Random Numbers and Monte Carlo Methods

Random Number Methods

The integration methods discussed so far all are based upon making a polynomial approximations to the integrand. Another class of numerical methods relies upon using random numbers. These methods have come to be known under the general rubric **Monte Carlo methods,** after the famous gambling casino. Before discussing Monte Carlo integration, we must digress to learn about random numbers.

Random Number Generators

Any standard random number generator produces a set of uniformly distributed (equal probability) numbers on some interval. For the interval [a,b] we define

   P(r)dr = probability of generating a random number in the
            interval dr near r

For this probability idea to make sense we must have

$$\int_a^b P(r)dr = 1 =$$ total probability of generating a random number

In the case of a uniform distribution, P(r) = constant and hence

$$P(r) = \frac{1}{b-a}$$

All such numerical random number generators give a sequence of numbers with a large repeat cycle on some interval. A good generator will have a very large repeat cycle($10^8 - 10^9$ non-repeating numbers).

In MATLAB, the rand(n,m) function generates uniformly distributed pseudo-random numbers. We illustrate the uniform distribution of 100,000 random numbers below:
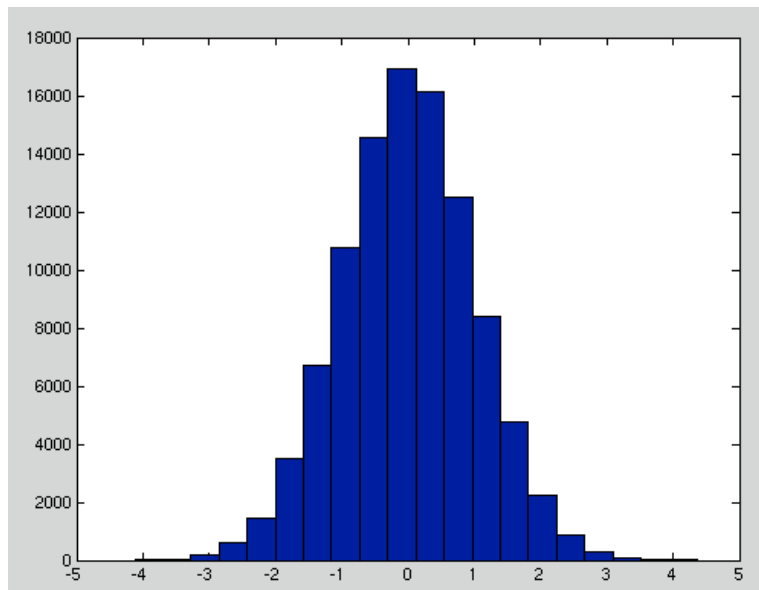
```
>> hist(rand(1,100000),10)
```
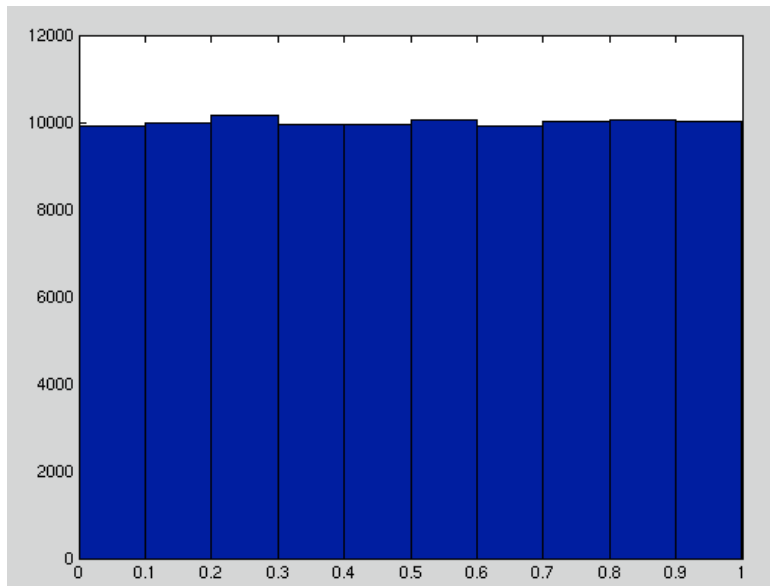
which obviously shows the uniform distribution on the interval [0,1].

