Computational Physics (PHYS6350)

Lecture 17: Random numbers

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

March 28, 2023

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 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 not truly random, but a good generator produces numbers that reflect the desired properties of a random variable, hence they are called pseudo-random.

Pseudo-random numbers on a computer

- The most basic routine typically 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 various 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

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

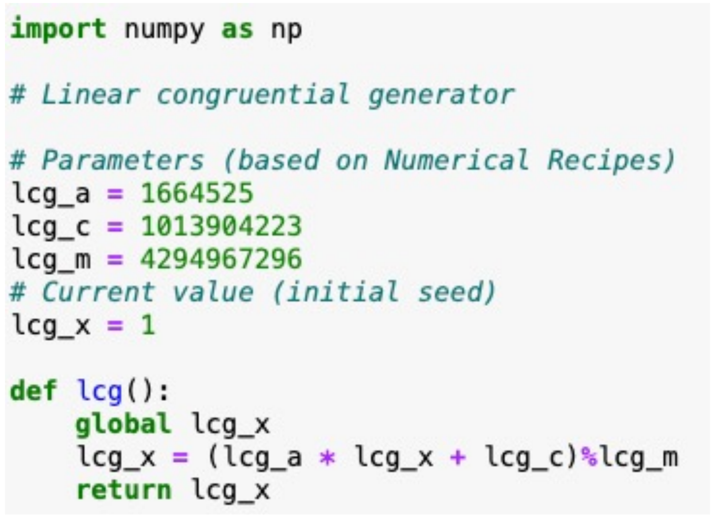
for some parameters *a*, *x*, *m*.

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

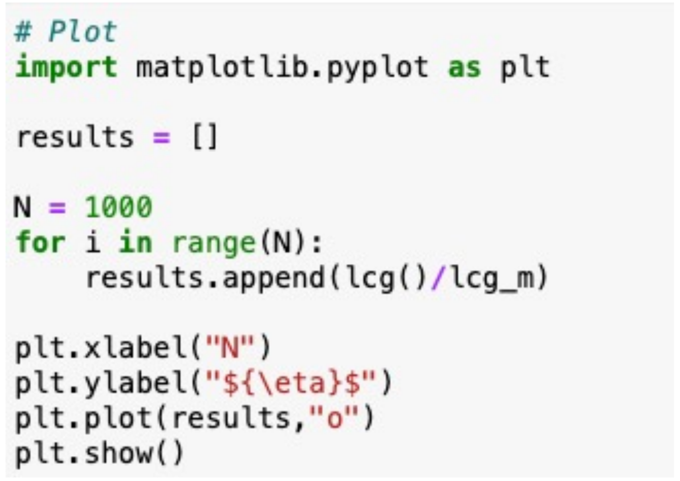
The sequence is periodic with a period of at most *m*

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

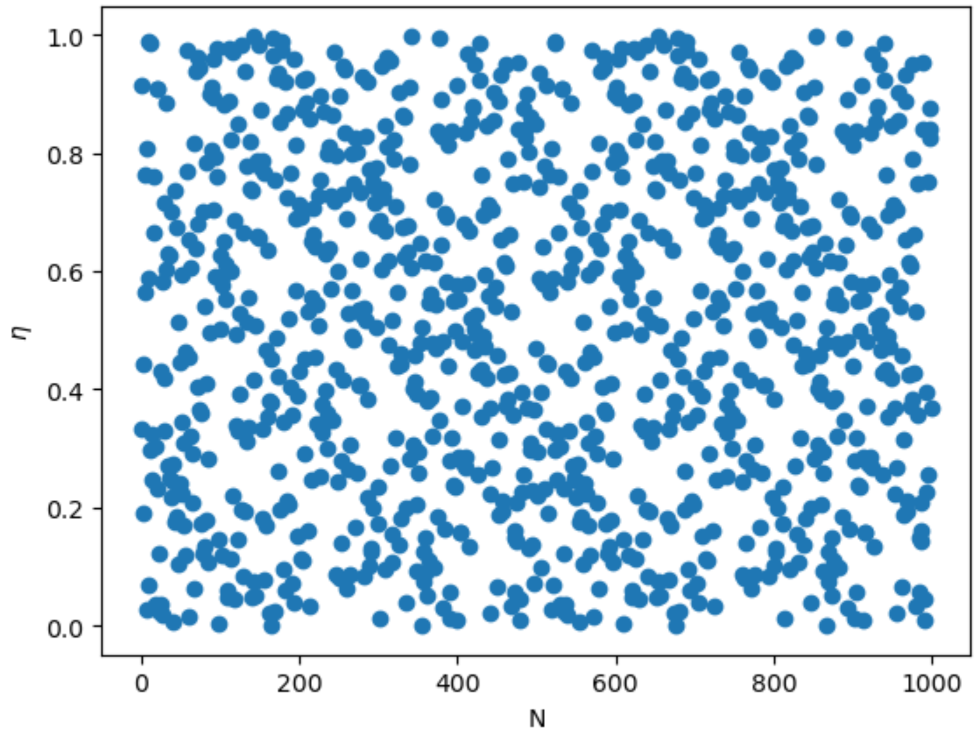
Linear congruential generator: Example



```
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
```



```
# 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.

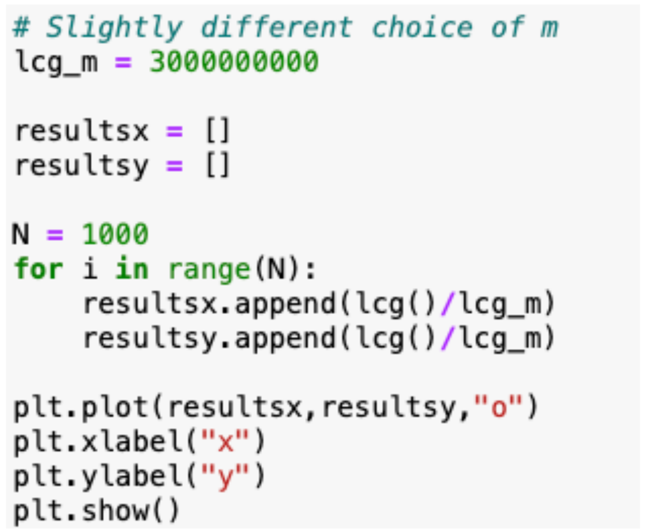
For instance, if one regards random numbers as components of a vector (x,y,...), the method tends to generate these points on a hyperplane (spectral test).

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).



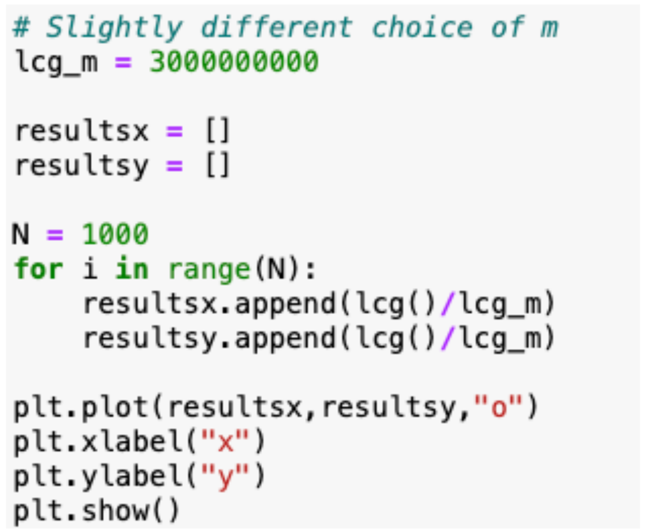
```
# Slightly different choice of m
lcg_m = 3000000000

