Computational Physics (PHYS6350)

Lecture 17: Random numbers

Reference: Chapter 10 of *Computational Physics* by Mark Newman

March 27, 2025

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics

(Pseudo-)random numbers

Random numbers play important role, both in modelling of the physics processes (some of which are regarded as truly random, such as radioactive decay) and as a tool to tackle otherwise intractable problems.

Examples:

- Numerical integration (especially in many dimensions)
- Sampling microstates in statistical mechanics
- Simulating quantum processes
- Monte Carlo event generators

Numbers generated on a computer are usually not truly random, but a good generator produces numbers that reflect the desired properties of a random variable, hence it is called *pseudo-random number generator*.

Pseudo-random numbers on a computer

- The most basic routine produces a random integer number x between 0 and some maximum value m.
- By dividing over m one can get a real pseudo-random number $\eta = x/m$ which is uniformly distributed in an interval $\eta \in (0,1)$
- By applying various transformations and techniques to the sequence of η one can sample other (non-uniform) distributions.

How to sample pseudo-random numbers x?

Linear congruential generator

Historically, one of the simplest RNG is linear congruential generator (LCG)*.

It generates a sequence of pseudo-random numbers in accordance with an iterative procedure

$$x_{n+1} = (ax_n + c) \bmod m$$

for some parameters a, m, x_0.

The next number in a sequence depends only on the present one.

The sequence is periodic with a period of at most m

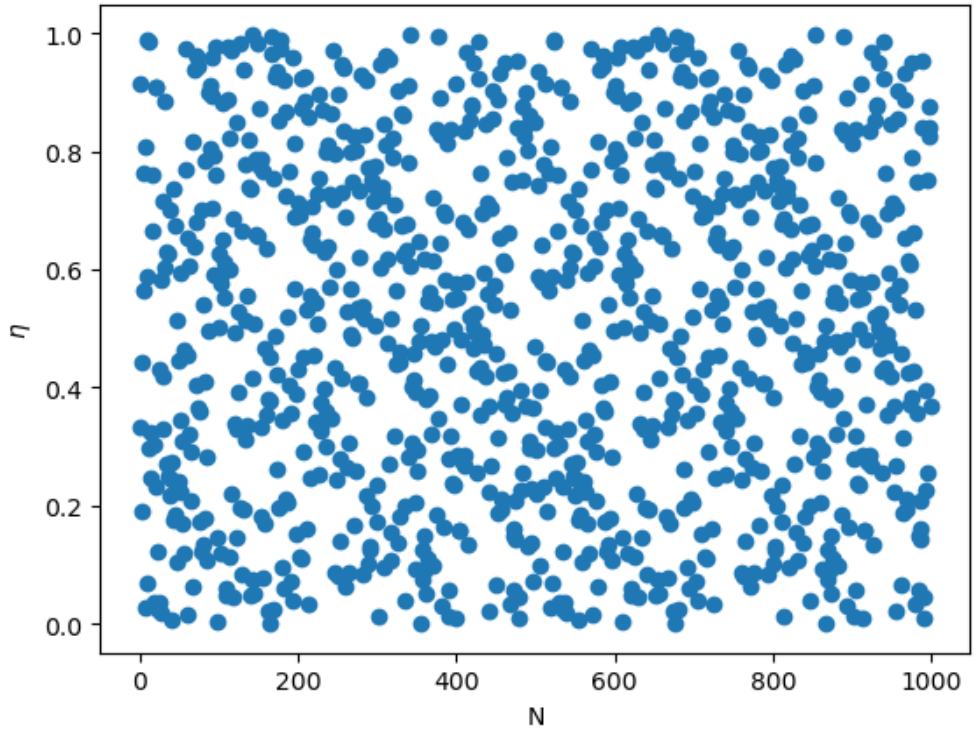
**Do not use LCG in any serious calculation(!)*

C++: avoid using rand()

Linear congruential generator: Example

```python
import numpy as np

# Linear congruential generator

# Parameters (based on Numerical Recipes)
lcg_a = 1664525
lcg_c = 1013904223
lcg_m = 4294967296
# Current value (initial seed)
lcg_x = 1

def lcg():
    global lcg_x
    lcg_x = (lcg_a * lcg_x + lcg_c)%lcg_m
    return lcg_x
```

```python
# Plot
import matplotlib.pyplot as plt

results = []

N = 1000
for i in range(N):
    results.append(lcg()/lcg_m)

plt.xlabel("N")
plt.ylabel("${\eta}$")
plt.plot(results,"o")
plt.show()
```



Linear congruential generator

LCG has some serious drawbacks:

Apart from a rather short period, it also fails many statistical randomness tests.

For instance, if one regards random numbers as components of a vector (x,y,...),
the method tends to generate these points on a hyperplane (spectral test).

Linear congruential generator

LCG has some serious drawbacks:

Apart from a rather short period, it also fails many statistical randomness tests.

For instance, if one regards random numbers as components of a vector (x,y,...), the method tends to generate these points on a hyperplane (spectral test).

```
# Slightly different choice of m
lcg_m = 3000000000

resultsx = []
resultsy = []

N = 1000
for i in range(N):
    resultsx.append(lcg()/lcg_m)
    resultsy.append(lcg()/lcg_m)

plt.plot(resultsx,resultsy,"o")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

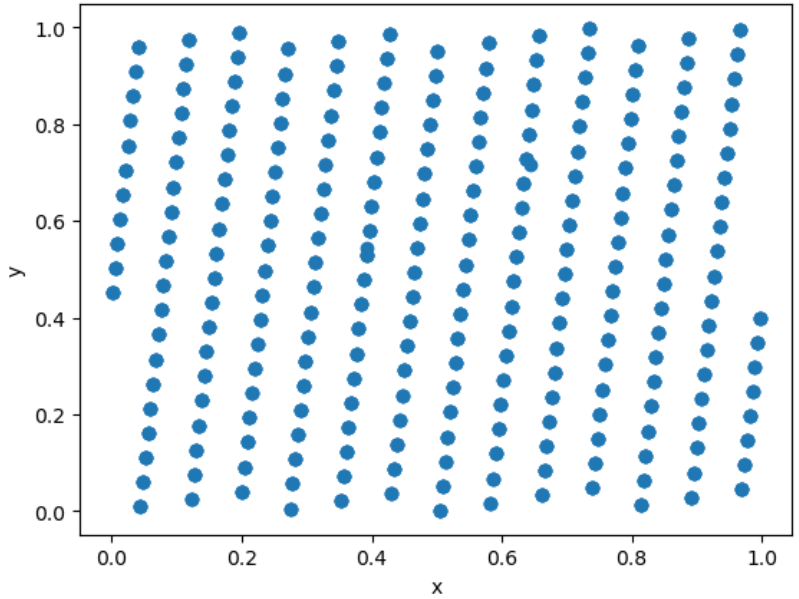
Linear congruential generator

LCG has some serious drawbacks:

Apart from a rather short period, it also fails many statistical randomness tests.

For instance, if one regards random numbers as components of a vector (x,y,...), the method tends to generate these points on a hyperplane (spectral test).

```
# Slightly different choice of m
lcg_m = 3000000000

resultsx = []
resultsy = []

N = 1000
for i in range(N):
    resultsx.append(lcg()/lcg_m)
    resultsy.append(lcg()/lcg_m)

plt.plot(resultsx,resultsy,"o")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Mersenne Twister

LCG is not and should not be used in any serious calculations.

Other methods have been developed over the years and the general method of choice is **Mersenne Twister** random number generator which is implemented by default in many programming environments.

MT has a long period of 2^{19937} - 1, passes most statistical randomness tests, fast, and suitable for most physical applications (except cryptography).

It now implemented by default in many languages and we will take it for granted.

Python:

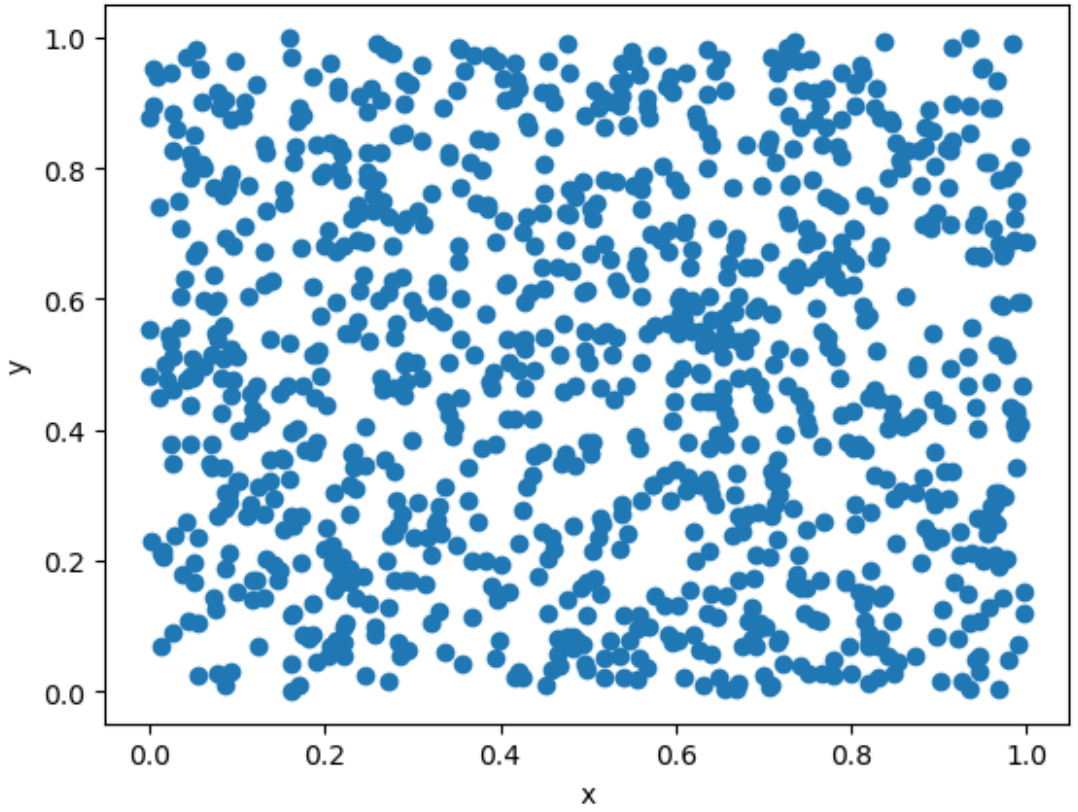
```python
# Use Mersenne Twister
import numpy as np

np.random.rand() # Random number \eta uniformly distributed over (0,1)
```

C++ (since C++11):

```cpp
#include <ctime>
#include <iostream>
#include <random>
using namespace std;

int main()
{
  // Initializing the sequence
  // with a seed value
  // similar to srand()
  mt19937 mt(time(nullptr));

  // Printing a random number
  // similar to rand()
  cout << mt() << '\n';
  return 0;
}
```

