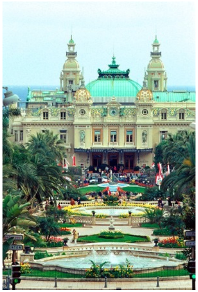
HPSC – Lecture 21

Outline:

- Monte Carlo methods
- Random number generators
- Monte Carlo integrators
- Random walk solution of Poisson problem

Monte Carlo Methods



Computational methods that use random (or pseudo-random) sampling to obtain numerical approximations.

Originally developed in 1940’s at Los Alamos for neutron diffusion problems.

Monte Carlo methods

Examples:

- Approximate a definite integral by sampling the integrand at random points (rather than on a regular grid, as with Trapezoid or Simpson).
- Random walk solution to a Poisson problem
- Given a probability distribution of inputs to some problem, estimate probability distribution of output.

  Sensitivity analysis

  Uncertainty quantification
- Simulate processes that have random data or forcing.

Classical quadrature

Midpoint rule in 1 dimension:

$$\int_a^b f(x)\,dx \approx h \sum_{i=1}^{n} f(x_i)$$

There are n terms in sum and error is O(h^2) = O$(1/n^2)$

Midpoint rule in 2 dimensions:

$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2)\,dx_1\,dx_2 \approx h^2 \sum_{j=1}^{n} \sum_{i=1}^{n} g(x_1^{[i]}, x_2^{[j]})$$

There are $N = n^2$ terms in sum and accuracy is
O(h^2) = O$(1/n^2)$ = O$(1/N)$

Classical quadrature

Midpoint rule in 20 dimensions:

$$\int_{a_{20}}^{b_{20}} \cdots \int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2, \ldots, x_{20})\, dx_1\, dx_2 \cdots dx_{20}$$

$$\approx h^{20} \sum_{k=1}^{n} \cdots \sum_{j=1}^{n} \sum_{i=1}^{n} g(x_1^{[i]}, x_2^{[j]}, \ldots, x_{20}^{[k]})$$

There are $N = n^{20}$ terms in sum and accuracy is
O(h^2) = O$(1/n^2)$ = O$((1/N)^{1/10})$

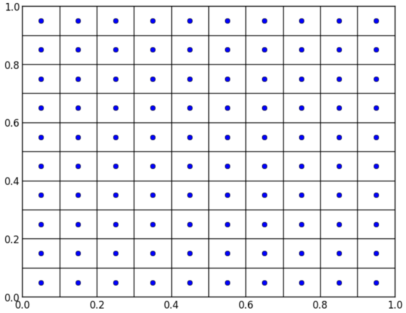
Note: with only $n = 10$ points in each direction, $N = 10^{20}$.

On 1 GFlop computer, would take 10^{11} seconds > 3000 years to compute sum and get accuracy $\approx 1/n^2 = 0.01$.

Also each evaluation of g might be expensive!

Classical quadrature

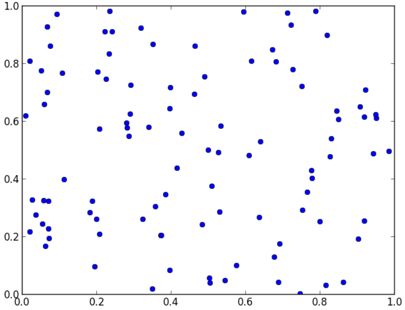
$N = 100$ points in two space dimensions for Midpoint:



$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2)\, dx_1\, dx_2 \approx h^2 \sum_{j=1}^{n} \sum_{i=1}^{n} g(x_1^{[i]}, x_2^{[j]})$$

Monte Carlo integration

$N = 100$ random points in the same 2-dimensional region:



$$\int_{a_2}^{b_2} \int_{a_1}^{b_1} g(x_1, x_2)\, dx_1\, dx_2 \approx \frac{V}{N} \sum_{k=1}^{N} g(x_1^{[k]}, x_2^{[k]})$$

$V = (b_2 - a_2)(b_1 - a_1)$ is volume.

Monte Carlo integration

Accuracy: With N random points, error is O$(1/sq\overline{rt}(N)$

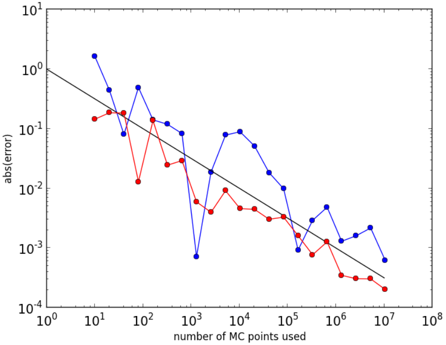
This is true independent of the number of dimensions!

In 20 dimensions, if $g$is smooth then can expect error ≈ 0.01 with $N = 10000$. (vs. $N = 10^{20}$ for Midpoint.)

