Floating points

IEEE Standard unifies arithmetic model

by Cleve Moler

If you look carefully at the definition of fundamental arithmetic operations like addition and multiplication, you soon encounter the mathematical abstraction known as the *real numbers*. But actual computation with real numbers is not very practical because it involves limits and infinities. Instead, MATLAB and most other technical computing environments use *floating-point* arithmetic, which involves a finite set of numbers with finite precision. This leads to phenomena like *roundoff error*, *underflow*, and *overflow*. Most of the time, MATLAB can be effectively used without worrying about these details, but every once in a while, it pays to know something about the properties and limitations of floating-point numbers.

Twenty years ago, the situation was far more complicated than it is today. Each computer had its own floating-point number system. Some were binary; some were decimal. There was even a Russian computer that used trinary arithmetic. Among the binary computers, some used 2 as the base; others used 8 or 16. And everybody had a different precision.

In 1985, the IEEE Standards Board and the American National Standards Institute adopted the ANSI/IEEE Standard 754-1985 for Binary Floating-Point Arithmetic. This was the culmination of almost a decade of work by a 92-person working group of mathematicians, computer scientists and engineers from universities, computer manufacturers, and microprocessor companies.

All computers designed in the last 15 or so years use IEEE floating-point arithmetic. This doesn't mean that they all get exactly the same results, because there is some flexibility within the standard. But it does mean that we now have a machine-independent model of how floating-point arithmetic behaves. MATLAB uses the IEEE *double precision* format. There is also a single precision format which saves space but isn't much faster on modern machines. And, there is an extended precision format, which is optional and therefore is one of the reasons for lack of uniformity among different machines.

Most floating point numbers are *normalized*. This means they can be expressed as

$$x = \pm (1 + f) \cdot 2^e$$

where f is the fraction or mantissa and e is the exponent. The fraction must satisfy

$$0 \le f < 1$$

and must be representable in binary using at most 52 bits. In other words, $2^{52}f$ must be an integer in the interval

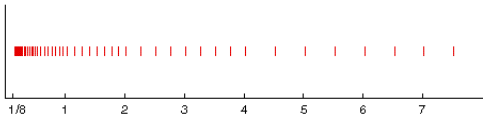
$$0 \le 2^{52}f < 2^{53}$$

The exponent must be an integer in the interval

$$-1022 \le e \le 1023$$

The finiteness of f is a limitation on *precision*. The finiteness of e is a limitation on *range*. Any numbers that don't meet these limitations must be approximated by ones that do.

Double precision floating-numbers can be stored in a 64-bit word, with 52 bits for f, 11 bits for e, and 1 bit for the sign of the number. The sign of e is accommodated by storing e+1023, which is between 1 and 2^{11}-2. The two extreme values for the exponent field, 0 and 2^{11}-1, are reserved for exceptional floating-point numbers, which we will describe later.

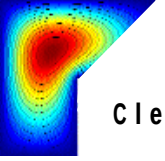


The picture above shows the distribution of the positive numbers in a toy floating-point system with only three bits each for f and e. Between 2^e and 2^{e+1} the numbers are equally spaced with an increment of 2^{e-3}. As e increases, the spacing increases. The spacing of the numbers between 1 and 2 in our toy system is 2^{-3}, or $\frac{1}{8}$. In the full IEEE system, this spacing is 2^{-52}. MATLAB calls this quantity eps, which is short for *machine epsilon*.

```
eps = 2^(–52)
```

What is the output?

```
a = 4/3
b = a – 1
c = b + b + b
e = 1 – c
```



Before the IEEE standard, different machines had different values of eps.

The approximate decimal value of eps is $2.2204 \cdot 10^{-16}$. Either `eps/2` or `eps` can be called the *roundoff level*. The maximum relative error incurred when the result of a single arithmetic operation is rounded to the nearest floating-point number is `eps/2`. The maximum relative spacing between numbers is `eps`. In either case, you can say that the roundoff level is about 16 decimal digits.

A very important example occurs with the simple MATLAB statement

```
t = 0.1
```

The value stored in `t` is not exactly 0.1 because expressing the decimal fraction $1/10$ in binary requires an infinite series. In fact,

$$1/10 = 1/2^4 + 1/2^5 + 0/2^6 + 0/2^7 + 1/2^8 + 1/2^9 + 0/2^{10} + 0/2^{11} + 1/2^{12} + \ldots$$

After the first term, the sequence of coefficients 1, 0, 0, 1 is repeated infinitely often. The floating-point number nearest 0.1 is obtained by rounding this series to 53 terms, including rounding the last four coefficients to binary 1010. Grouping the resulting terms together four at a time expresses the approximation as a base 16, or *hexadecimal*, series. So the resulting value of `t` is actually

$$t = (1 + 9/16 + 9/16^2 + 9/16^3 + \ldots + 9/16^{12} + 10/16^{13}) \cdot 2^{-4}$$

The MATLAB command

```
format hex
```

causes `t` to be printed as

```
3fb999999999999a
```

The first three characters, `3fb`, give the hexadecimal representation of the biased exponent, e+1023, when e is -4. The other 13 characters are the hex representation of the fraction f.

So, the value stored in `t` is very close to, but not exactly equal to, 0.1. The distinction is occasionally important. For example, the quantity

```
0.3/0.1
```

is not exactly equal to 3 because the actual numerator is a little less than 0.3 and the actual denominator is a little greater than 0.1.

Ten steps of length `t` are not precisely the same as one step of length 1. MATLAB is careful to arrange that the last element of the vector

```
0:0.1:1
```

is exactly equal to 1, but if you form this vector yourself by repeated additions of 0.1, you will miss hitting the final 1 exactly.

