Can you trust your computer?

Khalil Kalbasi

Floating-point formats, round-off, and machine constants—they could be making liars out of your data

Computers are a part of daily life in areas from banking, game playing, reservation systems, traffic control and business to science and engineering. Their use, in the latter, requires much more diligence and care than is usually extended, especially when performing many arithmetic operations, such as solving differential equations or a system of algebraic equations. In such instances, the computer, which is perceived as an "exact" tool, can turn out to be a monster number cruncher that delivers nothing but "garbage." This is due to the way computers handle real numbers.

For example, consider the determination of the following sum.

$$10^{48} + 914 - 10^{48} + 10^{32} + 615 - 10^{32} = 1529.$$

By summing from left to right, most digital computers will return zero as the answer. This gross error is the result of floating-point formats in these computers. What can be done about these problems? Are they machine dependent? If so, what are the characteristics of different computers, and if not, what algorithms would eliminate or reduce these errors?

Real vs. floating point

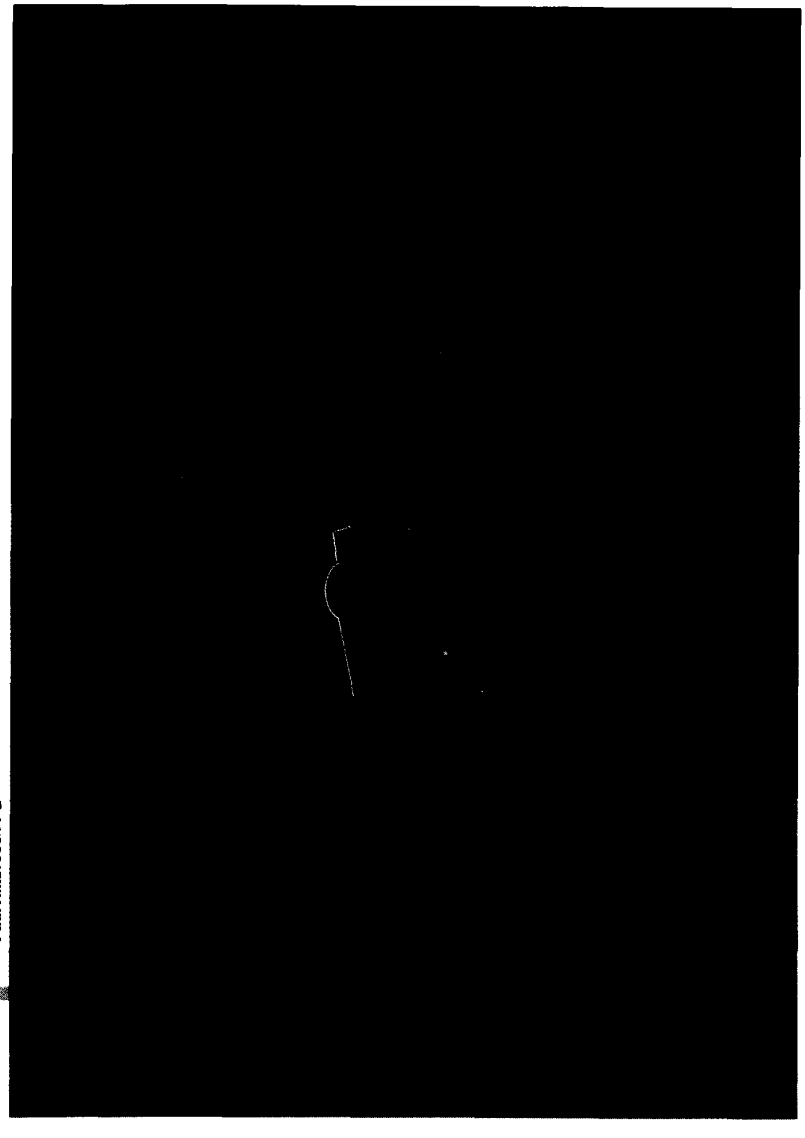
The *real number system* is one of the triumphs of the human mind, underlying calculus and higher analysis. In spite of the infinite span of the real number system, computers deal with a finite numbering system called the *floating-point number system*. A floating-point number system consists of a finite number of elements having the appearance of a screen placed over the real numbers. Indeed, the expression floating-point screen is sometimes used. Thus for an approximation of the real numbers on a computer, floating-point numbers are used.