MATLAB also generate normally (Gaussian) distributed random numbers using the randn function.

```
>> hist(randn(1,100000),20))
```


which is obviously a normal or gaussian distribution.

These two methods are examples of "**simple sampling**" techniques. In many problems we need to use probability distributions that reflect the appropriate physics of the system under study. In this case we use a different technique called "**importance sampling**".

The algorithm for generating random numbers with a specified distribution(not uniform) goes this way. Suppose that we have a set of uniformly distributed numbers (as above) in the interval [0,1]. Then the rule
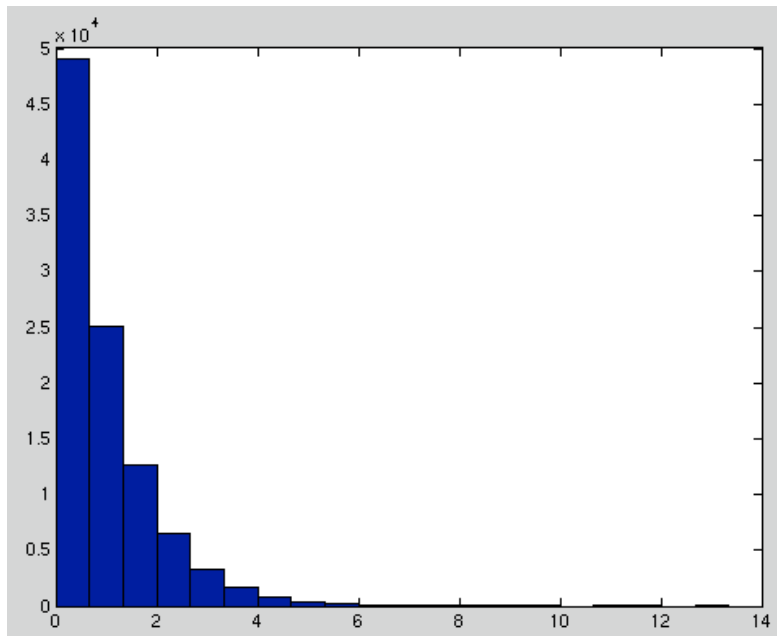
$$r_j = \int_0^{x_j} dx' w(x')$$

will generate a set of numbers $\{x_i\}$ distributed according to the rule w(x). For example, suppose $w(x) = e^{-x}$ 0 ≤ x ≤ 4. Then we obtain

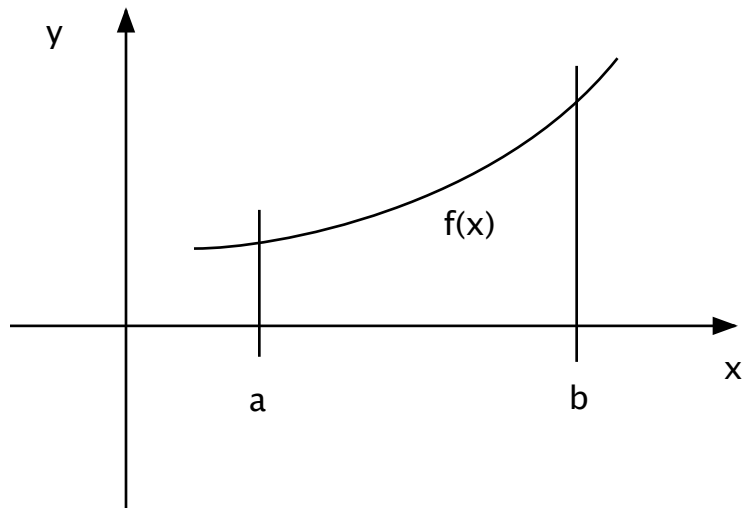$$r_j = 1 - e^{-x_j} \rightarrow x_j = -\log(1 - r_j)$$

which is the desired result. This is illustrated in the plot below:

```
>> hist((-log(1-rand(1,100000))),20)
```