Another example is provided by the MATLAB code segment in the margin on the previous page. With exact computation, `e` would be 0. But in floating-point, the computed `e` is not 0. It turns out that the only roundoff error occurs in the first statement. The value stored in `a` cannot be exactly $4/3$, except on that Russian trinary computer. The value stored in `b` is close to $1/3$, but its last bit is 0. The value stored in `c` is not exactly equal to 1 because the additions are done without any error. So the value stored in `e` is not 0. In fact, `e` is equal to `eps`. Before the IEEE standard, this code was used as a quick way to estimate the roundoff level on various computers.

The roundoff level `eps` is sometimes called "floating-point zero," but that's a misnomer. There are many floating-point numbers much smaller than `eps`. The smallest positive normalized floating-point number has $f = 0$ and $e = -1022$. The largest floating-point number has f a little less than 1 and $e = 1023$. MATLAB calls these numbers `realmin` and `realmax`. Together with `eps`, they characterize the standard system.

Name	Binary	Decimal
`eps`	`2^(-52)`	`2.2204e-16`
`realmin`	`2^(-1022)`	`2.2251e-308`
`realmax`	`(2-eps)*2^1023`	`1.7977e+308`

When any computation tries to produce a value larger than `realmax`, it is said to *overflow*. The result is an exceptional floating-point value called `Inf`, or *infinity*. It is represented by taking $f = 0$ and $e = 1024$ and satisfies relations like `1/Inf = 0` and `Inf+Inf = Inf`.

When any computation tries to produce a value smaller than `realmin`, it is said to *underflow*. This involves one of the optional, and controversial, aspects of the IEEE standard. Many, but not all, machines allow exceptional *denormal* or *subnormal* floating-point numbers in the interval between `realmin` and `eps*realmin`. The smallest positive subnormal number is about 0.494e-323. Any results smaller than this are set to zero. On machines without subnormals, any result less than `realmin` is set to zero. The subnormal numbers fill in the gap you can see in our toy system between zero and the smallest positive number. They do provide an elegant model for handling underflow, but their practical importance for

MATLAB style computation is very rare.

When any computation tries to produce a value that is undefined even in the real number system, the result is an exceptional value known as *Not-a-Number*, or NaN. Examples include `0/0` and `Inf-Inf`.

MATLAB uses the floating-point system to handle integers. Mathematically, the numbers 3 and 3.0 are the same, but many programming languages would use different representations for the two. MATLAB does not distinguish between them. We like to use the term *flint* to describe a floating-point number whose value is an integer. Floating-point operations on flints do not introduce any roundoff error, as long as the results are not too large. Addition, subtraction and multiplication of flints produce the exact flint result, if it is not larger than `2^53`. Division and square root involving flints also produce a flint when the result is an integer. For example, `sqrt(363/3)` produces 11, with no roundoff error.

As an example of how roundoff error effects matrix computations, consider the two-by-two set of linear equations

$$10x_1 + x_2 = 11$$
$$3x_1 + 0.3x_2 = 3.3$$

The obvious solution is $x_1 = 1$, $x_2 = 1$. But the MATLAB statements

```
A = [10   1;   3   0.3]
b = [11   3.3]'
x = A\b
```

produce

```
x =
   -0.5000
   16.0000
```

Why? Well, the equations are singular. The second equation is just 0.3 times the first. But the floating-point representation of the matrix `A` is not exactly singular because `A(2,2)` is not exactly 0.3.

Gaussian elimination transforms the equations to the upper triangular system

```
U*x = c
```

where

```
U(2,2)= 0.3 - 3*(0.1)
      = -5.5551e-17
```

and

```
c(2)  = 3.3 - 33*(0.1)
      = -4.4409e-16
```

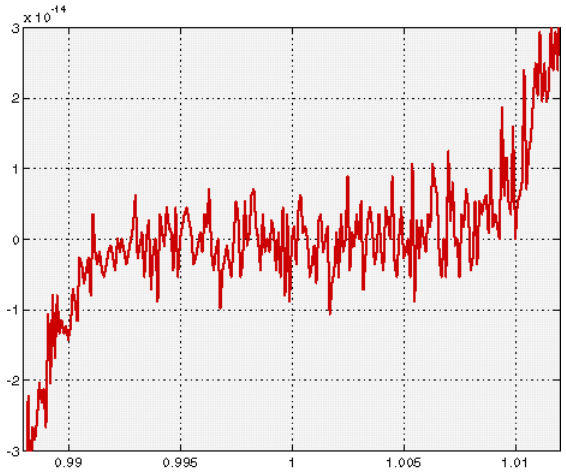
MATLAB notices the tiny value of `U(2,2)` and prints a message warning that the matrix is close to singular. It then computes the ratio of two roundoff errors

```
x(2)  =      c(2)/U(2,2)
      =      16
```

This value is substituted back into the first equation to give

```
x(1)  =      (11 - x(2))/10
      =      -0.5
```

The singular equations are consistent. There are an infinite number of solutions. The details of the roundoff error determine which particular solution happens to be computed.



Our final example plots a seventh degree polynomial.

```
x = 0.988:.0001:1.012;
y = x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-...
    21*x.^2+7*x-1;
plot(x,y)
```

But the resulting plot doesn't look anything like a polynomial. It isn't smooth. You are seeing roundoff error in action. The y-axis scale factor is tiny, 10^{-14}. The tiny values of y are being computed by taking sums and differences of numbers as large as $35 \cdot 1.012^4$. There is severe subtractive cancellation. The example was contrived by using the Symbolic Toolbox to expand $(x - 1)^7$ and carefully choosing the range for the x-axis to be near $x = 1$. If the values of y are computed instead by

```
y = (x-1).^7;
```

then a smooth (but very flat) plot results. ■

Cleve Moler is chairm
and co-founder of
The MathWorks.
His e-mail address is
moler@mathworks.com