Random Numbers

Random numbers are ubiquitous in physics – thermodynamics, radioactivity, particle collision and everything in between

Two basic methods to generate random number with varying degree of randomness

- *True* RNG
- *Pseudo* RNG

True RNG : uses natural phenomenon like coin flipping, dice rolling, radioactive decay, thermal noise, atmospheric radio-noise etc.

`Requires post-processing, slow ⇒ not useful for regular usage`

Pseudo RNG : based on algorithms, generated iteratively

`Deterministic, finite sequence length, correlated but extremely fast and portable`

Sequence length can be made veryyy long by proper choice of parameters

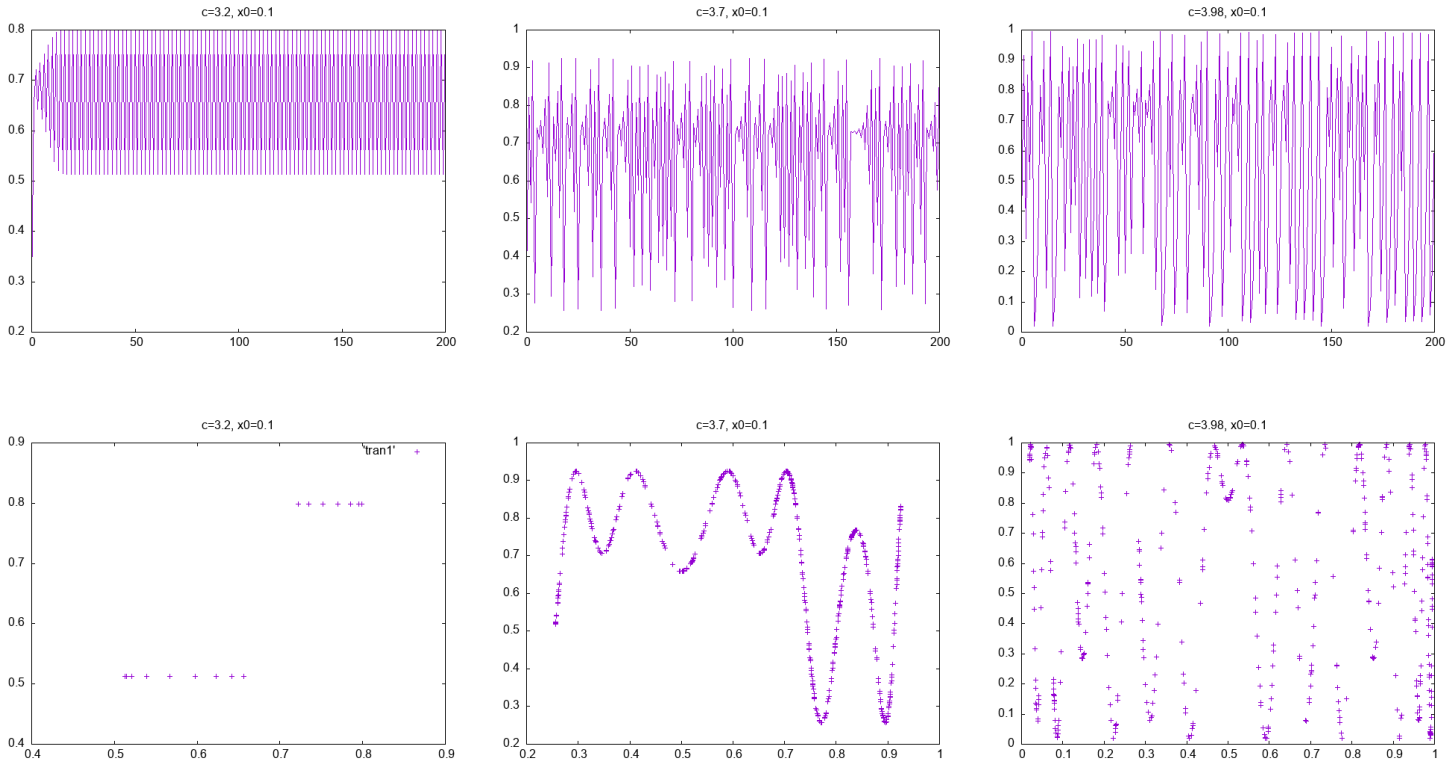
Basic goal : Write pRNG and test it for randomness

Example : a quick and dirty pRNG

$$x_{i+1} = c\, x_i\, (1 - x_i)$$

x_0 is the seed which defines the random sequence.

