Probabilistic Methods

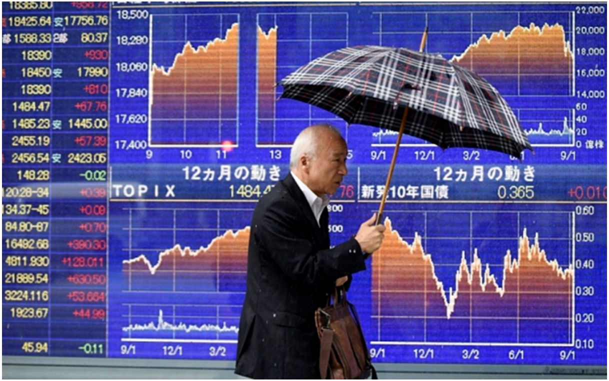
Alvaro Ridruejo (UPM)

October 15, 2018

Table of contents

Motivation: Why Monte Carlo Methods?

Short answer: **UNCERTAINTY**



Very frequently, our knowledge of a system is not complete

Some practical examples about uncertainty

- Planning and operation of a wind generator: *direction and magnitude of wind?*
- Large structure building process: *unexpected delays or extra costs might be caused by problems in soil, supply of materials, strikes, lack of funding,...+ black swans*

Note: a highly improbable event is called a **black swan** (such as the accident of the Fukushima nuclear power plant)

Monte Carlo strategy

Key idea

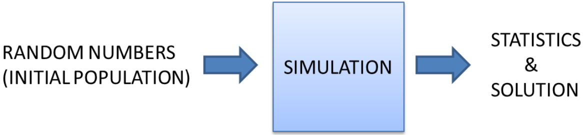
*To transfer **input uncertainties** into **results** in a **quantitative way***

This is done through **random sampling**. For example, we can simulate the winds for the wind generator in a certain moment by using two random numbers. For each day:

- The first number represents the wind speed, from 0 to 140 km/h
- The second number is the angle of the wind direction with respect to North. We can consider numbers in the interval [0º, 360º] or [-180º, 180º]
- Later on, we can perform statistical calculations on both parameters

Monte Carlo strategy

In general, Monte Carlo methods have a common structure:



1. The initial population is generated by using random numbers
2. The evolution of the system can be fully deterministic or include additional random contributions
3. The final solution is obtained from statistical estimations

Some applications

- Finance: market evolution, risk assessment
- Physics: Brownian motion, nuclear processes, diffusion
- Math: evaluation of integrals
- Management: prediction of schedule and cost overruns

the "rand" command in Octave and Matlab

```
rand            % Single random number
rand(1000,1)    % Column vector with 1000 random components
rand(1,12)      % Row vector with 12 random components
rand(25)        % 25x25 random matrix
```

This command produces **uniformly distributed** pseudo-random numbers in the interval [0,1]

- *Pseudo-random* means that they are not perfect random numbers, but a good approximation
- *Uniformly distributed* implies that all numbers in the interval are equally probable
- Generating uniformly distributed random numbers **in another interval [a,b]** can be done through a **linear function** $y = \alpha x + \beta$, imposing $y(0) = a$ and $y(1) = b$

How pseudo-random number generating algorithms work

Computers cannot produce real random numbers, but there are algorithms whose outputs look really random

An algorithm for pseudo-random numbers

$X_{n+1} = (aX_n + b) \pmod{m}$

X_0 is a seed, and a, b and m are constants. It produces a "random" uniform distribution from 0 to $m - 1$.

The constants are critical to have a good quality random number. Its conditions are (Greenberger, 1961):

- *b and m are relatively prime.*
- *$a - 1$ is divisible by all prime factors of m*
- *$a - 1$ is a multiple of 4 if m is a multiple of 4.*

For instance: m = 232, a = 1664525, b = 1013904223.

Testing the quality of random numbers

There are three main tests to assess the quality of random numbers

1. Mean. The mean value of a list of uniformly distributed random numbers in [0,1] should be close to 0.5
2. Histogram. A histogram divides the interval into subintervals (bins) and represents how many numbers lie on each. It is a visual method, and a good generator should display a flat histogram (fluctuations are acceptable)
3. Cumulative plot. Similar to the integral of the histogram. If the numbers are correctly distributed, 25% of numbers should be below 0.25, half of the numbers should be less than 0.5, etc. Then, the line is expected to be a ramp.