Mersenne Twister

```
# Use Mersenne Twister
import numpy as np

np.random.rand() # Random number \eta uniformly distributed over (0,1)

resultsx = []
resultsy = []

N = 1000
for i in range(N):
    resultsx.append(np.random.rand())
    resultsy.append(np.random.rand())

plt.plot(resultsx,resultsy,"o")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



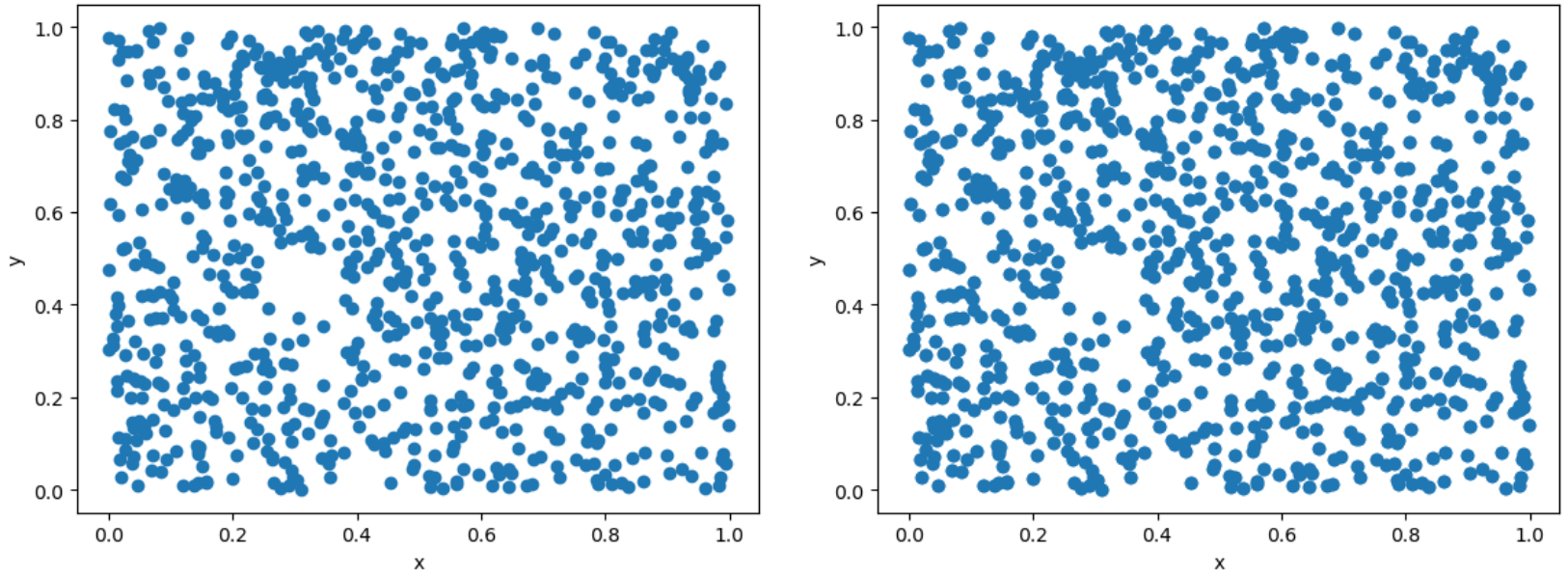
Random seed

Most RNGs (like LCG, Mersenne Twister,...) maintain state variables and iteratively generate a pre-determined sequence of (pseudo)-random numbers

The initial state can be changed by specifying the *seed*

Running the program from the same seed will generate identical outcome

```
resultsx = []
resultsy = []

N = 1000
np.random.seed(1)
for i in range(N):
    resultsx.append(np.random.rand())
    resultsy.append(np.random.rand())

plt.plot(resultsx,resultsy,"o")
plt.xlabel("x")
plt.ylabel("y")
plt.show()

np.random.seed(1)
for i in range(N):
    resultsx.append(np.random.rand())
    resultsy.append(np.random.rand())

plt.plot(resultsx,resultsy,"o")
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



Using the same seed is good for debugging... but bad for parallel production runs on a cluster

Simulation example: Radioactive decay

Example 10.1 from M. Newman, Computational Physics

Some physical processes are truly random (recall quantum mechanics), for instance **radioactive decay**

The number of radioactive isotopes with a half-life of τ evolves as

$$N(t) = N(0)2^{-t/\tau},$$

therefore, the probability for a single atom to decay over the time interval t is

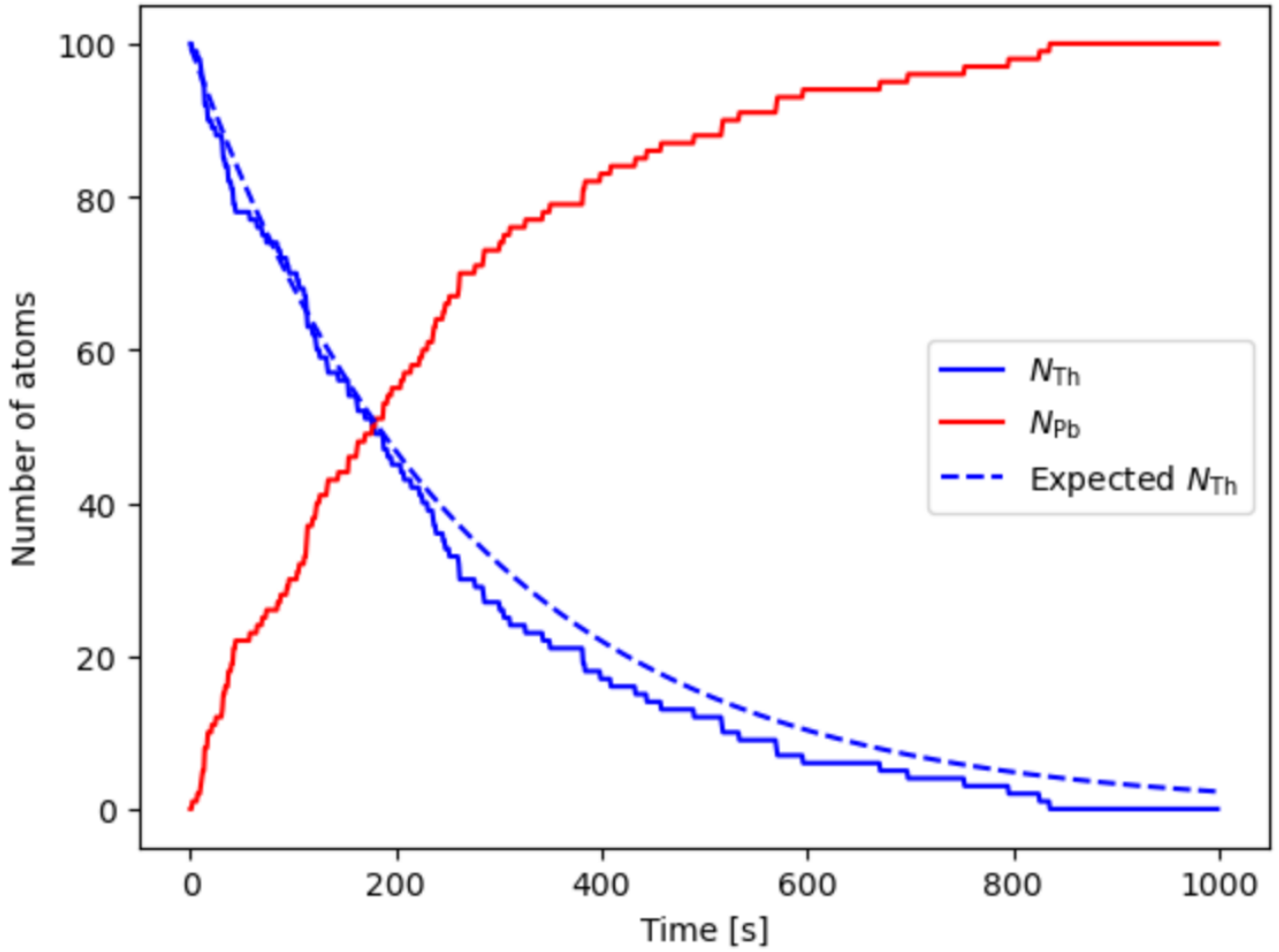
$$p(t) = 1 - 2^{-t/\tau}.$$

Let us simulate the time evolution for a sample of thallium atoms decaying (half-life of $\tau = 3.053$ mins) into lead atoms.

```
# Decay constants
NTl = 100             # Number of thallium atoms
NPb = 0               # Number of lead atoms
tau = 3.053*60        # Half life of thallium in seconds
h = 1.0               # Size of time-step in seconds
p = 1 - 2**(-h/tau)   # Probability of decay in one step
tmax = 1000           # Total time
ctime = 0             # Current time

# Lists of plot points
tpoints = np.arange(0.0,tmax,h)
Tlpoints = []
Pbpoints = []
```

```
# Main loop
for t in tpoints:
    Tlpoints.append(NTl)
    Pbpoints.append(NPb)

    # Calculate the number of atoms that decay
    decay = 0
    for i in range(NTl):
        if np.random.rand()<p:
            decay += 1
    NTl -= decay
    NPb += decay
```

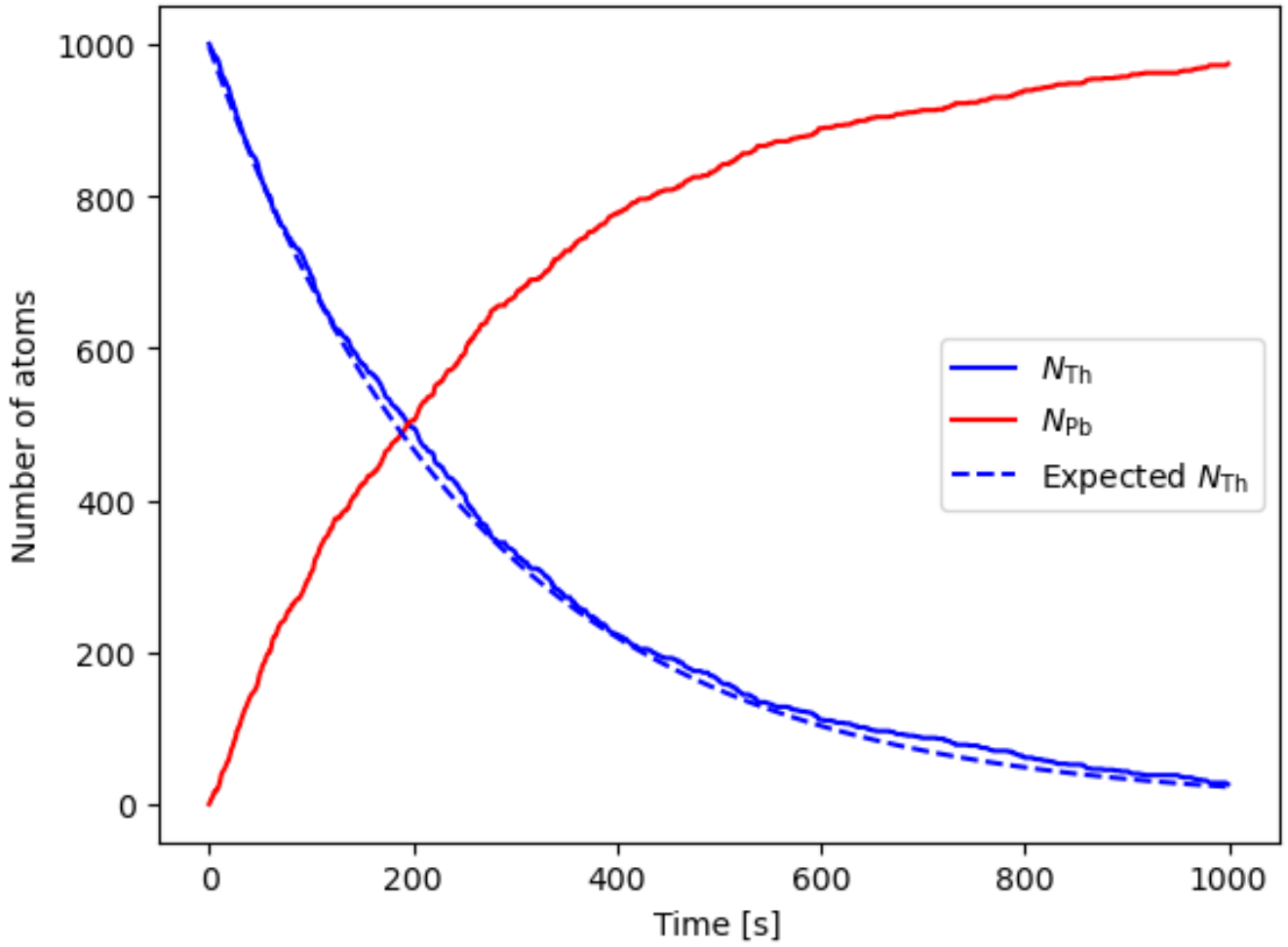
Simulation example: Radioactive decay



Expected:

$$N(t) = N(0)2^{-t/\tau},$$

Simulation example: Radioactive decay



Expected:

$$N(t) = N(0)2^{-t/\tau},$$

Simulation example: Brownian motion

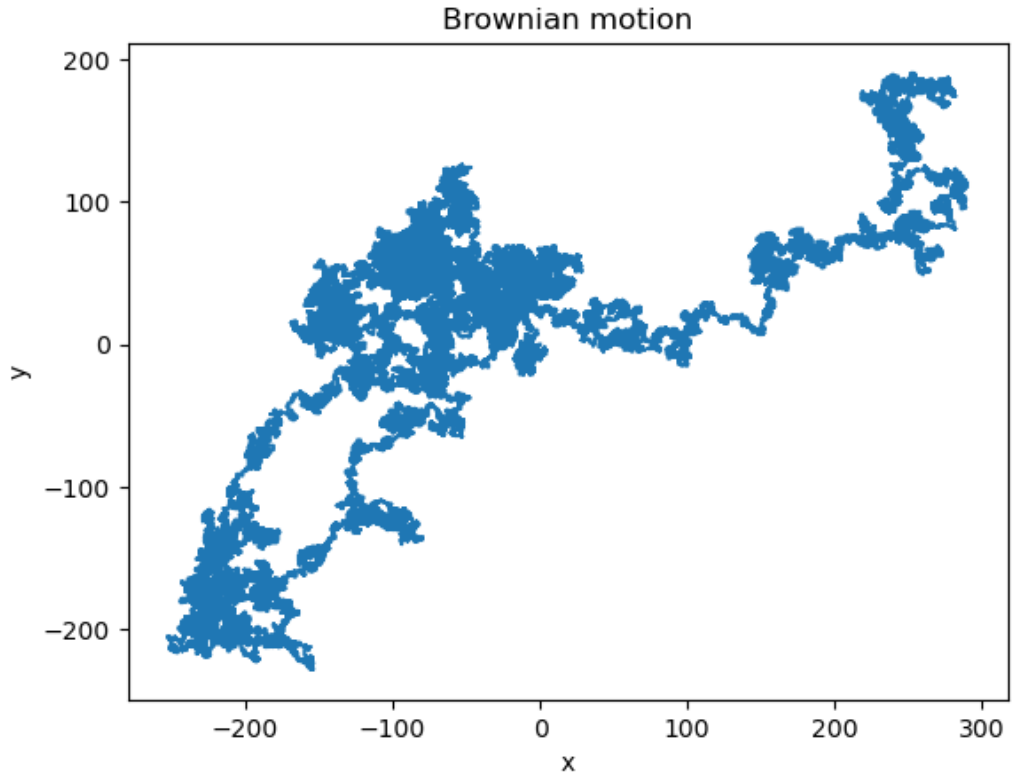
Brownian motion is a motion of a heavy particle in a gas colliding with the lighter gas particles.
We can consider a simplified 2D motion of particle by randomly making
a small step at each iteration in one of the four directions.

```
N = 100000
x = 0
y = 0

dirs = [ [1,0], [-1,0], [0,1], [0,-1] ]

points_x = [x]
points_y = [y]
for i in range(N):
    direction = np.random.randint(4)
    x += dirs[direction][0]
    y += dirs[direction][1]
    points_x.append(x)
    points_y.append(y)
```

Simulation example: Brownian motion

Brownian motion is a motion of a heavy particle in a gas colliding with the lighter gas particles.
We can consider a simplified 2D motion of particle by randomly making
a small step at each iteration in one of the four directions.

```
N = 100000
x = 0
y = 0

dirs = [ [1,0], [-1,0], [0,1], [0,-1] ]

points_x = [x]
points_y = [y]
for i in range(N):
    direction = np.random.randint(4)
    x += dirs[direction][0]
    y += dirs[direction][1]
    points_x.append(x)
    points_y.append(y)
```

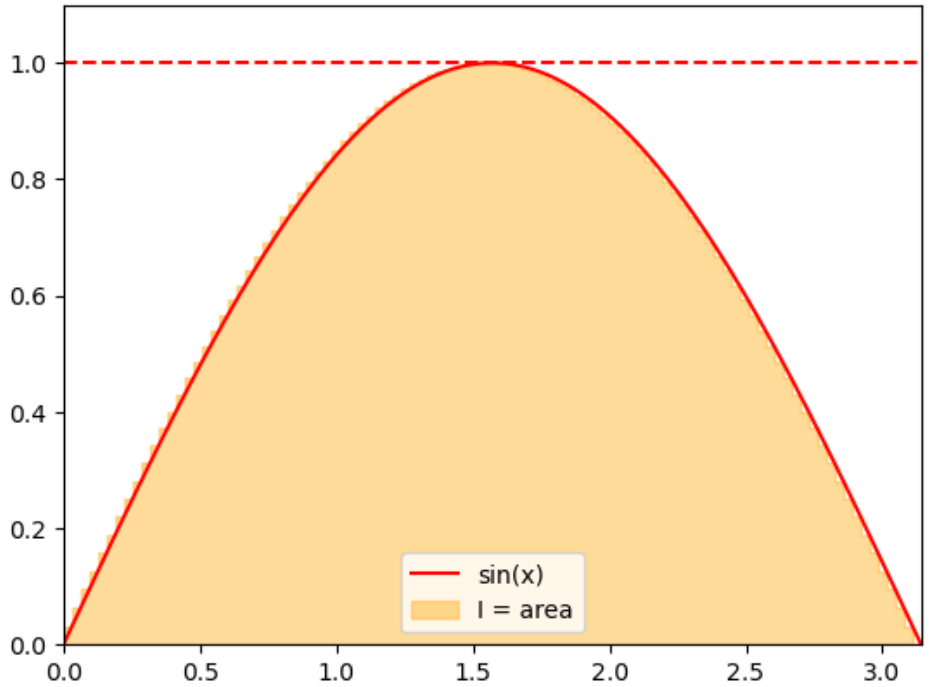


Computing integrals: Estimating the area under the curve

Recall the interpretation of a definite integral as the area under the curve.
We can use this interpretation to apply random numbers for approximating integrals.

Consider

$$I = \int_0^{\pi} \sin(x)dx$$



Computing integrals: Estimating the area under the curve

We can estimate the area by sampling the points uniformly from an enveloping rectangle and counting the fraction of points under the curve given by the integrand f(x).

Assuming an integral

$$I = \int_a^b f(x)dx$$

where f(x) $>$ 0 and f(x) $<$ y_{max}, the integrand can be evaluated as

$$I = (b-a)y_{\max}\frac{C}{N},$$

where C is the number of the sampled points that fall under f(x).

The statistical error of the integrand can be estimated using the properties of the binomial distribution with p = C/N:

$$\delta I = (b-a)y_{\max}\sqrt{\frac{p(1-p)}{N}}$$

The error scales with $N^{-1/2}$

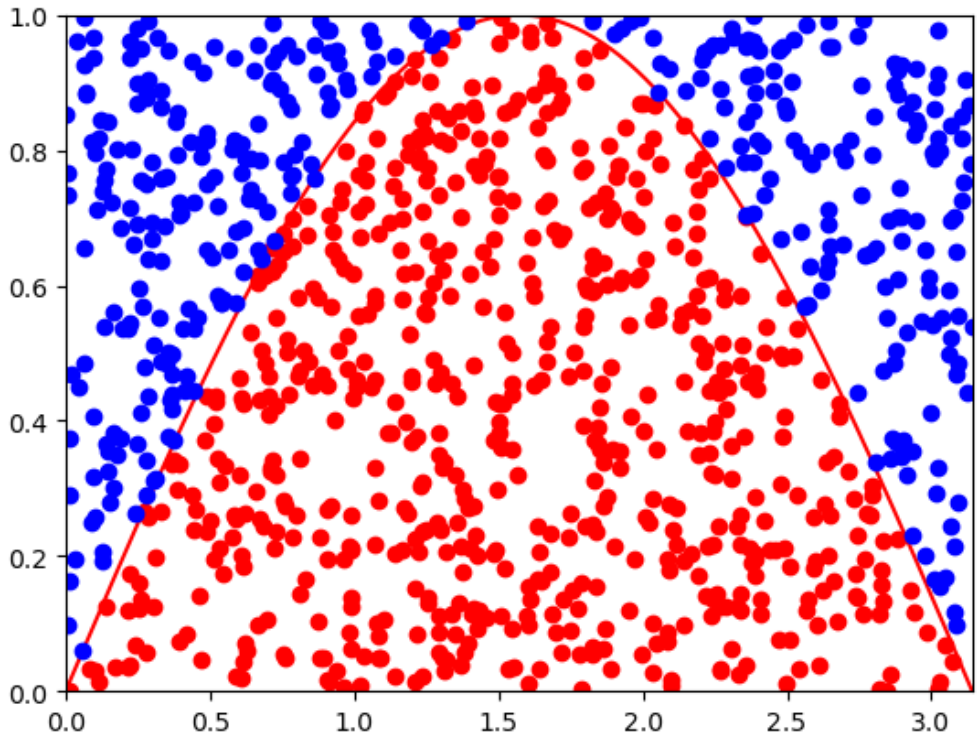
To reduce the error by factor x2 we need to sample x4 more numbers – true for most Monte Carlo methods.

