# Ch. 04
# *Random Numbers & a primer to Monte Carlo Methods*

Andrea Mignone

Physics Department, University of Torino

AA 2023-2024

## Pseudo-Random Numbers

- It may seem a conceptual impossibility in producing “random numbers” with a computer which, by definition, is the most precise and deterministic of all machines conceived by the human mind.

- A program will produce output that is entirely predictable, hence not truly “random.”

- Computer-generated sequences are termed *pseudo-random*: a deterministic algorithm that produces a random sequence which should be different from, and — in all measurable respects — statistically uncorrelated with, the computer program that *uses* its output.

- In other words, any two different random number generators ought to produce statistically the same results when coupled to your particular applications program. If they don’t, then at least one of them is not (from your point of view) a good generator.

# Generating Pseudo Random Numbers

- Let's understand how pseudo random numbers (***prn***) work with an example

```
srand(seed);                  // Initialize the sequence
for (i=0; i<10; i++){
  number = rand()%100+1;      // Generate a random number in [1,100]
  cout << number << endl;
}
```

- The pseudo-random number generator `srand()` is initialized using the argument passed as *seed*.

- Using the same *seed* will generate the same succession of  results in subsequent calls to `rand()`, which generates integer random numbers.

- For different *seed* value used in a call to `srand()`, the prn generator `rand()` is expected to generate a different succession of results in the subsequent calls.

- To produce different sequences at each execution, we can use the `time()` function:

```
srand(time(NULL));                  // Initialize the sequence
```

## Generating Pseudo Random Numbers

- Double precision random numbers in the interval [0,1] can be generated using `drand48()` rather than `rand()`:

```
srand48(time(NULL));     // Initialize the sequence
for (i=0; i<10; i++){
  double x = drand48();  // Generate a random number in [0,1]
  cout << number << endl;
}
```

- As before, the ***prn*** sequence is initialized using `srand48()` (instead of `srand()`).

- At each code execution, the sequence will now be different since the seed depends on the call to `time(NULL)`.

## *Practice Session #1*

- `guess.cpp`: write a program that extract an integer random number in the range `[1,100]` and try to guess it.

  Based on the most recent guess, have the code suggest the interval containing the random number. Count the number of guesses.
- For instance, suppose the number to guess is 35, then your code should produce an output similar to [input from keyboard is colored in red]:

```
n in [1,100]
type your guess #1 > 5
n in [5,100]
type your guess #2 > 27
n in [27,100]
type your guess #3 > 60
n in [27,60]
type your guess #4 > 50
n in [27,50]
type your guess #5 > 32
n in [32,50]
type your guess #6 > 38
n in [32,38]
type your guess #7 > 35
Number found in 7 iterations !!
```

# Practice Session #2

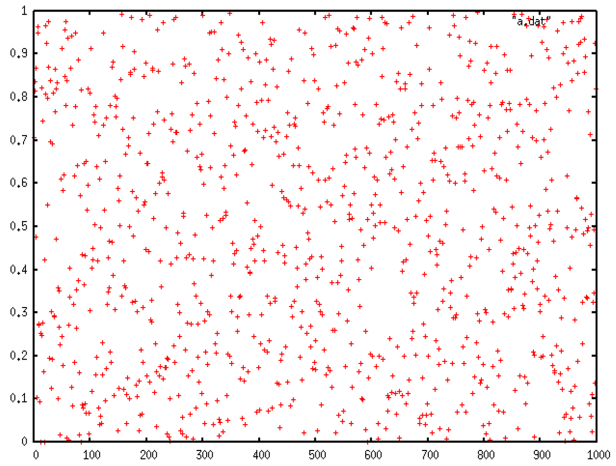
- `prn_uniformity.cpp`: test a random-number generator in the range [0,1] (use double precision !) to obtain a numerical measure of its uniformity and randomness before you stake your scientific reputation on it.
  - Generate a quick visual test by plotting $r_i$ as a function of `i` ($0 \leq$ `i` $< 10^3$) : a uniform distribution of uncorrelated point values should appear.
  - Evaluate the kth-moment of the distribution:

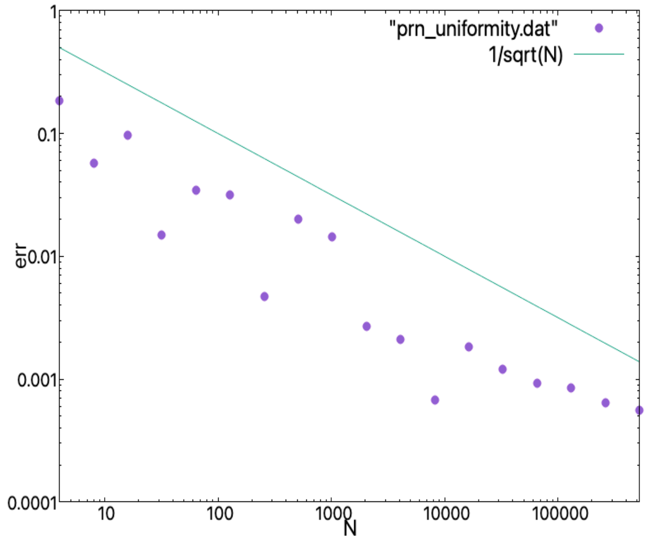
$$\langle x^k \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i^k .$$

If the number are distributed uniformly, then the previous equation can be approximated by

$$\frac{1}{N} \sum_{i=1}^{N} x_i^k \simeq \int_0^1 dx\, x^k P(x) \simeq \frac{1}{k+1} + O\left(\frac{1}{\sqrt{N}}\right)$$

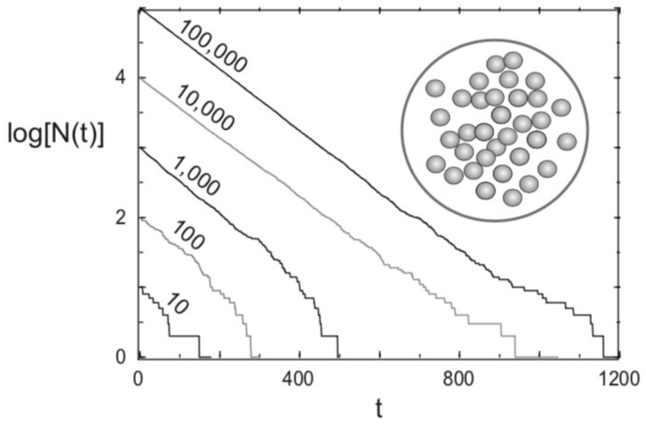
Check that the deviation varies as 1/sqrt(N) by plotting the error as a function of N (double N at each iteration to produce a log-log plot). Limit to k = 1 and k = 2.





# Application to Radioactive Decay

- Spontaneous decay is a natural process in which a particle decays into other particles. It is random event with constant probability.
- Of course, as the total number of particles decreases with time, so will the number of decays.



- Imagine having a sample of N(t) radioactive nuclei at time t.
- Let ΔN be the number of particles that decay in some small time interval Δt; the probability P of any one particle decaying per unit time is a constant:

$$\mathcal{P} = \frac{\Delta N(t)/N(t)}{\Delta t} = -\lambda \qquad \Longrightarrow \qquad \frac{\Delta N(t)}{\Delta t} = -\lambda N(t)$$

- Because the exponential decay law is a large-number approximation to a natural process that always ends with small numbers, our simulation should be closer to nature than is the exponential decay law:

$$N(t) = N_0 \exp(-\lambda t)$$

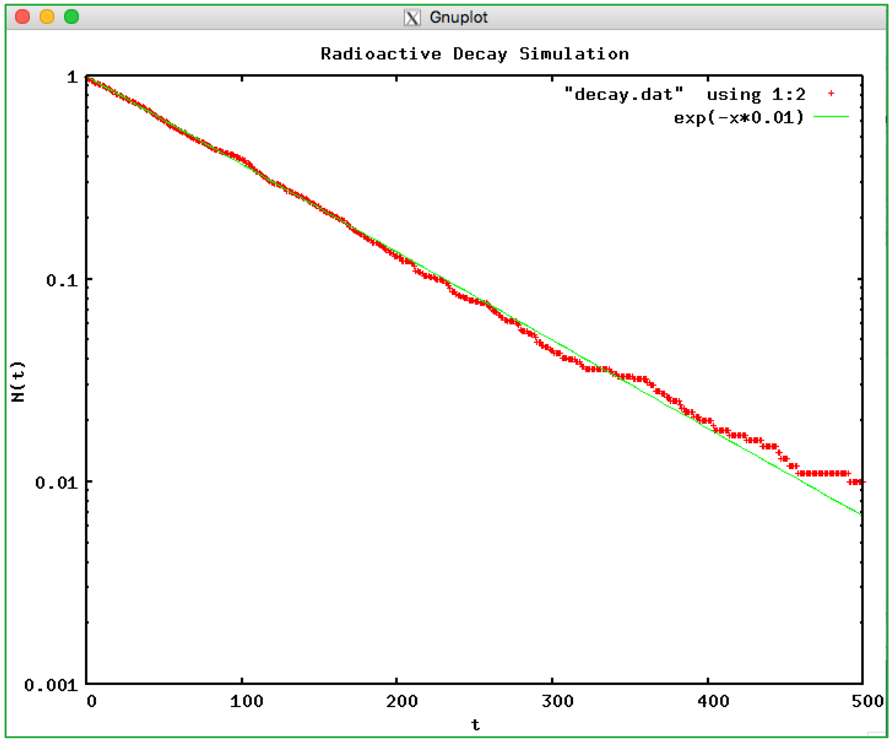
## Practice Session #3: Radioactive Decay

- decay.cpp: using random deviates in [0,1] simulate the radioactive decay of an initial distribution of N atoms having a decay rate λ (use λ = 0.01).

  - We increase time in discrete steps Δt (= 1), and for each time interval we count the number of nuclei that have decayed during that Δt.

  - The simulation quits when there are no nuclei left to decay or when time exceeds a given threshold (e.g. t < 500). Such being the case, we have an outer loop over the time steps Δt and an inner loop over the remaining nuclei for each time step.

  - Produce a plot.



# Acceptance-Rejection Method

- A convenient method for generating random variables according to a specific distribution is von Neumann acceptance / rejection, whose geometrical basis is illustrated in the figure.

- Suppose we are interested in generating x between 0 and 1 with distribution W(x) and let C(x) be a positive function such that C(x) > W(x) over the region of integration.

- C(x) may be chosen as a constant greater than the maximum value of W(x).

- If we generate points in 2D that uniformly fill the area under C(x), and accept only those for which y < W(x), then the accepted point will be distributed according to W(x).

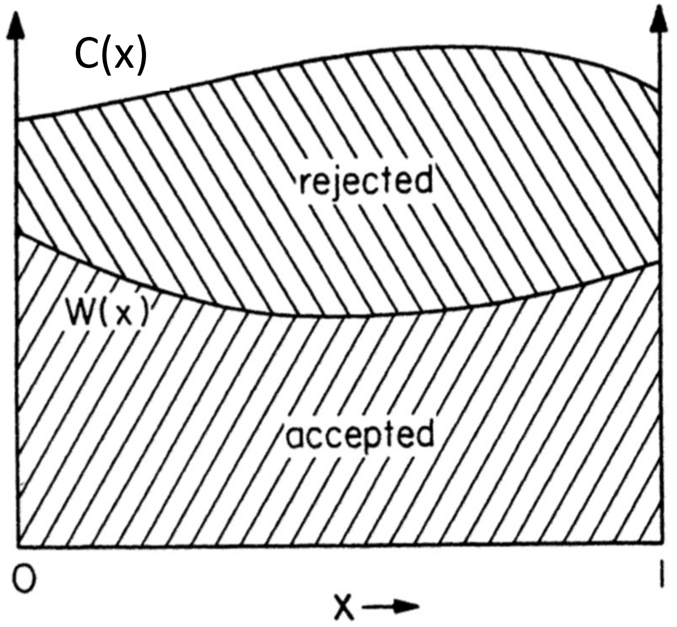


**Figure 8.1** Illustration of the von Neumann rejection method for generating random numbers distributed according to a given distribution $w(x)$.

