Introduction to Programming and Computational Physics

Lesson n.12

An introduction to Monte Carlo method

- Random numbers
- Statistical inference
- Generators

Bibliography

J. E. Gentle: Random Number Generation and Monte Carlo Methods

R.Y. Rubinstein, D.P. Kroese: Simulation and the Monte Carlo Methods (advanced)

W. H. Press, B. P. Flannery, S.A. Teukolsky, W. V. Wetterling: Numerical recipes in C, the art of scientific computing

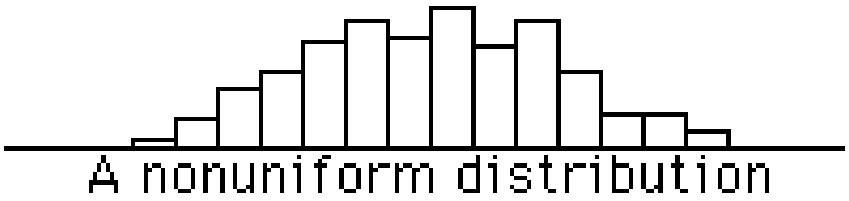
http://www.nrbook.com/a/bookcpdf.php

Uniform distribution

A uniform distribution is one for which the probability of occurrence is the same for all values of X. For example, if a fair dice is thrown, the probability of obtaining any one of the six possible outcomes is 1/6.

Since all outcomes are **equally probable**, the distribution is *uniform*. If a uniform distribution is divided into equally spaced intervals, there will be an equal number of members of the population in each interval

A uniform distribution



A nonuniform distribution

Probability and statistics

Typically the probability is unknown and we want to *infer* it from the experiment: we have the dice!

Over n throws we observe r times the value 1.

$$\overline{p_1} = \frac{r}{n}$$ It is an estimation of the probability.

Frequentistic definition of probability:
Limit of the relative frequency of occurrence. If an event E occurs r times in n trials

$$P(E) = \lim_{n \to \infty} \frac{r}{n} \qquad P(E) \in [0,1]$$

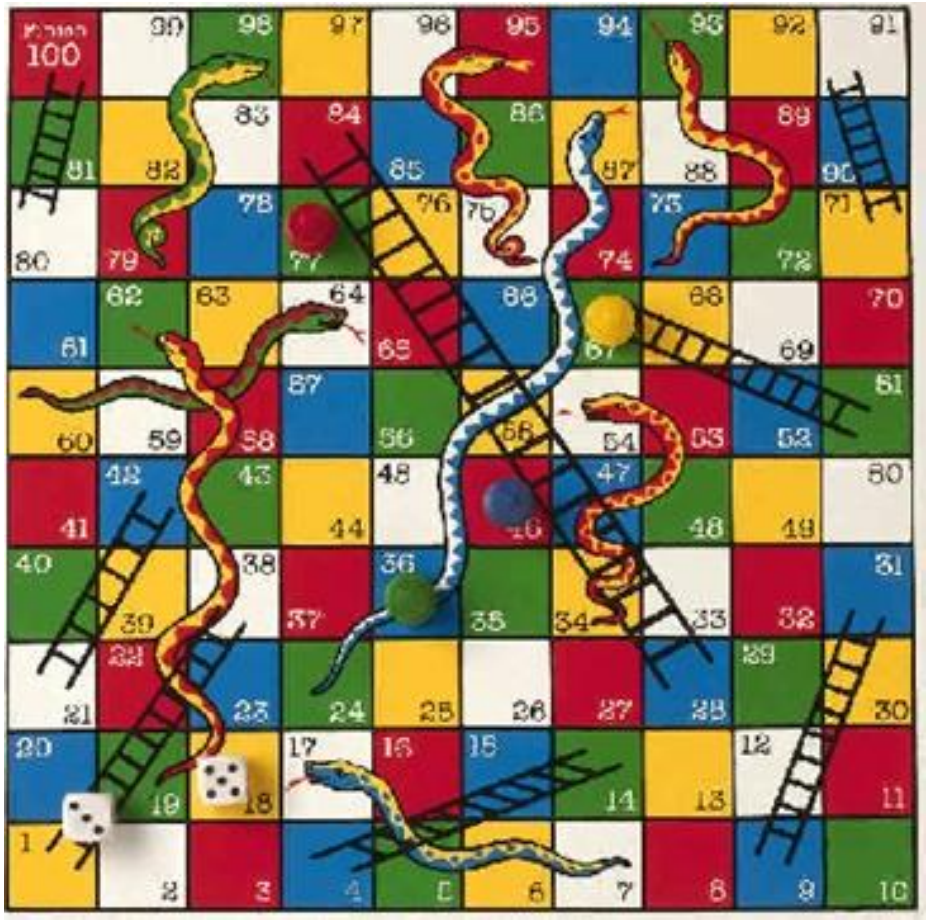
Let´s do the experiment...