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

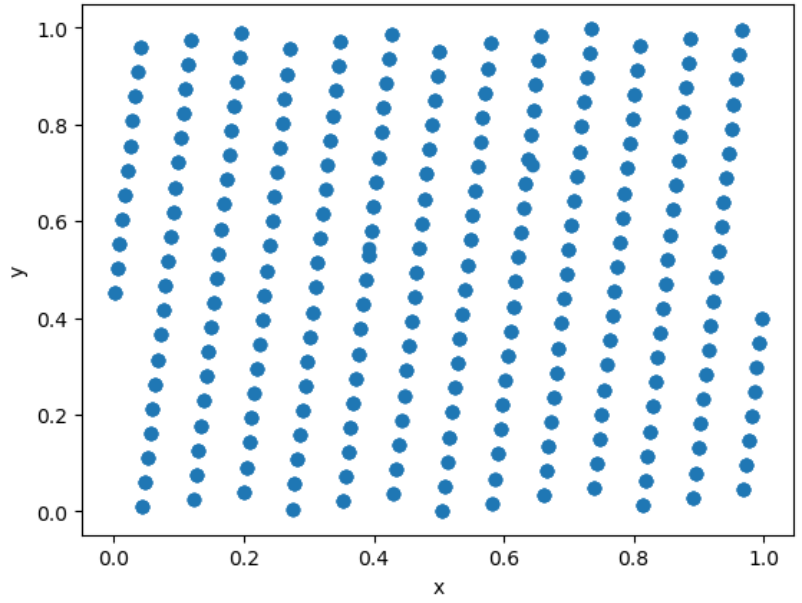
LCG has some serious drawbacks:

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

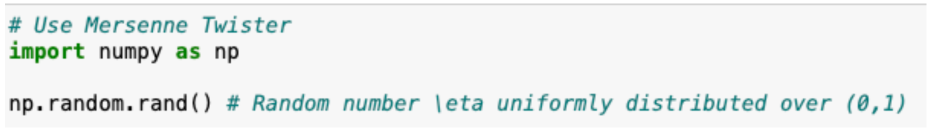
LCG is not and should not be used in any serious calculations.

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.

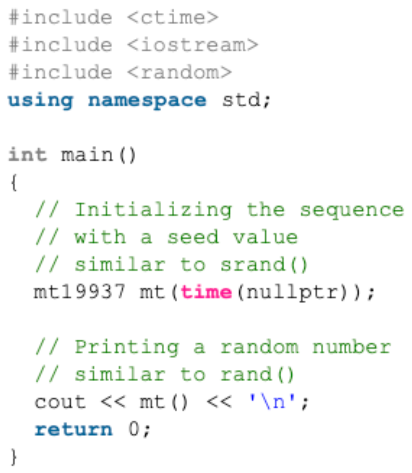
It now implemented by default in many languages and we will take it for granted.

Python:



```
# Use Mersenne Twister
import numpy as np

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

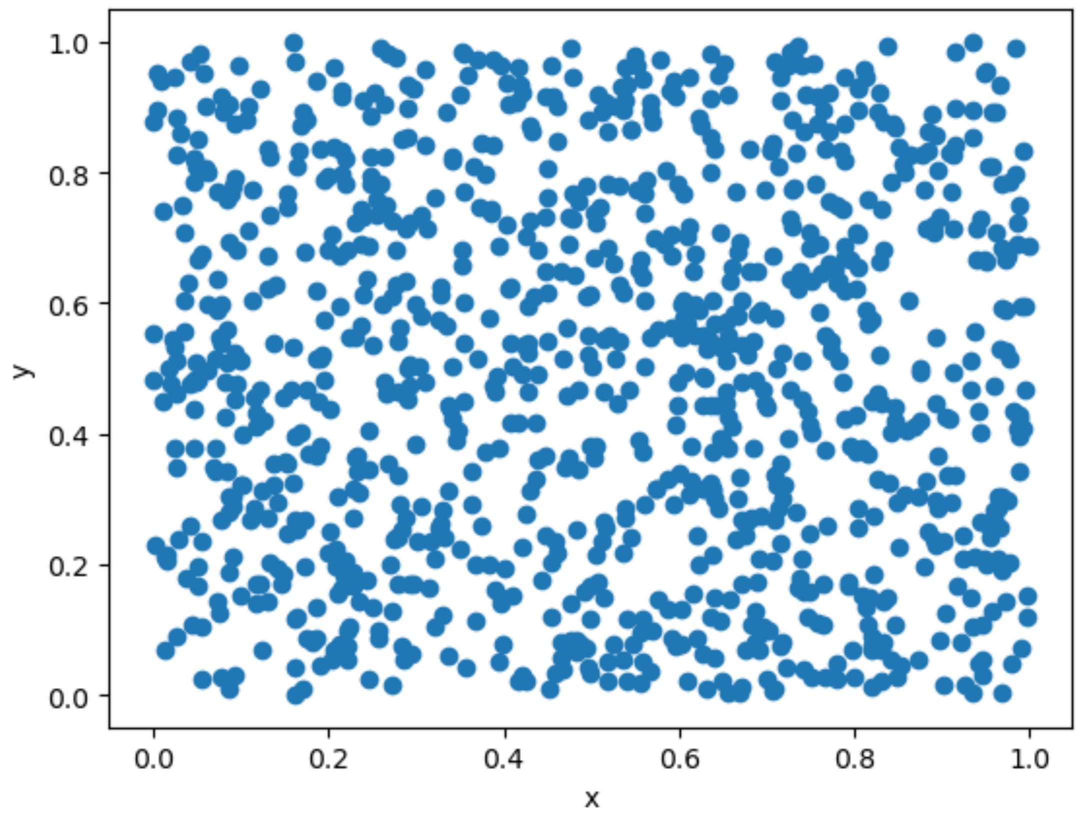
C++ (since C++11):



```
#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

```python
# 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()
```

