Floating-Point Arithmetic in Matlab

Since real numbers can not be coded with finite number of bits, Matlab and most other technical computing environments use floating-point arithmetic, which involves a finite set of numbers with finite precision. This leads to the phenomena of *roundoff*, *underflow*, and *overflow*. Most of the time, it is possible to use Matlab effectively 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.

Before 1985, 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 since 1985 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 has traditionally used the IEEE double-precision format. There is a single-precision format that saves space, but that isn't much faster on modern machines. Below we will deal exclusively with double precision. There is also an extended-precision format, which is optional and therefore is one of the reasons for lack of uniformity among different machines.

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

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

The quantity f is the fraction or mantissa and e is the exponent. The fraction satisfies

$$0 \le f < 1$$

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

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

The exponent e is an integer in the interval

$$-1022 \leq e \leq 1023.$$

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-point numbers are 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 2 extreme values for the exponent field, 0 and $2^{11} - 1$, are reserved for exceptional floating-point numbers that we will describe later.

The entire fractional part of a floating-point number is not f, but $1 + f$, which has 53 bits. However, the leading 1 doesn't need to be stored. In effect, the IEEE format packs 65 bits of information into a 64-bit word.

The program `floatgui` shows the distribution of the positive numbers in a model floating-point system with variable parameters. The parameter t specifies the number of bits used to store f. In other words, $2^t f$ is an integer. The parameters e_{min} and e_{max} specify the range of the exponent, so $e_{min} \leq e \leq e_{max}$. Initially, `floatgui` sets $t = 3$, $e_{min} = -4$, and $e_{max} = 2$ and produces the distribution shown in Figure 1.7.

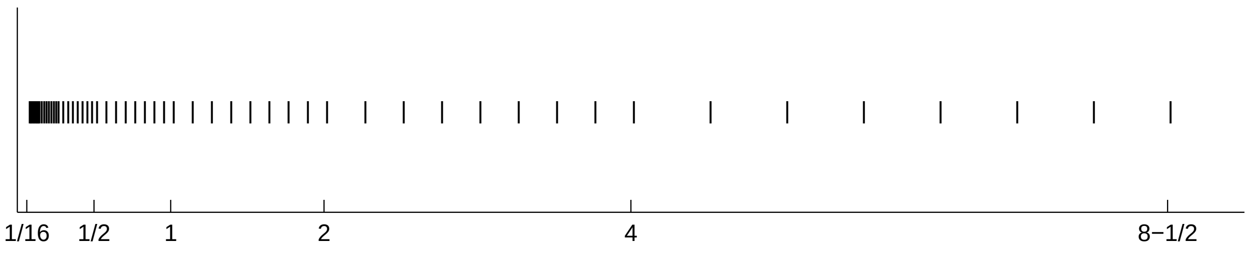

Figure 1: `floatgui`.

Within each binary interval $2^e \leq x \leq 2^{e+1}$, the numbers are equally spaced with an increment of 2^{e-t}. If $e = 0$ and $t = 3$, for example, the spacing of the numbers between 1 and 2 is 1/8. As e increases, the spacing increases.

It is also instructive to display the floating-point numbers with a logarithmic scale. Figure 1.8 shows `floatgui` with `logscale` checked and $t = 5$, $e_{min} = -4$, and $e_{max} = 3$. With this logarithmic scale, it is more apparent that the distribution in each binary interval is the same.

A very important quantity associated with floating-point arithmetic is highlighted in red by `floatgui`. Matlab calls this quantity `eps`, which is short for *machine epsilon.*

`eps` *is the distance from 1 to the next larger floating-point number.*

For the `floatgui` model floating-point system, `eps = 2^(-t)`.

Before the IEEE standard, different machines had different values of `eps`. Now, for IEEE double-precision,

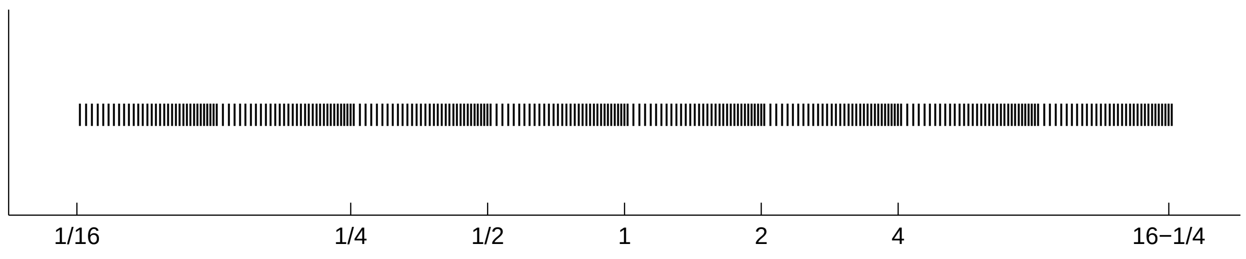

Figure 2: floatgui(logscale).

```
    eps = 2^(-52).
```

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 an 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 frequent instance of roundoff occurs with the simple MATLAB statement

```
    t = 0.1
```