Computing integrals: Estimating the area under the curve

```
# For visualization
points_in = []
points_out = []

# Compute integral \int_a^b f(x) dx as an area below the curve
# Assumes that f(x) is non-negative and bounded from above by ymax
# Returns the value of the integral and the error estimate
def areaMC(f, N, a, b, ymax):
    global points_in, points_out
    points_in = []
    points_out = []
    count = 0
    for i in range(N):
        x = a + (b-a)*np.random.rand()
        y = ymax * np.random.rand()
        if y<f(x):
            count += 1
            points_in.append([x,y])
        else:
            points_out.append([x,y])
    p = count/N
    return (b-a) * ymax * p, (b-a) * ymax * np.sqrt(p*(1-p)/N)
```

```
def f(x):
    return np.sin(x)

N = 1000
I, err = areaMC(f, N, 0, np.pi, 1)
print("I = ",I," +- ",err)
```

```
I =  2.004336112990288  +-  0.04774352682885915
```



Computing π

Consider a circle of unit radius r = 1. Its area is:

$$A = \pi r^2 = \pi$$

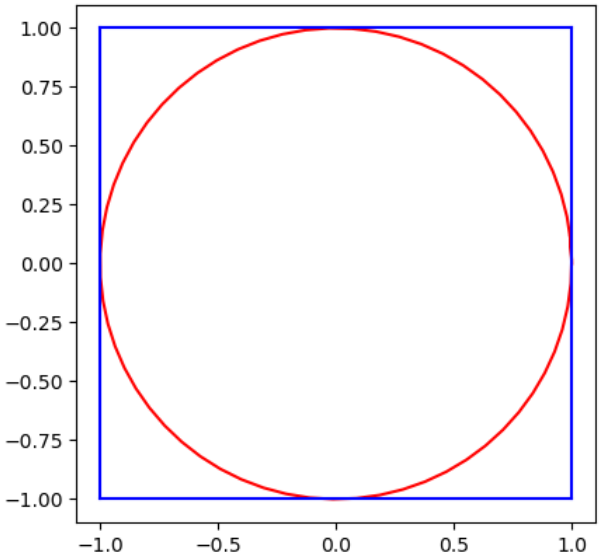
The circle can be embedded into a square with a side length of two. The area of the square is: $A_{sq} = 2^2 = 4$

Consider now a random point anywhere inside the square. The probability that it is also inside the circle is the ratio of their areas:

$$P = \frac{A}{A_{\text{sq}}} = \frac{\pi}{4}.$$

This probability can be estimated by sampling points inside the square many times and counting how many fall inside the circle. π can therefore be estimated as:

$$\pi = 4\frac{A}{A_{\text{sq}}}$$



Computing π

Consider a circle of unit radius r = 1. Its area is:

$$A = \pi r^2 = \pi$$

The circle can be embedded into a square with a side length of two. The area of the square is: $A_{sq} = 2^2 = 4$

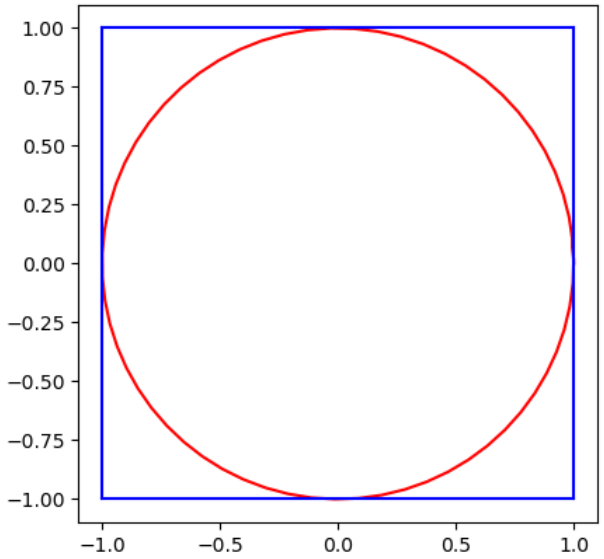
Consider now a random point anywhere inside the square. The probability that it is also inside the circle is the ratio of their areas:

$$P = \frac{A}{A_{\text{sq}}} = \frac{\pi}{4}.$$

This probability can be estimated by sampling points inside the square many times and counting how many fall inside the circle. π can therefore be estimated as:

```
# Compute the value of \pi through the fraction of random points inside a square
# that are also inside a circle around the origin
# Returns the value of the integral and the error estimate
def piMC(N):
    global points_in, points_out
    points_in = []
    points_out = []
    count = 0
    for i in range(N):
        x = -1 + 2 * np.random.rand()
        y = -1 + 2 * np.random.rand()
        r2 = x**2 + y**2
        if (r2 < 1.):
            count += 1
            points_in.append([x,y])
        else:
            points_out.append([x,y])
    p = count/N
    return 4. * p, 4. * np.sqrt(p*(1-p)/N)
```

$$\pi = 4\frac{A}{A_{\text{sq}}}$$



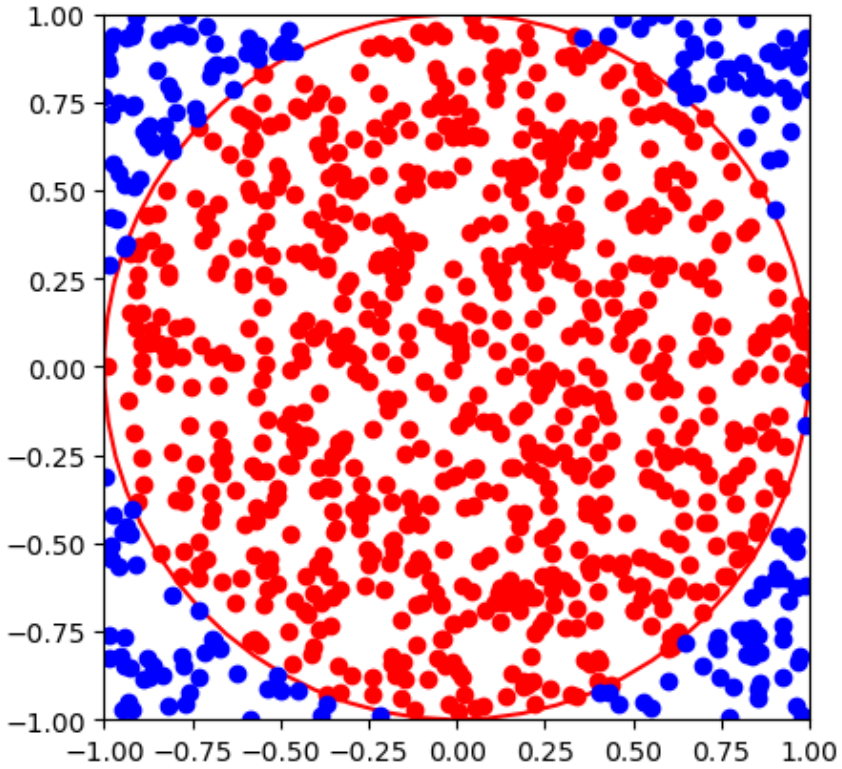
This method is known as the **Monte Carlo estimation of π**.

Computing pi

```
N = 1000
piMC, piMCerr = piMC(N)
print("pi = ",piMC," +- ",piMCerr)
```

```
pi =  3.208  +-  0.050405713961811906
```

Try a larger number of points



Computing integral as the average

- The integral of a function over an interval (a, b) is given by:

$$I = \int_a^b f(x)dx$$

- The mean value of f(x) over (a, b) is:

$$\langle f \rangle = \frac{\int_a^b f(x)dx}{b-a} = \frac{I}{b-a}, \qquad \text{which gives:} \quad I = (b-a)\langle f \rangle$$

- The integral can be estimated by computing the average value of f(x), where x is randomly sampled over (a, b):

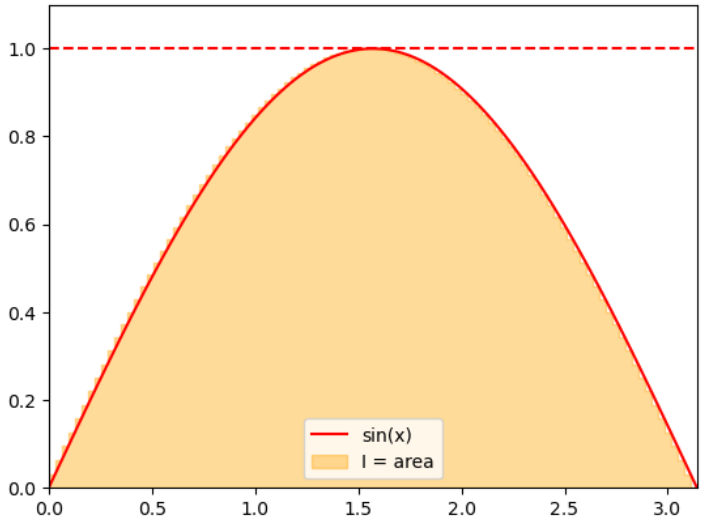
$$\langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i).$$

- Using the law of averages, the error estimate involves the variance of f(x):

$$\delta I = (b-a)\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$

```
def intMC(f, N, a, b):
    total = 0
    total_sq = 0
    for i in range(N):
        x = a + (b-a)*np.random.rand()
        fval = f(x)
        total += fval
        total_sq += fval * fval
    f_av = total / N
    fsq_av = total_sq / N
    return (b-a) * f_av, (b-a) * np.sqrt((fsq_av - f_av*f_av)/N)
```

Computing integral as the average




```
def f(x):
    return np.sin(x)

N = 1000
I, err = intMC(f, N, 0, np.pi)
print("I = ",I," +- ",err)
```

```
I =  1.964605422837963  +-  0.030792720278272654
```

Advantages:

- the method works also if $f(x)$ is negative
- no need to know its maximum value

Another way to compute pi

Consider an integral

$$4 \int_0^1 \frac{1}{1+x^2} dx = 4 \arctan(x)|_0^1 = \pi$$

Another way to compute pi

Consider an integral

$$4\int_0^1 \frac{1}{1+x^2}dx = 4\arctan(x)|_0^1 = \pi$$

```
def fpi(x):
    return 4 / (1 + x**2)

N = 10000
I, err = intMC(fpi, N, 0, 1)
print("pi = ",I," +- ",err)
```

```
pi =  3.1365784339451928  +-  0.006449180867490663
```

Computing multi-dimensional integrals

Monte Carlo methods really shine when it comes to numerical evaluation of integrals in multiple dimensions. Consider the following D-dimensional integral

$$I = \int_{a_1}^{b_1} dx_1 \ldots \int_{a_D}^{b_D} dx_D f(x_1, \ldots, x_D).$$

Computing it numerically using for instance the *rectangle rule* would involve the evaluation of a multi-dimensional sum

$$I \approx \sum_{k_1=1}^{N_1} \ldots \sum_{k_D=1}^{N_D} f(x_{k_1}, \ldots, x_{k_D}) \prod_{d=1}^{D} h_d,$$

where $h_d = (b_d - a_d)/N_d$ and $x_{k_d} = a_d + h_d(k_d - 1/2)$.

The total number of integrand evaluations is $N_{tot} = \prod_{d=1}^{N_D} N_d$,
e.g. if we use the same number N of points in each dimension, N_{tot} scales exponentially with D

$$N_{\text{tot}} = N^D$$

curse of dimensionality

Computing multi-dimensional integrals: Monte Carlo

Similar to 1D case, replace

$$I = \int_{a_1}^{b_1} dx_1 \ldots \int_{a_D}^{b_D} dx_D f(x_1, \ldots, x_D).$$

by the mean

$$I = \langle f(x_1, \ldots, x_D) \rangle \prod_{k=1}^{D} (b_k - a_k).$$

Here $x_1,\ldots,x_D$ are independent random variables distributed uniformly in intervals x_k in $[a_k,b_k]$.