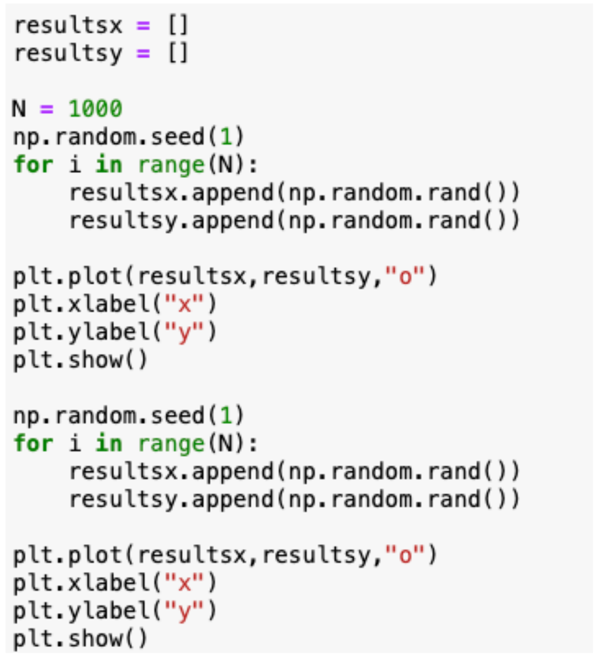


Random seed

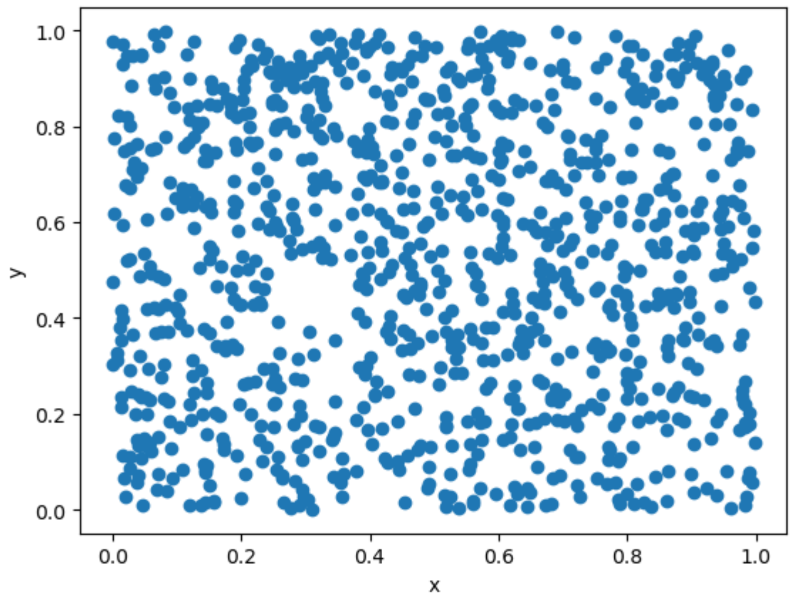
Most RNGs (like LCG, Mersenne Twister,...) maintain state variables and iteratively generate a pre-determined sequence of (pseudo)-random numbers

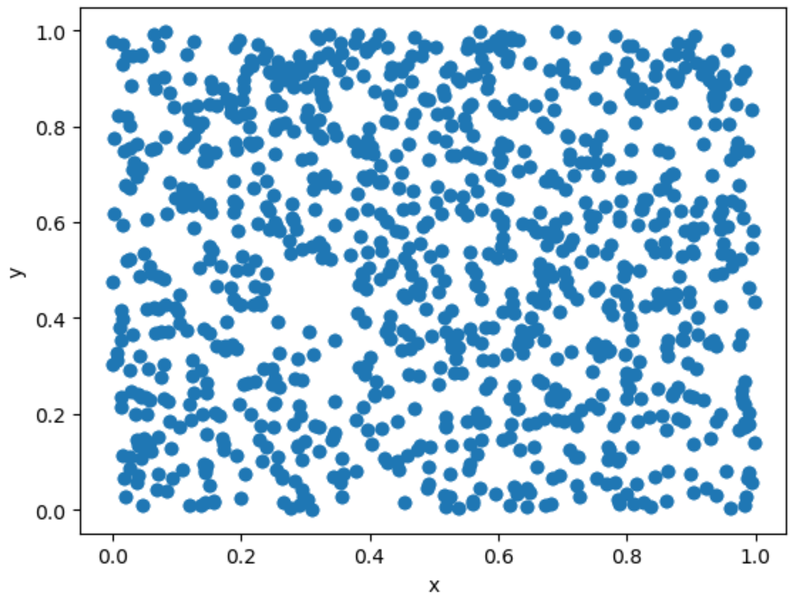
The initial state can be changed by specifying the *seed*

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

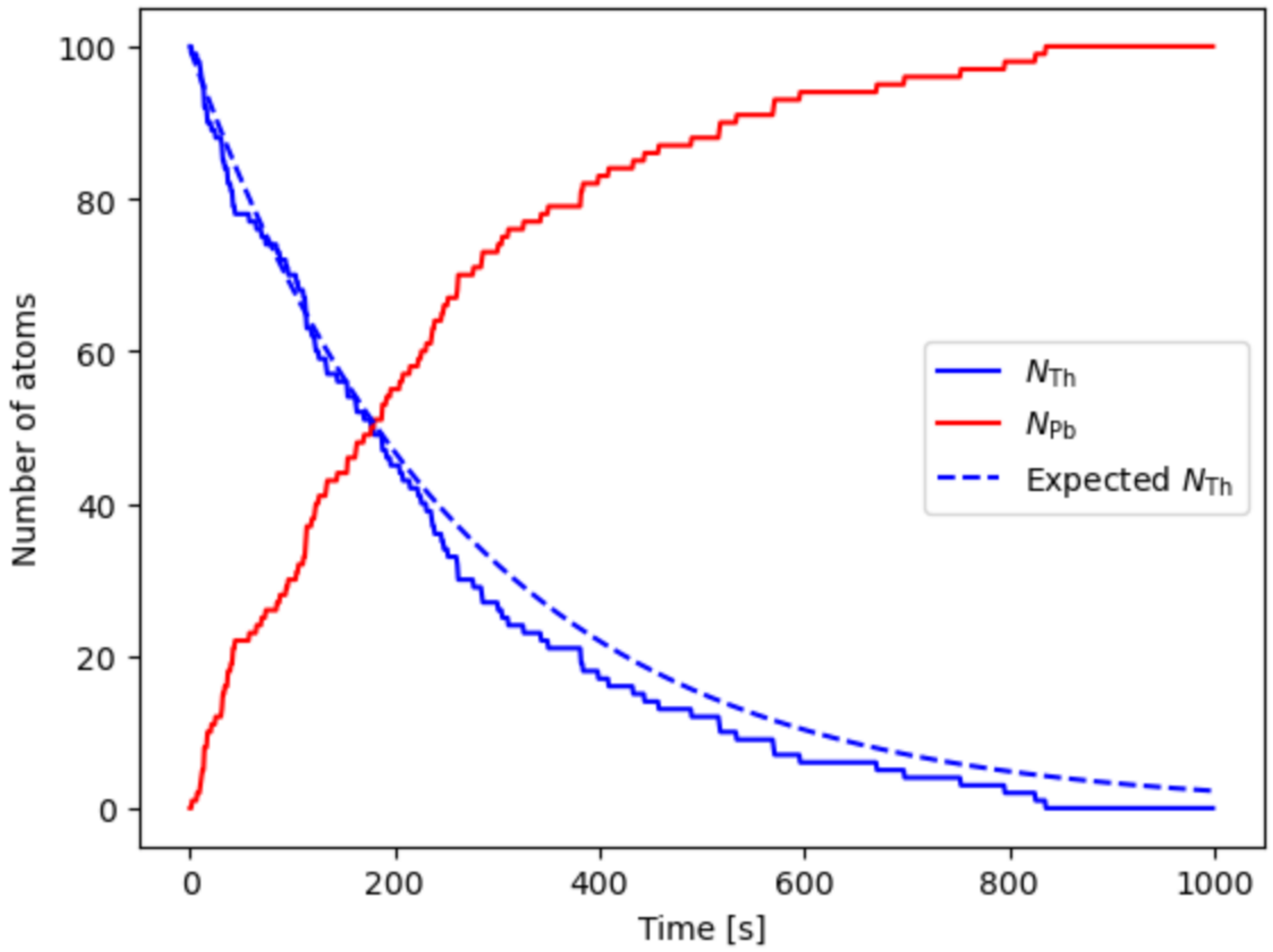
$$p(t) = 1 - 2^{-t/\tau}.$$

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

```python
# 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 = []
```

```python
# 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
```

