Numerical Methods II

Prof. Mike Giles

mike.giles@maths.ox.ac.uk

Oxford University Mathematical Institute

Two computing issues

In preparation for lectures 7 and 8, this lecture looks at two computing issues:

- finite precision arithmetic
- numerical differentiation

Finite precision arithmetic

On the computer, integer arithmetic is exact
(but leads to overflow if the numbers are too big)
but floating point arithmetic is not exact.

A floating point number is represented by

$$f = x \times 2^n$$

where n is the integer exponent which is given by some number of bits, and $1 > |x| \geq 1/2$ is also represented by some number of bits:

e.g. $\qquad 2/3 \equiv 0.10101010101010$

Finite precision arithmetic

In MATLAB, and also C/C++, we have a choice between double precision (default in MATLAB) and single precision.

type	total bits	exponent	significand
double	64	11	53
single	32	8	24

The number of exponent bits affects the range – what is the biggest/smallest number which can be represented.

The number of significand bits S affects the accuracy – in general, when multiplying two numbers y, z the error is roughly of size $2^{-S}|y\,z|$.

Finite precision arithmetic

How does computer arithmetic work?

Conceptually, first do exact computation:

$$\begin{aligned} z &= x + y \\ z &= x \times y \end{aligned}$$

then round the result z to the nearest floating point number which can be represented.

Relative error in rounding:

$$2^{-S} \approx \begin{cases} 10^{-16} & \text{double precision} \\ 10^{-7} & \text{single precision} \end{cases}$$

In MATLAB these are `eps` and `eps('single')`.

Finite precision arithmetic

What happens with the following MATLAB code?

```
sum = single(0);

for n = 1:10^9
  sum = sum + 1;
end

sum
```

. . . and why?

Finite precision arithmetic

In Monte Carlo simulation we need to average over N samples.

If the sum is s, error in each increment is of size $e_n \approx 2^{-S} s$, but the errors are equally likely to be positive or negative, so

$$\mathbb{E}[e_n] = 0, \quad \mathbb{V}[e_n] \sim 2^{-2S} s^2$$

and hence

$$\mathbb{E}\left[\sum_n e_n\right] = 0, \quad \mathbb{V}\left[\sum_n e_n\right] \sim N\, 2^{-2S} s^2$$

so final fractional error is roughly of size $2^{-S}\sqrt{N}$.

Finite precision arithmetic

In single precision, this might be about $O(10^{-4})$ which is not a huge error compared to other errors:

- Monte Carlo sampling errors
- model errors
- errors in uncertain parameters
- bias due to timestep discretisation

but maybe still better to use double precision for averaging.

The only other concern in computational finance applications is with next topic – numerical differentiation and "bumping" for Greeks.

But why not just do everything in double precision?

Single or double precision?

Modern CPUs have vector units:

- 256 bits wide $\equiv$ 4 doubles or 8 floats
- future CPUs will handle 8 doubles or 16 floats, so single precision twice as fast as double precision
- not using the vector unit “throws away” most of the CPU’s compute capability

GPUs (graphics chips which can now be used for compute) are also at least twice as fast in single precision compared to double precision.

My view: do things in single precision, except for averaging and “bumping”.

Numerical differentiation

Suppose we have MATLAB code to compute $f(x)$ (with x and $f(x)$ both scalar) and we want to compute the derivative $f'(x)$.

What can we do? Performing a Taylor series expansion,

$$f(x+\Delta x) \approx f(x) + \Delta x\, f'(x) + \tfrac{1}{2}\Delta x^2\, f''(x) + \tfrac{1}{6}\Delta x^3\, f'''(x)$$

$$\begin{aligned}
\Longrightarrow \qquad \frac{f(x+\Delta x) - f(x)}{\Delta x} &\approx f'(x) + \tfrac{1}{2}\Delta x\, f''(x), \\
\frac{f(x+\Delta x) - f(x-\Delta x)}{2\Delta x} &\approx f'(x) + \tfrac{1}{6}\Delta x^2\, f'''(x), \\
\frac{f(x+\Delta x) - 2f(x) + f(x-\Delta x)}{\Delta x^2} &\approx f''(x) + \tfrac{1}{24}\Delta x^2\, f''''(x).
\end{aligned}$$

Numerical differentiation

These are finite difference approximations, and they are the basis for the finite difference method for approximating PDEs.

In Monte Carlo methods, we use similar ideas (often referred to as “bumping”) for computing sensitivities (the “Greeks”)

Numerical example: trying to estimate $f'(1)$ for $f(x) = \sin x$

Numerical differentiation

The problem with taking $\Delta x \ll 1$ is inaccuracy due to finite precision arithmetic.

Error in computing $f(x+\Delta x) - f(x)$ is roughly of size $2^{-S} f(x)$, so error in computing one-sided difference estimate for $f'(x)$ is of order

$$\frac{2^{-S} f(x)}{2\Delta x}$$

while the finite difference error is $O(\Delta x)$.

Numerical differentiation

To balance errors, want

$$\frac{2^{-S}}{\Delta x} \sim \Delta x \quad \Longrightarrow \quad \Delta x \sim 2^{-S/2}.$$

In single precision, this means taking $\Delta x \sim 10^{-3}$, and getting an error which is roughly of size 10^{-3}.

This is not great, and making Δx smaller or bigger will make things worse.

This is why many banks use double precision when doing “bumping” for sensitivity analysis.

Complex Variable Trick

This is a very useful “trick”, which I learned about from this very short article:

> “Using Complex Variables to Estimate Derivatives of Real Functions”, William Squire and George Trapp, SIAM Review, 40(1):110-112, 1998.

which now has 331 citations according to Google Scholar.

Complex Variable Trick

Suppose $f(z)$ is a complex analytic function, and $f(x)$ is real when x is real.

Then

$$f(x+i\,\Delta x) \approx f(x) + i\,\Delta x\, f'(x) - \tfrac{1}{2}\Delta x^2\, f''(x) - i\,\tfrac{1}{6}\Delta x^3\, f'''(x)$$

and hence

$$\frac{\text{Im} f(x+i\,\Delta x)}{\Delta x} \approx f'(x) - \tfrac{1}{6}\Delta x^2\, f'''(x)$$

Now, we can take $\Delta x \ll 1$, and there is no problem due to finite precision arithmetic.

I typically use $\Delta x = 10^{-10}$!

Complex Variable Trick

There are just a few catches, because $f(z)$ must be analytic:

- need analytic extensions for $\min(x,y)$, $\max(x,y)$ and $|x|$
- need analytic extensions to certain MATLAB functions, e.g. `normcdf`
- in MATLAB, must be aware that A' is the Hermitian of A (complex conjugate transpose), so use $A.'$ for the simple transpose.

Using this, can very simply “differentiate” almost any MATLAB code for a real function $f(x)$.