Error estimate:

$$\delta I = \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}} \prod_{k=1}^{D} (b_k - a_k),$$

Increasing the number of dimensions by one: sample one more number each iteration.

linear complexity in D

Computing multi-dimensional integrals: Monte Carlo

Implementation:

```python
# Evaluating a multi-dimensional integral
# by sampling uniformly distributed numbers
# and calculating the average of the integrand
def intMC_multi(f, nMC, a, b):
    dim = len(a)

    total = 0
    total_sq = 0
    for iMC in range(nMC):
        x = [a[idim] + (b[idim] - a[idim]) * np.random.rand() for idim in range(dim)]
        fval = f(x)
        total += fval
        total_sq += fval * fval

    f_av = total / nMC
    fsq_av = total_sq / nMC

    vol = 1.
    for idim in range(dim):
        vol *= (b[idim] - a[idim])

    return vol * f_av, vol * np.sqrt((fsq_av - f_av*f_av)/nMC)
```

Computing multi-dimensional integrals: Monte Carlo

Our example:

$$I = \int_0^{\pi/2} dx_1 \ldots \int_0^{\pi/2} dx_D \; \sin(x_1 + x_2 + \ldots + x_D).$$

```
%%time

def f(x):
    xsum = 0
    for i in range(len(x)):
        xsum += x[i]
    return np.sin(xsum)

Ndimmax = 10
NMC = 1000000
for Ndim in range(1,Ndimmax + 1):
    a = [0. for i in range(Ndim)]
    b = [np.pi/2 for i in range(Ndim)]
    I, Ierr = intMC_multi(f,NMC,a,b)
    print("D =",Ndim, " I =",I,"+-",Ierr)
```

```
D = 1  I = 1.0001760548105423 +- 0.00048329078936716987
D = 2  I = 2.0003828593503097 +- 0.000526917899834823
D = 3  I = 1.999779600266565 +- 0.0018730526051484867
D = 4  I = 0.0016542203843071606 +- 0.003935579972226937
D = 5  I = -4.003942552547165 +- 0.00545377224016235
D = 6  I = -8.007809542583617 +- 0.007499080458630796
D = 7  I = -7.997053750388268 +- 0.01464987689110119
D = 8  I = 0.012579382216618208 +- 0.02586481622201921
D = 9  I = 15.948080294527836 +- 0.03792482973598219
D = 10  I = 31.965268795252253 +- 0.056583114947530044
CPU times: user 19.7 s, sys: 220 ms, total: 19.9 s
Wall time: 19.9 s
```

Analytic result:

D	I =
1	1
2	2
3	2
4	0
5	-4
6	-8

Volume of a D-dimensional ball (hypersphere)

Let us consider an D-dimensional ball of radius R.
Its volume is given by a D-dimensional integral

$$V_D(R) = \int_{\sqrt{x_1^2+\ldots x_D^2}<R} dx_1 \ldots dx_D.$$

This can be written with the recursion formula

$$V_D(R) = R^D \int_{-1}^{1} V_{D-1}\left(\sqrt{1-t^2}\right) dt,$$

with $V_0(R) = 1$.

Rectangle (non-MC) method (recursive)

```
# Computes volume of a D-dimensional ball
# using a recursion relation and rectangle rule
# with nrect slices for each dimension
def VD(D, R, nrect):
    if (D == 0):
        return 1.

    ret = 0.
    h = 2. / nrect;
    for k in range(nrect):
        xk = -1. + h * (k+1/2.)
        ret += VD(D-1,np.sqrt(1-xk**2), nrect)
    ret *= h * R**D
    return ret
```

```
nrect = 50
for n in range(5):
    print("V",n,"(1) = ",VD(n,1,nrect))
```

```
V 0 (1) =  1.0
V 1 (1) =  2.0
V 2 (1) =  3.144340711294003
V 3 (1) =  4.193292772581682
V 4 (1) =  4.940233310235603
CPU times: user 2.81 s, sys: 53 ms, total: 2.86 s
Wall time: 2.86 s
```

Volume of a D-dimensional ball (hypersphere)

Monte Carlo approach:

Observe that the ball $\sqrt{x_1^2 + \cdots + x_D^2} < R$ is a subvolume of a hypercube $-R < x_1, \ldots, x_D < R$.

If we now randomly sample points that are uniformly distributed inside the hypercube,
the fraction C/N of those that are also inside the ball will reflect
the ratio of the ball and hypercube volumes $V_D(R)$ and $V_{cube}(R) = (2R)^D$

Therefore,

$$V_D(R) = (2R)^D \frac{C}{N}$$

Volume of a D-dimensional ball (hypersphere)

$$V_D(R) = (2R)^D \frac{C}{N}$$

```
def VD_MC(D, R, N = 100):
    if (D == 0):
        return 1., 0.
    count = 0
    for iMC in range(N):
        xs = [-R + 2 * R * np.random.rand() for i in range(n)]
        r2 = 0.
        for i in range (D):
            r2 += xs[i]**2
        if (r2 < R**2):
            count += 1

        p = count/N
    return (2*R)**D * p, (2*R)**D * np.sqrt(p*(1-p)/N)
```

```
nMC = 100000
for n in range(11):
    Vnval, Vnerr = VD_MC(n, 1, nMC)
    print("V",n,"(1) = ",Vnval, "+-", Vnerr)
```

```
V 0 (1) =  1.0 +- 0.0
V 1 (1) =  2.0 +- 0.0
V 2 (1) =  3.13532 +- 0.00520677299063441
V 3 (1) =  4.18496 +- 0.012635580635016342
V 4 (1) =  4.94176 +- 0.023376733754397767
V 5 (1) =  5.2224 +- 0.037395633199613025
V 6 (1) =  5.1008 +- 0.054811772399731784
V 7 (1) =  4.73088 +- 0.0763656607661847
V 8 (1) =  4.20864 +- 0.10294169171673836
V 9 (1) =  3.05152 +- 0.12462208735571717
V 10 (1) =  2.51904 +- 0.16041045469290335
```

Nonuniformly distributed random numbers

In many cases we deal with random numbers ξ that are distributed non-uniformly.

Common examples are:

- Exponential distribution $\rho(x) = e^{-x}$.
- Gaussian distribution $\rho(x) \propto e^{-\frac{x^2}{2\sigma^2}}$.
- Power-law distribution $\rho(x) \propto x^{\alpha}$.
- Arbitrary peaked distributions.

There are two common methods for generating nonuniform random variates.
They both make use of uniformly distributed variates.

- Inverse transform sampling
- Rejection sampling

Inverse transform sampling

The basic idea is that if η is a uniformly distibuted random variable, some function of it, $\xi = f(\eta)$, is not. The idea is to sample η and calculate ξ via this function such that ξ corresponds to a desired probability density $\rho(\xi)$. How to find the function $f(\eta)$?

Without the loss of generality assume that $\xi \in (-\infty, \infty)$ and that $f(\eta)$ maps η to ξ such that $f(0) \to -\infty$. Consider now the cumulative distribution function

$$G(x) = Pr(\xi < x) = \int_{-\infty}^{x} \rho(\xi) d\xi.$$

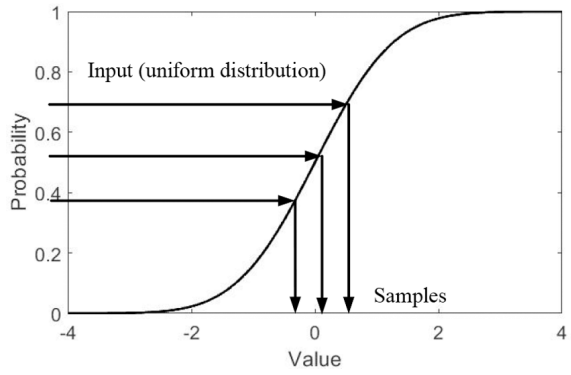
It corresponds to the probability that $\eta < y$ where y is such that $x = f(y)$. Since η is uniformly distributed, this probablity equals to y. Therefore,

$$G[x = f(y)] = y,$$

thus

$$f(y) = G^{-1}(y).$$

If we can calculate the inverse of $G^{-1}(y)$ of the cumulative distribution function for ξ, we are good.



Inverse transform sampling

The algorithm follows these steps:

1. Calculate the cumulative distribution function (CDF)

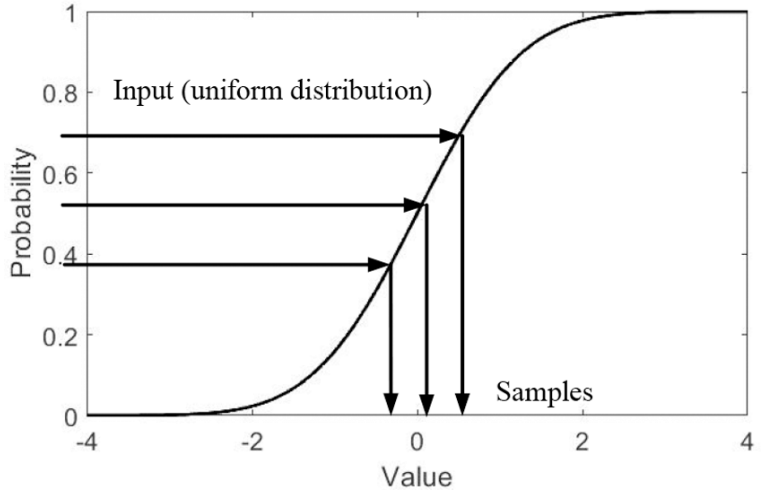
$$G(x) = \int_{-\infty}^{x} \rho(\xi) d\xi$$

2. Find the inverse function $G^{-1}(y)$ as the solution to the equation

$$G(x) = y$$

3. Sample uniformly distributed random variables η and compute ξ using the inverse function:

Challenges: Sometimes, evaluating G(x) and/or $G^{-1}(y)$ explicitly is difficult. In such cases, numerical integration and/or non-linear equation solvers may be required.