Throws	Probability	Error
10	0.20	0.14
10^2	0.18	0.04
10^3	0.158	0.013
10^4	0.170	0.004
10^5	0.1672	0.0013
10^6	0.1667	0.0004
10^7	0.16673	0.00013

$$\overline{P_1} = \frac{r}{n} \qquad \varepsilon(\overline{P_1}) = \frac{\sqrt{r}}{n}$$

(results obtained with a dice *simulator*...)

Snakes and ladders

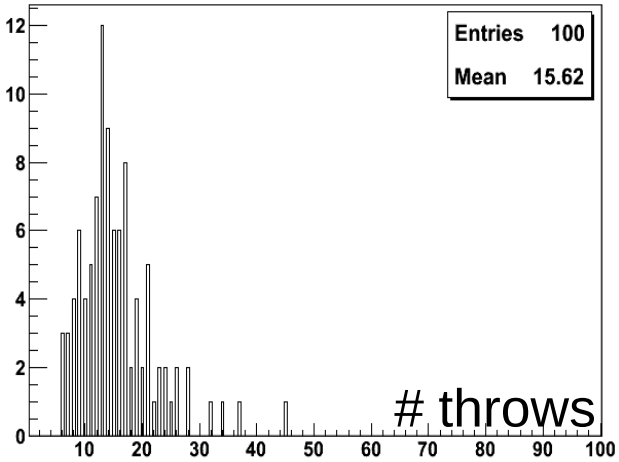


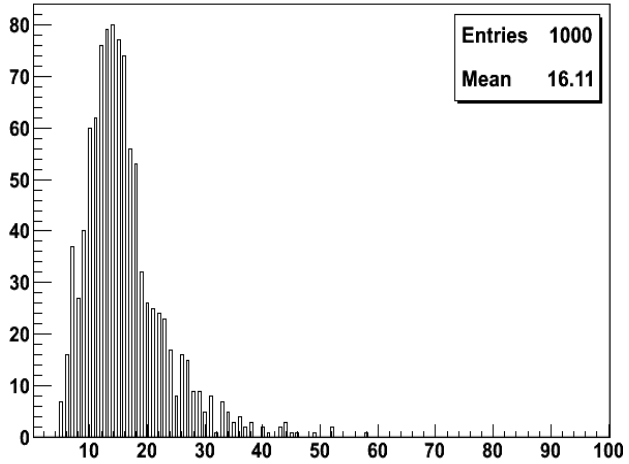
Rules: starting from 0, throw 2 dices and advance up to 100. Ladders leads up and snakes leads down

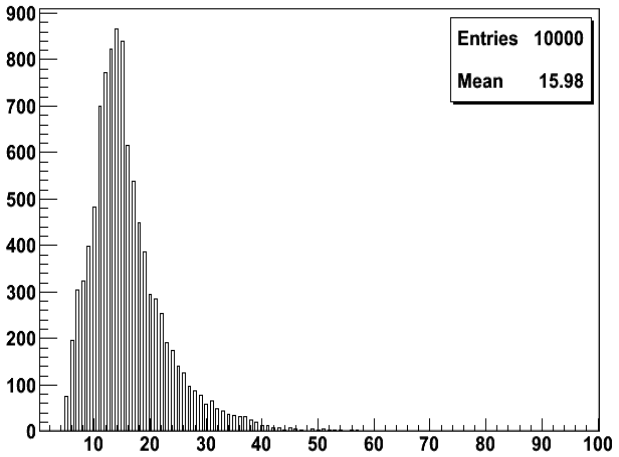
How many times we need to throw our dices on average to reach the end?
What is the minimum/maximum number or throws to reach the end?
What is the probability that 70 or more throws are needed to reach the end?