A t-digit base b floating-point number is one of the form

$$\pm .d_1 d_2 d_3 \ldots . d_t \, b^e.$$

Here $.d_1 d_2 d_3 \ldots . d_t$ is the mantissa, b is the base of the number system in use (an integer greater than unity) and e is the exponent. The exponent is an integer between two fixed integer bounds $e1$, $e2$ and, in general, $e1 \leq 0 \leq e2$. With the condition $d_1 = 0$, floating-point numbers are said to be normalized. The set of normalized floating-point numbers does not contain zero. (For a unique representation of zero, we assume a positive (+) sign, a mantissa of 0.0000 . . . 0 (t zeros after the radix point) and $e = e1$.) Therefore, a floating-point system depends on the constants b, t, $e1$ and $e2$. Let's denote it by $R = R(b, t, e1, e2)$. The base b is usually 2 and the length t of the mantissa is typically 24 bits for single precision arithmetic.

A floating-point system R consists of a finite number of elements spaced between successive powers of base b and their negatives. It has exactly $2(b - 1)b^{t-1}(e2 - e1 + 1) + 1$ numbers in it. Figure 1 shows a simple floating-point system $R = R(2, 3, -1, 2)$ consisting of 33 elements. These are



Paul Ambrose/FPG

0278-6648/90/0004-0015$01.00 ©1990 IEEE

not equally spaced throughout their range, but only between successive powers of b and their negatives.

Rounding and chopping, over- and under-flow

Because of the structure of floating-point numbers, the arithmetic operations for real numbers are only approximated by floating-point numbers. If x and y are floating-point numbers, the exact point $x \times y$ will have more than t digits of mantissa and therefore the least significant digits in excess of t are either "chopped" or "rounded," leading to error. The same is true for exact sum of $x + y$. Since a computer has to represent the results of its own operations as a floating-point number, it rounds (or chops) the exact result into a floating-point number screen and takes the outcome as the definition of floating-point operation.

We say that a number x is *chopped* to t digits or figures when all the digits following the tth digit are discarded and none of the remaining t digits is changed. Conversely, x is *rounded* to t digits or figures when x is replaced by a t-digit number that approximates x with minimum error. The question of rounding up or down a $(t + 1)$-digit decimal number that ends with a 5 is best handled by always selecting the rounded t-digit number with *even* tth digit.

The relative error of representing a number in floating-point format is at most equal to b^{1-t} and $\frac{1}{2}b^{1-t}$ for chopping and rounding, respectively. This implies that if you have a relative error of say 10^{-7} in calculating a number, you expect to have about 6 digits of accuracy.

Let $fl(x)$ represent the floating-point representation of a real number x. Furthermore, let $fl(x + y)$ represent the floating-point addition of two real numbers x and y. In the 33-point system described above, let $x = 5/4$ and $y = 3/8$. The operation $x + y$ on a computer is simulated via a *floating-point addition* approximation. We would like $fl(x + y)$ to be closest to the true $x + y$, but in most computers this does not happen. For instance, in our 33-point floating point number system the result of $fl(5/4 + 3/8)$ would be either $3/2$ or $7/4$ instead of $13/8$. The difference between $x + y$ and $fl(x + y)$ is called *rounding error* in addition. When the result of some floating-point operation is greater than the largest floating-point number available in a given floating-point system ($7/2$ in our toy system), the phenomenon of *overflow* occurs. *Underflow* happens in floating-point multiplication when two non-zero numbers have a non-zero product that is smaller in magnitude than the smallest number in R (i.e., $1/4$ in our 33-point system). When overflow occurs, almost all computers signal an error message, but underflow happens unnoticed in some computers because the machine quietly sets the number equal to zero. How great is the relative error when this happens?

