Instructors: Prof. Christopher Bergevin (cberge@yorku.ca)

Schedule: Lecture: MWF 11:30-12:30 (CLH M)

Website: http://www.yorku.ca/cberge/2030W2018.html
Challenge

- Write a code to use a random # generator to estimate pi
Ex. Estimating \( \pi \)

```matlab
% ### EXestimatePI.m ###      04.22.11 {C. Bergevin}
% Use a random # generator to estimate pi by considering the ratio of areas
% of a circle to a square

clear

% N=1000; % # of points to use

figure(1); clf; hold on; grid on;

% generate array of (uniformly distributed) x and y values
A= 2*rand(N,1)-1;  % x coord.
B= 2*rand(N,1)-1;  % y coord.

% loop thru to test if each coordinate pair is inside or out
Ac= 0;  % indexer
for nn=1:N
    x= A(nn); y= B(nn);
    if (sqrt(x^2+y^2) <= 1), Ac= Ac+1; end
    plot(x,y,'kx');     % also plot for visual purposes
end

% estimate pi as the ratio of the areas
piEST= 4*(Ac/N); fprintf('
 estimate for pi = %g (using %g points) 

',piEST, N);

axis([-1 1 -1 1]);  xlabel('x'); ylabel('y')
% also draw a unit circle
theta= linspace(0,2*pi,100); xC= cos(theta); yC= sin(theta); plot(xC,yC,'r-');
```
Ex. Estimating $\pi$

$N=1000; \quad \% \text{ # of points to use}$

$\Rightarrow$ Simply just using random numbers and ratios of areas

estimate for $\pi = 3.12$ (using 1000 points)
Ex. Estimating $\pi$

wikipedia (Monte Carlo method)
Monte Carlo
Monte Carlo

Stanislaw Ulam

- Interesting bit of 20th century physics history tied back to the Manhattan Project....

The computations for the first-ever numerical experiment were performed by a young woman named Mary Tsingou. After decades of omission, it is time to recognize her contribution.

The Fermi-Pasta-Ulam (FPU) problem, first written up in a Los Alamos report in May 1955, marked the beginning of both a new field, nonlinear physics, and the age of computer simulations of scientific problems. The idea was to simulate the one-dimensional analogue of atoms in a crystal: a long chain of masses linked by springs that obey Hooke’s law (a linear interaction), but with a weak nonlinear term. A purely linear interaction would ensure that energy introduced into a single Fourier vibrational mode always remains in that mode; the nonlinear term allows the transfer of energy between modes. Under certain conditions, the weakly nonlinear system exhibits surprising behavior: The energy does not drift toward the equipartition predicted by statistical physics but periodically returns to the original mode. That highly remarkable result, known as the FPU paradox, shows that nonlinearity is not enough to guarantee the equipartition of energy.

In the 1960s, pursuing the solution of the FPU paradox, Norman Zabusky and Martin Kruskal looked at the problem in real space rather than in Fourier space. They were able to explain the periodic behavior in terms of the dynamics of localized excitations now known as solitons. Those localized, or solitary, waves with the properties of particles (hence the suffix “-on”) have many physical applications and are today
Games of chance....
Monte Carlo Methods

- The purpose of **statistics** is to deal with randomness that is beyond our control

- ‘Monte carlo’ methods turn this around (not that intuitive at first!)

  → *How can we use random numbers (that we create) to our advantage?*

  1. Define a domain of possible inputs.
  2. Generate inputs randomly from a **probability distribution** over the domain.
  3. Perform a **deterministic** computation on the inputs.
  4. Aggregate the results.

- ‘Random’ ↔ **Stochastic** (i.e., non-deterministic, state is determined probabilistically)

- Wide range of applications:
  - simulating aspects of thermodynamics and statistical mechanics  
  - theoretical neuroscience & computational biology
  - statistics
  - artificial intelligence (AI)
  - financial/market modeling (e.g., insurance, options pricing)

  We’ll explore a handful of basic examples relevant to these
How does one generate random numbers?

