Week 4: aim to cover

- Monte Carlo integration, floating point numbers, roundoff error (Lecture 7)
- Confidence intervals, MC integration, roundoff error (Lab 4)
- Error propagation (Lecture 8)

Monte Carlo integration

It is frequently necessary to compute definite integrals. The functions being integrated are often quite complicated and it may not be possible to find an indefinite integral in closed form. Since what is wanted is a numerical result, computers are used to find numerical approximations to the definite integral. In the simplest case, we are given a function $f(x)$ and two limits of integration a and b and the object is to approximate

$$\int_a^b f(x)\,dx.$$

There are methods that are completely deterministic — every time you run the program you will get exactly the same answer.

There is another class of methods that use random numbers to compute the value of definite integrals — they are called *Monte Carlo methods*, after the famous Monaco casino. They are uncompetitive for simple one-dimensional integrals, but prove to be superior for complicated integrals over many variables.

Hit-and-Miss method

One method — so-called *hit-and-miss Monte Carlo* — uses the relation of the integral to the area under the graph of $f(x)$. Suppose we want the value of

$$\int_a^b f(x)dx$$

where $f(x) \geq 0$ and we know the maximum value of $f(x)$ over [a,b]:

$$0 \leq f(x) \leq M$$

Then the rectangle just including all the graph of $f(x)$ over [a,b] has area $(b-a)M$. In other words, the fraction of the rectangle lying under the curve $y = f(x)$ is just

$$\frac{\int_a^b f(x)dx}{(b-a)M}$$

In the hit-and-miss method, we generate points uniformly in the rectangle (generate an x-coordinate from $U(a, b)$, generate a y-coordinate from $U(0, M)$, then form the point (x, y)) and count the fraction of them that lie under the curve $y = f(x)$.

Buffon's needle (1733)

If we randomly toss a needle of length $l = 1$ onto a floor with floorboards of width $d = 2$, what is the probability the needle crosses over a crack between floorboards?

The needle crosses a crack if $y < \frac{l}{2}\sin\theta$ where y is the distance of the centre of the needle to the nearest crack, and θ is the angle of the upper part of the needle relative to the cracks, measured from the positive direction. Since $0 < y < \frac{d}{2} = 1$ and $0 < \theta < \pi$, the probability is exactly

$$\int_0^\pi \frac{1}{2}\sin\theta \; d\theta/\pi = \frac{1}{\pi}$$

So by counting the frequency of needles where $y < \frac{1}{2}\sin\theta$, we can estimate $\frac{1}{\pi}$.

$\triangleleft$ **Example:** We could also use this idea to estimate π by finding the fraction of random points inside the square of side 2 that also lie inside the unit circle — this fraction is just $\frac{\pi \times 1^2}{2^2} = \pi/4$.
Since we are using indicator variables, we would use the standard error for the sample proportion to obtain a suitable CI.

(Improved) MC integration

A second method uses the connection of the integral to the mean value of $f(x)$, not to be confused with the expected value of a random variable. The mean value of a function $f(x)$ over [a,b] is just

$$< f > \equiv \frac{1}{b-a} \int_a^b f(x) dx$$

In the simple Monte Carlo method, we generate values of x from $U(a,b)$ and calculate the sample mean of the set of values

$$\bar{f} \equiv \frac{1}{n} \sum_i f(x_i)$$

Our estimate of the integral is then just $b-a$ times the sample mean.

