MONTE CARLO ARITHMETIC: HOW TO GAMBLE WITH FLOATING POINT AND WIN

How sensitive to rounding errors are the results generated from a particular code running on a particular machine applied to a particular input? Monte Carlo arithmetic illustrates the potential for tools to support new kinds of a posteriori round-off error analysis.

Monte Carlo arithmetic is a variant of floating-point arithmetic in which arithmetic operators and their operands are randomized. For example, rather than insist that floating-point addition obey $x \oplus y = fl[x + y] = (x \oplus y)(1 + \delta)$, where δ is some deterministically defined value, we allow δ to be a random variable. 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, $x + y$ can yield different values if evaluated multiple times.

With MCA, then, if a program runs with the same input n times, each run yields a slightly different answer. These answers constitute a sample distribution to which we can apply the whole array of standard statistical methods. Typically, after n such runs, the sample mean $\hat{\mu}$ estimates the exact solution, the sample standard deviation $\hat{\sigma}$ estimates the error in the answer of any single run, and the sample standard error $\hat{\sigma}/\sqrt{n}$ estimates the error in $\hat{\mu}$ 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.

MCA makes numerical computing empirical. It has practical applications—end users can gauge the number of significant digits in computed values as well as a program's stability. They can also use Monte Carlo methods to estimate round-off error accumulation. By treating individual rounding errors as random variables, we can use statistical methods to analyze errors. MCA also has theoretical applications. In fact, we can use MCA to circumvent certain well-known anomalies in floating-point operations.[1]

This article gives a short review of several years' investigation.[2] For a copy of our full work and a C demonstration program that runs all the examples in this article, visit www.cs.ucla.edu/~stott/mca. We assume the reader is familiar with basic issues in floating-point arithmetic.[2] David Goldberg's tutorial[3] and Nicholas Higham's encyclopedia[4] are great references.

1521-9615/00/$10.00 © 2000 IEEE

D. STOTT PARKER
University of California, Los Angeles
BRAD PIERCE
Cadence Design Systems
PAUL R. EGGERT
Twin Sun

Inexactness: What it is and why it's contagious

Exact values are real numbers that a given floating-point format can exactly represent. *Inexact values*, by contrast, are either real values that must be rounded (to an approximation) to fit this format, or real values that are not completely known. Rounding discards information; thus, it usually increases inexactness. (This inexactness is often compounded during a computation's course because inexact floating-point numbers take part in arithmetic operations that yield even more inexact results.) In scientific computation most quantities are inexact; we know constants such as the speed of light in a vacuum ($\sim 2.99792458 \times 10^8$ m/s) only to a few digits. This inexactness can come from 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:

- a value known to only a few significant digits (such as the speed of light),
- the result of rounding a higher-precision computation, or
- the result of an arithmetic operation on inexact values.

Because an inexact value could represent a whole range of possibilities, we can only guess which of these real numbers it actually represents. Conventional floating-point arithmetic 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,[5] the inexact flag bit in the status word indicates whether the result would be exact if the operands were exact.

Modeling inexactness: The Monte Carlo method

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

$$\tilde{x} = \text{inexact}(x, s, \xi) = x + 2^{e+1-s}\xi$$

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

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

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

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

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, $x \odot y = \text{round}(\text{randomize}(\text{randomize}(x) \bullet \text{randomize}(y)))$. Although this definition requires real arithmetic in theory, finite precision can efficiently implement it.

Full MCA achieves two important properties. First, random unrounding detects catastrophic cancellation, which is the primary way we lose significant digits 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 we can change the virtual precision t as needed.

An additional use of randomization is in varying the effective precision of computation (even dynamically). The virtual precision t is the precision to which arithmetic values are represented (in memory). If we implement in floating point with register precision p, we require $t \le p$. By varying t in the definition of 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 a particular computation's hardware-precision requirements), so we let t differ from p.

Table 1. The second root of $7x^2 - 8686x + 2 = 0$, computed with IEEE floating-point arithmetic.

Precision	Second root
IEEE single precision	.000279018
Exact solution (rounded)	.00023025562642454231

Table 2. The second root of $7x^2 - 8686x + 2 = 0$, computed with single-precision Monte Carlo arithmetic.