An exercise with $x_0 = 0.1$ and $c = 3.2, 3.7, 3.98$



Bad pRNG for many choices of c and seed x_0 – settles down into regular pattern. However, no specific patter for $c = 3.98,\ x_0 = 0.1$

Quantifying good and bad pRNG : Need mathematical tests for determining randomness $\Rightarrow$ if pRNG fails test, then don't use

Eyes are good at discerning patterns but can fool us too!

Basic test : correlation, moments
Advanced test : chi-square, Kolmogorov-Smirnov

Ideally random numbers generated should have no correlations and the error statistical only *i.e.* scale as $1/\sqrt{N}$

Correlations test :

$$\epsilon(n, N) = \frac{1}{N}\sum_{i=1}^{N} x_i\, x_{i+n} - \left(\frac{1}{N}\sum_{i=n}^{N} x_i\right)^2$$

If tuplets of RN not correlated, $\epsilon(n, N) \to 0$ with statistical error $1/\sqrt{N}$.

Moments test : Assuming pRNG produces RN $\in [0, 1]$, for a uniform distribution k-th moment $= 1/(k+1)$.

$$\mu(N, k) = \left|\frac{1}{N}\sum_{i=1}^{N} x_i^k - \frac{1}{k+1}\right|$$

Linear Congruential Generator

One of the oldest and most common serious choice of pRNG having a uniform distribution,

$$x_{i+1} = \Big(ax_i + c\Big) \bmod m \quad \equiv \quad x_{i+1} = \text{remainder}\left(\frac{ax_i + c}{m}\right)$$

m determines period of the generator *i.e.* produces random numbers between $[0,\ m]$, whereas x_i/m yields randoms in interval $[0,1]$.

- m is typically chosen to be 2^{32}
- a is *multiplier* and usually $0 < a < m$. Numerical Recipes uses $a = 1664525$ and gcc uses $a = 1103515245$
- c is *increment* and usually $0 < c < m$. Numerical Recipes uses $c = 1013904223$ and gcc uses $c = 12345$

Not all rosy with **LCG**

- $a,\ c,\ m,\ x_0 = 6, 7, 5, 2 : x_{i+1} = (6x_i + 7)\text{mod}(5)$ 4,1,2,0,2,4,1,2,0,2, ...
- $a,\ c,\ m,\ x_0 = 27, 11, 54, 2 : x_{i+1} = (27x_i + 11)\text{mod}(54)$ 11,38,11,38, ...

General approach linear feedback shift register generators

$$x_i = \Big(a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_n x_{i-n} + c\Big) \bmod m$$

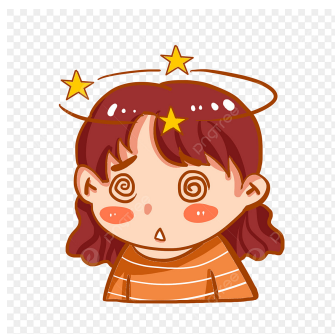
LCG : Hull-Dobell theorem

LCG is extremely sensitive to a, c, x_0, m. Particularly, a has to be chosen with great care else short / very short periodicity will set in.

Hull-Dobell theorem : LCG has a period m *iff* $c \neq 0$ and

1. c is coprime to m,
2. $a - 1$ is a multiple of p for every prime p dividing m
3. $a - 1$ is a multiple of 4, if m is a multiple of 4.

LCG works well for m having many repeated prime factors p, such as power of 2. But if m are square-free integer (having no n^2 factor for any n), then only $a = 1$ is allowed and it is a very bad pRNG.



LCG is extremely fast, least memory footprint but period severely limited by choice of m : for $m \sim 10^{32} \rightarrow 10^9$ pRN. Gets exhausted in seconds!!