$$\int_a^b f(x) dx = (b-a) < f > \approx (b-a)\bar{f}$$

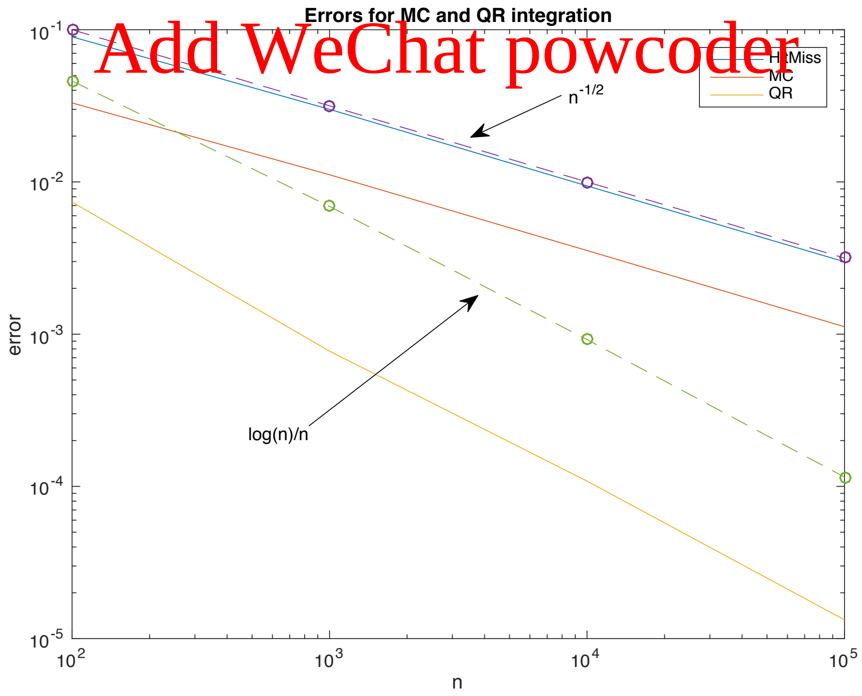
Both MC methods extend easily to higher-dimensional integrals.

A comparison

Here is a comparison for a simple 1-dimensional integral, so not really typical of the applications that MC integration is used for. We approximate

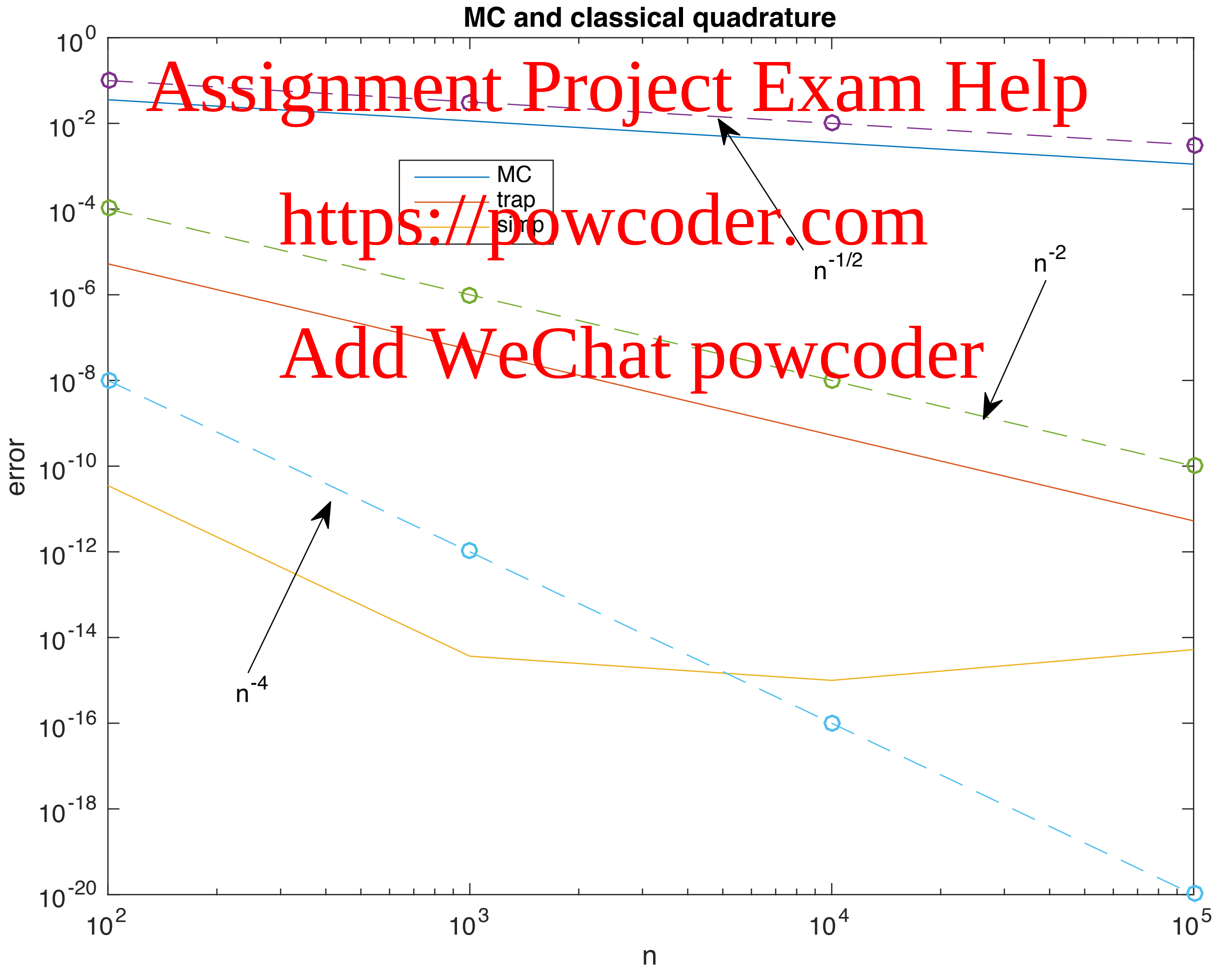
$$I = \int_0^1 \exp(-x)\, dx = 1 - \exp(-1) \approx 0.63212$$