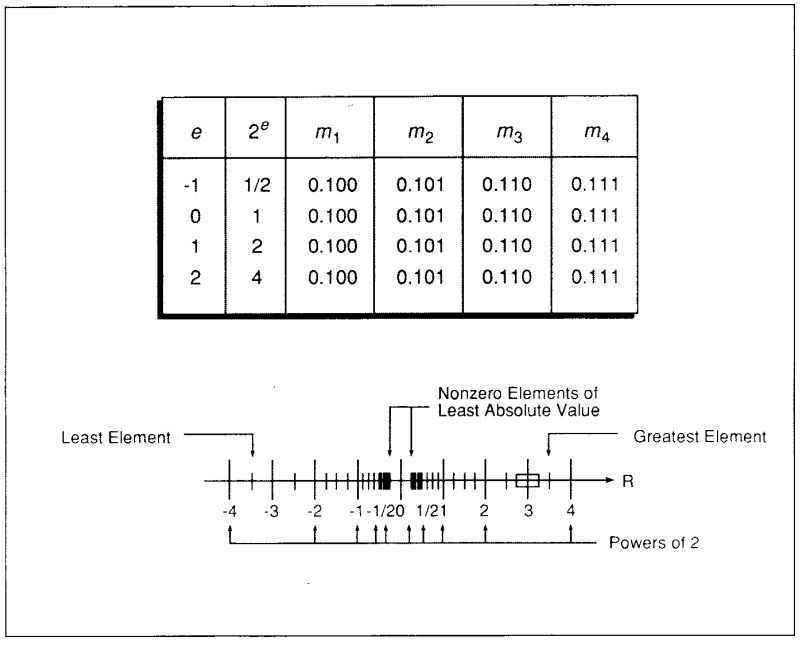


Fig. 1. A simple floating-point system.

An example

To show the implications of round-off in a very common type of calculation, let's look at an example. The following Fortran and Pascal Programs are roughly equivalent. Predict how many lines of output you would expect from running one of them on a computer that uses binary, rounded arithmetic.

```
program demo(output);
var
  i : integer
  h : real;
  x : real;
begin
  i := 0;
  x := 0.0;
  h := 0.1;
  repeat
    i := i + 1;
    x := x + h
    writeln(i , x)
  until x<1;
end.
```

```
integer i
real h
real x
i = 0
x = 0.0
h = 0.1
10  continue
    i = i + 1
    x = x + h
    print* i , x
if(x .le. 1.0)
go to 10
end
```

If your answer to this question is "11 lines" run the program and see for yourself. Now, if the statement x = x + h is replaced with x = i * h, are there going to be any changes in the printed output? If your answer is "no," try the program with the modification again and see what happens. If you could not see what is happening exactly, here is an explanation: Using exact arithmetic, the program should print eleven lines in either case. But using 24-digit base 2 arithmetic with rounding (such as single precision VAX arithmetic), the program prints only 10 lines when the x=x+h version is used. The sequence of values attained by x (in double-precision format) is

i	x
1	0.1000000014901161
2	0.2000000029802322
3	0.3000000119209290
4	0.4000000059604645
5	0.5000000000000000
6	0.6000000238418579
7	0.7000000476837158
8	0.8000000715255737
9	0.9000000953674316
10	1.000000119209290

Why did this happen? The fraction 1/10 cannot be represented exactly by 24 binary digits. The nearest 24-binary-digit number is slightly larger than 1/10. Observe the direction of the rounding errors. In particular, when i=10, x is slightly greater than 1, and the loop stops.

With the modification x=i*h, the results are as follows:

i	x
1	0.1000000014901161
2	0.2000000029802322
3	0.3000000119209290
4	0.4000000059604645
5	0.5000000000000000
6	0.6000000238418579
7	0.6999999880790710
8	0.8000000119209290
9	0.9000000357627869
10	1.000000000000000
11	1.100000023841858

The closest floating-point number to $fl(10 \times 0.10000000149012)$ is 1, so the loop runs eleven times. The moral of the story is that round-off error could alter the results completely without any "apparent" reason and that care must be taken to avoid pitfalls like this.

