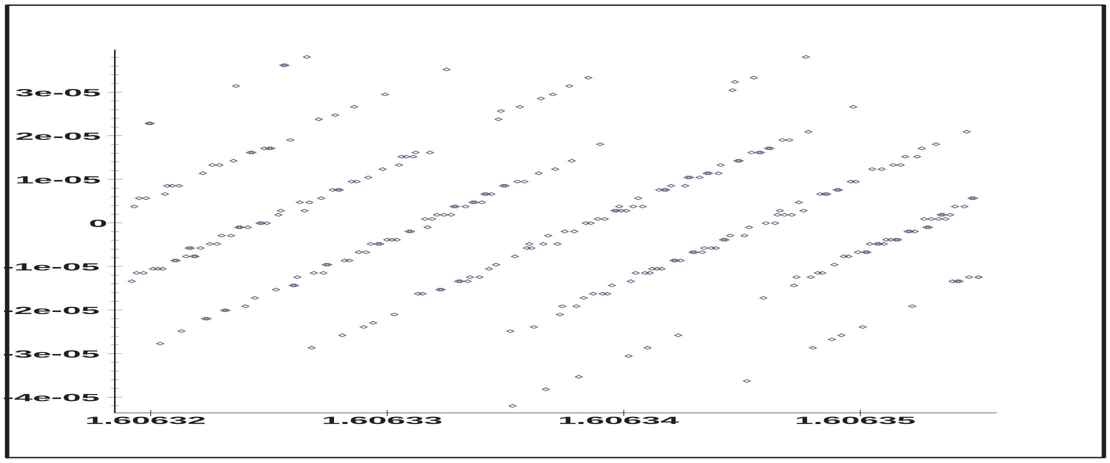

Figure 9: $rp(u+\delta) - rp(u)$, for $u = 1.60631924$, $\delta = 0, \epsilon, \ldots, 300\epsilon$ and $\epsilon = 2^{-23}$ ——— computed with single precision IEEE arithmetic.

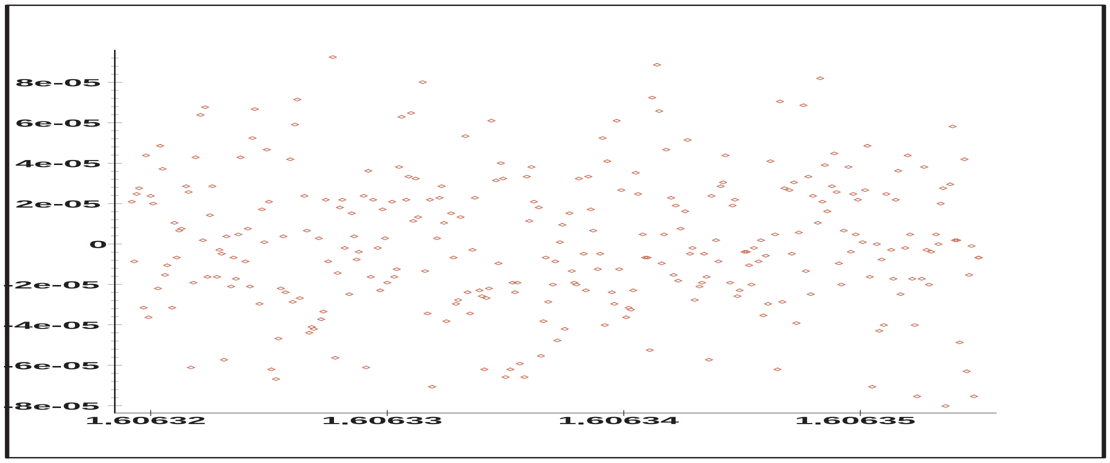

Figure 10: $rp(u+\delta) - rp(u)$, for $u = 1.60631924$, $\delta = 0, \epsilon, \ldots, 300\epsilon$ and $\epsilon = 2^{-23}$ ——— computed with single precision MCA (uniform random unrounding and random unrounding).

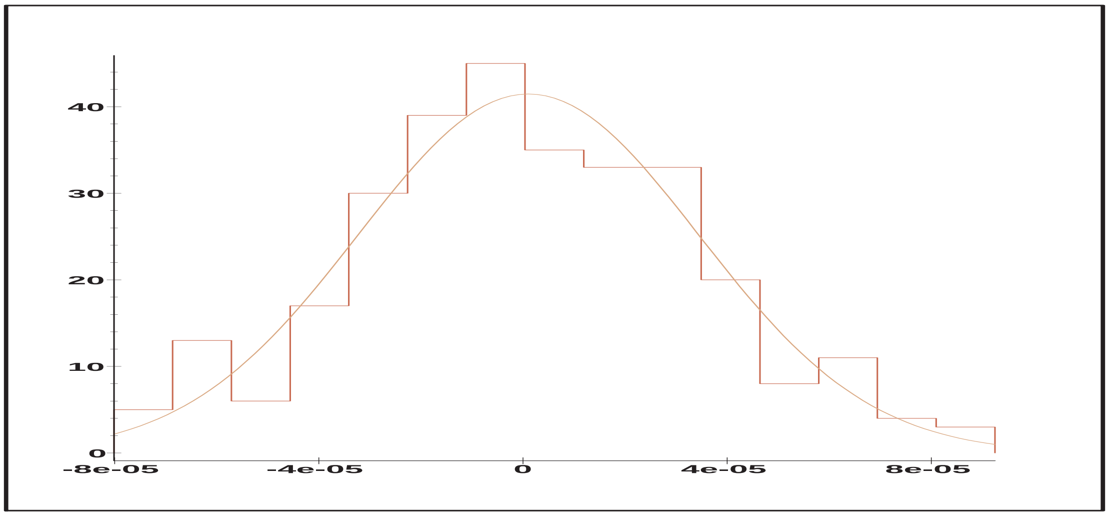

Figure 11: Histogram of values of $rp(u+\delta) - rp(u)$ computed with MCA in the previous figure. The normal density (for the average and standard deviation of these values) is superimposed.