First using both MC methods and a method using quasirandom numbers.



A comparison

Then we compare simple MC integration with 2 classical methods — the trapezoid rule and Simpson's rule.



In higher dimensions

The various integration methods differ in how they extend to higher dimensions. The classical methods require many points in each dimension, which makes them very expensive as the dimension d grows. Stated another way, the error falls very slowly with the number of evaluation points once the dimension gets larger than, say, 10.

Method	Error
MC	$n^{-1/2}$
QR	$\log(n)^d n^{-1}$
trapezoid	$n^{-2/d}$
Simpson's rule	$n^{-4/d}$

Table: Asymptotic error bound for large n, d

We see that MC methods have an error bound independent of the dimension, and win out for $d > 8$ or so.

This is why for many complicated processes that are equivalent to high-dimensional integrals (like pricing exotic stock options), Monte Carlo methods are the method of choice.

Overview of MC integration

Advantages:

- error decay rate independent of dimension
- can estimate the statistical error
- easy to program and parallelize

Disadvantages:

- error decays slowly for low dimensions
- sometimes the variance is very large, so need special methods to reduce it
- sometimes the estimates are correlated, so more care required to estimate error

More advanced topics: not in MAST30028

- Variance reduction methods e.g. importance sampling
- Markov chain methods
- Monte Carlo methods for optimization

How numbers are stored in a computer

To understand one ubiquitous source of error in scientific computing (roundoff error), have to understand how numbers are stored.

In scientific problems, numbers can vary greatly in magnitude $\implies$ we try to store numbers with a fixed relative error (**precision**), not absolute error. **Floating Point numbers** were invented to do this.

We describe the current standard: IEEE 754 used by most computers (perhaps not completely) for **double precision** numbers

Floating point numbers

To represent numbers with greatly varying size, we use scientific notation

Example

$6.023 \times 10^{23}, 1.055 \times 10^{-34}$

In the computer, we do the same but store both the exponent and fraction in base 2 (**binary representation**).

1011_2 represents the integer $2^3 + 2^1 + 2^0 = 8 + 0 + 2 + 1 = 11_{10}$

$\implies$ we approximate real numbers by a nearby rational number with a finite binary expansion

Floating point numbers

$$x = \pm(1.f_1 f_2 \cdots f_t) \times 2^e = \pm(1 + f)\, 2^e$$

f is the **fraction or mantissa**: $0 \leq f < 1$, each binary digit $f_i = 0$ or 1
e is the **exponent**, a binary integer

Example

$125.75 = 64 + 32 + 16 + 8 + 4 + 1 + 0.5 + 0.25 = 1111101.11_2 =$
$1.111110111_2 \times 2^6 = 1.111110111_2 \times 2^{110_2}$
so $1 + f = 1.111110111_2$, $e = 110_2$

e determines the range of numbers that can be represented
f determines the number of significant digits carried $\rightarrow$ the precision of numbers that can be represented

Double precision numbers

In double precision, use 64 bits (8 bytes) to store each number

- 1 for sign $\pm$
- 11 for exponent $\rightarrow 2^{11} = 2048$ possible values for e
- 52 for fraction $\implies t = 52$

$\implies$ to store 10^6 double precision numbers takes 8 Mbytes.
To get the exponent of the number, subtract 1023 from the 11-bit integer $\in [1, 2046] \rightarrow e \in [-1022, 1023]$ (so we get equal numbers of 'small' and 'large' numbers).
The extreme values $e = -1023, 1024$ are used for special cases or error flags.