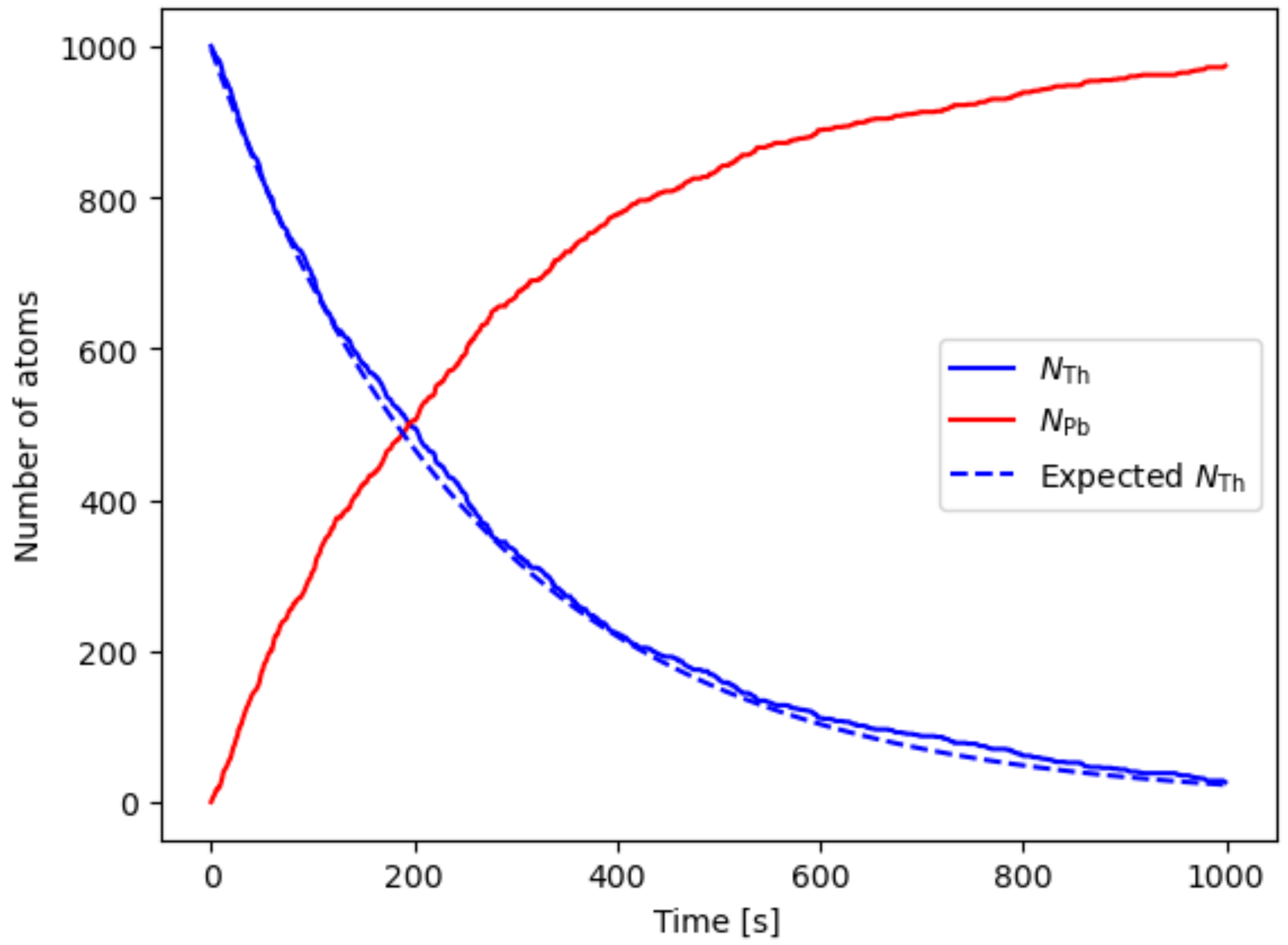
Simulation example: Radioactive decay



Expected:

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

Simulation example: Radioactive decay

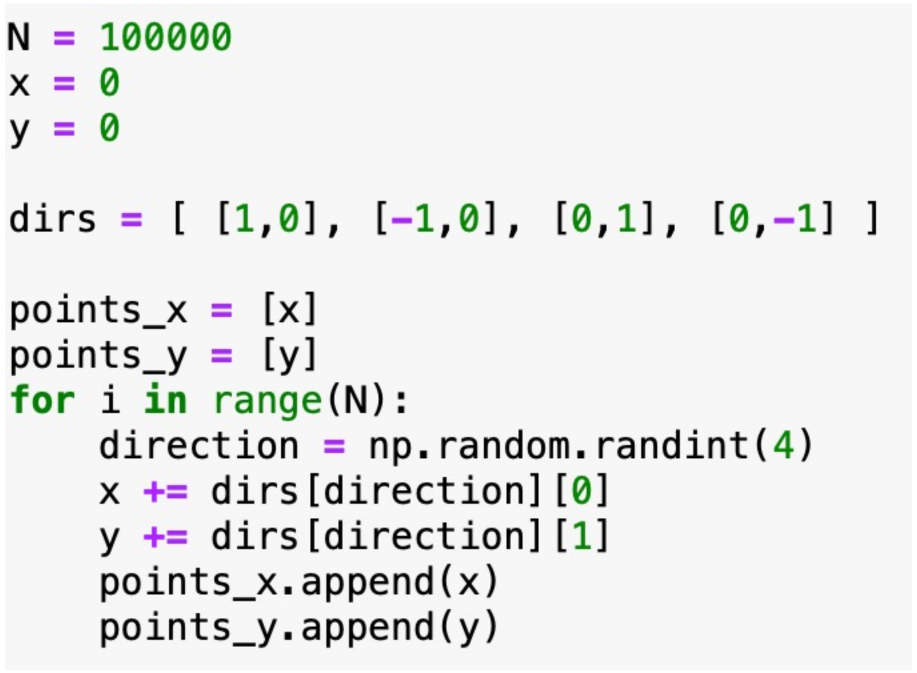


Expected:

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

Simulation example: Brownian motion

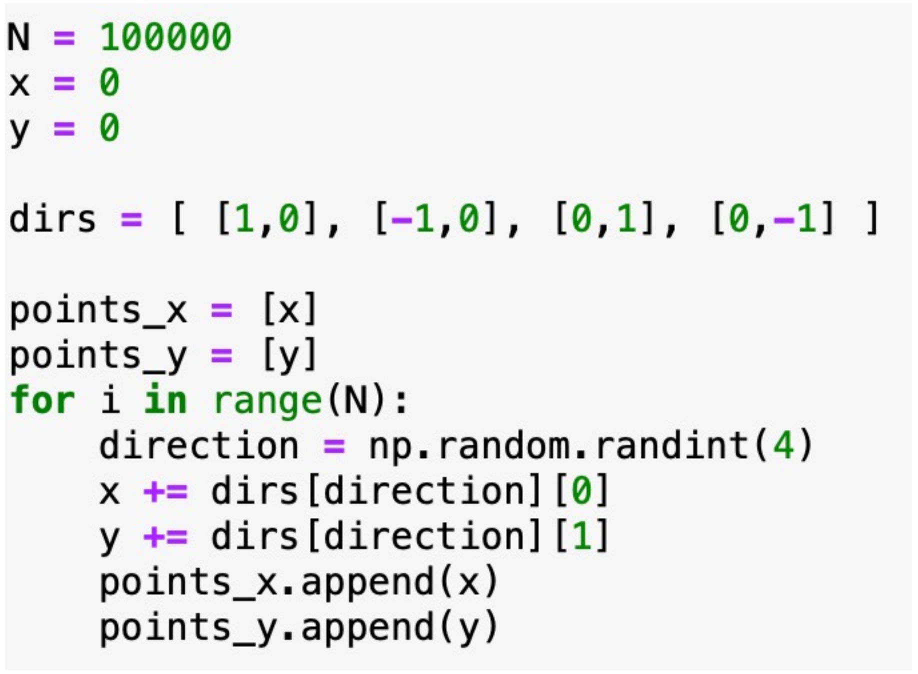
Brownian motion is a motion of a heavy particle in a gas colliding with the lighter gas particles.
We can consider a simplified 2D motion of particle by randomly making
a small step at each iteration in one of the four directions.



