Outline:

- Monte Carlo methods
- Random number generators
- Monte Carlo integrators
- Random walk solution of Poisson problem
Part of Final Project will be available tomorrow.
Announcements

Part of Final Project will be available tomorrow.

No Class on Monday, June 2
Announcements

Part of Final Project will be available tomorrow.

No Class on Monday, June 2

Wednesday, June 4: Please come for course evaluations.
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.
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 accuracy is \( O(h^2) = O(1/n^2) \)
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 accuracy 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:

\[
\int_{a_1}^{b_1} \int_{a_2}^{b_2} \cdots \int_{a_{20}}^{b_{20}} g(x_1, x_2, \ldots, x_{20}) \, dx_1 \, dx_2 \cdots dx_{20} \\
\approx h^{20} \sum_{k=1}^{n} \cdots \sum_{j=1}^{n} \sum_{i=1}^{n} g(x_1^{[i]}, x_2^{[j]}, \ldots, x_{20}^{[k]})
\]

There are \( N = n^{20} \) terms in sum and accuracy is

\( O(h^2) = O(1/n^2) = O((1/N)^{1/10}) \)
Midpoint rule in 20 dimensions:

\[
\int_{a_1}^{b_1} \cdots \int_{a_2}^{b_2} \int_{a_20}^{b_20} g(x_1, x_2, \ldots, x_{20}) \, dx_1 \, dx_2 \cdots \, dx_{20}
\]

\[
\approx h_1^{20} \sum_{k=1}^{n} \cdots \sum_{j=1}^{n} \sum_{i=1}^{n} g(x_1^i, x_2^j, \ldots, x_{20}^k)
\]

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 \( \approx 1/n^2 = 0.01 \).
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!
High dimensions might arise from many parameters...

**Example:** Solve chemical kinetics equations $u'(t) = F(u(t))$ for system with 20 reacting species and “mass action kinetics” in a stirred tank (so concentrations vary with time but not space).
High dimensions might arise from many parameters...

**Example:** Solve chemical kinetics equations $u'(t) = F(u(t))$ for system with 20 reacting species and “mass action kinetics” in a stirred tank (so concentrations vary with time but not space).

Need initial concentrations, e.g.

$$u_1(0) = x_1, \quad u_2(0) = x_2, \quad \ldots, \quad u_{20}(0) = x_{20}.$$ 

Suppose we want to determine $u_{15}(T) = g(x_1, x_2, \ldots, x_{20})$. 
High dimensions might arise from many parameters...

**Example:** Solve chemical kinetics equations $u'(t) = F(u(t))$ for system with 20 reacting species and “mass action kinetics” in a stirred tank (so concentrations vary with time but not space).

Need initial concentrations, e.g.

$$u_1(0) = x_1, \quad u_2(0) = x_2, \ldots, \quad u_{20}(0) = x_{20}.$$ 

Suppose we want to determine $u_{15}(T) = g(x_1, x_2, \ldots, x_{20})$.

Suppose initial conditions are not known exactly, but we know

$$a_1 \leq x_1 \leq b_2, \quad \ldots, \quad a_{20} \leq x_{20} \leq b_{20}.$$
Example: Solve chemical kinetics equations \( u'(t) = F(u(t)) \) for system with 20 reacting species and “mass action kinetics” in a stirred tank (so concentrations vary with time but not space).

Need initial concentrations, e.g.

\[
u_1(0) = x_1, \quad u_2(0) = x_2, \quad \ldots, \quad u_{20}(0) = x_{20}.
\]

Suppose we want to determine \( u_{15}(T) = g(x_1, x_2, \ldots, x_{20}) \).

Suppose initial conditions are not known exactly, but we know

\[
a_1 \leq x_1 \leq b_2, \quad \ldots, \quad a_{20} \leq x_{20} \leq b_{20}.
\]

We might want to estimate the expected value of \( u_{15}(T) \)

\[
= \frac{1}{\text{Volume}} \int_{a_2}^{b_2} \cdots \int_{a_1}^{b_1} g(x_1, x_2, \ldots, x_{20}) \, dx_1 \, dx_2 \cdots \, dx_{20}.
\]
$N = 100$ 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.
Accuracy: With $N$ random points, error is $O(1/\sqrt{N})$.

This is true independent of the number of dimensions!
Monte Carlo integration

Accuracy: With $N$ random points, error is $O(1/\sqrt{N})$.

This is true independent of the number of dimensions!

In 20 dimensions, if $g$ is smooth than can expect error $\approx 0.01$ with $N = 10000$. (vs. $N = 10^{20}$ for Midpoint.)
Log-log plot of errors with Monte Carlo

Black line: \( \frac{1}{\sqrt{N}} = N^{-1/2} \).

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.

Linear congruential generator:

\[ X_{n+1} = aX_n + c \mod m \]

e.g. from Numerical Recipes:

\[ a = 1664525, \quad c = 1013904223, \quad m = 2^{32} \]

Requires a seed \( X_0 \) to get started.
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:

\[ X_{n+1} = aX_n + c \mod m \]

e.g. from *Numerical Recipes*:

\[ a = 1664525, \quad c = 1013904223, \quad m = 2^{32} \]

Requires a seed \( X_0 \) to get started.
In Python: Plot of 100 random points was generated using...

```python
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...

```python
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.
In Fortran:

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

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

seed = ... ! array of integers

call random_seed(put = seed)
deallocate(seed)
```
Pseudo-Random number generators

To reduce to a single seed1:

\[
\text{if (seed1 == 0) then}
\]
\[
! \text{ randomize the seed: not repeatable}
\]
\[
\text{call system\_clock (count = clock)}
\]
\[
\text{seed1 = clock}
\]
\[
\text{endif}
\]

\[
\text{do i=1,nseed}
\]
\[
\text{seed}(i) = \text{seed1} + 37 \times (i - 1)
\]
\[
\text{enddo}
\]
To generate \( n \) random numbers, uniformly distributed in \([0, 1]\):

\[
\text{real(kind=4), allocatable :: } r(:) \\
\text{allocate}(r(n)) \\
\text{call random_number}(r) \\
\]

\[
r_{ab} = a + r \times (b-a) \quad ! \text{uniform in } [a,b]
\]
Pseudo-Random number generators

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

```fortran
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.)

```fortran
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

\[
\text{call random_number}(r)
\]

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

\[
call \text{ random\_number}(r)
\]

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.

\[
\text{seed}(i) = \text{seed1} + 37 \ast (i-1) + 97 \ast \text{proc\_num}
\]
Suppose we want to compute an approximate solution to

\[ u_{xx} + u_{yy} = 0 \]

with \( u \) 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.
Suppose we want to compute an approximate solution to

\[ u_{xx} + u_{yy} = 0 \quad \text{with } u \text{ given on boundary} \]

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.
Suppose we want to compute an approximate solution to

\[ u_{xx} + u_{yy} = 0 \quad \text{with } u \text{ given on boundary} \]

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
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 \(u(x_0, y_0) = 0.32.\)
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:

\[
\begin{align*}
0 \leq r < 0.25 & \quad \Rightarrow \quad \text{move left} \\
0.25 \leq r < 0.5 & \quad \Rightarrow \quad \text{move right} \\
0.5 \leq r < 0.75 & \quad \Rightarrow \quad \text{move down} \\
0.75 \leq r \leq 1.0 & \quad \Rightarrow \quad \text{move down}
\end{align*}
\]
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 $u = 0.577500$