# Practice Session #04: Gaussian Distribution

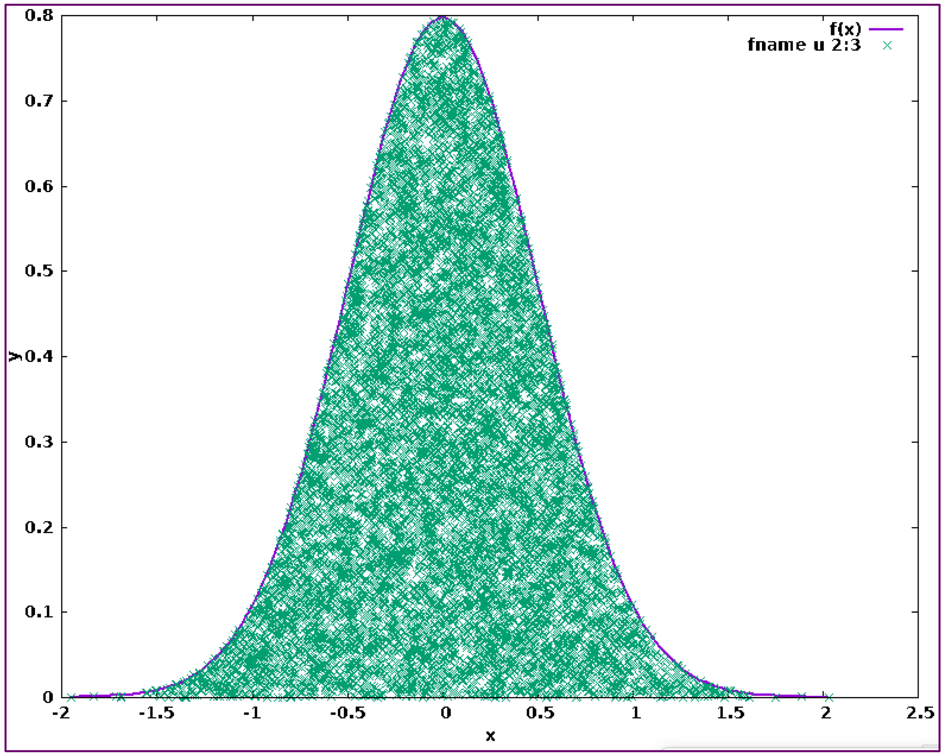
- Using the acceptance-rejection method, generate random deviates following a Gaussian-like distribution:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

with $\mu$=0, $\sigma$=0.5.

- Choose C to be a constant, e.g. C ≥ W(x) = f(x) and choose x in the range [-5,5];

- Generate pairs of random numbers $(x_k,y_k)$ for k = 0… N-1 (take N = $10^5$) and construct the distribution.



## Monte Carlo Methods & Quadrature

- **Monte Carlo methods** are a class of stochastic algorithms – optimization, numerical integration and generating probability distribution- relying on repeated random sampling to obtain the solution to a problem which might be deterministic in principle.

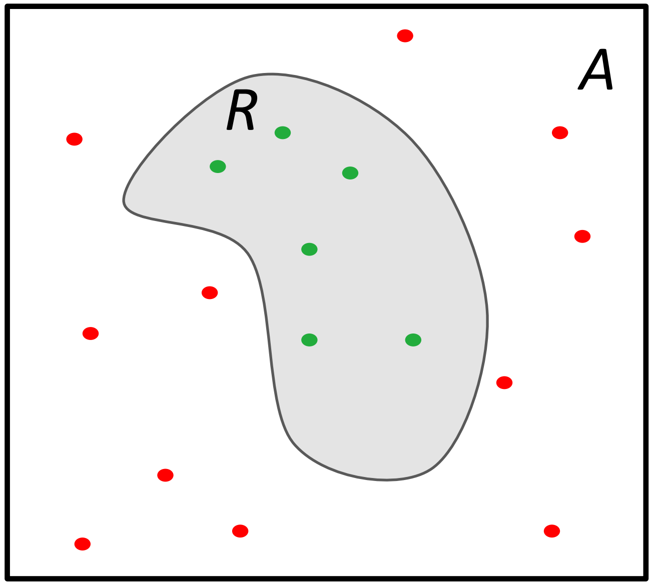
- Commonly employed in physics and/or mathematics when other approaches become impracticable.