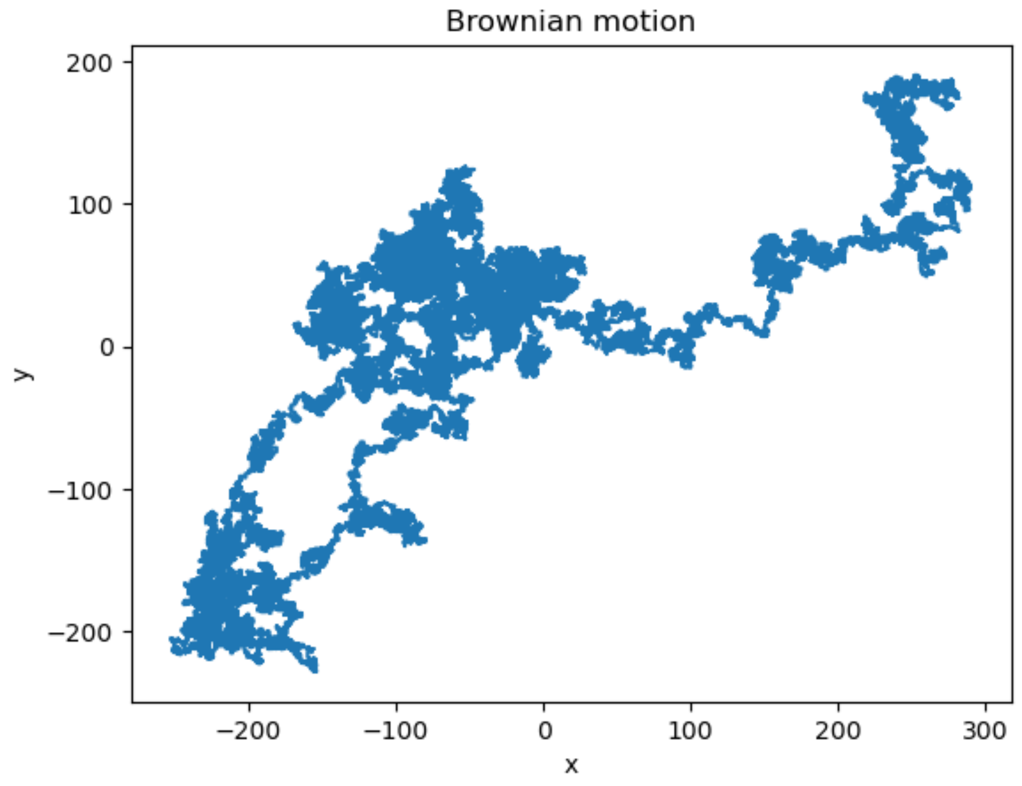
```python
N = 100000
x = 0
y = 0

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)
```

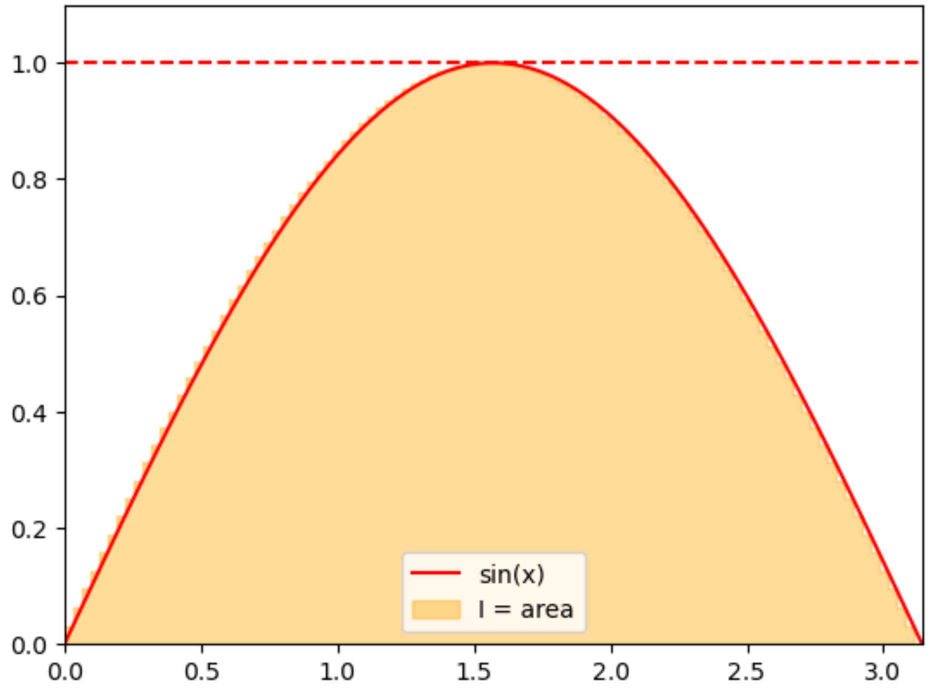


Computing integrals: Estimating the area under the curve

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 $N^{-1/2}$

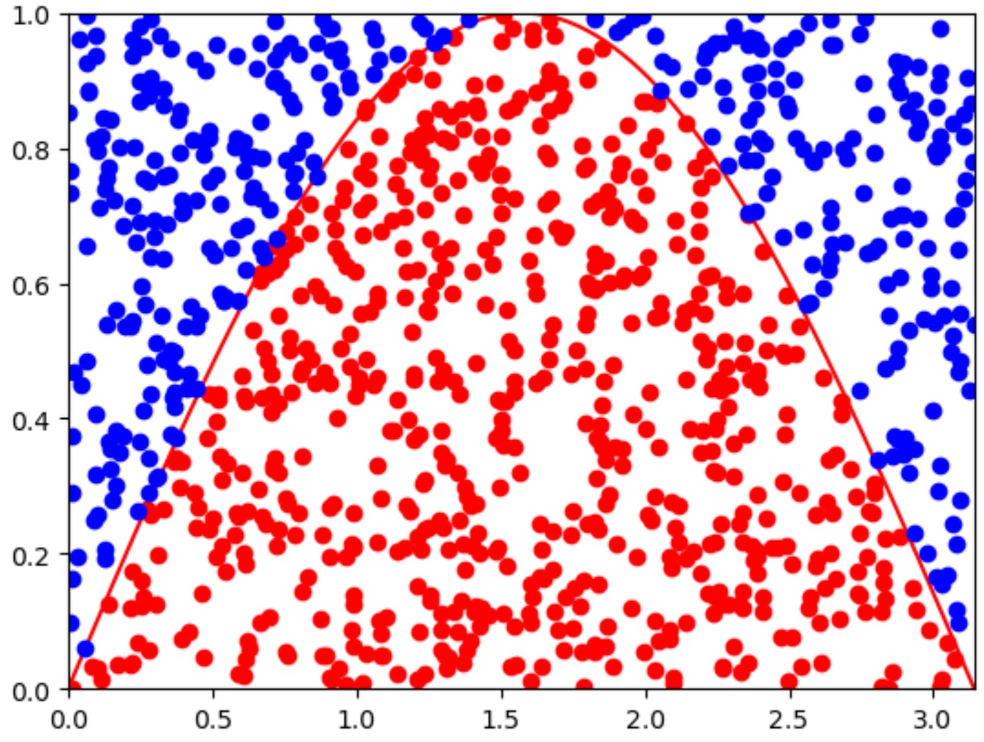
To reduce the error by factor x2 we need to sample x4 more numbers – true for most Monte Carlo methods.