Machine constants

Table 1 shows that round-off depends on the floating-point format of your computer. Therefore, one needs to know the machine constants of his or her favorite computer before beginning to do serious work on it. By machine constants we mean the set of numbers related to the floating-point format described in Table 1. Obviously, different computer hardware and architecture implementations would result in different machine constants. These constants present themselves during the course of computation in one or more ways. For instance, in order to quantify the round-off errors in the computation, we need a round-off unit. This unit, which is called *machine epsilon* or *machine precision* (ϵ), is, roughly speaking, the fractional accuracy to which floating-point numbers are represented and corresponds to a change of one in the least significant bit of the mantissa. In other words, machine epsilon is a positive floating-point number for which

$$fl(1 + \epsilon) \geq 1.$$

Note that ϵ is not the smallest floating-point number on a given machine. That number depends on the number of bits in the exponent, while ϵ depends on the number of bits in the mantissa.

Table 1 depicts the more important machine constants of some common computers both for single- and double-precision arithmetic. Note that extended precision is not implemented on hardware in these machines and is normally done via an internal algorithm, thereby requiring much more CPU usage.

If you need to know some of the machine constants of your favorite computer (without having to read the manuals), a set of programs on the *Netlib* public library are very helpful. In particular, the programs I1MACH, R1MACH and D1MACH are very handy. These programs will give you the integer, real and double-precision constants of different computers. These constants represent values checked by extensive testing and are not naive counts of "how many bits in a word." Because some machines use extended precision registers, trying to calculate such quantities directly is problematical. Here is the list of constants that can be obtained by running these codes for integer and floating-point (single- and double-precision) arithmetic:

- The number of bits per integer storage unit;
- The number of characters per character storage unit;
- The base for integer and floating-point numbers;
- The number of digits in the integer and floating-point base;
- The largest magnitude integer and floating-point number;
- The smallest and largest exponent (single- and double-precision);
- The smallest positive magnitude number (single- and double-precision);
- The smallest relative spacing (single- and double-precision);
- The largest relative spacing (single- and double-precision).

To get a copy of these codes, and the list of computers you can run them on, use the *Machine* or *Core* library from *Netlib*. Instructions on how to use *Netlib* are in the February 1989 issue of *Potentials*.

Computer	R/C	β	t	L	U	e
CDC CYBER 170	R	2	48	-976	1071	3.55×10^{-15}
CDC CYBER 205	C	2	47	-28,626	28,718	1.42×10^{-14}
Cray-1	C	2	48	-8,192	8,191	7.11×10^{-15}
DEC VAX (single)	R	2	24	-127	127	5.96×10^{-8}
DEC VAX (double)	R	2	56	-1,023	1,023	1.11×10^{-16}
HP-11C,15C	R	10	10	-99	99	5.00×10^{-10}
IBM 3033 (single)	C	16	6	-64	63	9.54×10^{-7}
IBM 3033 (double)	C	16	14	-64	63	2.22×10^{-16}
IBM/PC (single)	R	2	24	-126	127	5.96×10^{-8}
IBM/PC (double)	R	2	53	-1,022	1023	1.11×10^{-16}
PRIME 850 (single)	C	2	23	-128	127	2.38×10^{-7}
PRIME 850 (double)	C	2	47	-32,896	32,639	1.42×10^{-14}

Table 1

If you are not interested in all those constants and just need to know an approximate value for your machine epsilon, a quick and efficient way is to write a couple of lines of code. It is usually sufficient to know ϵ within a factor of two. The following short program in FORTRAN will do the job:

```
      implicit double precision
        (a–h,o–z)
      eps = 1.0
   10 continue
         eps = eps/2.0
         t = 1.0 + eps
         if (t .gt. 1.0) go to 10
         eps = 2.0 * eps
         print *, eps,' .le. machine
           epsilon .le. ',2.0 * eps
      end
```

The value of ϵ will become handy, especially when you want to terminate an iteration. Suppose you are finding the root of a nonlinear equation using the Newton method. You want the value you get to be as close as possible to the actual value, but at the same time you do not want the iteration process to continue forever by expecting the computer to give you absolutely no error. Therefore, you need a stopping criterion. This is when you need the value of ϵ. You use this value of ϵ to compare your error bounds or termination criterion.