Integration using Importance Sampling (Monte Carlo Method)

Now consider a function to be integrated, as shown below:

The integral is just the area under the curve. The width of the
interval (b-a) times the average value of the function is also
the value of the integral, that is,

$$\int_a^b f(x)dx = (b-a)f_{average} = (b-a)\langle f \rangle$$

So if we had some independent way of calculating the average
value of the integrand, then we could evaluate the integral.

That is where we can use random numbers.

Imagine that we have a list of random numbers, x_j, uniformly
distributed between a and b.

To calculate the function average, we simply evaluate f(x) at
each of the randomly selected points, and divide by the number
of points:

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

As the number of points used in calculating the average
increases, $\langle f \rangle_N$ approaches the true average value, $\langle f \rangle$.

Therefore, as a numerical approximation we can write

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

Alternatively, we can look at this so-called Monte Carlo
integration method in the following way:

To integrate the function f(x) over the interval [a,b] we

[1] find some value M such that f(x) < M over the interval [a,b]
[2] select a random number x from a uniform distribution over
    the interval [a,b]
[3] select a random number y from a uniform distribution over
    the interval [0,M]
[4] determine if y > f(x) or y ≤ f(x)
[5] repeat this process N times, keeping track of the number of
    times y ≤ f(x) or under the curve (= successes); call the
    total number of successes S.

The estimated probability of success is then

$$\frac{S}{N} = \frac{\text{Area under curve}}{\text{Total area inside rectangle}} = \frac{\int_a^b f(x)dx}{M(b-a)}$$

where the rectangle used is shown in the figure below:
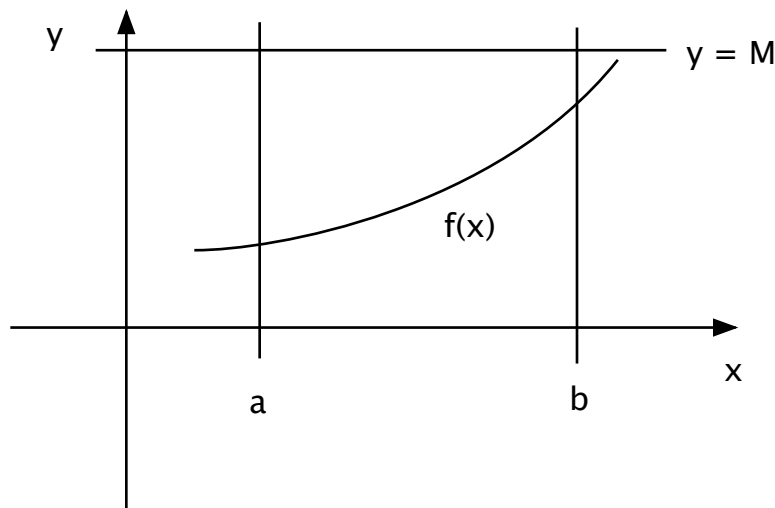
After a number of trials, the value of the integral can be
calculated from the above formula

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

Think about **throwing darts** and counting the number of darts that
land in the area representing the integral.