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 pi

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

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

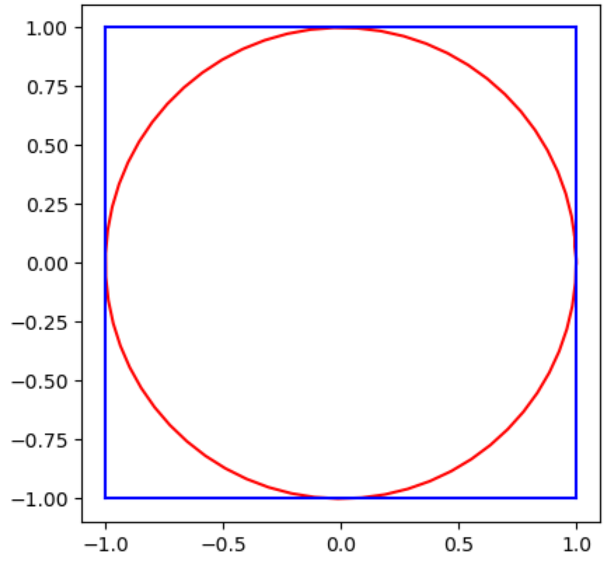
The circle can be embedded into a square with the side length of two. The area of the square is $A_{\mathrm{sq}} = 2^2 = 4$.

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

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

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

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



Computing pi

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

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

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

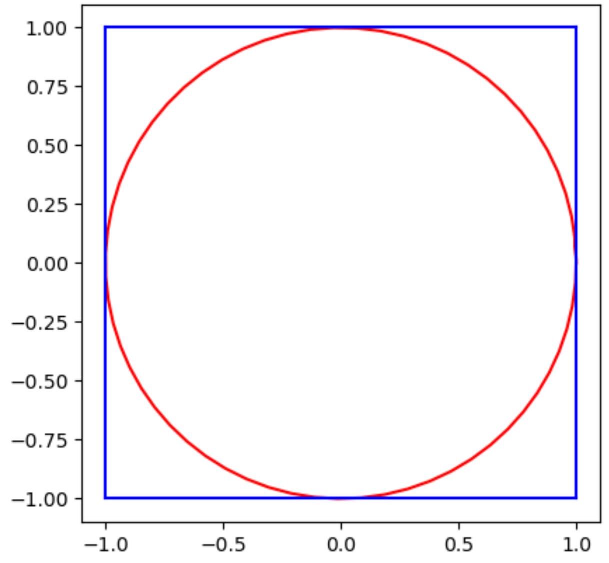
Consider now a random point anywhere inside the square. The probability that it is also inside the circle corresponds to the ratio of their areas

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

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

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

```python
# 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)
```