- Some ‘brute force’ methods exist:
  - roll some die or flip a coin
  - get a ‘bingo cage’
  - count neutrinos or radioactive decays
  - measure background acoustic noise
  - look up a table of ‘random’ numbers

- As odd as it might sound, ideally we’d like a method that has some degree of reliability and/or reproducibility → psuedorandomness

- Can develop some simple computational methods to demonstrate such

Note: We will not linger too much on numerical methods to create random numbers, but instead chiefly rely upon Matlab’s built-in routines rand.m and randn.m
Uses the computer’s clock time to create a **seed**

Such a value to then used in a **mod** function (i.e., modulus after division)

[see wikipedia re Modulo operation]

Ex. \( \text{mod}(5,3)=2 \)  \( \text{mod}(73,6)=1 \)
N = 100; % # of random values to compute
modB = 1000; % base value for mod

→ Flattish distribution, but not all that random per se.... (e.g., temporal correlations over short intervals)
Re-running has different effects (e.g., initial seed matters?). So there are some pluses and minuses....
Uses the computer’s clock time to create an initial seed
Then uses last ‘random’ value to determine the next (i.e., recursive)

% Generate a rough pseudo-random number distribution using the computer clock as the initial 'seed', then use the last number for the next
clear
% ---
N= 20;     % # of random values to compute
modB= 1000; % base value for mod
% ---
c= clock;   temp1= c(6);    % use computer clock as the initial 'seed'
% loop to generate random values
for nn=1:N
    temp2= mod(modB,temp1);     % compute the mod
    randVal(nn)= ceil(temp2)-temp2; % normalize to [0,1] interval
    temp1= rem(temp2*10,1);     % use (last few digits of) last value to 'reset' seed
end
figure(1); clf;
subplot(211);
hist(randVal,20); xlim([0 1]);
ylabel('Occurrences'); title(['Pseudo-random number over [0,1] distribution (',num2str(N),', values)']);
subplot(212)
plot(randVal,'.-'); xlabel('Iterate');
Pseudo-random number over [0,1] distribution (20 values)

Iterate
This routine seems to quickly converge (not a good thing here!) and thus seems even worse than our last one....
Note (again): We will not linger too much on numerical methods to create random numbers, but instead chiefly rely upon Matlab’s built-in routines \texttt{rand.m} and \texttt{randn.m}.
Figure 1.5.2. *Plots of pairs of* $10^6$ *random uniform deviates* $(U_i, U_{i+1})$ *such that* $U_i < 0.0001$. *Left: MATLAB 4; Right: MATLAB 5.*
Updating Your Random Number Generator Syntax

Description of the Former Syntaxes

In earlier versions of MATLAB®, you controlled the random number generator used by the `rand` and `randn` functions with the `seed`, `state` or `twister` inputs. For example:

```matlab
rand('seed',sd)
randn('seed',sd)
rand('state',s)
randn('state',s)
rand('twister',5489)
```

These syntaxes referred to different types of generators, and they will be removed in a future release for the following reasons:

- The terms `seed` and `state` are misleading names for the generators.
- All of the former generators except `twister` are flawed.
- They unnecessarily use different generators for `rand` and `randn`.

To assess the impact this change will have on your existing code, execute the following commands at the start of your MATLAB session:

```matlab
warning('on','MATLAB:RandStream:ActivatingLegacyGenerators')
warning('on','MATLAB:RandStream:ReadingInactiveLegacyGeneratorState')
```

Description of Replacement Syntaxes

Use the `rng` function to control the shared generator used by `rand`, `randn`, `randi`, and all other random number generation functions like `randperm`, `sprand`, and so on. To learn how to use the `rng` function to replace the former syntaxes, take a few moments to understand what the former syntaxes did. This should help you to see which new `rng` syntax best suits your needs.