Log-log plot of errors with Monte Carlo

Black line: $1/\sqrt{N} = N^{-1/2}$.

Note that $E(N) = C/\sqrt{N} \implies \log(E(N)) = \log(C) - \frac{1}{2}\log(N)$



Red points: For an integral in 2 dimensions

Blue points: For an integral in 20 dimensions

Pseudo-Random number generators

Hard to generate a truly random number on the computer.

Instead generally use pseudo-random number generators that produce a sequence of numbers by some deterministic formula, but designed so that numbers generated are approximately distributed according to desired distribution.

Pseudo-Random number generators

Hard to generate a truly random number on the computer.

Instead generally use pseudo-random number generators that produce a sequence of numbers by some deterministic formula, but designed so that numbers generated are approximately distributed according to desired distribution.

Linear congruential generator:

$$X_{n+1} = aX_n + c \bmod m$$

e.g. from *Numerical Recipes*:

$$a = 1664525,\ c = 1013904223,\ m = 2^{32}$$

Requires a seed X_0 to get started.

Pseudo-Random number generators

In Python: Plot of 100 random points was generated using...

```
from numpy.random import RandomState
random_generator = RandomState(seed=55)
r = random_generator.uniform(0., 1., size=200)
plot(r[::2],r[1::2],’bo’)
```

Pseudo-Random number generators

In Python: Plot of 100 random points was generated using...

```
from numpy.random import RandomState
random_generator = RandomState(seed=55)
r = random_generator.uniform(0., 1., size=200)
plot(r[::2],r[1::2],'bo')
```

Initializing with `seed=None` will use a “random” seed.

Specifying a seed makes it possible to reproduce the same results later.

Pseudo-Random number generators

In Fortran:

```
integer, dimension(:), allocatable :: seed

! determine how many seeds needed:
call random_seed(size = nseed)
allocate(seed(nseed))

seed = ...        ! array of integers

call random_seed(put = seed)
deallocate(seed)
```

Pseudo-Random number generators

To reduce to a single seed1:

```
if (seed1 == 0) then
    ! randomize the seed: not repeatable
    call system_clock(count = clock)
    seed1 = clock
  endif

do i=1,nseed
    seed(i) = seed1 + 37*(i-1)
    enddo
```

Pseudo-Random number generators

To generate n random numbers, uniformly distributed in $[0, 1]$:

```
real(kind=4), allocatable :: r(:)
allocate(r(n))
call random_number(r)

r_ab = a + r*(b-a)    ! uniform in [a,b]
```

Pseudo-Random number generators

To generate n random numbers, uniformly distributed in $[0, 1]$:

```
real(kind=4), allocatable :: r(:)
allocate(r(n))
call random_number(r)

r_ab = a + r*(b-a)    ! uniform in [a,b]
```

Note: More efficient in general to call `random_number` once for array of length n rather than n times in succession, but same sequence of numbers will be generated.

State at end of one call is used at start of next call!

Pseudo-Random number generators in parallel

With OpenMP...

State is changed whenever any thread calls `random_number`.

Different threads share same global state.

(Should be thread safe, but can't generate in parallel.)

```
real(kind=4) :: r, x(100)

!$omp parallel do private(r)
do i=1,100
    call random_number(r)
    x(i) = r
    enddo
```

Should produce same set of random numbers but may not end up in same order!

Pseudo-Random number generators in parallel

With MPI... (Processes cannot share the state)

If each process initializes with same seed, then each process will generate the same sequence of random numbers

```
    call random_number(r)
```

Will produce the same `r` on each process.

Pseudo-Random number generators in parallel

With MPI... (Processes cannot share the state)

If each process initializes with same seed, then each process will generate the same sequence of random numbers

```
    call random_number(r)
```

Will produce the same `r` on each process.

This might not be what you want, e.g. if splitting up Monte Carlo integration between processes — want each to sample a different set of points on each process.

Pseudo-Random number generators in parallel

With MPI... (Processes cannot share the state)

If each process initializes with same seed, then each process will generate the same sequence of random numbers

```
    call random_number(r)
```

Will produce the same `r` on each process.

This might not be what you want, e.g. if splitting up Monte Carlo integration between processes — want each to sample a different set of points on each process.

Would need to seed differently on each Process, e.g.

```
  seed(i) = seed1 + 37*(i-1) + 97*proc_num
```

Monte Carlo solution of Poisson problem

Suppose we want to compute an approximate solution to

$$u_{xx} + u_{yy} = 0 \qquad \text{with } u \text{ given on boundary}$$

at a single point (x_0, y_0).

Finite difference approach: Discretize domain and solve linear system for approximations U_{ij} at all points on grid.

Monte Carlo solution of Poisson problem

Suppose we want to compute an approximate solution to

$$u_{xx} + u_{yy} = 0 \qquad \text{with } u \text{ given on boundary}$$

at a single point (x_0, y_0).