Overflow

Using the largest exponent $e = 1023$ and the largest $1 + f = 1.11\cdots1_2$
we get the largest number representable $\approx 2^{1024} \approx 10^{308}$

called `realmax` in MATLAB

Any number larger than this causes Overflow

and is stored as infinity, with $e = 1024, f = 0$
called `Inf` in MATLAB

Underflow

Using the smallest exponent $e = -1022$ and the smallest $1 + f = 1.00 \cdots 0_2$ we get the smallest (non-zero, normalized) number representable `realmin`$= 2^{-1022} \approx 10^{-308}$
If $e = -1023$, use **denormalized** numbers ($x = \pm f \; 2^{-1022}$)
$\implies$ smallest non-zero number representable $= 2^{-1074} \approx 5 \times 10^{-324}$

Any number smaller than this gives Underflow and $\to 0$

Exceptions

Finally

Results of illegal operations (divide by zero, etc.) use $e = 1024, f \neq 0$

and are stored as `NaN` (Not a Number) in MATLAB.
Both `NaN` and `Inf` propagate in subsequent calculations, once they have been produced.
So you can diagnose the problem!

Machine numbers

Only numbers $\in$ [`realmin, realmax`] with a binary fraction expansion that terminates in less than 53 digits + the denormalized numbers can be represented exactly as a Floating Point number $\to$ the set of **machine numbers** , denoted by $\mathcal{F}$

Example

$125.75 = 1.111110111_2 \times 2^{110_2}$ which needs only 9 binary digits in the fraction $\implies$ 125.75 is a machine number

all the rest will be **approximated** with an error $\to$ Roundoff error

> *If rounding errors vanished, 95% of numerical analysis would remain.*

Trefethen's Maxims

The structure of the Floating Point number system

Since there are 2^{52} numbers in [1,2), 2^{52} numbers in [2,4), 2^{52} numbers in [4,8) etc.

The machine numbers are nonuniformly distributed

The granularity of the machine numbers is specified by the gap between 1 and the next machine number $1 + 2^{-t}$, called machine epsilon ε_M
Hence IEEE double precision numbers have machine epsilon $2^{-52} \approx 2 \times 10^{-16}$, given by eps in MATLAB.

Roundoff error

If you try to produce a non-machine number x, must store it as the nearest machine number produced by rounding — we denote this number $fl(x)$.

The default rounding mode: round to nearest, then even

$\implies$ maximum error $= \varepsilon_M/2 \times 2^e = u \times 2^e$

where unit roundoff $u = \varepsilon_M/2$.

$\implies$ max. relative error

$= u \times 2^e/(1+f) \times 2^e = u/(1+f) \in (u/2, u] \leq u$

unit roundoff determines the precision with which floating point numbers are stored

So

$$-u \leq \frac{fl(x) - x}{x} \leq u$$

Floating point precision

Since $t = 52$, IEEE double precision numbers have precision $2^{-53} \approx 10^{-16}$

16 decimal digits of precision

Can also have IEEE single precision numbers with unit roundoff $2^{-24} \approx 10^{-7}$

7 decimal digits of precision e.g. ANSI C `float`, MATLAB `single`

Floating point precision sets a lower bound to the level of (relative) data error

We make this error just in storing the numbers in a computer.

Floating point arithmetic

From the error on storing a number

$$-u \leq \frac{fl(x) - x}{x} \leq u$$

we get

$$fl(x) = x(1+\delta), \quad |\, \delta \,| \leq u \quad \forall x \in \mathbb{R}$$

i.e. the nearest machine number $fl(x)$ to x is at most a factor $1 \pm u$ away.
Since arithmetic operations on machine numbers produce results that usually are not machine numbers so must get rounded, we get a model for Floating point arithmetic (ignoring underflow/overflow)

$$\boxed{fl(x \text{ op } y) = (x \text{ op } y)(1+\delta), \quad |\, \delta \,| \leq u \quad \forall x, y \in \mathcal{F}, \text{ op } = +, -, \times, \div}$$

Summary

1. Some numbers (machine numbers) can be represented exactly as a Floating Point number; most can't

2. there's a smallest and largest Floating Point number

3. Floating Point numbers have a precision u — the source of inevitable roundoff error

End of Lecture 7