Run	Second root
1	.000198747
2	.000248582
3	.000251806
4	.000177380
5	.000203571
Sample mean	.000216017
Sample standard deviation	.000032739
Sample standard error	.000014641

Random rounding

Random rounding is rounding of a randomized value: random_round (x) = round (randomize(x)). When t equals the machine precision, round(randomize(x)) implements the same random-rounding method other analysts have proposed.[2]

Theoretically, the advantage of this choice for the distribution is the absence of bias: as we explained earlier, the resulting rounding method has zero expected error. For a real value x, the expected value (average value) of random_round(x) is E[random_round(x)] = x. Stott Parker showed for general x that E[randomize(x)] = E[x] and E[round(randomize(x))] = E[x] for ξ uniform over $(-\frac{1}{2}, \frac{1}{2})$, using elementary statistical analysis.[2] We have also verified these in practice.

In practice, of course, randomize(x) is implemented with a pseudorandom number generator.[1] Hardware implementations of these generators (based, for example, on shift registers) are not difficult to incorporate into a floating-point unit.

Random unrounding

A natural randomized arithmetic comes from using input randomization: if $\odot$ is the floating-point approximation to the $\bullet$ operation with input randomization, then $x \odot y$ = round(randomize(x) $\bullet$ randomize(y)).

Input randomization is random unrounding: the random conversion of a (rounded-off, inexact) floating-point value to a real value. Brad Pierce[6] developed 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 proposed implementing $p > t$, 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.

MCA overcomes important weaknesses

William Kahan[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 $7x^2 - 8686x + 2 = 0$. With `a` = 7, `b` = –8686, and `c`= 2, the C statement

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

yields Table 1, using IEEE floating-point arithmetic with default rounding.

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 are not randomized because they are exact—they're representable precisely in floating-point format.) Using the Gnu C Compiler version 2.7.2 with no options (except `-lm`) on a Sun SparcStation 20 running SunOS 5.5, we ran the program five times with single-precision MCA, which yielded Table 2. For simplicity, we used IEEE single-precision floating-point representation in our MCA implementation, but MCA will work in any precision.

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

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2}$$

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 arithmetic.

An estimate of the error in $\hat{\mu}$ is the sample standard error, $\sigma/\sqrt{n}$ which we approximate as $\hat{\sigma}/\sqrt{n}$. Just as the standard deviation of the samples $\hat{\sigma}$ gives an estimate of the error in any one sample, the standard error is the standard deviation of $\hat{\mu}$, and gives an estimate of the error in $\hat{\mu}$.

Using standard error is common in scientific work.[8] A widely used notation expressing the order of magnitude of error is "μ = sample mean ± sample standard error" ($\mu = \hat{\mu} \pm \hat{\sigma}/\sqrt{n}$). For example, "r = .00216017 ± .000014641." This notation gives one-standard-deviation estimates on the error in $\hat{\mu}$ and not bound. It merely says that the sample standard error is an order-of-magnitude estimate of the error in $\hat{\mu}$. This estimate is often more useful than a bound.

A startling example

Consider the following iteration.[9] Define the sequence (x_k) by

$$x_0 = 1.510005072139$$

$$x_{k+1} = \frac{(3x_k^4 - 20x_k^3 + 35x_k^2 - 24)}{(4x_k^3 - 30x_k^2 + 70x_k - 50)}.$$

As Table 3 illustrates, and depending on your machine's precision, the sequence converges to either 1, 2, 3, or 4. Actually, IEEE double precision converges to 3, and IEEE single precision converges to 2.

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

Precision (digits)	x_{30} (computed with decimal arithmetic)
30	3.00000000000000000000000000000
24	3.000000000000000000000000
20	3.00000000000000000000
16	3.000000000000000
12	1.99999999990
10	4.000000000
8	1.9999980
6	1.99990
4	2.097
2	1.0

With MCA, we got the values in Table 4. With four samples, again using uniform input and output randomization, the iteration's extremely unstable nature 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, we can obtain results other than 2 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.