$c = 0$ corresponds to Lehmer, Park-Miller pRNG

$$x_{i+1} = ax_i \cdot \bmod m$$

m can be a prime or a prime just less than a power of 2 (Mersenne primes $2^{31} - 1$, $2^{61} - 1$ etc.) or can be a simple power of 2. a has be chosen with care else short to very short periodicity will set in. A variant of **LCG** is *shift register generators*

$$x_i = (ax_{i-j} + cx_{i-k}) \cdot \bmod p, \quad c = 0, \ p = \text{prime}, \ \text{period} = p^k - 1$$

Using bitwise *XOR* instead of '+': $x_i \equiv x_{i-j} \oplus x_{i-k}$ leads to larger period. Example – $x_{i-j} = 6 = 0110$ and $x_{i-k} = 11 = 1011$ in 4-bit representation, then $\oplus$ yields 1101=13

Another improvement of LCG is *Lagged Fibonacci generator*,

$$x_l = (x_{l-i} \star x_{l-j}) \cdot \bmod m \quad \text{where} \ \ 0 < i < j$$

where $\star$ is a binary operator – addition, multiplication or XOR. Typically, $m = 2^M$, $M = 32$ or 64.

Fibonacci generators are fast, have very long periods and known to pass all statistical quality tests. This can also be parallelized.

In above cases, underlying PDF of the pRNG is uniform distribution, $p(x) = 1$ for $x \in [0, 1]$. PDF for uniform distribution in interval $[a, b]$ is

$$p(x) = \begin{cases} \frac{1}{b-a} & a \le x \le b \\ 0 & \text{otherwise} \end{cases}$$

Mean or expectation value and variance of such random variables are

$$\langle x \rangle = \int_a^b x\, p(x)\, dx = \frac{1}{b-a} \int_a^b x\, dx = \frac{a+b}{2}$$

$$\sigma^2 = \langle x^2 \rangle - \langle x \rangle^2 = \int_a^b x^2\, p(x)\, dx - \left(\frac{a+b}{2} \right)^2 = \frac{(b-a)^2}{12}$$

LCG and its variants are deterministic, two successive *RN* may not be independent. Degree of correlation, *auto-correlation* function

$$C_k = \frac{\langle x_{i+k}\, x_i \rangle - \langle x_i \rangle^2}{\sigma_i^2}, \quad \text{where } \langle x_{i+k}\, x_i \rangle = \frac{1}{n-k} \sum_{i=1}^{n-k} x_i\, x_{i+k}$$

$C_k \neq 0$ for $k \neq 0 \Rightarrow$ RNs are not independent over a *length* of k. Two consecutive RNs x_1, x_2 distributed uniformly $\in [0, 1] \Rightarrow p(x_1) = p(x_2) = 1$, if independent, then

$$\langle x_1\, x_2 \rangle = \int_0^1 dx_1 \int_0^1 dx_2\; x_1 x_2\; p(x_1, x_2) = \int_0^1 dx_1 \int_0^1 dx_2\; x_1 x_2\; p(x_1)\, p(x_2) = \frac{1}{4}$$

Non-uniform RN

Standard **pRNG** generates uniform random integers $\in [0, INT\_MAX]$ or floating point numbers $\in [0, 1)$.

Non-uniform RN – gaussian, poisson, exponential etc. distributed

Transformation method

- Uniform RN $u \in [a, b)$ from $x \in [0, 1)$ using transformation

$$u = a + (b - a)\, x$$

For non-uniform, we need probability theory,

$$p(x)\, dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad \int_{-\infty}^{\infty} p(x)\, dx = 1$$

Suppose $p(x)$ is **pdf** of a uniform RN x and target **pdf** is $q(y)$. From transformation law of probability distribution

$$|q(y)\, dy| = |p(x)\, dx| \;\; \rightarrow \;\; q(y) = p(x) \left| \frac{dx}{dy} \right|$$

Consider only RN $x \in [0, 1) \;\Rightarrow\; p(x) = 1$. Idea is to generate uniform RN x and then invert the cumulative **pdf** to get y
– possible if this inversion can be done analytically!

- Exponentially distributed RN

$$q(y) = a\, e^{-ay} \quad \text{for} \quad y \geq 0,\ a > 0$$

From the transformation law

$$a\, e^{-ay} = \left| \frac{dx}{dy} \right| \ \rightarrow\ x = \int_0^y q(y)\, dy = 1 - e^{-ay} \ \Rightarrow y = -\frac{1}{a} \ln(1-x) \equiv -\frac{1}{a} \ln x$$

because $(1 - x) \in [0, 1)$ as well.