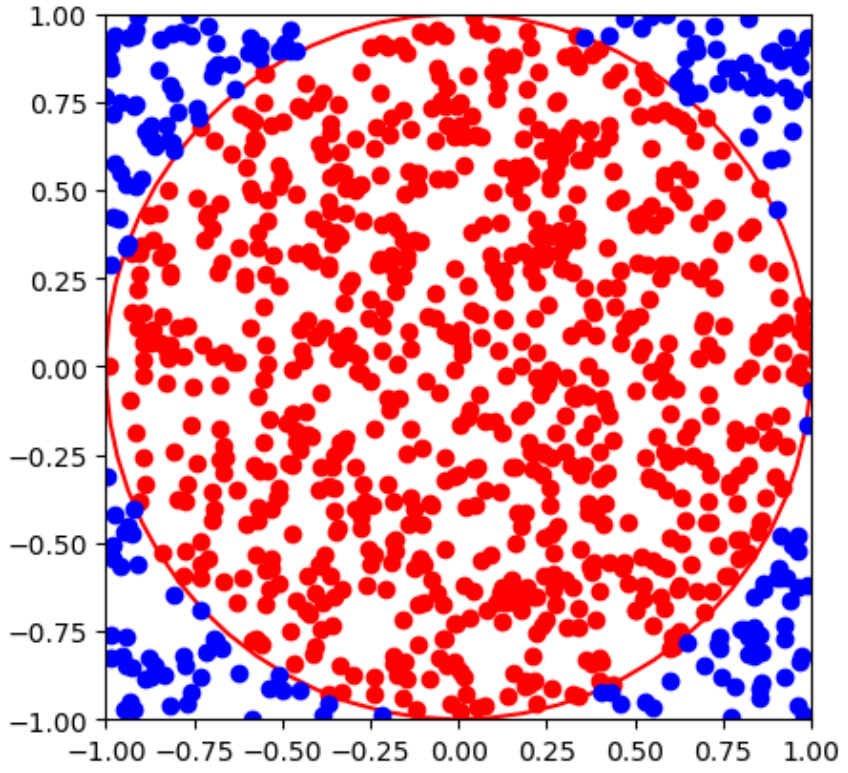


Computing pi

```
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

An integral

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

corresponds to the mean value $\langle f \rangle$ over (a, b)

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

so that

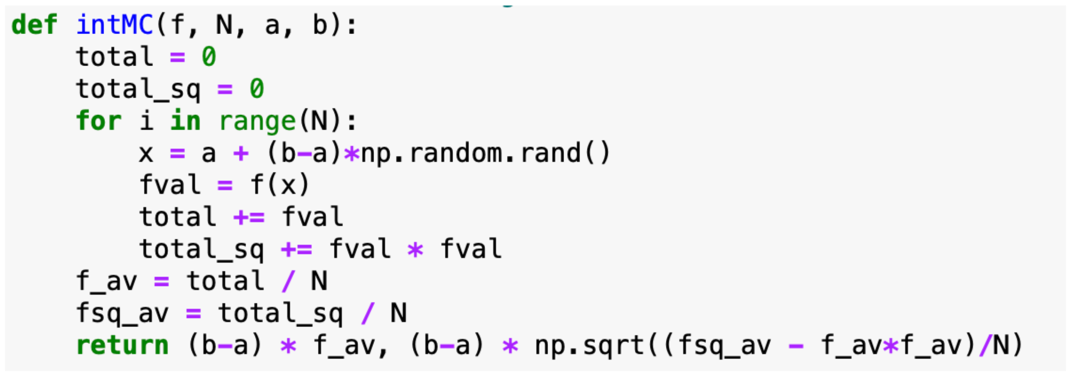
$$I = (b-a)\langle f \rangle$$

The integral can therefore be estimated by evaluating $\langle f \rangle$ as the average value of $f(x)$ obtained through random sampling of the variable x uniformly over the interval (a, b):

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

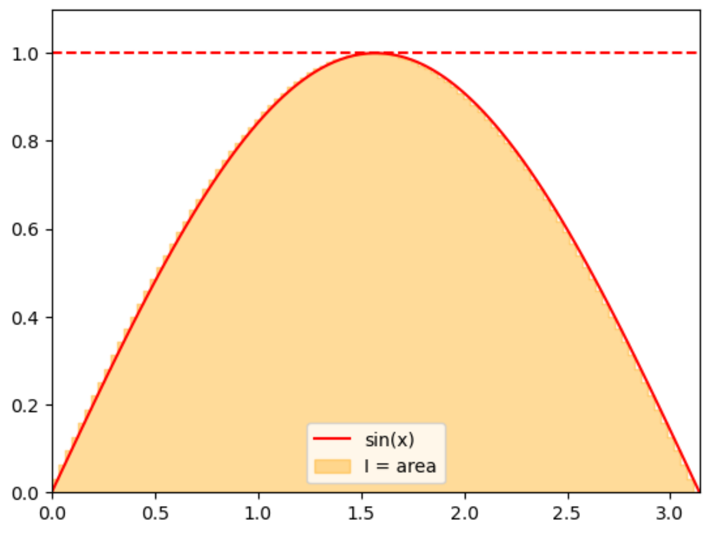
The error estimate comes from the law of averages and involves the estimate of $\langle f^2 \rangle$

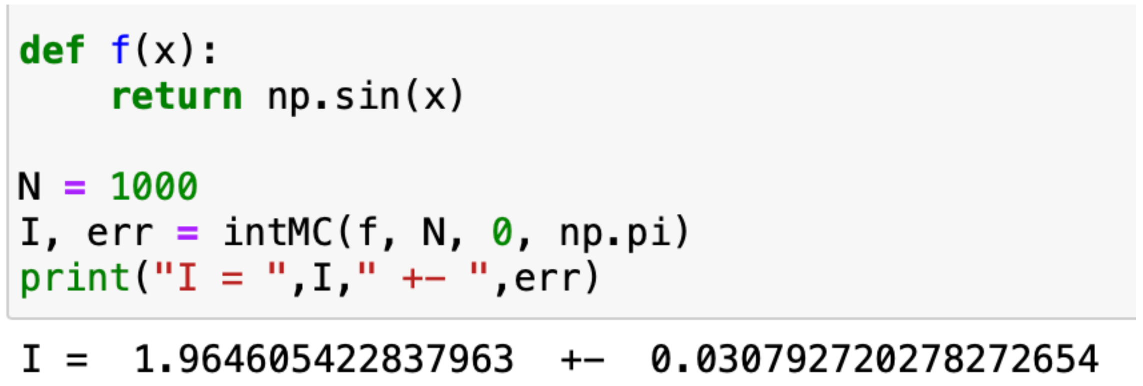
$$\delta I = (b-a)\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$



```
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
```

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

Another way to compute pi

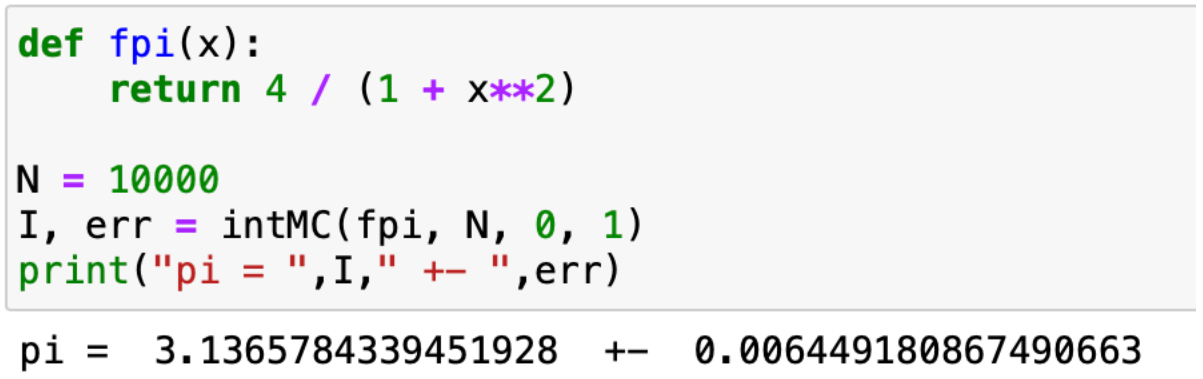
Consider an integral

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

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
```

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 is the following:

1. Calculate the cumulative distribution
$$G(x) = \int_{-\infty}^{x} \rho(\xi) d\xi$$
2. Find the inverse function $G^{-1}(y)$ as the solution to the equation
$$G(x) = y$$
with respect to x.
3. Sample uniformly distributed randon variables η and calculate $\xi = G^{-1}(\eta)$

Sometimes, evaluating $G(x)$ and/or $G^{-1}(y)$ explicitly is challenging. In such cases one would resort to numerical integration and/or non-linear equation solvers.

Inverse transform sampling

The algorithm is the following:

1. Calculate the cumulative distribution
$$G(x) = \int_{-\infty}^{x} \rho(\xi) d\xi$$
2. Find the inverse function $G^{-1}(y)$ as the solution to the equation
$$G(x) = y$$
with respect to x.
3. Sample uniformly distributed randon variables η and calculate $\xi = G^{-1}(\eta)$

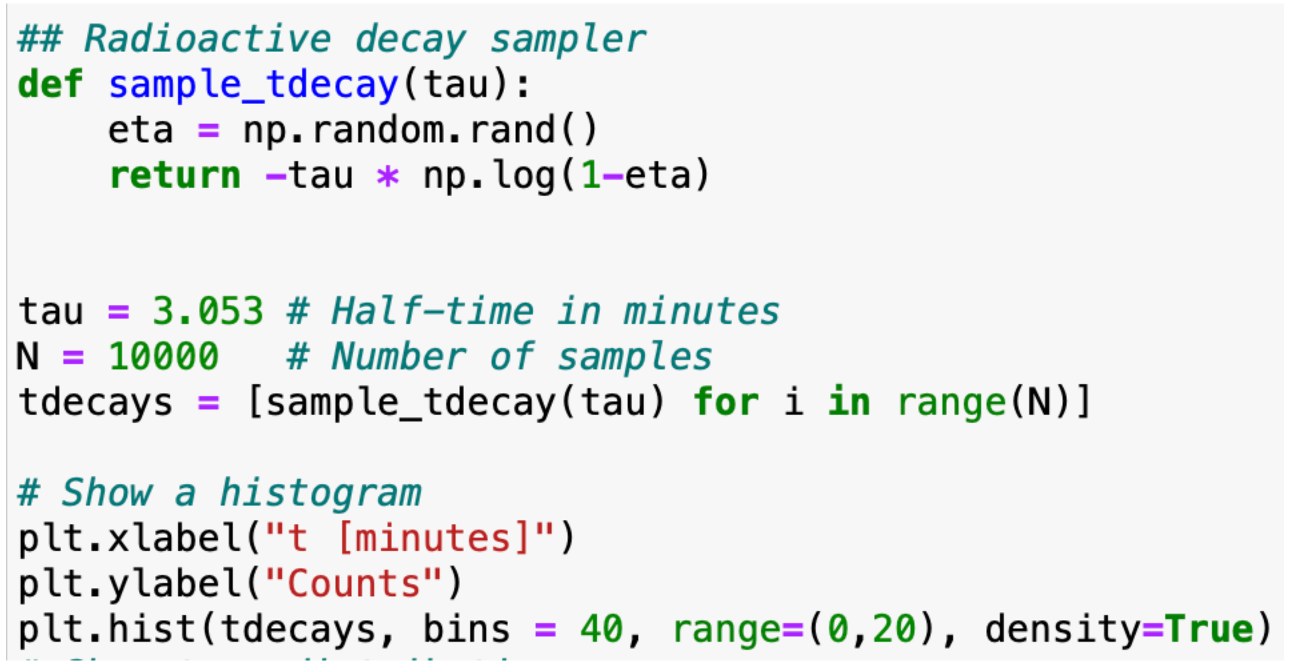
Sometimes, evaluating $G(x)$ and/or $G^{-1}(y)$ explicitly is challenging. In such cases one would resort to numerical integration and/or non-linear equation solvers.