There is no simple way to identify unstable iterations. However, huge variances among intermediate iterates should raise concerns about the iteration, and in this example, they do reflect instability.

Mild instability

For another example, consider the Tchebycheff polynomial James Wilkinson studied:

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

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

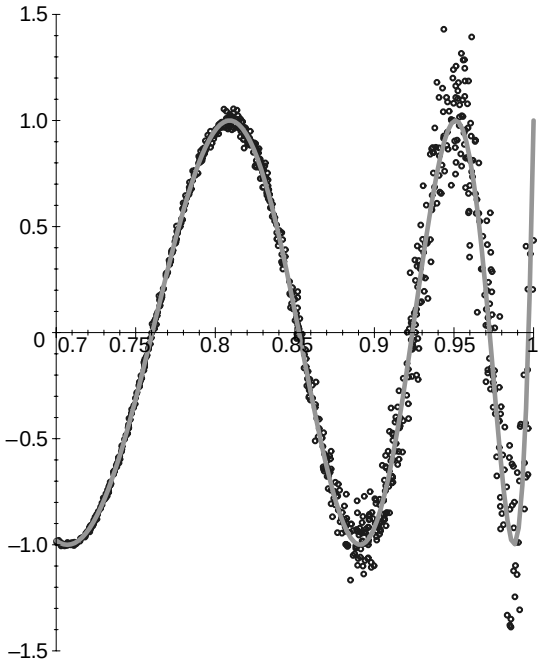

Figure 1. Monte Carlo arithmetic single-precision results performed at random points, exhibiting the Horner $T_{20}(z)$ computation's instability.

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

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 (the binary representation of 6553600 is 23 digits long). MCA and a scatterplot can help us visualize the increase in ill-conditioning when z nears 1 (see Figure 1).

Gaussian elimination, ill-conditioning, and significant digits

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 $\|\hat{\mathbf{x}}-\mathbf{x}\|/\|\mathbf{x}\|$ in the computed solution $\hat{\mathbf{x}}$ 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 round-off (bound on relative error due to rounding).[3] 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 MCA implementation, we ran Gaussian elimination with partial pivoting on the 10×10 inexact Hilbert matrix and a specific $\mathbf{b}$ vector. With five samples, we obtained the solution in Table 5.

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 $\|\hat{\mathbf{x}}-\mathbf{x}\|/\|\mathbf{x}\| \approx 10^{-3}$. This is often interpreted as predicting that $\hat{\mathbf{x}}$ will be accurate to about three digits. Table 5 bears out this prediction—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 ill-conditioning. For this problem, randomization appears to give a good empirical estimate of the actual number of significant digits in each entry of $\hat{\mathbf{x}}$. Moreover, the standard error's relative size measures algorithm stability (sensitivity to perturbation) and problem ill-conditioning.

Gastinel's Theorem tells us that the more ill-conditioned a nonsingular matrix A is, the

Table 5. Gaussian elimination solution vectors. Nonsignificant digits are italicized; underlined digits are within the magnitude of the standard error.

b	Exact solution to Ax = b	IEEE double solution	MCA double solution
1	2.4609375	2.460*8240*	2.460*8335*
2^{-1}	-216.5625	-216.5*526178*	-216.5*534088*
2^{-2}	4439.53125	4439.*3195461*	4439.*3359787*
2^{-3}	-36599.0625	-36597.*1294209*	-36597.*2756677*
2^{-4}	148507.734375	14849*8.4820907*	14849*9.1670631*
2^{-5}	-323760.9375	-32373*5.4324287*	-32373*7.2859033*
2^{-6}	387105.46875	3870*63.5276615*	3870*66.5268827*
2^{-7}	-239301.5625	-23926*0.9547177*	-23926*3.8179388*
2^{-8}	59825.390625	5980*4.0379124*	5980*5.5248358*
2^{-9}	0	*4.7021712*	*4.3783411*

```
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)
```

(a)

```
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) )
```

(b)

Figure 2. The (a) textbook algorithm and the (b) two-pass algorithm for computing standard deviations. Despite the textbook algorithm's enormous popularity, it's unstable.

smaller is the perturbation that would be needed to convert it into a singular matrix. If A were singular, $A\mathbf{x} = \mathbf{b}$ would have an infinite number of solutions. So it should not be surprising that even the tiny amounts of noise MCA introduces are amplified here into distributions with such large variances.