Your program above only works if the integrand is greater than
or equal to zero everywhere over the range of integration.
Suppose, in fact, that the function f(x) was not always greater
than zero in the interval [a,b] as shown below.
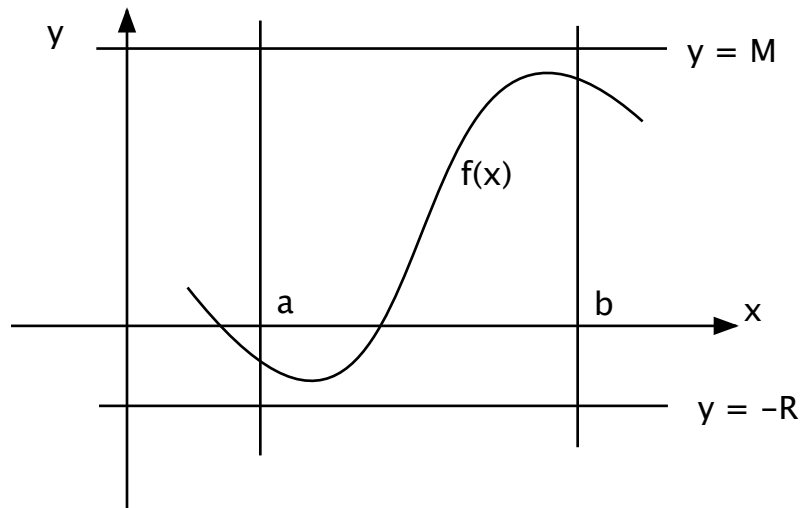
We can modify the Monte Carlo integration method to handle such cases, i.e., fix the problem with f(x) possibly being less than zero as follows. To integrate the function f(x) over the interval [a,b] we

[1] find some value M such that f(x) < M over the interval [a,b]
[2] find some R such that f(x) > - R over the interval [a,b]
[2] select a random number x from a uniform distribution over
    the interval [a,b]
[3] select a random number y from a uniform distribution over
    the interval [-R,M]
[4] determine if y > f(x) or y ≤ f(x)
[5] repeat this process N times, keeping track of the number of
    times y ≤ f(x) or under the curve (= successes); call the
    total number of successes S.

The estimated probability of success is then

$$\frac{S}{N} = \frac{\text{Area under curve}}{\text{Total area inside rectangle}} = \frac{\int_a^b f(x)dx}{(M+R)(b-a)}$$

$$\int_a^b f(x)dx = (M+R)(b-a)\frac{S}{N}$$

This must now be corrected for the fact that the line y = -R has been used as the baseline for the integral instead of the line y = 0. This is accomplished by subtracting the rectangular area R(b-a). The final integral is then

$$\int_a^b f(x)dx = (M+R)(b-a)\frac{S}{N} - R(b-a)$$

The Metropolis Algorithm

Suppose that we want to generate a set of points in some, possibly multidimensional, space of variables X distributed with probability density w(X) (not necessarily uniform). The Metropolis algorithm generates a set of points $X_0, X_1, X_2, \ldots\ldots$ as those visited successively by a random walker moving through the X space. As the walk becomes longer and longer, the points it connects will approximate closely the desired distribution.

The rules for the random walk are as follows:

[1] Suppose that the walker is at a point X_n in the sequence. To generate X_{n+1} it makes a trial step to a new point X_t. This new point can be chosen in any convenient manner, for example, uniformly at random within a multidimensional cube of small side δ about X_n.

[2] This trial step is then "accepted" or "rejected" according to the ratio

$$r = \frac{w(X_t)}{w(X_n)}$$

that is, if r is larger than one, then the step is always accepted (i.e., we put $X_{n+1} = X_t$, while if r is less than one, the step is accepted with probability r.