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

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \cdots.$$

After the first term, the sequence of coefficients 1, 0, 0, 1 is repeated infinitely often. Grouping the resulting terms together four at a time expresses 1/10 in a base 16, or *hexadecimal*, series.

$$\frac{1}{10} = 2^{-4} \cdot \left(1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^2} + \frac{9}{16^3} + \frac{9}{16^4} + \cdots\right)$$

Floating-point numbers on either side of 1/10 are obtained by terminating the fractional part of this series after 52 binary terms, or 13 hexadecimal terms, and rounding the last term up or down. Thus

$$t_1 < 1/10 < t_2,$$

where

$$t_1 = 2^{-4} \cdot \left(1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \cdots + \frac{9}{16^{12}} + \frac{9}{16^{13}}\right),$$

$$t_1 = 2^{-4} \cdot \left(1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \cdots + \frac{9}{16^{12}} + \frac{10}{16^{13}}\right).$$

It turns out that 1/10 is closer to t_2 than to t_1, so t is equal to t_2. In other words,

$$t = (1 + f) \cdot 2^e,$$

where

$$f = \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \cdots + \frac{9}{16^{12}} + \frac{10}{16^{13}},$$
$$e = -4.$$

The Matlab command

```
format hex
```

causes t to be displayed as

```
3fb999999999999a
```

The characters a through f represent the hexadecimal "digits" 10 through 15. The first three characters, `3fb`, give the hexadecimal representation of decimal 1019, which is the value of the biased exponent $e + 1023$ if e is -4. The other 13 characters are the hexadecimal representation of the fraction f.

In summary, 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.

What does the floating-point approximation to the golden ratio look like?

```
format hex
phi = (1 + sqrt(5))/2
```

produces

```
phi =
    3ff9e3779b97f4a8
```

The first hex digit, 3, is 0011 in binary. The first bit is the sign of the floating- point number; 0 is positive, 1 is negative. So `phi` is positive. The remaining bits of the first three hex digits contain $e + 1023$. In this example, `3ff` in base 16 is $3 \cdot 16^2 + 15 \cdot 16 + 15 = 1023$ in decimal. So

$$e = 0.$$

In fact, any floating-point number between 1.0 and 2.0 has $e = 0$, so its hex output begins with 3ff. The other 13 hex digits contain f. In this example,

$$f = \frac{9}{16} + \frac{14}{16^2} + \frac{3}{16^3} + \cdots + \frac{10}{16^{12}} + \frac{8}{16^{13}}.$$

With these values of f and e,

$$(1+f)2^e \approx \phi.$$

Another example is provided by the following code segment.

```
format long
a = 4/3
b = a - 1
c = 3*b
e = 1 - c
```

With exact computation, e would be 0. But with floating-point, the output produced is

```
a =
    1.33333333333333
b =
    0.33333333333333
c =
    1.00000000000000
e =
    2.220446049250313e-016
```

It turns out that the only roundoff occurs in the division in the first statement. The quotient cannot be exactly 4/3, except on that Russian trinary computer. Consequently the value stored in a is close to, but not exactly equal to, 4/3. The subtraction b = a - 1 produces a b whose last bit is 0. This means that the multiplication 3*b can be done without any roundoff. The value stored in c is not exactly equal to 1, and so the value stored in e is not 0. 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.

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

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

If 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`. `NaN` is represented by taking $e = 1024$ and f nonzero.

If 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 0. On machines without subnormals, any results less than `realmin` are set to 0. The subnormal numbers fill in the gap you can see in the `floatgui` model system between 0 and the smallest positive number. They do provide an elegant way to handle underflow, but their practical importance for MATLAB-style computation is very rare. Denormal numbers are represented by taking $e = -1023$, so the biased exponent $e + 1023$ is 0.

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 sometimes 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 int result if it is not larger than 2^{53}. Division and square root involving flints also produce a flint if the result is an integer. For example, `sqrt(363/3)` produces 11, with no roundoff.