Often it is pointed out that $\hat{\mathbf{x}}$ is the exact solution of a perturbed problem $(A + E)\,\mathbf{x} = \mathbf{b}$, where E is a matrix of small errors ($\|E\| \approx \mathbf{u}\|A\|$).[10] Although backward-error results such as this give reassurance and show what floating-point arithmetic guarantees, it is important to recognize that they do not give guarantees about the number of significant digits in any particular computed value.

Standard deviations: Caveat user

Everyone knows the textbook algorithm for standard deviation shown in Figure 2. It is probably executed many billions of times every day, yet it's unstable. When the standard deviation is orders of magnitude smaller than the mean, its results might 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 approximately 36.8058.

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

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

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

$$\hat{\sigma} = \text{standard deviation } (\{1, \ldots, 127\}) = \sqrt{170,688/126} \approx 36.8058$$

the two algorithms give different results. Figure 3 shows how 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 algorithm choice.

Chaos: Your results will vary

Chaotic phenomena are often defined as those

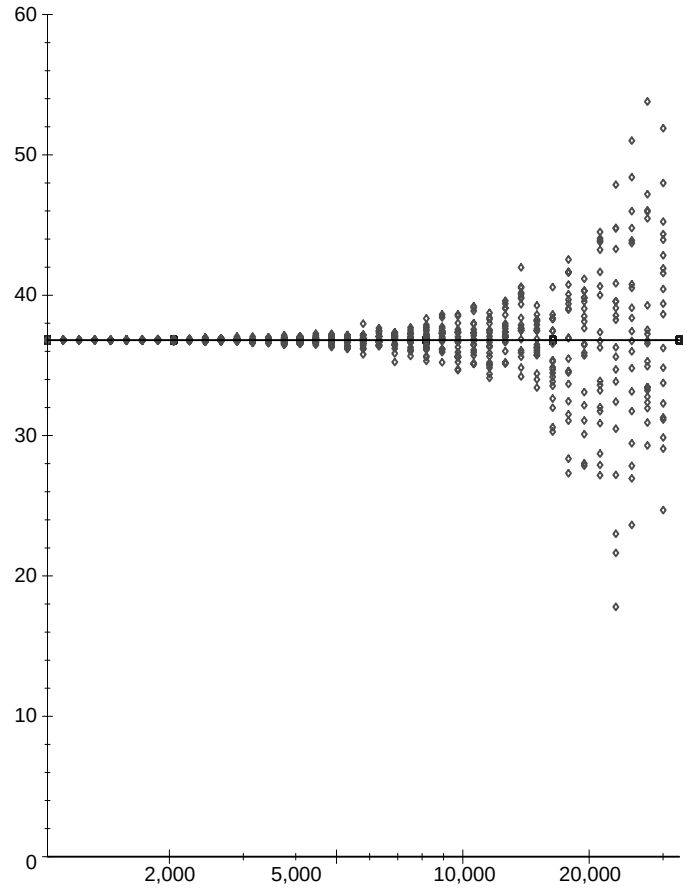

Figure 3. Computed `SIGMA` values for the standard deviation of {1 + *c*, 2 + *c*, . . ., 127 + *c*} using the textbook algorithm (scattered points) and the two-pass algorithm (the line at 36.8058), which gives the correct result. The horizontal axis gives the value of *c*, and for each *c* value chosen, 20 single-precision MCA samples are plotted.