This latter step is conveniently accomplished by comparing r with a random number η uniformly distributed in the interval and accepting the step if $\eta < r$. If the trial step is not accepted, then it is rejected, and we put $X_{n+1} = X_n$. This generates X_{n+1}, and we may proceed to generate X_{n+2} by the same process. Any arbitrary point X_n can be used to start this random walk.
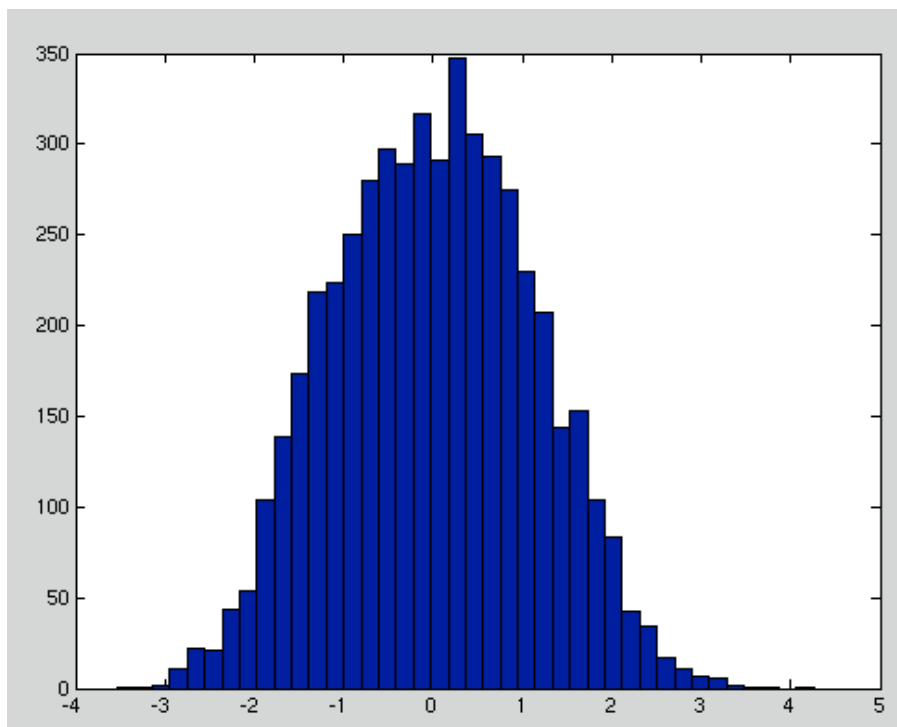
A MATLAB code to use the Metropolis algorithm for the case $w(X) = e^{-0.2X^2}$ looks like:

```
x=[];
X0=0;
delta=4;
naccept=0;
```

```
n=0;
while (naccept < 5000)
   n=n+1;
   XT = X0 + delta*(2*rand(1,1)-1);
   ratio=exp(0.5*(X0^2-XT^2));
   if (ratio > rand)
          x=[x,XT];
          X0=XT;
           naccept=naccept+1;
      end
end
hist(x,40)
```


which clearly reflects the proposed Gaussian distribution.

A good rule is to choose δ so that about 1/3 of the trials is
accepted and to choose X_0 such that w(X) is near a maximum.

Because successive points in this distribution are not
statistically independent of each other, some care must be taken
when choosing a set of points to use from a larger set that has
been generated earlier. Generally, the accepted method is to
choose points separated by some interval, say every k^{th} point,
where k is such that any correlations are washed out.

A simple Metropolis method function is :

```
function z=metropolis(input)
% must have another funct.m file defined
rand('seed',sum(100*clock));
x=input(1);
delta=input(2);
xtrial=x+delta*(2*rand(1)-1);
w=funct(xtrial)/funct(x);
if (rand < w)
    z=xtrial;
else
    z= x;
end
```

The code below tests this function for $w(X)=e^{-0.2X^2}$:

```
zz=zeros(100000)
z=metropolis([0.0,4.0]);
for j=1:100000
 z=metropolis([z,4.0]);
 zz(j)=z
end
hist(zz,40);
```
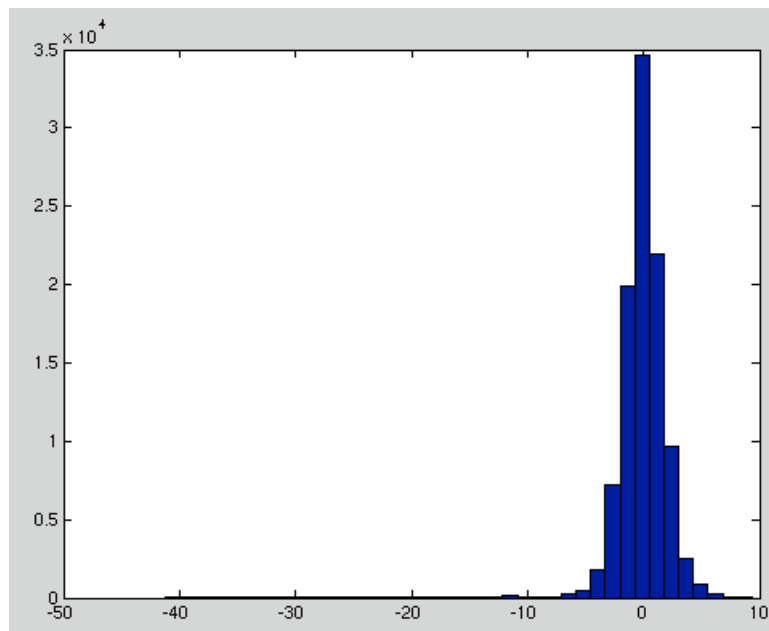