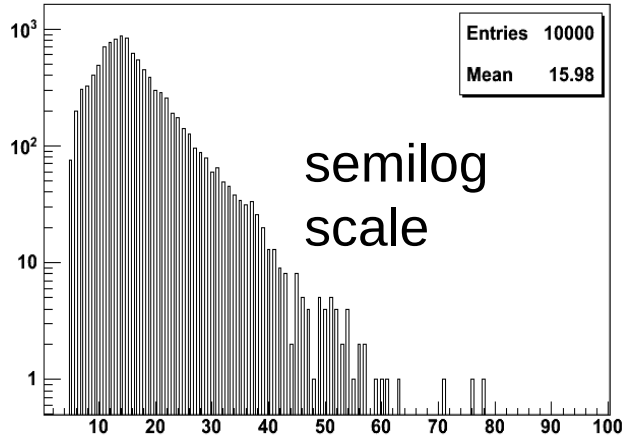
Difficult to calculate

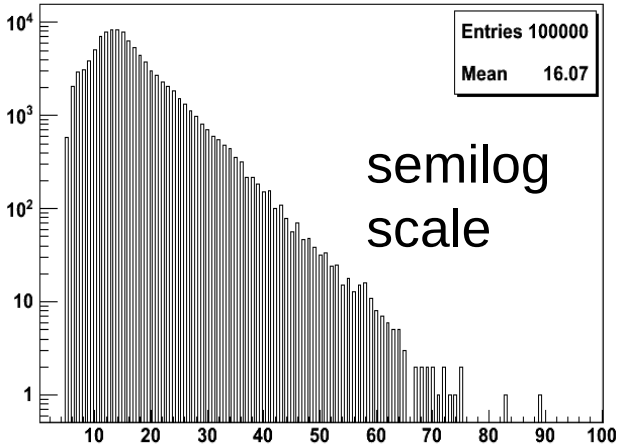
Easy to *simulate*… we just need a generator of random numbers with uniform distribution [1,6]

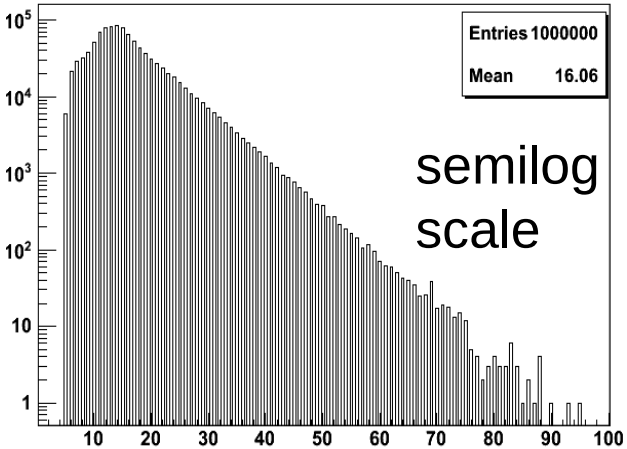










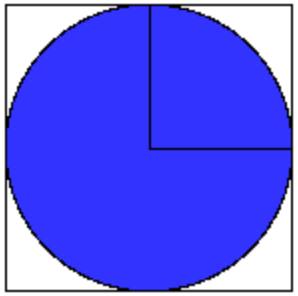


Trials = 10^2	Min = 6,	Max = 45	N>70 = 0	
Trials = 10^3	Min = 5 (7 times)	Max = 58	N>70 = 0	
Trials = 10^4	Min = 5 (75 times)	Max = 78	N>70 = 3	
Trials = 10^5	Min = 5 (587 times)	Max = 89	N>70 = 9	
Trials = 10^6	Min = 5 (5957 times)	Max = 95	N>70 = 122	$P \sim 10^{-4}$

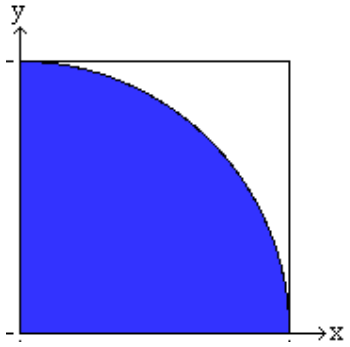
Maximum value found with 10^{10} trials is 164 $P \sim 10^{-10}$

Random numbers to calculate π

Let's assume that we have a random number generator with uniform distribution (0,1)



Area of circle is **π** (r=1)



Area = π/4

Given a set of randomly generated points (x,y) we consider the number of "hits" (if $x^2 + y^2 \leq 1$) over the total number of trials and we will have:

$$\pi = 4 * \frac{\#hits}{\#trials}$$

Random numbers to calculate π

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

int main()
{
   double x,y;
   int trials;
   int hits   = 0;
   srand ( time(NULL) );
   double pi;
   for (trials=0; trials<1000000;trials++)
   {
      x = (double)rand()/(double)RAND_MAX;
      y = (double)rand()/(double)RAND_MAX;
      if(x*x+y*y<=1)
         hits++;
   }
   pi = 4.*(double)hits/(double)trials;
   printf("value of pi is %.8lf \n",pi);
return 0;
}
```

srand(time(NULL))
set the seed (see next slides)

rand() returns random values between 1 and a maximun value which depends on the gcc version

How good is the estimation for π depends on how many iterations (trials) are done, and to a lesser extent on the quality of the random number generator
With 10^9 iteration (a few minutes of CPU time) I got $\pi = 3.14150$
Real value is $\pi = 3.14159$ (error 10^{-4})

Monte Carlo method

Monte Carlo methods are a class of computational algorithms that rely on repeated random sampling to provide approximate solutions to a variety of mathematical/physical problems. Monte Carlo methods tend to be used when it is infeasible or impossible to compute an exact result with a deterministic algorithm.

The method is called after the city in the Monaco principality, because of a roulette, a simple random number generator. The name and the systematic development of Monte Carlo methods dates from about 1944.

The real use of Monte Carlo methods as a research tool stems from work on the atomic bomb during the second world war. This work involved a direct simulation of the probabilistic problems concerned with random neutron diffusion in fissile material.

Random numbers

Old reference books contained tables of random number (output of physical processes)

Computer-generated sequences are deterministic and predictable (the algorithms to produce them are conceived by the human mind)
We will call them *pseudorandom* numbers, which are deterministic but *look like* they were generated randomly

There is a list of statistical tests which are used to prove the goodness of a (pseudo)random number generator.

Multiple recursion

A generator of pseudorandom process updates a current sequence of number in a manner that appears to be random. Such a deterministic generator *f* yields number recursively, in a fixed sequence.
The previous *k* numbers determine the next number:

$$x_i = f(x_{i-1}, \ldots, x_{i-k})$$

k is the *order* of the generator.

Because the set of numbers directly representable in the computer is finite, the sequence will necessarily repeat (the length of the sequence prior to beginning to repeat is called **period**)
The initial set of values is called **seed**.

The seed is typically a collection of integers. It sets the generator to a random starting point. A unique seed returns a unique random number sequence.

It is important to use a new seed every time that a random selection is initiated.

A typical error is the use of the same seed for multiple generation, which leads to the generation of the same sample of random numbers.

Often the seed is set using the current universal time (number of seconds lasted from January 1st 1970, 00:00)

Uniform deviates

Uniform deviates are just random numbers that lie within a specified range, typically (0,1). The probability density function (pdf) will be in this case:

$$p(x) = 1 \quad \text{if} \quad 0 < x < 1$$
$$= 0 \quad \text{elsewhere}$$

Other kinds or deviates (gaussian, poissonian…) are almost always generated by performing appropriate operation on one or more uniform deviates.

Generators

Usually generators produce random integers over same fixed range and then scale them into the interval (0,1)

Modular arithmetic

Two numbers are said to be equivalent (congruent) modulo *m* if their difference is an integer evenly divisible by *m*

$$a \equiv b \bmod m$$

Reduction modulo m:

Given a number *b*, find *a* such that $a \equiv b \bmod m$ and $0 \leq a < m$ (a is the reminder of the integer division b/m, or a=b%m)

Simple Linear Congruential Generators

D. H. Lehmer (1948)

$$x_i \equiv (ax_{i-1} + c) \bmod m$$

order = 1

max. period = m-1 (0 can't be a result)

To scale into the unit interval (0,1):

$$u_i = x_i / m$$

Quite limited in its ability to produce very long streams of numbers but it is often a basic element of more sophisticated generators

Exemples:

$$a = 7 \quad c = 0 \quad m = 31 \quad x_0 = 19$$

The sequence will be:

$$9,1,7,18,2,14,5,4,28,10,8,25,20,16,19$$

and the period is 15.

If we choose instead:

$$a = 3 \quad c = 0 \quad m = 31 \quad x_0 = 19$$

the period will be maximum (30) and we will say that 3 is a primitive root modulo 31.

The moduli in common use range typically from 10^9 to 10^{15}

The generator RANDU for many years was the most widely used random number generator in the world. It is essentially:

$$x_i = 65539 x_{i-1} \bmod 2^{31}$$

Multiple recursive generators

$$x_i \equiv (a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_k x_{i-k}) \bmod m$$

Order is k

Period can be > m (if m is prime the max period is $m^k - 1$)

Ex.:

$$x_i \equiv (107374182 x_{i-1} + 104480 x_{i-5}) \bmod (2^{31} - 1)$$

Random numbers in C

The most common function used for random number generation in C is *rand* and it is implemented in the stdlib.

It returns an integer between 1 and a max value which depends on the gcc version.

My gcc has max value = 2^{15}-1 = 32767
gcc in A94-A95 has max value = 2^{31}-1= 2147483647
The max value is called RAND_MAX

Period is *anyway* 2147483647 (enough for our purposes)
Use the code:

```
#include <time.h>
#include <stdlib.h>
…
srand ( time(NULL) );   //set the seed using the universal time
double  rand = (double)rand()/(double)RAND_MAX; //generate the number
```

Plotting histograms with Root

```
void my_histo()
{
   TH1F *h1 = new TH1F("h1", "My first root histogram", 100,
0., 40.);
   FILE *fp = fopen("data.txt","r");
   float data;
   while (!feof(fp))
   {
      fscanf(fp,"%f",&data);
    h1->Fill(data);
   }
   fclose(fp);
   h1->GetXaxis()->SetTitle("Value");
   h1->GetYaxis()->SetTitle("# of entries");
   h1->Draw();
}
```