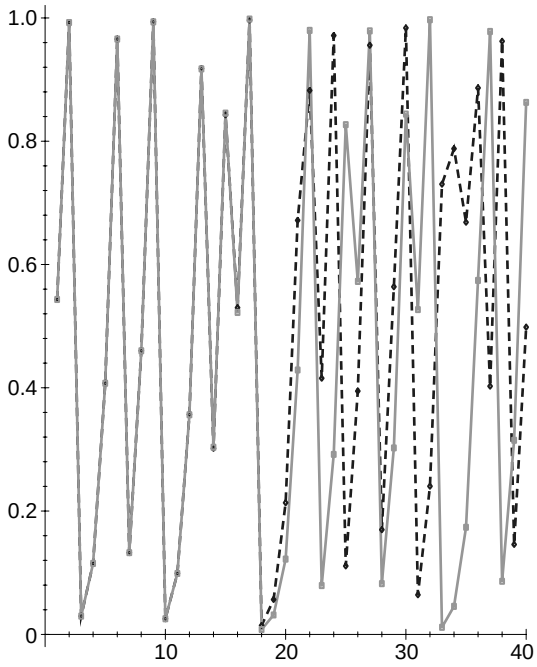

Figure 4. Two independent iterations of the chaotic logistic equation $x_{n+1} = 4x_n(1 - x_n)$ using single-precision MCA ($x_1 = 0.54321$). Rounding-error differences quickly lead to very different results.

that are "highly sensitive to initial conditions." Formally, this can be the equivalent of requiring their models to be numerically unstable. Nevertheless, tutorials about chaos often start with this definition and then proceed to report results of straightforward numeric computations.

For example, consider the two iterations of an instance of the logistic equation $x_{n+1} = 4\,x_n\,(1 - x_n)$, plotted in Figure 4. We used single-precision MCA, so different iterations obtain different pseudorandom 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 approximately 50 iterations.

Although 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 the integration of chaotic differential equations.[2]

Operand 1	$+3.495683 \times 10^{0}$
Operand 2	$+3.495681 \times 10^{0}$
Difference	$+0.000002 \times 10^{0}$
Normalized Difference	$+2.000000 \times 10^{-6}$

Figure 5. Catastrophic cancellation. Boxed values denote floating-point values.

MCA gives a way to detect high sensitivity to round-off in—and to apply statistical analysis to—results of chaotic computations. Many people seem to misunderstand the basic instability of chaotic computations, apparently believing that double precision is more than enough for accurate results. It's not.

Why MCA worked in these examples

Full MCA uses randomization in all arithmetic operations. 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, which 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—two operands whose difference has a smaller exponent than either operand (see Figure 5).

If one of the operands is inexact, then the difference computed in Figure 5 might have just one significant digit, yet there is no way to detect this in modern computers. Furthermore, if either 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.

Operand x (inexact)	$+3.495683 \times 10^{0}$	
x' = randomize (x)		$+3.495683\ 20391695941600884\ldots$
Operand y (inexact)	$+3.495681 \times 10^{0}$	
y' = randomize (y)		$+3.495680\ 91870795420835463\ldots$
Unnormalized difference ($x' - y'$)		$+0.000002\ 28520900520765421\ldots$
Normalized result round ($x' - y'$)	$+2.285209 \times 10^{-6}$	

Figure 6. An 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.

Operand x	$+3.495683 \times 10^0$
Operand y	$+3.495681 \times 10^0$
Unnormalized difference $(x - y)$	+0.000002
Random padded unnormalized difference $(x - y)'$	+0.000002 31083217525704126...
Normalized result round $((x - y)')$	$+2.310832 \times 10^{-6}$

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.

Catastrophic cancellation occurs in all the examples discussed earlier and 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

$$(-b) - (\sqrt{b^2 - 4ac})$$

because 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 (see Figure 6). When computed multiple times, these randomized results will disagree on the trailing ℓ digits.

An alternative

Random padding[5] is an alternative method inspired by MCA that focuses exclusively on detecting catastrophic cancellation. Wherever conventional floating-point arithmetic would pad on the right with zeros, random padding would instead add an unbiased random real value. The most important case is prenormal randomization (see Figure 7).

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.

MCA's other strengths

MCA strengths center around how it makes computer arithmetic more like real arithmetic. Randomization transforms round-off 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.

The statistical simulation of round-off error is only one way to look at MCA. Another possibility is to interpret it as an implementation of a nonstandard version of real arithmetic in which the operations are always already clouded by truly random noise. Might such a fundamentally finite-precision arithmetic lead to a mathematics that is more realistic than "real" arithmetic?