Finite difference approach: Discretize domain and solve linear system for approximations U_{ij} at all points on grid.

Instead can take a random walk starting at (x_0, y_0) and evaluate u at the first boundary point the walk reaches.

Do this N times and average all the values obtained.

Monte Carlo solution of Poisson problem

Suppose we want to compute an approximate solution to

$$u_{xx} + u_{yy} = 0 \qquad \text{with } u \text{ given on boundary}$$

at a single point (x_0, y_0).

Finite difference approach: Discretize domain and solve linear system for approximations U_{ij} at all points on grid.

Instead can take a random walk starting at (x_0, y_0) and evaluate u at the first boundary point the walk reaches.

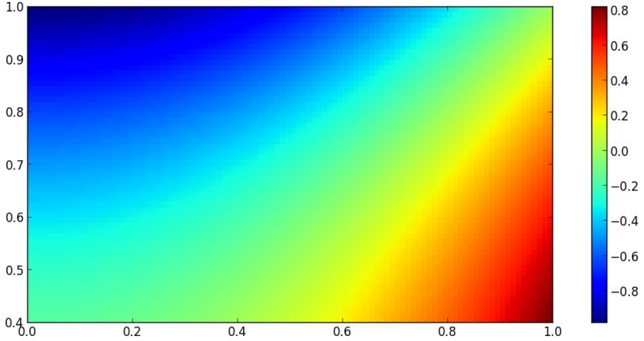
Do this N times and average all the values obtained.

This average converges to $u(x_0, y_0)$ with rate $1/sqrt(N)$

Monte Carlo solution of Poisson problem

$u_{xx} + u_{yy} = 0$ with solution $u(x, y) = x^2 - y^2$.

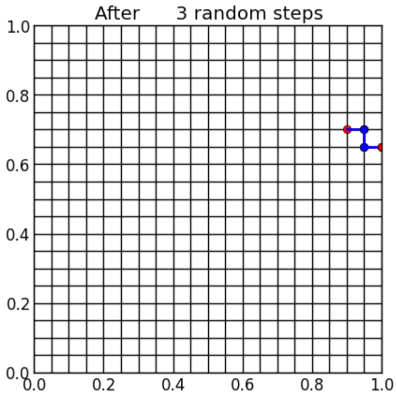
Estimate solution at $(x_0, y_0) = (0.9, 0.7)$ where $u(x_0, y_0) = 0.32$.



Random walk on a lattice

$u_{xx} + u_{yy} = 0$ with solution $u(x, y) = x^2 - y^2$.

Estimate solution at $(x_0, y_0) = (0.9, 0.7)$ where $u(x_0, y_0) = 0.32$.

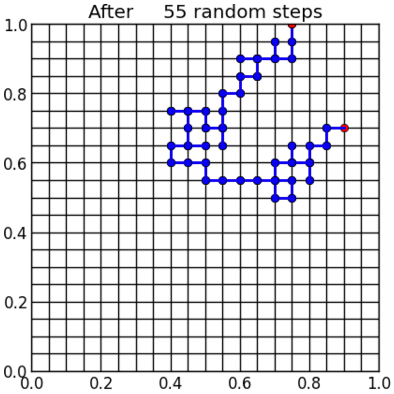


Hit boundary where u = 0.577500

Random walk on a lattice

$u_{xx} + u_{yy} = 0$ with solution $u(x, y) = x^2 - y^2$.

Estimate solution at $(x_0, y_0) = (0.9, 0.7)$ where $u(x_0, y_0) = 0.32$.



Hit boundary where u = -0.437500

Random walk on a lattice

Strategy:

Start at (x_0, y_0).

Each step, move to one of 4 neighbors, choosing with equal probability.

If $0 \leq r \leq 1$ is a uniformly distributed random number then decide based on:

$0 \leq r < 0.25 \Longrightarrow$ move left

$0.25 \leq r < 0.5 \Longrightarrow$ move right

$0.5 \leq r < 0.75 \Longrightarrow$ move down

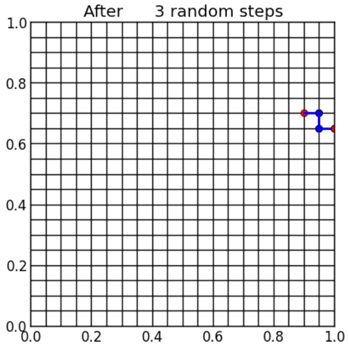
$0.75 \leq r \leq 1.0 \Longrightarrow$ move down

Random walk on a lattice

Why does this work? Let E_{ij} be expected value of boundary value reached if starting at grid point (i,j).

Then $E_{ij} = \frac{1}{4}(E_{i-1,j} + E_{i+1,j} + E_{i,j-1} + E_{i,j+1})$

The same equation as finite difference method for Poisson!



Hit boundary where u = 0.577500

Discuss Python, Fortran and parallelised MPI codes