- Lorentz distributed RN (mean $= 0$, $b =$ half width at half maximum)

$$\begin{aligned} q(y) &= \frac{1}{\pi} \frac{b}{x^2 + b^2} \ \rightarrow\ x = \int_{-\infty}^y q(y)\, dy = \frac{1}{2} + \frac{1}{\pi} \tan^{-1}\left(\frac{y}{b}\right) \\ y &= b \tan\left[\pi\left(x - \frac{1}{2}\right)\right] \quad \text{by inverting} \end{aligned}$$

- Gaussian distributed RN (mean $= 0$, variance $= 1$)

$$q(y) = \frac{1}{\sqrt{\pi}}\, e^{-x^2/2} \ \rightarrow\ x = \int_{-\infty}^y q(y)\, dy = \frac{1}{2}\left[1 + \text{erf}\left(\frac{y}{\sqrt{2}}\right)\right]$$

No closed form expression for erf, hence not possible to invert! At least not in one space dimension.

Achieved by Box-Muller transformation that uses two uniform RN $x_1,\ x_2$ to generate two Gaussian distributed randoms y_1, y_2,

$$\begin{aligned} q(y_1)\, q(y_2) &= \frac{1}{2\pi}\, e^{-(y_1^2+y_2^2)/2} dy_1\, dy_2 \\ &= \frac{1}{2\pi}\, e^{-r^2/2} r\, dr\, d\theta \quad \text{where } y_1 = r\, \sin\theta,\ y_2 = r\, \cos\theta \\ &\equiv p(r)\, p(\theta)\, dr\, d\theta \quad \text{where } p(\theta) = 1/2\pi,\ p(r) = r\, e^{-r^2/2} \end{aligned}$$

Therefore, upon integrating θ and r we get

$$\int_0^{2\pi} p(\theta) d\theta = 1,\ \ x = \int_0^r r\, e^{-r^2/2} dr = 1 - e^{-r^2/2} \ \Rightarrow\ r = \sqrt{-2\ln(1-x)}$$

Trick is to generate two uniform RN $x_1,\ x_2 \in [0, 1)$,

$$\begin{cases} \theta &= 2\pi\, x_1 \\ r &= \sqrt{-2\ln\, x_2} \end{cases} \Rightarrow \begin{cases} y_1 &= \sqrt{-2\ln(x_2)}\, \sin(2\pi x_1) \\ y_2 &= \sqrt{-2\ln(x_2)}\, \cos(2\pi x_1) \end{cases}$$

Box-Muller implies at each step of the algorithm two uniform RN $x_1,\ x_2$ are converted into two Gaussian distributed RN $y_1,\ y_2$.

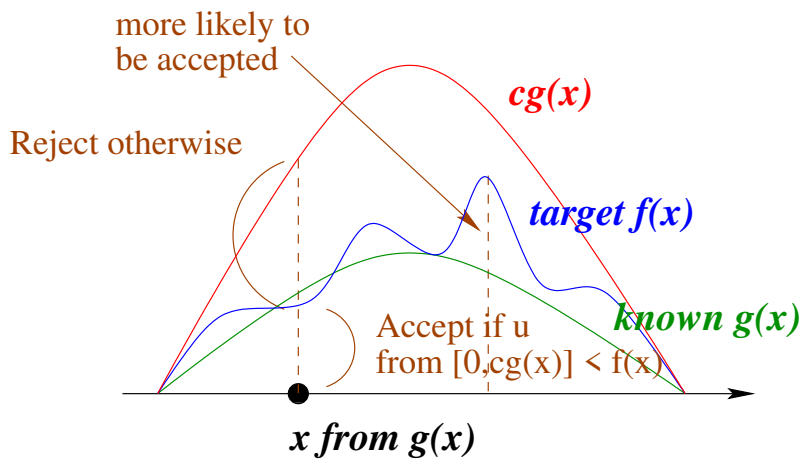
But what if no transformation yields to inversion?

Acceptance-Rejection method

Cumulative **pdf** *i.e.* integral in transformation method may not be invertible. Even if invertible, it may not be efficient compared to some other alternative method(s). Accept-reject method is one such clever method.

Basic idea is if RN from a probability density function $f(x)$ is sought then task is to find an alternative pdf $g(x)$ from which efficient method of generating RN already exists. The condition being $f(x) \leq c\, g(x)$ where $c > 0$ is a constant and is as close to 1 as possible.