Anomalies

Floating-point arithmetic is well-known for not being associative. Donald Knuth gives this example for eight-digit decimal arithmetic:[1]

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

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.

Table 6 gives a computational demonstration of how MCA avoids these anomalies, 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 could say that addition would be associative up to the standard error of the sum. With the right statistical caveats, we can thus formally prove that MCA avoids certain floating-point anomalies.[2]

Although we can look at MCA as a way to get greater accuracy in some situations, it is an extremely inefficient method. Assume the negative log of the relative error measures the number of significant digits, and the ratio of the standard error

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.

	$(11111113 \oplus -11111111) \oplus 7.5111111$	$11111113 \oplus (-11111111 \oplus 7.5111111)$
n	$\hat{\mu} \pm \hat{\sigma}/\sqrt{n}$	$\hat{\mu} \pm \hat{\sigma}/\sqrt{n}$
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

to the sample mean estimates relative error. This measure grows as $O(\log(n))$, where n is the number of samples. So, to get twice as many significant digits requires squaring the number of samples.

Variable precision

Figure 8 gives the result of varying the virtual precision while computing the Tchebycheff polynomial $T_{20}(0.75)$, using the factored representation $T_{20}(z) = 1 + 8z^2 (z - 1) (z + 1) (4z^2 + 2z - 1)^2 (4z^2 - 2z - 1)^2 (16z^4 - 20z^2 + 5)^2$. 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 result's accuracy, modulo peculiarities in the binary representation of the intermediate results. Thus we can get a qualitative sense of the accuracy of 15-bit or 10-bit computation, for example. Being able to explore accuracy–precision trade-offs can prove important in signal-processing applications.

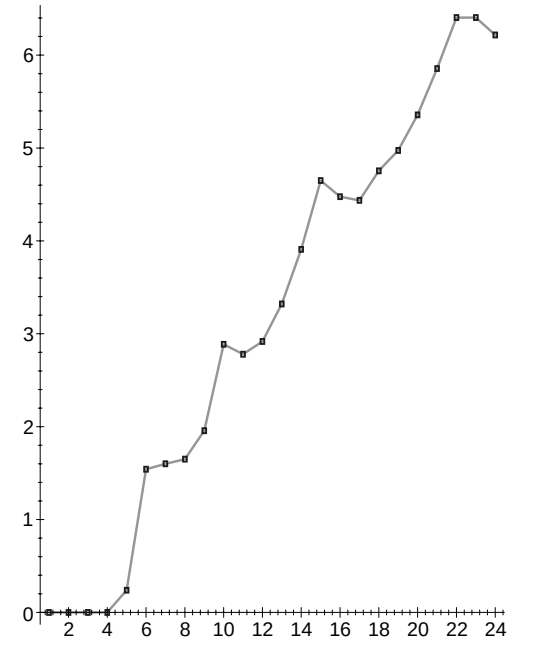

Figure 8. The number of significant decimal digits (negative base-10 log of the relative error) in the value $T_{20}(0.75)$ computed with full MCA (100 samples), at the indicated binary virtual precision t.

Round-off errors behave like random variables

Kahan[11] and others[4] argue that statistical analyses of round-off 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 such as

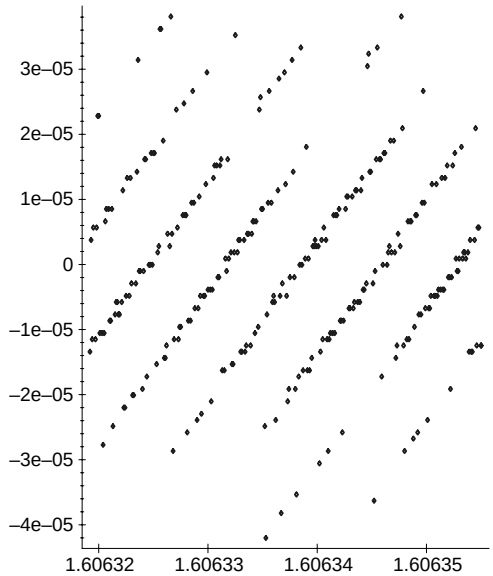

Figure 9. $rp(u + \delta) - rp(u)$, for $u = 1.60631924$, $\delta = 0, \epsilon, \ldots, 300\epsilon$, and $\epsilon = 2^{-23}$—computed with single-precision IEEE arithmetic.