Two MATLAB functions that take apart and put together floating-point numbers are LOG2 and POW2.

```
help log2
help pow2
```

produces

```
[F,E] = LOG2(X) for a real array X, returns an array F
of real numbers, usually in the range 0.5 <= abs(F) < 1,
and an array E of integers, so that X = F .* 2.^E.
Any zeros in X produce F = 0 and E = 0.

X = POW2(F,E) for a real array F and an integer array E
computes X = F .* (2 .^ E). The result is computed quickly
by simply adding E to the floating-point exponent of F.
```

The quantities F and E used by `log2` and `pow2` predate the IEEE floating-point standard and so are slightly different from the f and e we are using in this section. In fact, f = 2*F-1 and e = E-1.

```
    [F,E] = log2(phi)
```

produces

```
  F =
     0.80901699437495
  E =
       1
```

Then

```
    phi = pow2(F,E)
```

gives back

```
    phi =
          1.61803398874989
```

As an example of how roundoff error affects matrix computations, consider the 2-by-2 set of linear equations

$$17x_1 + 5x_2 = 22,$$

$$1.7x_1 + 0.5x_2 = 2.2.$$

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

```
  A = [17 5; 1.7 0.5]
  b = [22; 2.2]
  x = A\b
```

produce

```
  x =
    -1.0588
     8.0000
```

Where did this come from? Well, the equations are singular, but consistent. The second equation is just 0.1 times the first. The computed `x` is one of infinitely many possible solutions. But the floating-point representation of the matrix A is not exactly singular because `A(2,1)` is not exactly 17/10.

The solution process subtracts a multiple of the first equation from the second. The multiplier is `mu = 1.7/17`, which turns out to be the floating-point number obtained by truncating, rather than rounding, the binary expansion of 1/10. The matrix `A` and the right-hand side `b` are modified by

```
   A(2,:) = A(2,:) - mu*A(1,:)
   b(2) = b(2) - mu*b(1)
```

With exact computation, both `A(2,2)` and `b(2)` would become zero, but with floating-point arithmetic, they both become nonzero multiples of `eps`.

```
A(2,2) = (1/4)*eps
       = 5.5511e-17
b(2) = 2*eps
       = 4.4408e-16
```

MATLAB notices the tiny value of the new `A(2,2)` and displays a message warning that the matrix is close to singular. It then computes the solution of the modified second equation by dividing one roundoff error by another.

```
x(2) = b(2)/A(2,2)
     = 8
```

This value is substituted back into the first equation to give

```
x(1) = (22 - 5*x(2))/17
     = -1.0588
```

The details of the roundoff error lead MATLAB to pick out one particular solution from among the infinitely many possible solutions to the singular system.

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)
```

The resulting plot in Figure 1.9 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.

References

[1] CLEVE MOLER, *Numerical Computing with MATLAB*, SIAM, Philadelphia, 2004.

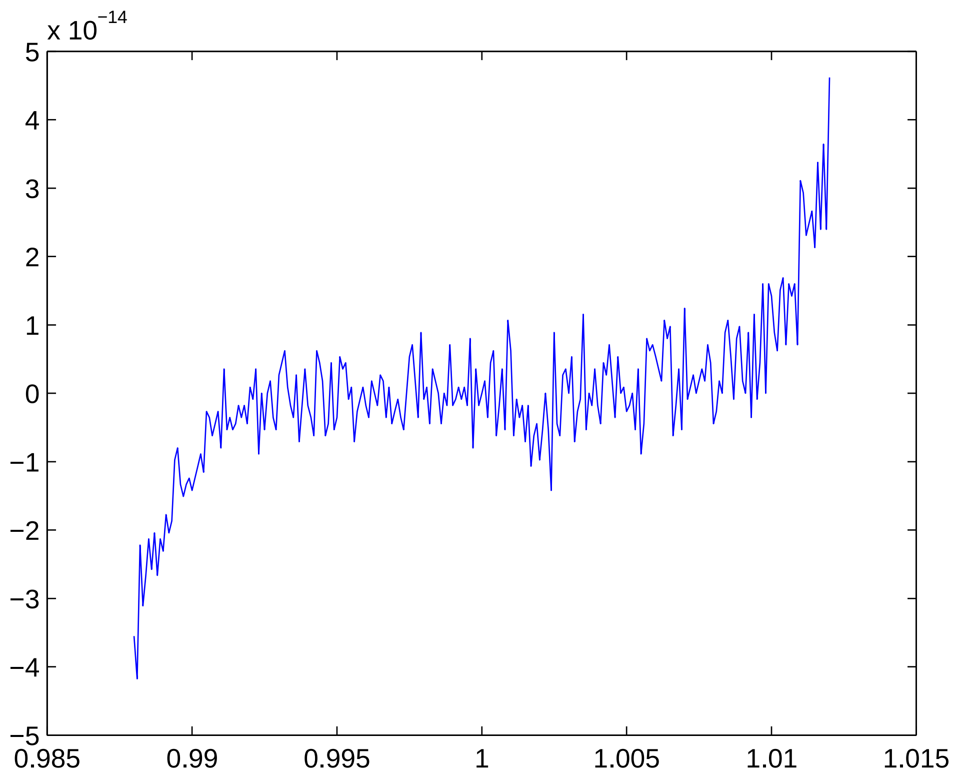

Figure 3: Figure 1.9. Is this a polynomial?