Again this clearly reflects the correct distribution.

Simple Simulation Example Using Random Numbers

Another example of using random numbers to simulate particles in a box with two distinct sides.

Suppose a particle can be at only two positions XR and XL and that

$$w(X) = \frac{n_R}{N}\delta_{X,XL} + \frac{n_L}{N}\delta_{X,XR}$$

that is, the probability of being on a given side of the box is given by the ratio of the number of particles on that side of the box to the total number of particles.

Consider the program below. We have N = 1000 particles in a box. We start with a random fraction of the N particles on the LHS of the box.

If all particles on the LHS, then we send one to the RHS. In all other cases, we then use the Metropolis algorithm to decide whether we decrease the number of particles on the LHS by 1 (increase the number on the RHS by 1) or vice versa. The ratio r in this case is the probability of being on the LHS.
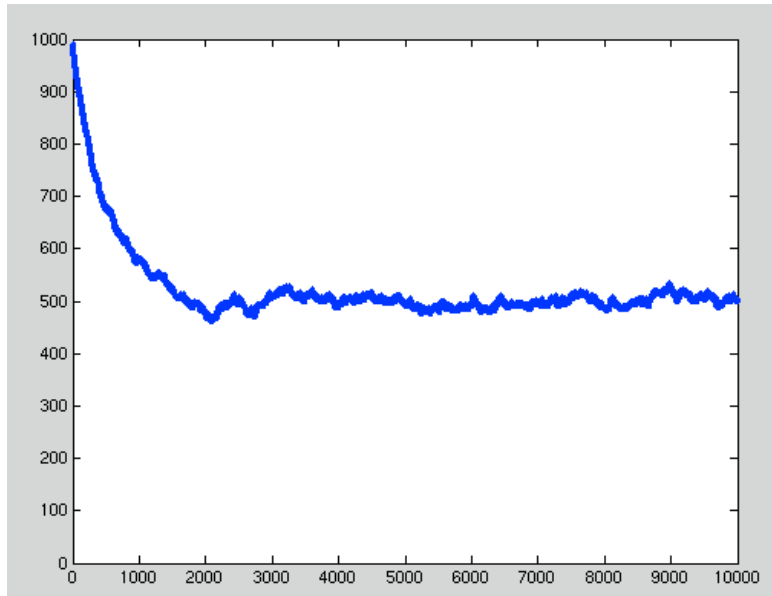
$$ratio = r = w(XL) = \frac{n_L}{N}$$

Consider the program below:

```
% Metropoplis simulation
rand('seed',sum(100*clock));
N=1000;
tmax=10000;
nl=round(rand*N);
t=0;
p=plot(t,nl,'.','EraseMode','none');
axis([0,tmax,0,N]);
while (t <= tmax)
   t=t+1;
   ratio=nl/N;
   if (ratio >= 1)
     nl=nl-1;
   else
     if (rand <= ratio)
       nl=nl-1;
     else
       nl=nl+1;
     end
```

```
    end
    set(p,'XData',t,'YData',nl)
    drawnow
end
```

The result is:

When we run this simulation, we always end up with approximately
1/2 of the particles on each side(the equilibrium configuration)
and the simulation accurately represents the fluctuations
present at equilibrium.

This example illustrates how chance or random motion can
generate deterministic behavior.