Practical example

Run the following code and evaluate the quality of the list of 1000 pseudo-random numbers

```
clear all
global seed = 51;
function r=myRand2()
  global seed
  a=1664525;
  b=1013904223;
  m=4294967296;
  seed = mod(a*seed + b, m);
  r = seed/m;
endfunction
mean = 0;
howMany=1000
for i=1:howMany
        A(i) = myRand2();
         mean = mean + A(i);
end
fprintf("%f\n", mean/howMany);
for i=1:100
 B(i) = 0;
   for j=1:howMany
     if (A(j) < i/100)
       B(i) = B(i) + 1/howMany;
     end
   end
end
figure;
 hist(A,10);
 figure;
 plot(B);
fid=fopen("rand1.txt","w");
fprintf(fid, "%f\n", A);
fclose(fid);
```

Exercise

Set now a=432, b=54534 and m=10000. What happens?

Sample mean integration

- The integral of a function $f(x)$ defined in the interval $[a, b]$ can be written as a Riemann sum:

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

- This allows to do the following approximation

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

- We can also understand this expression as the average area over N rectangles. By generating N random numbers in the interval $[a, b]$, we can evaluate them (height of rectangles), calculate the area of the N rectangles and calculate its mean value.

Sample mean integration

Exercise

Use the method to calculate the integral of $g(x) = x^3 + 1$ in the interval $[0, 1]$ with a) N=100 and b) N=10000 points

Solution:

```
N=100;
a=0;
b=1;
result=0;
for i=1:N;
 ev=(b-a)*(rand^3+1);
 result=result+ev;
end
result=result/N;
fprintf('The integral is %f', result);
```

Exercise

Exercise (home)

Use this method to calculate

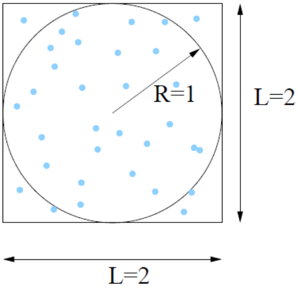
$$\int_1^5 ln(x)dx$$

- Do not forget to generate numbers in the correct interval.
- Exact value ≈ 4.047

Hit-and-miss in 2D

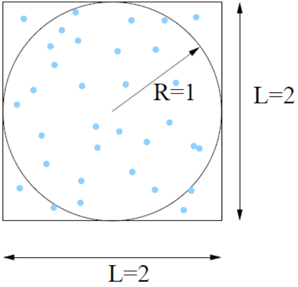
Hit-and-miss methods are used to calculate areas in a very straightforward way. Many points are randomly generated in a region which contains the area of interest. Some points may be inside (*hit*) or outside (*miss*) this area of interest. The hit-over-total ratio is approximately equal to the fraction of areas:

$$\frac{A_{area-of-interest}}{A_{total}} = \frac{n_{inside}}{N_{total}}$$



As a first example, we are going to calculate the area of a circle with radius R=1 (by definition, the value of this area is $A = \pi$). This circle will be centered and contained in a square (L=2). To "throw" our points, we must generate 2 random coordinates (x and y) in the interval [-1,1] per point.

Hit-and-miss in 2D



$$A_{circle} = \frac{n_{inside}}{N_{total}} L^2$$

```
clear all
N=1000;
n=0;
result=0;
for i=1:1:N
 x=2*rand-1;
 y=2*rand-1;
 if (x*x+y*y <= 1.0)
  n=n+1;
 end
end
result=4*n/N;
fprintf('pi is %f\n', result)
```

Hit-and-miss in 2D

Hit-and-miss methods...

- can easily be extended to higher dimensions
- require very little information and are easy to program
- but are NOT very ACCURATE.

How to improve accuracy

There are two strategies to get more accurate estimations

- *To increase the number of points (slow convergence rate)*
- *To calculate the average over several runs*

Hit-and-miss in 6D

An important application is the calculation of multi-dimensional integrals. As an exercise, we can calculate the hypervolume of the unit ball in 6D. Some considerations:

- In $\mathbf{R}^6$, 6 numbers (coordinates) are needed to define a point
- The unit ball is always defined by the condition: radius ≤ 1
- In 6D, the volume of a hypercube of side L is equal to L^6

You can employ $N = 1000$ points first, and later increase to improve the result

Hit-and-miss in 6D

Solution:

```
clear
N=1000;
vol=0.0;
ninside=0;
i=1
r=0.0
for i=1:N
 x=2*rand-1;
 y=2*rand-1;
 z=2*rand-1;
 u=2*rand-1;
 v=2*rand-1;
 w=2*rand-1;
 r=(x*x+y*y+z*z+u*u+v*v+w*w)^(0.5);
 if (r <= 1)
  ninside=ninside+1;
 end
end
vol=64*ninside/N
```

A faster version:

```
clear
tic
N=1000;
vol=0.0;
ninside=0;
i=1;
r2=0.0;
for i=1:N
 s=2*rand(1,6)-1;
 r2=s*s';
 if (r2 <= 1)
  ninside=ninside+1;
 end
end
vol=64*ninside/N
toc
```

Why is this version faster?