5 Conclusion

We have argued that Monte Carlo Arithmetic (MCA) is a tool with interesting practical uses in numerical computations. We sketched how MCA can give useful results without imposing a great deal of overhead. Other examples and case studies are presented in [10].

Let us emphasize that *randomized floating-point is not always needed.* There are many useful algorithms that were carefully tuned with IEEE 754 floating-point arithmetic in mind [5, 8]. It makes no sense to inject randomness into these algorithms.

MCA is *an extension* to floating-point: by disabling randomization, we obtain conventional floating-point as a special case. One uses randomization only when it will help. Very much like the rounding modes in IEEE 754 [1], it makes sense to be able to turn randomization on or off — depending on the type of computation being performed.

MCA also does give new perspectives, and it often gives reasonable estimates of the accuracy of computed results. MCA thus appears to give an alternative way for a wide spectrum of numerical program users, without special training, to gauge the sensitivity of the output of a program to perturbations in its input and to rounding errors. Running a program multiple times with MCA yields a distribution of sample values. Statistical analysis of the distribution then can give a rough intuitive sense (and sometimes rigorous confidence intervals) about the roundoff error and the instability of the program (i.e., its sensitivity to rounding errors).

Specific other points that can be argued about MCA include:

- MCA is a simple way to bring some of the benefits of the Monte Carlo method to floating-point computation. The Monte Carlo method offers simplicity; it replaces exact computation with random sampling, and replaces exact analysis with statistical analysis.

- MCA is a probabilistic way to detect occurrences of catastrophic cancellation in numeric computations. This gives an optional *'idiot light'* for numeric computations, that can be used at fairly low cost, and can be used without changing existing programs. While large standard deviation values are not guaranteed to reflect numerical instability, or vice versa, they are a warning signal that strongly recommends further analysis.

- MCA gives a way to maintain information about the number of significant digits in conventional floating-point values. We believe many users can appreciate the significance perspective on error analysis, though they find backward analysis hard to understand.

- MCA can be used like any other rounding method, and thus it can be used without changing existing programs. Thus MCA should have wide applicability.

- MCA is a way to formally circumvent some anomalies of floating-point arithmetic. Although floating-point summation is not associative, for example, Monte Carlo summation is 'statistically associative' (i.e., associative up to the standard error of the result).

Probably the strongest argument for MCA is that it is a way to make numerical computing more *empirical*, and may encourage consumers of numerical software to investigate the quality of the results they are getting. We suspect that many certified numerical algorithms are giving inaccurate results in practice, mainly because they are being misused (being applied to ill-conditioned or stiff problems, or resting on assumptions that do not hold). An experimental attitude can definitely help in getting high-quality numerical results.

References

[1] American National Standards Institute, *ANSI/IEEE Std 754-1985: IEEE standard for binary floating-point arithmetic*, New York, 12 Aug. 1985. Also: "An American National Standard, IEEE standard for binary floating-point arithmetic", *SIGPLAN Notices*, **22**:2, 9–25, 1987.

[2] J.-C. Bajard, D. Michelucci, J.-M. Moreau, J.-M. Muller, Introduction to the Special Issue on "Real Numbers and Computers", *Journal of Universal Computer Science* **1**:7, page 438, Jul 28, 1995. Available on the internet at many sites.

[3] W.J. Cody, J.T. Coonen, D.M. Gay, K. Hanson, et al. "A proposed radix- and word-length-independent standard for floating-point arithmetic", *SIGNUM Newsletter* **20**:1, 37–51, 1985.

[4] G.E. Forsythe, "Round-off errors in numerical integration on automatic machinery. Preliminary report", *Bull. AMS* **56**, 61, 1950.

[5] D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* **23**:1, 5–48, March 1991.

[6] N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, Philadelphia: SIAM, 1996.

[7] W. Kahan, "The programming environment's contribution to program robustness," *SIGNUM Newsletter*, Oct. 1981, p.10.

[8] W. Kahan, "The Improbability of PROBABILISTIC ERROR ANALYSES for Numerical Computations", lecture notes prepared for the UC Berkeley Statistics Colloquium, 28 February 1996, and subsequently revised (4 March 1996). (An earlier version of this lecture was presented at the third ICIAM Congress, 3–7 July, 1995.) Currently available as: `http://http.cs.berkeley.edu/~wkahan/improber.ps`

[9] D.E. Knuth, *The Art of Computer Programming. Vol. II: Seminumerical Algorithms*, Addison-Wesley, 1969.

[10] D.S. Parker, "Monte Carlo Arithmetic: systematic random improvements upon floating-point arithmetic", UCLA Computer Science Department, Technical Report CSD-970002, February 1997. Available at `http://www.cs.ucla.edu/~stott/mca/`

[11] B.A. Pierce, *Applications of randomization to floating-point arithmetic and to linear systems solution*, Ph.D. dissertation, UCLA Computer Science Department, 1997.

[12] J.R. Taylor, *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*, Mill Valley, CA: University Science Books (Oxford University Press), 1982.