Handout: floating point arithmetic

When implementing numerical algorithms on a digital computer it is advisable to know something about how the computer stores and manipulates numbers. A computer has finite memory and cannot represent the full (uncountably infinite) set of real numbers. Instead numerical computations are performed with with a very large but finite set of *floating point* numbers. Floating point numbers are numbers of the form

$$f = \pm d_0.d_1 d_2 \ldots d_t \times \beta^e, \tag{1}$$

for some base $\beta \in \mathbb{N}$, precision $t \in \mathbb{N}$, mantissa $d_0 \ldots d_t$ where the digits $d_i \in \mathbb{N}_0$ with $0 \leq d_i < \beta$, and exponent $e \in \mathbb{Z}$, which lies in some range $L \leq e \leq U$ for some fixed integers L and U. The IEEE standard 754 specifies the standard for the encoding and operation of floating point numbers, including the specification of several binary formats ($\beta = 2$) and decimal formats ($\beta = 10$).[1]

At present, the majority of numerical computations are performed in *double precision* arithmetic, i.e. with 64 bit binary ($\beta = 2$) floating point numbers, where each floating point number uses 64 bits of storage. The reason this is called *double precision* is because *single precision* arithmetic refers to arithmetic with 32 bit floating point numbers (which was more commonly used in older generation hardware).

The relevant values of β, L, U and t for double precision arithmetic are

$$\beta = 2, \quad t = 52, \quad L = -1022, \quad U = 1023,$$

Here 1 bit is used to store the sign, 52 bits are used to store the values of the d_i, $i = 1, \ldots, 52$, and 11 bits are used to store the exponent e, which amounts to 64 bits in total. For binary floating point numbers, it turns out that the value of d_0 can be inferred from the value of the exponent and is not stored explicitly, so d_0 is called the implicit bit (we won't go into detail here as to how exactly d_0 is inferred).

For an arbitrary real number x lying in the representable range, the relative error in the floating point representation $\mathrm{fl}(x)$ of x satisfies

$$\frac{|\mathrm{fl}(x) - x|}{|x|} \leq \frac{1}{2}\beta^{1-t} =: eps;$$

eps is called the *unit round-off*. For double precision 64 bit floating point numbers,

$$eps = 2^{-52} \approx 2 \times 10^{-16}.$$

[1]The standard also specifies some special values for floating point numbers such as $\pm\infty$ and `NaN` which stands for Not a Number and reports errors such as division by zero.

For instance, the closest double precision floating point number to 0.6 (in decimal notation) is (again in decimal notation)

$$\mathrm{fl}(0.6) = 0.59999999999999997779553950749686919152736663818359375,$$

which has a relative error below *eps*. Software that displays a floating point number will often round the floating point number to the nearest 15 or 16 digit decimal number, so it might print fl(0.6) as 0.6. However, the exact number used by the computer for computations will still be fl(0.6) shown above.

If a computation with floating point numbers achieves a relative error of size *eps* (or close to it) then one says the approximation is "correct to machine precision"; one cannot in general hope to do any better than this[2]. Whenever the computer carries out an arithmetic operations such as $+, -, \times, \div$ on a pair of floating point numbers, the result returned usually has to be rounded to a floating point number. This causes *rounding errors* which are usually at a relative level of machine precision.

Do rounding errors matter for numerical methods? In most cases, numerical methods can be designed and implemented so that the effects of rounding errors do not seriously impact the accuracy of the results, *as long as we are aware of some of the limitations of floating point numbers.*

One such limitation is that rounding errors can cause problems when one computes the difference of two numbers that are very close to one another. For instance, given $h > 0$, consider the calculation of

$$z = x - y, \qquad \text{where} \quad x = 1 + h \quad \text{and} \quad y = 1.$$

Clearly the exact value is $z = h$. But, assuming we first compute x and y, and then compute their difference, the relative error in the floating point version of the calculation is

$$\left|\frac{\mathrm{fl}(z) - z}{z}\right| = \frac{|\mathrm{fl}(\mathrm{fl}(1+h) - \mathrm{fl}(1)) - h|}{h},$$

and if h is small then $\mathrm{fl}(1+h) - \mathrm{fl}(1)$ may not be an accurate approximation of h, in which case $\mathrm{fl}(z)$ will not be an accurate approximation of z. In Matlab one may illustrate this by taking

```
>> h = 1.e-15; (((1+h)-1)-h)/h
   ans=0.1102
```

The loss of significant digits in the calculation of $\mathrm{fl}(1+h) - \mathrm{fl}(1)$ has caused an error of 11% in the calculation of z! If we take h smaller than the unit

[2]However, there exists software, such as Mathematica, that allow for the emulation of floating point arithmetic with an arbitrary large (but still finite) precision chosen by the user. The term machine precision is therefore slightly inaccurate since it refers to the precision of a given floating point arithmetic, rather than the precision of a given computer. Nevertheless, it has become common terminology.

round-off *eps* then things get even worse, since then $\mathrm{fl}(1+h)=1$ and the computed value of z would be $\mathrm{fl}(z)=\mathrm{fl}(\mathrm{fl}(1+h)-\mathrm{fl}(1))=\mathrm{fl}(1-1)=0$, giving a relative error of 100%! This can have catastrophic consequences if the difference $z=x-y$ appears on the denominator of an expression, since then one would be trying to divide by zero.

One important situation in which one has to calculate the difference between two close numbers is when one is approximating the derivative of a function by a difference quotient approximation

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

In exact arithmetic, the accuracy of this approximation improves as h tends to zero. However, in floating point arithmetic this only holds up to a point, because eventually rounding errors will mean that the difference $f(x+h)-f(x)$ is not being calculated sufficiently accurately.

To illustrate this, consider the function $f(x)=x^2$, and consider the above difference quotient approximation of $f'(1)=2$. In exact arithmetic, the relative error of this approximation would be

$$\frac{1}{2}\left|\frac{(1+h)^2-1}{h}-2\right|=h.$$

But when we compute the relative error in the floating point arithmetic version of the same approximation, we obtain the results in Figure 1. As h decreases the error initially decreases, but at $h\approx10^{-8}$ it reaches a minimum and then starts to increase, reaching 100% error once $h\approx eps\approx10^{-16}$.

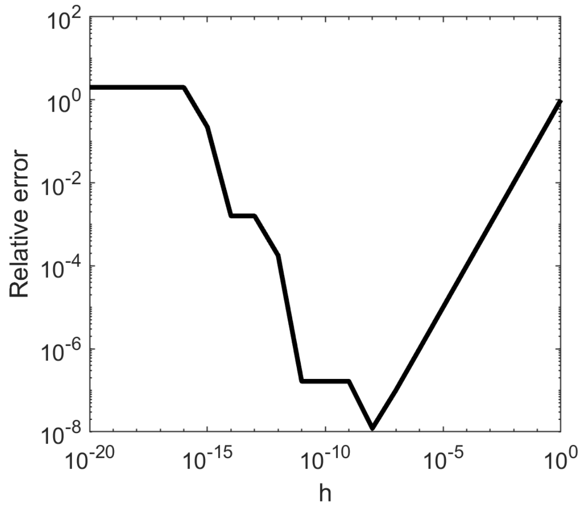

Figure 1: Relative error in difference quotient approximation of a derivative.