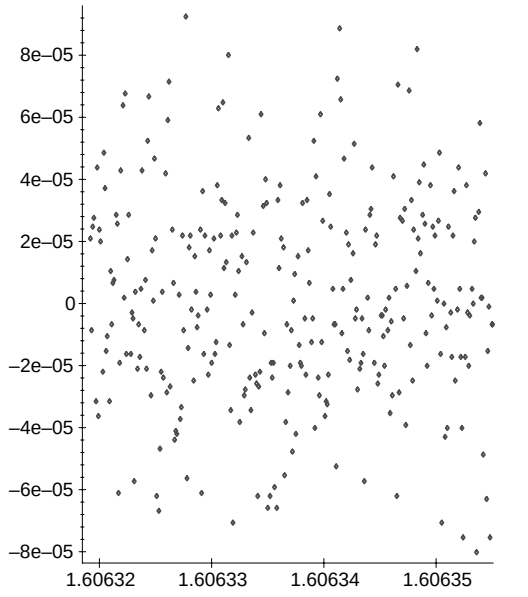

Figure 10. $rp(u + \delta) - rp(u)$, for $u = 1.60631924$; $\delta = 0, \epsilon, \ldots, 300\epsilon$, and $\epsilon = 2^{-23}$—computed with single-precision MCA (uniform random unrounding and random unrounding).

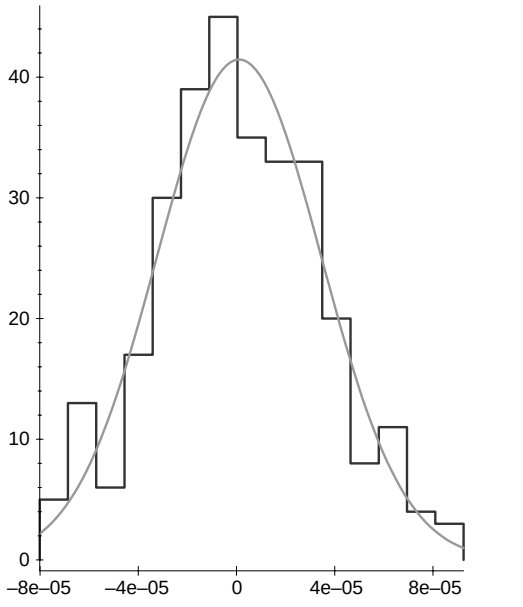

Figure 11. Histogram of values of $rp(u + \delta) - rp(u)$ computed with MCA in Figure 10. The normal density (for the average and standard deviation of these values) is superimposed.

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

is sensitive to perturbation in ways that suggest its round-off 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 are thousands of times greater in magnitude for some choices of δ.

With MCA, randomization forces the rounding errors to be pseudorandom. Randomization gives results such as the equivalent plot—Figure 10—produced with full MCA. George Forsythe predicted random rounding alone could have this effect,[12] but because this problem has considerable cancellation, random unrounding might 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.

Of course, Monte Carlo error analysis can give no ironclad guarantees that serious round-off error is absent in a computation. The practitioner must decide whether the level of sensitivity suggested by this simulation faithfully reflects reality. Randomized floating point is not always needed. There are many useful algorithms that other researchers carefully tuned with IEEE 754 floating-point arithmetic in mind.[3,11] Injecting randomness into these algorithms makes no sense.

We have seen in the earlier examples that MCA lets us assess the number of significant digits in computed values. However, many methods will do this: running in higher precision, computing condition numbers, using interval arithmetic, performing a forward round-off analysis, and so on. MCA does not replace any of these methods but rather complements them. Despite MCA's many advantages, it's not a panacea, and it is not a replacement for error analysis. It is a tool with certain strengths and weaknesses.

