AMTH250

Floating Point Arithmetic

Gary Bunting

July 25, 2011

Contents

1 Floating Point Numbers

The arithmetic used by Octave is called **floating point arithmetic**. Scientific notation is used to express floating point numbers; the number 1.2345×10^{-6} is written in Octave as `1.2345e-6`.

Internally floating point numbers are stored in a binary format known as IEEE double precision arithmetic. Each floating point number occupies 64 bits, which contain the sign, digits and exponent of the number. The details of the binary representation are not usually important in practical applications, but the finite precision of floating point numbers has important implications:

1. Each floating point number is capable of representing about 16 decimal digits (actually 53 bits).

2. There is a limit on the range of exponents; from about 10^{-308} to 10^{+308} (actually 2^{-1022} to 2^{+1023}).

The limit on precision means that it is impossible to calculate with more than 16 digits precision. This, in itself, is not a problem since it is rare in practical applications to require more accuracy than this. However it sometimes happens that accuracy deteriorates in numerical computations due to rounding errors.

Example

Consider the following sequence of calculations:

```
octave:> format long
octave:> a = 4/3
octave:> b = a - 1
octave:> c = 3*b
octave:> d = 1 - c
```

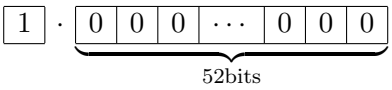
Of course the exact value of `d` is 0, but in floating point arithmetic

```
a =  1.33333333333333
b =  0.333333333333333
c =  1.000000000000000
d =  2.22044604925031e-16
```

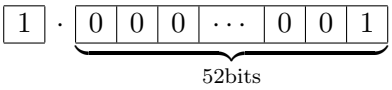
In this calculation rounding error only occurs at the first statement. The rational number 4/3 is not exactly representable as floating point number so number stored in `a` is not exactly 4/3. Then `b = a - 1` is not exactly 1/3 and `c = 3*b` is not exactly 1.

2 Precision

The usual measure of precision in floating point arithmetic is the number **machine epsilon**, written $\varepsilon_{\text{mach}}$, and defined to be difference between the number 1 and the next largest floating point number. In IEEE arithmetic the number 1 has the representation



and the next largest floating point number is



Thus

$$\varepsilon_{\text{mach}} = 2^{-52} \approx 2.220 \times 10^{-16}$$

In Octave $\varepsilon_{\text{mach}}$ is denoted by `eps`.

```
octave:> eps
ans =  2.22044604925031e-16
```

In the example of the previous section, the final result *is* $\varepsilon_{\text{mach}}$

```
octave:> d - eps
ans = 0
```

Before the introduction of the IEEE floating point standard, the code in the example was a quick way to determine $\varepsilon_{\text{mach}}$ for any particular machine.

3 Exceptional Values

Besides the usual floating point numbers, IEEE arithmetic has two special numbers `Inf`, for 'infinity', and `Nan`, for 'not-a-number'. `Inf` typically occurs when we try try to produce a number whose exponent is greater than the maximum exponent:

```
octave:> x = 1e300
x =  1.0000e+300
octave:> x*x
ans = Inf
```

This is called **overflow**. It is easy to recognize when overflow has occurred since an `Inf` always results.

If the result of a calculation has exponent less than the smallest exponent, the result is set to zero. This is called **underflow** and, while it is usually harmless it can sometimes cause unexpected errors.

Example

Issues of overflow and underflow need to be taken into account in writing numerical software. Consider, for example, the Octave function `hypot(x,y)` which computes $\sqrt{x^2 + y^2}$. Mathematically, computing `hypot(x,y)` is equivalent to computing `sqrt(x^2 + y^2)`, but they give different numerical results

```
octave:> x1 = 1e300; y1 = 1e300;
octave:> x2 = 1e-300; y2 = 1e-300;

octave:> hypot(x1, y1)
ans =  1.4142e+300
octave:> sqrt(x1^2 + y1^2)
ans = Inf

octave:> hypot(x2, y2)
ans =  1.4142e-300
octave:> sqrt(x2^2 + y2^2)
ans = 0
```

It is easy to see what has gone wrong with calculation using `sqrt`. In the first example, the numbers `x1^2` and `y1^2` overflow to `Inf` and `sqrt(Inf)` results in another `Inf`. In the second example, the numbers `x2^2` and `y2^2` underflow to 0 and `sqrt(0)` gives 0.