Inverse transform sampling: Exponential distribution

Example: Exponential Distribution

1. Recall the radioactive decay process. The time of decay is distributed according to the probability density function:

$$\rho(t) = \frac{1}{\tau} e^{-\frac{t}{\tau}}.$$

2. The cumulative distribution function is given by:

$$F(x) = \int_0^x \frac{1}{\tau} e^{-\frac{t}{\tau}} dt = 1 - e^{-\frac{x}{\tau}}.$$

3. To apply inverse transform sampling, we need to invert F(x) by solving:

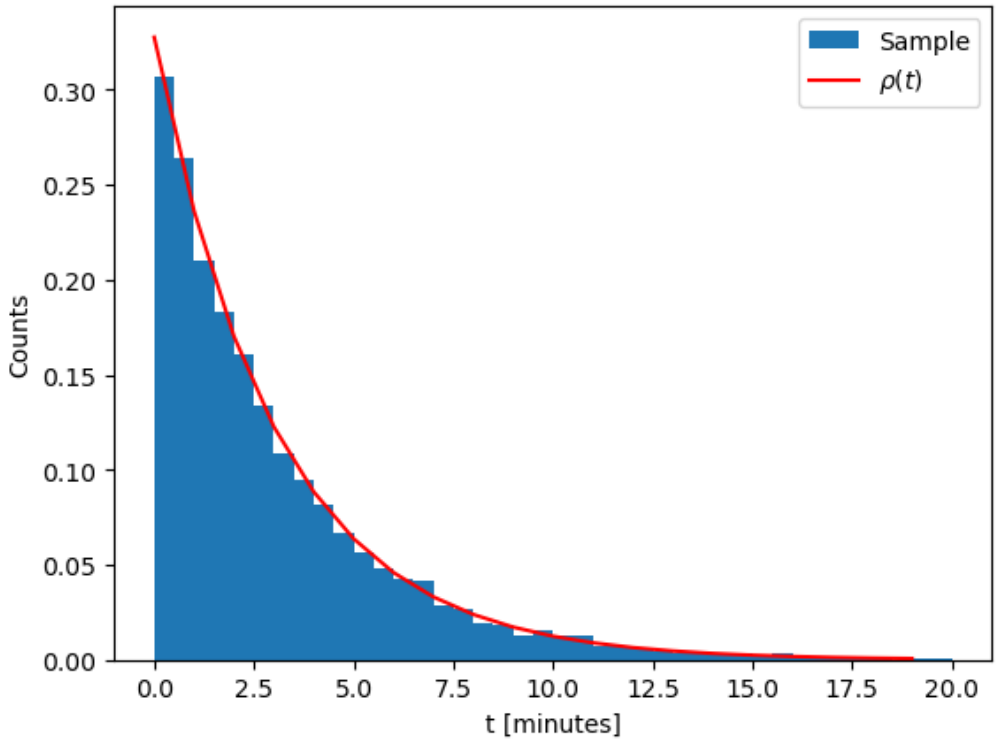
$$1 - e^{-\frac{t}{\tau}} = \eta.$$

4. Solving for t, we obtain:

$$t(\eta) = -\tau \ln(1 - \eta).$$

Sampling radioactive decay time

```python
## Radioactive decay sampler
def sample_tdecay(tau):
    eta = np.random.rand()
    return -tau * np.log(1-eta)


tau = 3.053 # Half-time in minutes
N = 10000   # Number of samples
tdecays = [sample_tdecay(tau) for i in range(N)]

# Show a histogram
plt.xlabel("t [minutes]")
plt.ylabel("Counts")
plt.hist(tdecays, bins = 40, range=(0,20), density=True)
```



Sampling points inside a circle

One way to sample points inside a unit circle is by switching to **polar coordinates**:

$$x = r\cos(\phi), \qquad y = r\sin(\phi),$$

where we sample r ∈ [0,1) and $\phi \in [0, 2\pi)$.

Naively, one could sample r and ϕ independently from two uniform distributions. Let's see what happens!

```
def sample_xy_naive():
    r = np.random.rand()
    phi = 2 * np.pi * np.random.rand()
    return r*np.cos(phi), r*np.sin(phi)

xplot = []
yplot = []
N = 1000
for i in range(N):
    x, y = sample_xy_naive()
    xplot.append(x)
    yplot.append(y)

plt.plot(xplot,yplot,'o',color='r')
plt.show()
```

Sampling points inside a circle

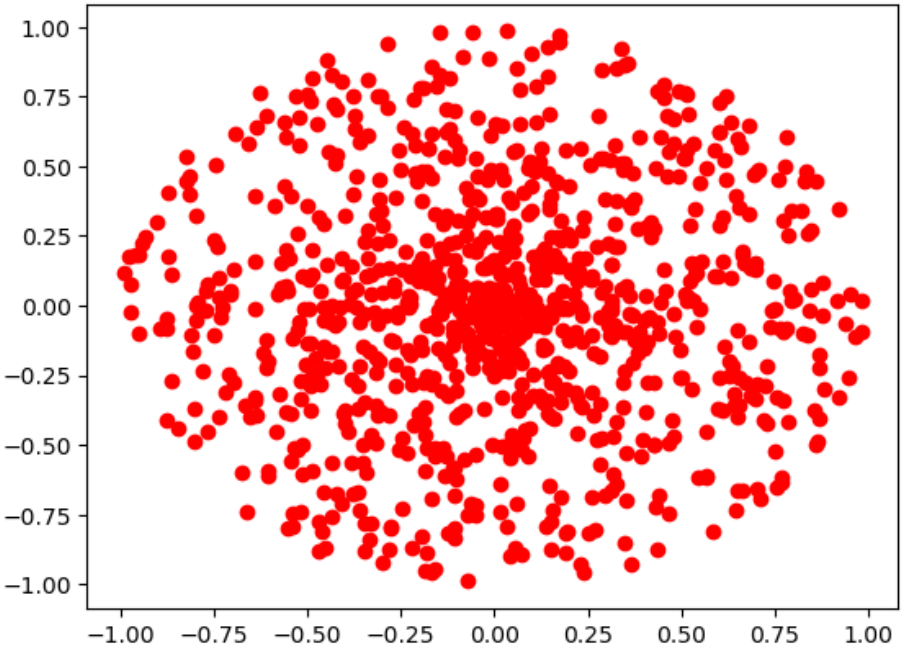
One way to sample points inside a unit circle is by switching to **polar coordinates**:

$$x = r\cos(\phi), \qquad y = r\sin(\phi),$$

where we sample r ∈ [0,1) and $\phi \in [0, 2\pi)$.

Naively, one could sample r and ϕ independently from two uniform distributions. Let's see what happens!

```
def sample_xy_naive():
    r = np.random.rand()
    phi = 2 * np.pi * np.random.rand()
    return r*np.cos(phi), r*np.sin(phi)

xplot = []
yplot = []
N = 1000
for i in range(N):
    x, y = sample_xy_naive()
    xplot.append(x)
    yplot.append(y)

plt.plot(xplot,yplot,'o',color='r')
plt.show()
```



The points clump more in the center!

Sampling points inside a circle

- The points **clump more in the center** because r is **not uniformly distributed**.
- Recall the **differential area element** in polar coordinates: $dxdy = rdrd\phi$
- This leads to the probability density functions: $\rho_r(r) = 2r, \qquad \rho_\phi(\phi) = \frac{1}{2\pi}.$
- **Cumulative Distribution Function (CDF)**

$$F_r(r) = \int_0^r \rho_r(r')dr' = r^2.$$

- To obtain a properly distributed r, we solve $F_r(r) = \eta$ which gives: $r = \sqrt{\eta}$

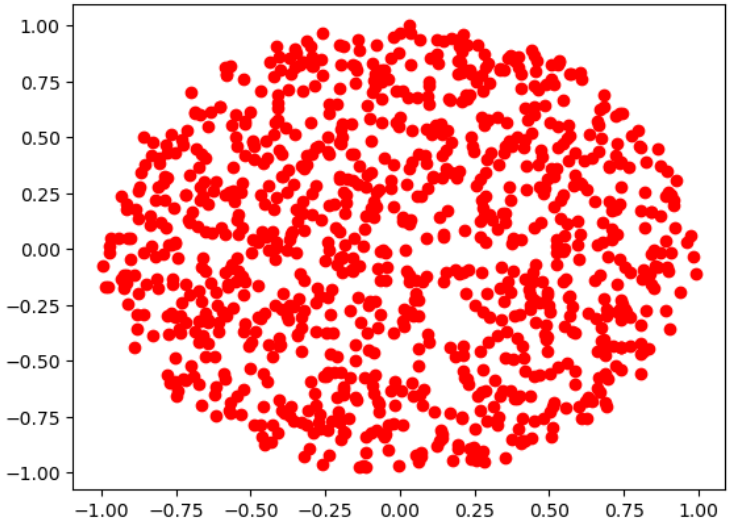
Sampling points inside a circle

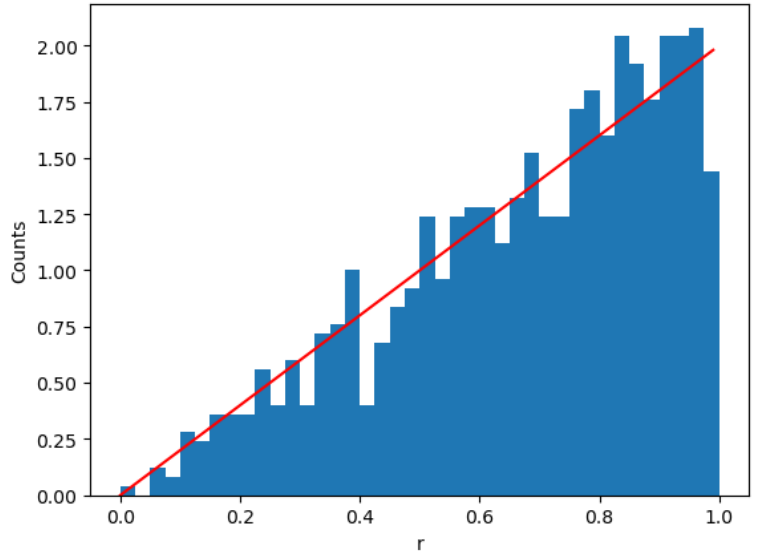
- The points **clump more in the center** because r is **not uniformly distributed**.
- Recall the **differential area element** in polar coordinates: $dxdy = rdrd\phi$
- This leads to the probability density functions: $\rho_r(r) = 2r, \qquad \rho_\phi(\phi) = \frac{1}{2\pi}.$
- **Cumulative Distribution Function (CDF)**

$$F_r(r) = \int_0^r \rho_r(r')dr' = r^2.$$

- To obtain a properly distributed r, we solve $F_r(r) = \eta$ which gives: $r = \sqrt{\eta}$

```
def sample_xy_correct():
    eta = np.random.rand()
    r = np.sqrt(eta)
    phi = 2 * np.pi * np.random.rand()
    return r*np.cos(phi), r*np.sin(phi)
```





Sampling of an Isotropic Direction in 3D

One common problem in **Monte Carlo simulations** is the **random sampling of an isotropic direction in 3D space**. This issue arises in various contexts, such as:

- Sampling a random orientation of an **axially symmetric object** (e.g., a rod).
- Sampling the **momentum direction** of a particle.