To sum up:

- 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, which we can use at fairly low cost and without changing existing programs. Although

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 this perspective on error analysis, although they find backward analysis hard to understand.
- MCA is a way to formally circumvent some anomalies of floating-point arithmetic. Although, for example, floating-point summation is not associative, Monte Carlo summation is statistically associative (up to the result's standard error).

Because MCA is an extension to floating-point arithmetic, we can disable randomization and obtain conventional floating point as a special case. You use randomization only when it will help. Very much like the rounding modes in IEEE 754,[5] being able to turn randomization on or off—depending on the type of computation being performed—makes sense.

MCA gives new perspectives, and it often gives reasonable estimates of a computed result's accuracy. MCA thus appears to give an alternative way for a wide spectrum of numerical program users, without special training, to gauge a program output's sensitivity 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 can then give a rough intuitive sense (and sometimes rigorous confidence intervals) about the round-off error and the program's instability—its sensitivity to rounding errors.

MCA might encourage numerical software consumers 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 (they're being applied to ill-conditioned or stiff problems or resting on assumptions that do not hold). An experimental attitude can definitely help get high-quality numerical results. CiSE

References

1. D.E. Knuth, *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*, Addison-Wesley, New York, 1969.
2. D.S. Parker, *Monte Carlo Arithmetic: Systematic Random Improvements upon Floating-Point Arithmetic*, Tech. Report CSD-970002, Computer Science Dept., Univ. of California, Los Angeles, 1997.
3. D. Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, Mar. 1991, pp. 5–48; www.validgh.com/goldberg/paper.ps (current June 2000).
4. N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM Press, Philadelphia, 1996.
5. *ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic*, American Nat'l Standards Inst., New York, 1985.
6. B.A. Pierce, *Applications of Randomization to Floating-Point Arithmetic and to Linear Systems Solution*, PhD diss., Computer Science Dept., Univ. of California, Los Angeles, 1997.
7. W. Kahan, "The Programming Environment's Contribution to Program Robustness," *SIGNUM Newsletter*, Vol. 16, No. 4, Oct. 1981, p.10.
8. J.R. Taylor, *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*, *2nd Ed.*, Univ. Science Books, Mill Valley, Calif., 2000.
9. J.-C. Bajard et al., Introduction to the Special Issue on Real Numbers and Computers, *J. Universal Computer Science*, Vol. 1, No. 7, July 1995, p. 438.
10. G.H. Golub and C.F. Van Loan, *Matrix Computations: 2nd Ed.*, Johns Hopkins Univ. Press, Baltimore, 1989.
11. W. Kahan, "The Improbability of Probabilistic Error Analyses for Numerical Computations," UC Berkeley Statistics Colloquium, 28 Feb. 1996; www.cs.berkeley.edu/~wkahan/improber.ps (current May 2000).
12. G.E. Forsythe, "Round-Off Errors in Numerical Integration on Automatic Machinery: Preliminary Report," *Bull. AMS*, Vol. 56, 1950, p. 61.

D. Stott Parker is a professor of computer science at the University of California, Los Angeles. His interests involve developing more effective information models, and computer systems that manage them for data mining, decision support, and scientific data management. Currently, he is investigating ways to exploit randomization in numeric algorithms. He received his MS and PhD from the University of Illinois. He is a member of the IEEE and the ACM. Contact him at stott@cs.ucla.edu.

Brad Pierce is a senior member of the technical staff at Cadence Design Systems, where he builds software tools that optimize deep submicron interconnection networks. He received his BS in mathematical sciences and MS in computer science from Indiana University and his PhD in computer science from UCLA. He is a member of the ACM and the IEEE Computer Society. Contact him at bpierce@cadence.com.

Paul R. Eggert is CTO of Twin Sun, Inc., a strategic technical service provider for the Japanese Internet market. He directs software research and development with projects ranging from financial trading simulations to internationalized collaboration technology. He has also contributed to several widely used free software packages. He received his BA in electrical engineering and mathematical sciences from Rice University, and his MS and PhD in computer science from UCLA. He is a member of the IEEE Computer Society, the ACM, and the American Association for the Advancement of Science. Contact him at eggert@twinsun.com.