- Here we consider Monte Carlo integration which is based on a random choice of points at which the integrand is evaluated. This method is particularly useful for higher-dimensional integrals.

- The approach is clearly non-deterministic:: each realization provides a different outcome.

- The final outcome is an approximation to the correct value with respective error bars, and the correct value is likely to be within those error bars.

## An Example: “Stone Throwing”



- Consider a function f(x,y) in a 2D space and its integral over a certain region *R* ∈ A

$$\int_R f(x,y)dxdy$$

- Random values of *x* and *y* are generated to sample A: only trial points (x,y) inside R are accepted.

- The ratio between successful attempts divided by the total number of samples represents the probability for samples to be accepted.

- Multiplying this probability by the volume of the sampling space A gives approximation of the correct value of the integral:

$$I \approx A\frac{N_{\text{accepted}}}{N}$$

- A large number of samples will result a closer value to the expected value.

$$\lim_{N\to\infty} Q_N = I$$

## Error

- The uncertainty associated with this quadrature formula, we consider f(x,y) as a random variable and invoke the central limit for large N. From statistic we have

$$\sigma_I^2 \approx \frac{1}{N}\sigma_f^2 = \frac{1}{N}\left[\frac{1}{N}\sum_{i=1}^{N} f_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} f_i\right)^2\right]$$

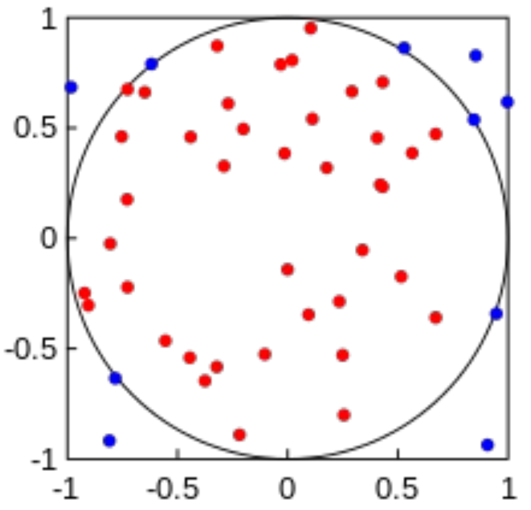
- Where $\sigma_f^2$ is the variance in f, a measure of the extent to which f deviated from its average over the region of integration.

- The uncertainty in $\sigma_I$ decreases as 1/sqrt(N) (not very fast, if compared with trapezoidal for which the error scales line $N^{-2}$).

- Precision is greater if $\sigma_f$ is smaller (function is as smooth as possible).

## Practice Session #5: Computing $\pi$

- `pi.cpp`: use random sampling to perform a 2-D integration on the unit disc, determine $\pi$:



  - Consider a circle of radius 1 enclosed in a square of side 2.
  - Generate pairs of random number, $\{x_i, y_i\}$ in the range [-1,1], with i = 1,…,N and count how many points fall inside the circle.
  - Since we now the area of the square, obtain an approximation of the area of the circle as

$$I \approx A_{\rm sq}\frac{N_{\rm in}}{N}$$

1. Give an input value of N and compute the integral;
2. Keep increasing N such that the relative error `err = |I/π-1| < tol` (use `tol = 10`$^{-4}$);
3. Plot the error[*] as a function of N = 4,8,16, … ≈ $10^6$-$10^7$ and compare it with estimate (≈1/sqrt(N)).

[*]From statistics, a more accurate error estimate may be obtained using the standard deviation, i.e., the amount of variation we should expect from samples. In this case the error is computed from the standard deviation σ, that is, the square root of

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}(\mu - f_i)^2 = \frac{N}{N-1}\left[\sum_{i=1}^{N}\frac{f_i^2}{N} - \left(\sum_{i=1}^{N}\frac{f_i}{N}\right)^2\right]$$