This problem is equivalent to **choosing a random point on a unit sphere**. The coordinates x, y, z on the sphere can be parametrized using **azimuthal** and **polar angles**:

- $\phi \in [0, 2\pi)$ (azimuthal angle)
- $\theta \in [0, \pi)$ (polar angle)

Using these angles, the Cartesian coordinates are given by:

$$x = \sin\theta \cos\phi$$

$$y = \sin\theta \sin\phi$$

$$z = \cos\theta$$

Sampling of an Isotropic Direction in 3D

$$x = \sin\theta\cos\phi$$
$$y = \sin\theta\sin\phi$$
$$z = \cos\theta$$

- The solid angle element is:

$$d\Omega = \sin(\theta)d\theta d\phi,$$

- The random variables ϕ and θ are independent.
- ϕ is uniformly distributed in $[0, 2\pi]$, making its sampling straightforward.
- However, θ has a **weighted probability density function**: $\rho_\theta(\theta) = \frac{1}{2}\sin(\theta),$

The cumulative distribution function is:

$$F_\theta(\theta) = \int_0^\theta \frac{1}{2}\sin(\theta')d\theta' = \frac{1 - \cos(\theta)}{2}$$

Solving $F_\theta(\theta) = \eta$, we obtain:

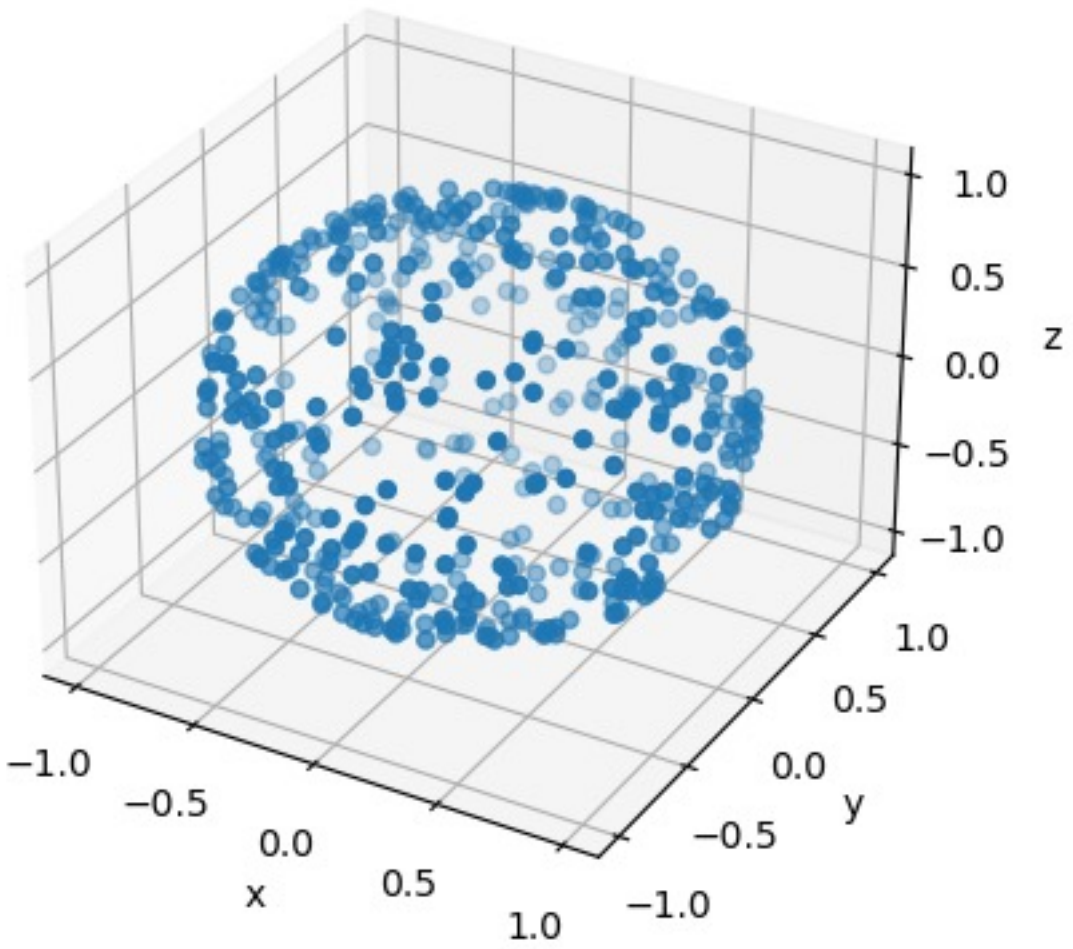
$$\theta = \arccos(2\eta - 1).$$

In practice, work directly with $\cos\theta$ and $\sin\theta$:

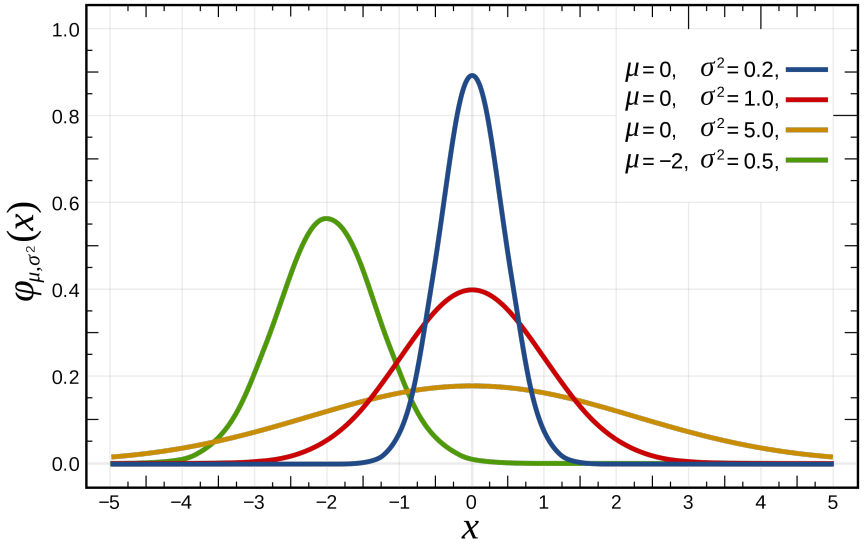
$$\cos(\theta) = 2\eta - 1, \qquad \sin(\theta) = \sqrt{1 - [\cos(\theta)]^2}$$

Sampling an isotropic direction

```python
def sample_xyz_isotropic():
    phi = 2 * np.pi * np.random.rand()
    costh = 2 * np.random.rand() - 1
    sinth = np.sqrt(1-costh*costh)
    return sinth * np.cos(phi), sinth * np.sin(phi), costh

xplot = []
yplot = []
zplot = []
N = 500
for i in range(N):
    x, y, z = sample_xyz_isotropic()
    xplot.append(x)
    yplot.append(y)
    zplot.append(z)

fig = plt.figure()
ax = fig.add_subplot(projection='3d')

ax.scatter(xplot,yplot,zplot)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

plt.show()
```



Sampling normally distributed variables



One of the most common probability distributions is the **normal (or Gaussian) distribution**, given by:

$$\rho(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

There are many standard methods for sampling from this distribution. One common approach is to **standardize the variable** by making the transformation $x \to \mu + \sigma x$

After this transformation, the new variable follows a **standard normal distribution** with **zero mean** and **unit standard deviation**:

$$\rho(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}.$$

Calculating the cumulative distribution function $F(x) = \int_{-\infty}^{x} \rho(x')dx'$ is **not straightforward**, use numerical methods.

Sampling normally distributed variables

Instead of one variable, we can consider a pair of independent normally distributed variables x, y:

$$\rho(x, y) = \frac{1}{2\pi} e^{-\frac{x^2}{2}} e^{-\frac{y^2}{2}},$$

Making a change of variables to polar coordinates

$$x = r\cos(\phi), \qquad y = r\sin(\phi),$$

and taking into account

$$dxdy = rdrd\phi$$

we get

$$\rho(r, \phi) = \frac{1}{2\pi} re^{-r^2/2}.$$

Therefore, we can sample x and y by sampling two independent random variables r and ϕ. ϕ is uniformly distributed in $[0, 2\pi)$. For r we have the following probability density

$$\rho_r(r) = re^{-r^2/2},$$

and the cumulative distribution function

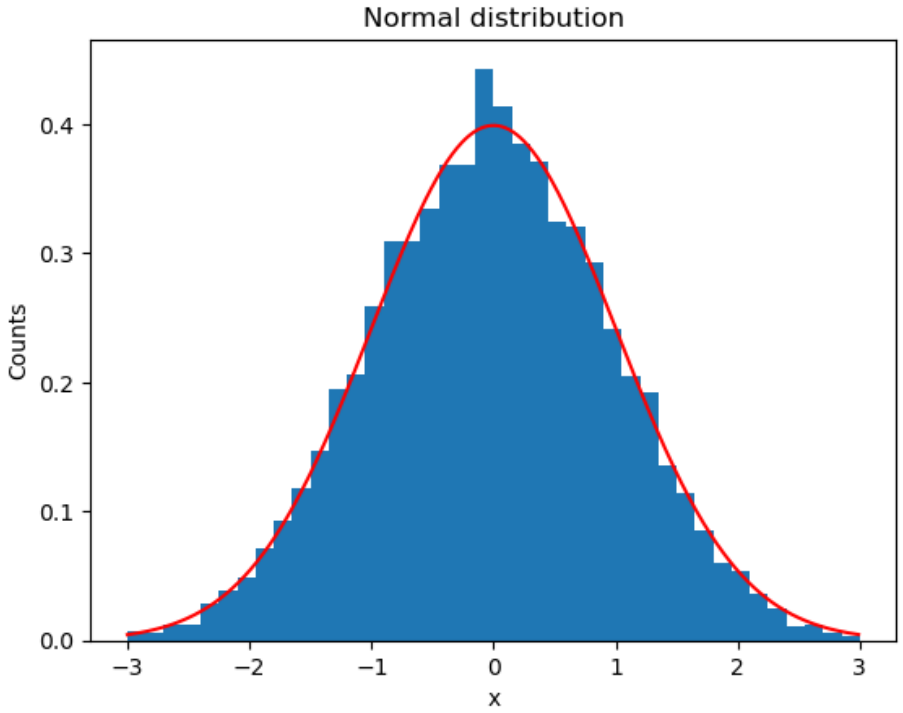
$$F_r(r) = \int_0^r r' e^{-r'^2/2} dr' = 1 - e^{-r^2/2},$$

therefore

$$r = \sqrt{-2\ln(1-\eta)}.$$

Sampling normally distributed variables

```python
def sample_xy_normal():
    phi = 2 * np.pi * np.random.rand()
    eta = np.random.rand()
    r = np.sqrt(-2*np.log(1-eta))
    return r * np.cos(phi), r * np.sin(phi)

N = 10000
samples = []
for i in np.arange(0,N,2):
    x, y = sample_xy_normal()
    samples.append(x)
    samples.append(y)
```



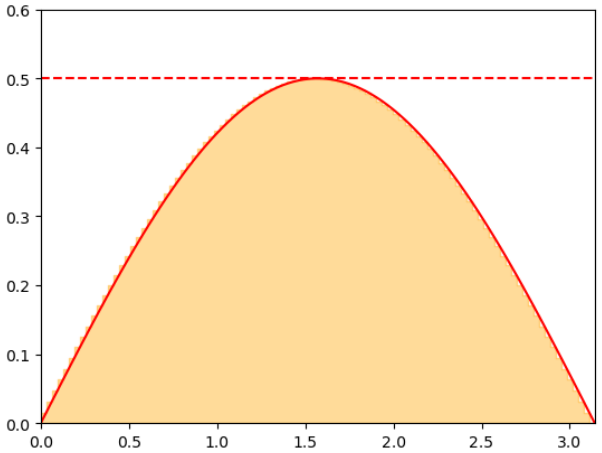
Rejection sampling

The **rejection sampling method** allows one to sample a variable ξ from an **envelope distribution** and accept or reject it with a certain probability.