Error propagation and analysis

An optimistic value for the round-off accumulation in performing N arithmetic operations is roughly $\sqrt{N}\epsilon$ where the square root is for the random walk. In many instances, the round-off error could grow to $N\epsilon$ or even more. For instance, when subtracting two large numbers that are nearly equal, the result depends only on the few less significant digits in which the two numbers differed. Thus, once the result is normalized, the rest of the digits in the mantissa other than those few ones are lost. This phenomenon is called *subtractive cancellation* and occurs every now and then, introducing considerable round-off error. For example, in 4-digit base 10 arithmetic, $fl[(10000 + 1) - 10000] = 0$ but $(10000 + 1) - (10000) = 1$ (exactly). The finite precision result has no correct significant digit. Subtractive cancellation happens when we attempt to compute the relatively small quantity 1 by subtracting the relatively large numbers 10001 and 10000.

The introduction of floating-point when combined with the enormous gains in speed of computers mandates methods of controlling the round-off errors. Traditionally, two techniques of error analysis called *forward error analysis* and *backward error analysis* have been used. In forward error analysis, the floating-point representation of error is subjected to the same mathematical operations as the data themselves, resulting in an equation for the accumulated error. In backward error analysis, attempts are made to regenerate the original mathematical problem from the computed solutions. Both of these methods are analytic, and, for large scale computations, a large number of error estimates will be needed in addition to their propagation throughout the computation process. For instance, in multiplying two complex matrices of order 100, about 8 million of such estimates are required. It is clear that one prefers to avoid such analysis. Instead, one should adopt one of the following techniques:

1. Compute a residual; i.e., feed the computed solution into the original problem and evaluate the remainder. A small remainder *usually* indicates a good solution. This process is sometimes referred to as *defect correction*.
2. Repeat the calculation in double or extended precision to verify a good agreement. In fact, avoid single precision arithmetic if you can; because, as my numerical analysis instructor used to say, "Single precision computing is only good for writing checks!"
3. Rerun your problem with slightly perturbed input data and notice the change in the results. *Usually*, small variations in the results indicate stability in the computational process.

One can dream of examples that show the above methods to be completely unreliable, but they frequently are a good indication of the quality of your computations.

New ideas for computer arithmetic

With the increased capability of computers and enormous growth and ramification of the problems that are dealt with in scientific computations, there remains no alternative but to furnish the computer with the capability of control and validation of the computational process. Recently developed concepts and methods of floating-point arithmetic provide a superior capability for modern digital computers with far-reaching consequences for scientific computation. For example, they go a long way toward eliminating errors of the type described in this article. There are also nonfloating-point arithmetic implementations for eliminating errors in scientific computation. Examples of these are rational arithmetic, the use of multiple and the full precision arithmetic found in such systems as SCRATCHPAD and MACSYMA.

The basic feature of advanced computer arithmetic is to augment the four basic operations $+, -, \times, /$ for floating-point numbers by another operation referred to as the scalar or dot product of two vectors. The new scalar product must be implemented with only one rounding. The augmented set of five basic operations is sufficient for the execution with maximum accuracy. The availability of exact scalar products, as well as matrix and matrix-vector operations with maximum accuracy, combined with defect correction and interval arithmetic concepts make it possible to verify or validate a given computational process.

Conclusions

Understanding the limitations of computers is the key step in their utilization. Although progress is being made to reduce the limitations of floating-point formats of modern computers, we are still a long way away from eliminating the negative effects of round-off in extensive computations. Therefore, the user is left with no alternative but to exercise care. This involves efficient algorithm and programming, verification of results, and keeping an eye on the specifications of the computers being used.

Read more about it

New York, 1981.

- Jack Dongarra and Iain Duff, "Advanced Computer Architectures," Argonne National Laboratory, ANL/MCS-TM-57, (Revision 1), January 1987.
- U. W. Kulisch and L. Miranker, "The Arithmetic of the Digital Computer: A New Approach," *SIAM Review* Vol. 28, No. 1, March 1986.
- G. E. Forsythe, "Pitfalls in Computation, or Why a Math Book Isn't Enough," Computer Science Department, Stanford Univ., Stanford, CA, 1970.

About the author

K. Kalbasi is a Ph.D. student at the Electrical and Computer Engineering Department of the University of Kansas, Lawrence, and a Graduate Research Assistant at the Radar Systems & Remote Sensing Laboratory (RSL) of the Center for Research Inc. His research interests include computational electromagnetics, digital signal processing, and radar systems. □