The first input to the former syntaxes of `rand(Generator,s)` or `randn(Generator,s)` specified the type of the generator, as described here.
Ex. Integrals (Method 1)

Riemann sums motivated the idea that we can treat integrals as adding up lots of areas (i.e., the total area ‘under’ the curve).

Our example earlier estimating π suggested a means to estimate ratios of areas using random numbers.

→ Why not apply the same basic idea to estimating integrals?
Consider what are the relevant ‘areas’ here?
Let’s also consider another idea in parallel.

Remember that a definite integral is related to the average value of the integrand over the interval of integration.

Suppose then that we have an array of $N$ random numbers $x_i$ uniformly distributed over $[a,b]$. Then the average value is just:

$$
\langle f \rangle_N = \frac{1}{N} \sum_{i=1}^{N} f(x_i)
$$

$$
\int_{a}^{b} f(x) \, dx \approx (b - a) \frac{1}{N} \sum_{i=1}^{N} f(x_i)
$$

→ So we have another ‘recipe’ for estimating a definite integral....
% ### EXintegrateMC1.m ###     10.08.14

% clear
% -----------------------------
N= 100;      % number of 'data' points
Np= 1000;     % number of random samples to compute
intLimts= [-3 8];   % limits to evaluate integral over
% ===
% user specifies function to integrate
% generic model parameters
AA= 2;      % model parameters
BB= 2;
CC= 0.5;
% function
z= @(x) AA*sin(x)-3;                 % sinusoid
z= @(x) AA*exp(-x)-BB;              % decaying exponential
% -------------------------------------------------
% user specifies function to integrate
x=linspace(intLimts(1),intLimts(2),N); % create x-values
y= z(x);        % determine associated y-values
% ***
figure(1); clf;     % plot the function
plot(x,y,'b-','LineWidth',2);    hold on; grid on; xlabel('x');  ylabel('y'); % ***

% Integrate via built-in Matlab function trapz.m
a1 = trapz(x,y);
msg= ['Integral calculated by trapz.m (i.e., Riemann sums)= ' num2str(a1,10)]; disp(msg);
% ****************
% Integrate via a brute-force Monte carlo routine
% 1. Create a bounding box about the function
if (min(y)>=0), yL=0;   else yL= min(y); end % handle case if entire curve sits above x-axis
if (max(y)<=0), yH=0;   else yH= max(y); end % handle case if entire curve sits below x-axis
yD= yH- yL;
xL= min(x); xH= max(x); xD= xH-xL;
% 2. Create uniform distribution of points inside that box; then for each
% point determine whether it falls between the curve and x-axis (if so, % counter is updated +1)
cntP= 0; cntN= 0;
for nn=1:Np
    xT= xD*rand(1)- abs(xL);
    yT= yD*rand(1)- abs(yL);
    zT= z(xT);  % evaluate the function at all the random x-values
    storeP(nn,:)= [xT yT];   % store away points (for visualization)
    if (yT>=0)
        if(yT<=zT), cntP= cntP+1; end
    end
    if (yT<0)
        if (yT>=zT), cntN= cntN+1;   end
    end
zComp(nn)= zT;  % compile all values of evaluated function
end
% estimate area as ratio determined via flagged counts
areaMC= ((cntP-cntN)/Np)*yDxD; % diff. here accounts for area being above or below x-axis
msg = ['Integral calculated via Monte carlo Method 1 (area ratios)= ' num2str(areaMC)]; disp(msg);
plot(storeP(:,1),storeP(:,2),'r+'); title('Method 1 (area ratios)') % visualize the points used
% also report back estimate via average value
msg = ['Integral calculated via Monte carlo Method 2 (average value)= ' num2str(mean(zComp)*diff(intLimts))]; disp(msg);
Integral calculated by trapz.m (i.e., Riemann sums) = 8.017256207
Integral calculated via Monte carlo Method 1 (area ratios) = 8.0003
Integral calculated via Monte carlo Method 2 (average value) = 8.0058
Integral calculated by trapz.m (i.e., Riemann sums) = -34.68724693
Integral calculated via Monte carlo Method 1 (area ratios) = -33.3834
Integral calculated via Monte carlo Method 2 (average value) = -34.1728
Ex. Integrals

➢ Can re-run a few times.....