The following shows some ways `Nan`'s can arise:

```
octave:> 0 * Inf
ans = NaN
octave:> Inf - Inf
ans = NaN
```

`Nan`'s have another use — in statistical computing they are often used to represent missing data values.

4 Rounding Error

When arithmetic operations are preformed on floating point numbers the exact result will not, in general, be representable as a floating point number. The exact result will be rounded to the nearest[1] floating point number, a process called, obviously enough, **rounding**. In IEEE arithmetic all results of floating point operations are correctly rounded, something that was rarely done on early computers.

[1] In the case of a tie, the round to even rule is used.

Rounding errors are unavoidable in numerical computation due to the finite precision of floating point arithmetic. The relative error due to rounding is less than $1/2\varepsilon_{\text{mach}} \approx 1.1 \times 10^{-16}$, but this can be expected to occur in every floating point operation, including binary/decimal conversion on reading/writing floating point numbers.

5 Cancellation Error

Cancellation error occurs when two nearly equal numbers floating point are subtracted. This result can have much larger relative error than the original numbers.

Example

Let $\Delta = 1/3 \times 10^{-10}$, $x = 1 + 2\Delta$ and $y = 1 + \Delta$. Then $x - y = \Delta$. Now let's see what happens in floating point arithmetic.

```
octave:> format long
octave:> del = (1/3)*1e-10
del =  3.33333333333333e-11
octave:> x = 1 + 2*del
x =  1.00000000006667
octave:> y = 1 + del
y =  1.00000000003333
octave:> diff = x - y
diff =  3.33333360913457e-11
```

Neither x not y can represented exactly in floating point and rounding error will also occur in the subtraction so the computed value of $x - y$ will not be exactly equal to Δ. The error is

```
octave:> abserr = abs(diff - del)
abserr =  2.75801236542194e-18
```

An error of this magnitude is in accord what we know about floating point. The relative error in x and y due to floating point conversion will be less than $1/2\varepsilon_{\text{mach}} \approx 1.1 \times 10^{-16}$, However the relative error in their difference is

```
octave:> relerr = abserr/del
relerr =  8.27403709626582e-08
```

This error is (at least) 10^7 times larger than the relative error in x and y.

Example

An instructive example of cancellation error occurs when we approximate the derivative of function by a formula such as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

As h gets smaller and smaller the approximation gets better until in the limit as $h \to 0$ we get the definition of the derivative.

Computing with this formula has the numerical problem that as h gets smaller, $f(x+h)$ and $f(x)$ get closer together and cancellation error will occur. In fact if h is small enough, $f(x+h)$ and $f(x)$ will be represented by the *same* floating point number giving the approximation $f'(x) \approx 0$.

We will take $f(x) = \sin x$ and approximate the derivative at $x = 1$ using values of h from 10^{-2} down to 10^{-17} [note the use of the dot operator]:

```
octave:> n = -2:-1:-17
octave:> h = 10 .^n
octave:> approx = (sin(1+h) - sin(1))./h
```

We compute the relative error in the approximation and do a log-log plot of the error as a function of h:

```
octave:> err = abs(approx - cos(1)) / cos(1)
octave:> loglog(h, err)
```

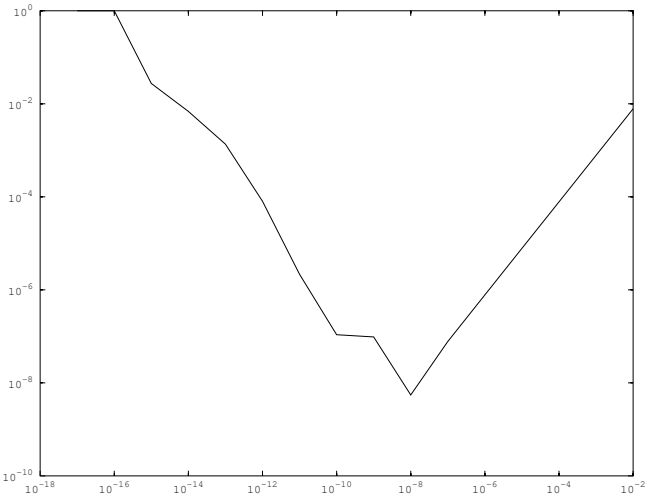


Figure 1: Log-log plot of error as a function of h. Note: h decreases from right to left

At first, as h decreases, the main cause of error is the approximation itself (this is called **truncation error**) and the error decreases with h. At $h \approx 10^{-8}$ the error begins to increase. The main cause of error now is cancellation and the error increases until at $h = 10^{-16}$ the approximate derivative is zero and completely useless.