Consider again the probability density function for the polar angle:

$$\rho_\theta(\theta) = \frac{\sin(\theta)}{2}.$$

Since ρ_θ is bounded from above, we define: $\rho_\theta^{\max} = 1/2.$



1. Sample a candidate value θ_{cand} from a uniform distribution over $(0, \pi)$.
2. Accept θ_{cand} with probability: $p = \rho_\theta(\theta_{\text{cand}})/\rho_\theta^{\max}$.
3. This step can be performed by sampling y **from a uniform distribution** over $(0, \rho_\theta^{max})$ and accepting θ_{cand} if $y < \rho_\theta(\theta_{cand})$

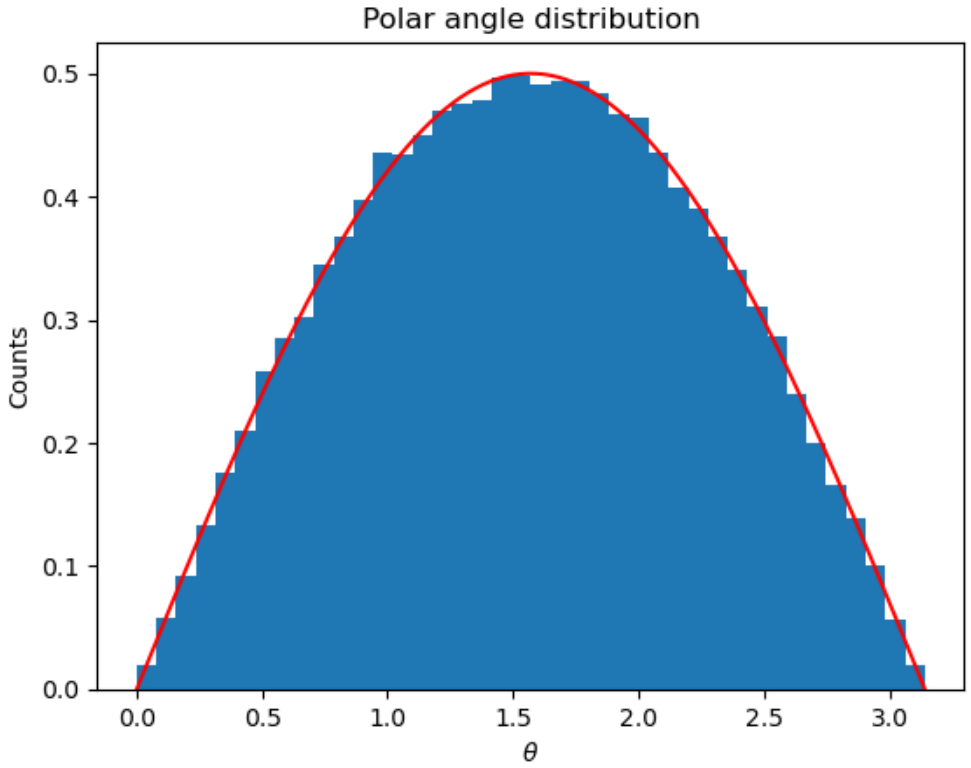
Geometric Interpretation: If we consider $\theta_{cand} = x$ and y as the **coordinates of a point** in a plane, we accept θ_{cand} **if it lies below the curve** defined by $\rho_\theta(\theta)$. This ensures that θ_{cand} values are accepted at a rate proportional to $\rho_\theta(\theta)$, as desired.

Advantages of Rejection Sampling:

$\rho_\theta(\theta)$ **does not need to be a normalized distribution** for the method to work.

Rejection sampling

```
def sample_rejection(rho, a, b, rhomax):
    while True:
        x_cand = a + (b-a)*np.random.rand()
        y = rhomax * np.random.rand()
        if (y < rho(x_cand)):
            return x_cand
    return 0.

def rho_theta(theta):
    return np.sin(theta) / 2.

N = 100000
samples = []
for i in np.arange(0,N,1):
    theta = sample_rejection(rho_theta, 0., np.pi, 0.5)
    samples.append(theta)
```



Pros and Cons of Rejection Sampling

Pros:

- Does not require the distribution to be normalized.
- Works even if y_{max} is **larger than the true maximum** of $\rho(x)$
- Applicable to **generic distributions** and does not require the evaluation of the cumulative distribution function.

Cons:

- Can be **inefficient** if the rejection rate is high (e.g., for highly peaked distributions).
- Not directly applicable to distributions **over infinite ranges**.

Generalizations of Rejection Sampling

To address some of its limitations, several generalizations of rejection sampling can be used, including:

- **Adaptive rejection sampling** by considering multiple **enveloping rectangles**.
- **Variable transformation** to map an **infinite interval** into a finite one.
- **Sampling from a non-uniform enveloping distribution** for better efficiency.

Importance sampling

Recall the calculation of an integral as statistical average

$$I = \int_a^b f(x)dx = (b-a)\langle f \rangle, \qquad \text{where} \qquad \langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i), \qquad x_i \in U(a,b)$$

Some issues with the method:

- Sample unimportant regions (e.g. f is highly peaked)
- Integrable singularities

$$I = \int_a^b \frac{f(x)}{w(x)} w(x)dx = \left\langle \frac{f(x)}{w(x)} \right\rangle_w .$$

Importance sampling:

Sample x_i from a *non-uniform* distribution w(x) that resembles f(x).

The integrand is then calculated as

$$I = \int_a^b \frac{f(x)}{w(x)} w(x)dx = \left\langle \frac{f(x)}{w(x)} \right\rangle_w$$

Normalization:

$$\int_a^b w(x)dx = 1.$$

Error:

$$\delta I = \frac{\sqrt{\left\langle \left[\frac{f(x)}{w(x)}\right]^2 \right\rangle_w - \left\langle \frac{f(x)}{w(x)} \right\rangle_w^2}}{\sqrt{N}}.$$

Importance sampling

$$I = \int_a^b \frac{f(x)}{w(x)} w(x)dx = \left\langle \frac{f(x)}{w(x)} \right\rangle_w \qquad \delta I = \frac{\sqrt{\left\langle \left[\frac{f(x)}{w(x)}\right]^2 \right\rangle_w - \left\langle \frac{f(x)}{w(x)} \right\rangle_w^2}}{\sqrt{N}}.$$

- For w(x)=1/(b-a) we recover the mean value method

$$I = \int_a^b f(x)dx = (b-a)\langle f \rangle$$

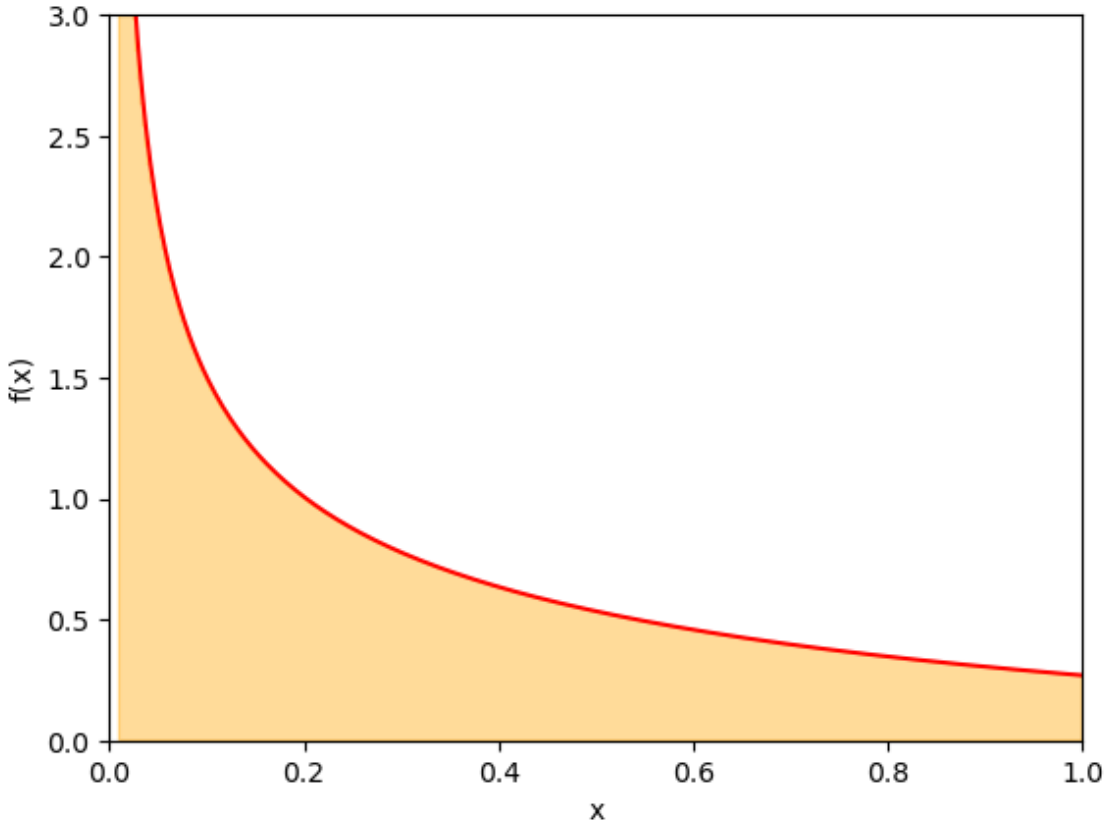
- For w(x) ∝ f(x) one has $\left\langle \frac{f(x)}{w(x)} \right\rangle = \text{const} = I$ and $\delta I = 0$

Importance sampling

```python
# Calculate integral \int_a^b f(x) dx using importance sampling
# f = f(x) is the integrand
# N is the number of random samples
# wx = w(x) is the normalized probability density from which
# the sampling takes place
# sampler is a function which samples a random number from w(x)
def intMC_weighted(f, N, wx, sampler):
    total = 0
    total_sq = 0
    for i in range(N):
        x = sampler()
        fval = f(x)
        total += fval / wx(x)
        total_sq += (fval / wx(x))**2
    fw_av = total / N
    fwsq_av = total_sq / N
    return fw_av, np.sqrt((fwsq_av - fw_av*fw_av)/N)
```

Importance sampling: Example

$$\int_0^1 \frac{x^{-1/2}}{e^x + 1} dx$$



Integrable singularity at x=0

Importance sampling: Example

$$\int_0^1 \frac{x^{-1/2}}{e^x + 1} dx$$

Mean value method

$$w(x) = \frac{1}{b-a}$$

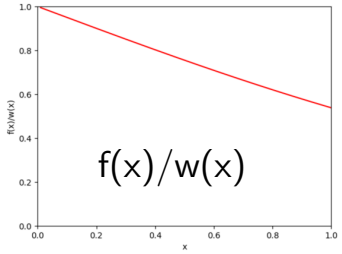
```
def uniform_sample():
    eta = np.random.rand()
    return eta

def uniform_w(x):
    return 1.

np.random.seed(1)
N = 1000000
I, err = intMC_weighted(f, N, uniform_w, uniform_sample)
print("I = ",I," +- ",err)
```

```
I =  0.8374063441946126  +-  0.0017772180714415427
```

Importance sampling

$$w(x) = \frac{1}{2\sqrt{x}} \qquad I = \left\langle \frac{2}{e^x + 1} \right\rangle_w \qquad x = \eta^2$$

```
def rsqrt_sample():
    eta = np.random.rand()
    return eta * eta

def rsqrt_w(x):
    return 1. / (2. * np.sqrt(x))
```




```
N = 1000000
I, err = intMC_weighted(f, N, rsqrt_w, rsqrt_sample)
print("I = ",I," +- ",err)
```

```
I =  0.839014917136739  +-  0.0001409071521618816
```

Statistical error is more than x10 smaller than in the mean value method.
We would need more than x100 samples in the mean value method to reach the same accuracy as importance sampling in this case.