Sample RN from $g(x)$ and accept with probability $f(x)/c\, g(x)$.



Steps involved are :

1. Choose a **pdf** $g(x)$ that is easy to sample.
2. Sample RN x from $g(x)$.
3. Sample RN u from uniform distribution $[0, 1]$.
4. Accept x if

$$u \leq \frac{f(x)}{c\, g(x)}$$

reject otherwise. All accepted samples x will be from **pdf** $f(x)$.

If success probability is p and n is the number of successful iterations then average number of iterations required is $E(n)$

$$p = P\left(u \leq \frac{f(x)}{cg(x)}\right) \;\Rightarrow\; E(n) = \frac{1}{p} = c$$

Hence, it is important not to have c large.

Important to note that above ratio is bounded between 0 and 1

$$0 < \frac{f(x)}{cg(x)} \leq 1$$

Some examples

- RN with **pdf** $f(x) = 2x,\ 0 < x < 1$ from uniform RN $g(x);\ x \in [0,1]$

$$g(x) = 1 \ \Rightarrow \ \frac{f(x)}{g(x)} = 2x < 2 \ \rightarrow \ c = 2$$

- RN with **pdf** $f(x) = e^{-2x}/2,\ x \geq 0$ from uniform RN $g(x);\ x \in [0,1]$

$$g(x) = 1 \ \Rightarrow \ \frac{f(x)}{g(x)} = \frac{e^{-2x}}{2} < 1/2 \ \rightarrow \ c = 1/2$$

- RN with **pdf** $f(x) = (\sqrt{2/\pi})e^{-x^2/2},\ x \geq 0$ from RN $g(x) = e^{-x};\ x \geq 0$

$$\frac{f(x)}{g(x)} = \sqrt{\frac{2}{\pi}}\, e^{x - x^2/2} \ \xrightarrow[x=1]{\max} \ \sqrt{\frac{2e}{\pi}} = c$$

- RN with Beta(α, β) **pdf** $f(x) = 6x(1-x),\ 0 < x < 1$ from RN $g(x);\ x \in [0,1]$

$$\frac{f(x)}{g(x)} = 6x(1-x) \ \xrightarrow[x=1/2]{\max} \ \frac{3}{2} = c$$

RN on N-sphere

Often it is necessary to generate random number on a N-sphere. There are two possible approaches, stated here without proof, both involving the same idea – dividing random vector of $N+1$ RN's by norm of the vector gives RNs on the N-sphere.

1. Box-Muller
   - 1.1 Use Box-Muller on each component of a uniform random vector $\vec{x}$ to obtain a Gaussian distributed vector.
   - 1.2 Normalize the length of $\vec{x}$ to unity *i.e.* $\hat{e} = \vec{x}/||\vec{x}||$. Angles are uniformly distributed.
2. Accept-Reject
   - 2.1 Generate a uniform random vector $\vec{x}$ with each component $\in [-1, 1]$.
   - 2.2 If $||\vec{x}|| > 1$, choose a new vector; otherwise normalize $\vec{x}$ to unity. Angles are uniformly distributed.

Library implementation of pRNG

- For assignment purpose it is advisable to implement one's own pRNG subjected to testing its randomness using both auto-correlation and ENT (needed to be installed from https://www.fourmilab.ch/random/).
- For research purpose it is highly recommended to use GSL (https://www.gnu.org/software/gsl/) or Boost (https://www.boost.org). Both of these are extremely efficient, have super long period and contain vast selection of pdf's. A quick guide to use GSL is given below.

Compilation codes to include GSL requires the flags `-lgsl -lgslcblas`

A quick guide to use GSL pRNG is here:

```
 #include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
int main(int argc,char *argv[])
{
   double rn;
   const gsl_rng_type *T;
   gsl_rng *r;
   gsl_rng_env_setup();
   T=gsl_rng_default;
   r=gsl_rng_alloc(T);
// Uniform distributed
   rn=gsl_ran_flat(r,-1.0,1.0);
// Gaussian distributed
   rn=gsl_ran_gaussian(r,1.0);
// Lorentz / Cauchy distributed
   rn=gsl_ran_cauchy(r,1.0);
} // end of main
```