➔ Each run yields a slightly different value....... (except for trapz.m! why so stable?)

Integral calculated by trapz.m (i.e., Riemann sums)= -34.68724693
Integral calculated via Monte carlo Method 1 (area ratios) = -33.3834
Integral calculated via Monte carlo Method 2 (average value) = -34.1728

Integral calculated by trapz.m (i.e., Riemann sums)= -34.68724693
Integral calculated via Monte carlo Method 1 (area ratios) = -35.1434
Integral calculated via Monte carlo Method 2 (average value) = -34.7353

Integral calculated by trapz.m (i.e., Riemann sums)= -34.68724693
Integral calculated via Monte carlo Method 1 (area ratios) = -35.1434
Integral calculated via Monte carlo Method 2 (average value) = -34.7353

Integral calculated by trapz.m (i.e., Riemann sums)= -34.68724693
Integral calculated via Monte carlo Method 1 (area ratios) = -35.9683
Integral calculated via Monte carlo Method 2 (average value) = -35.8865
So the distributions looks like Gaussians..... (this is telling us something important!)

FIGURE 4.12 Distributions of 10,000 Monte Carlo estimates of the integral $\int_0^1 e^x \, dx$. On the left, each integral was evaluated with $N = 100$ points; on the right, with $N = 400$ points.
Simple code to see if we can ‘create’ a normal distribution.....

```matlab
% ### EXgaussian1.m ###
clear
%  -----------
M= 1000;        % # of (uniformly distributed) random #s to average
N=1000;         % # of repeats (i.e., how many averages to compute) for histogram
binN= 20;      % # of bins for histogram
%  -----------
figure(1); clf; hold on; grid on;
% +++
% loop thru to compute the N averages (each loop deals with the M random #s)
for nn=1:N
    xR= rand(M,1);  % determine array of M random #s
    mu(nn)= mean(xR);   % compute/store mean value
end
% +++
[jj,kk]=hist(mu,binN);   % determine histogram distribution
bar(kk,jj);             % plot the histogram (as a bar plot)
```

→ Simply determines a group of uniformly distributed numbers, then averages them. Subsequently, we keep track of the those mean values and plot as a ‘histogram’
Ex. Making a Gaussian distribution

\[ M = 1000; \quad \% \text{ # of (uniformly distributed) random #s to average} \]
\[ N = 1000; \quad \% \text{ # of repeats (i.e., how many averages to compute) for histogram} \]

So within a given sample, the \( M \) points are uniformly distributed...

\[ \rightarrow \text{This sort of observation demonstrates the notion of a normal distribution and is ultimately telling us something important about the nature of the underlying probability distribution!} \]
Gaussian distributions

- Gaussian or ‘normal’ distributions (i.e., the ‘bell-shaped curve’) arise throughout many contexts in physics and engineering applications.

- The parent distribution has a relatively simple analytic form:
  \[
  f(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)
  \]

  Note: This expression can be generalized further and scaled.

- This equation contains many of the quantities that we have come across already (e.g., the mean, standard deviation) and forms the basis for many common statistical measures (e.g., 95% confidence intervals).

- Standard deviation (STD, or σ here) tells you what fraction of the area lays underneath the curve. For example:
  - 2/3 of the area is contained within +/- σ
  - 95% of the area is within +/- 2σ
  - Hence σ is the basis for: confidence intervals, standard error, Student’s t-tests, the coefficient of variation, .....
**Gaussian distributions**

*Be careful!*

→ A normal distribution doesn’t necessarily tell you where an individual measurement lies, but the mean value across a set of measurements (i.e., what happens for compiling across REPEATED measurements)