Example: Exponential distribution

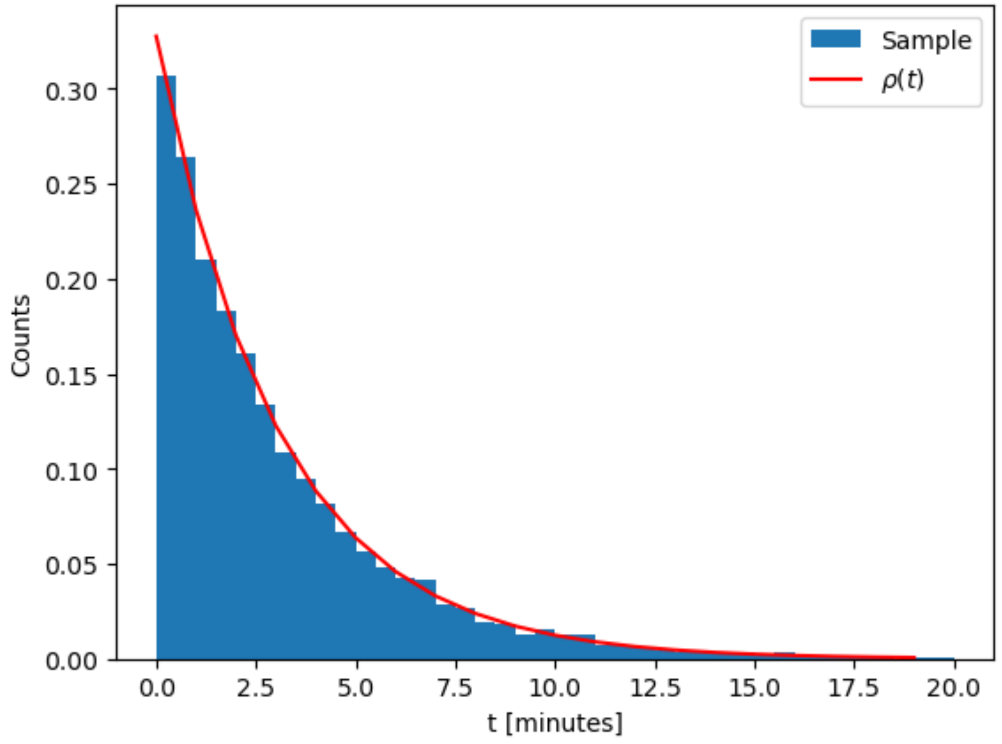
Recall the radioactive decay process. The time of decay is distributed in accordance with
$$\rho(t) = \frac{1}{\tau} e^{-\frac{t}{\tau}}.$$

The cumulative distibution function reads
$$F(x) = \int_0^x \frac{1}{\tau} e^{-\frac{t}{\tau}} dt = 1 - e^{-\frac{x}{\tau}}.$$
To apply inverse transform sampling we have to invert $F(x)$ by solving the equation
$$1 - e^{-\frac{t}{\tau}} = \eta.$$
This can be done straightforwardly to give
$$t(\eta) = -\tau \ln(1 - \eta).$$

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

Let us sample points on a plane inside a unit circle. One way to do that is to switch to polar coordinates

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

and sample r and ϕ.

Since $r \in [0, 1)$ and $\phi \in [0, 2\pi)$, naively one could sample r and ϕ independently from two uniform distributions. Let us 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

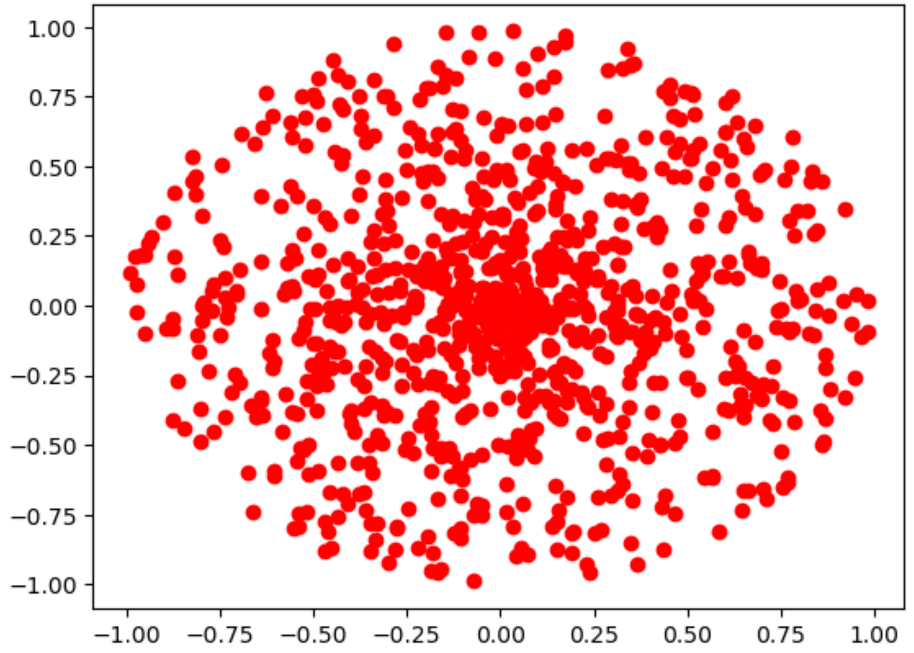
Let us sample points on a plane inside a unit circle. One way to do that is to switch to polar coordinates

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

and sample r and ϕ.

Since $r \in [0, 1)$ and $\phi \in [0, 2\pi)$, naively one could sample r and ϕ independently from two uniform distributions. Let us 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 centre! Why? Because r is not uniformly distributed. Recall

$$dxdy = rdrd\phi,$$

therefore

$$\rho_r(r) = 2r, \qquad \rho_\phi(\phi) = \frac{1}{2\pi}.$$

Cumulative distribution function

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

Solving $F_r(r) = \eta$ we get

$$r = \sqrt{\eta}.$$

Sampling points inside a circle

The points clump more in the centre! Why? Because r is not uniformly distributed. Recall

$$dxdy = rdrd\phi,$$

therefore

$$\rho_r(r) = 2r, \qquad \rho_\phi(\phi) = \frac{1}{2\pi}.$$

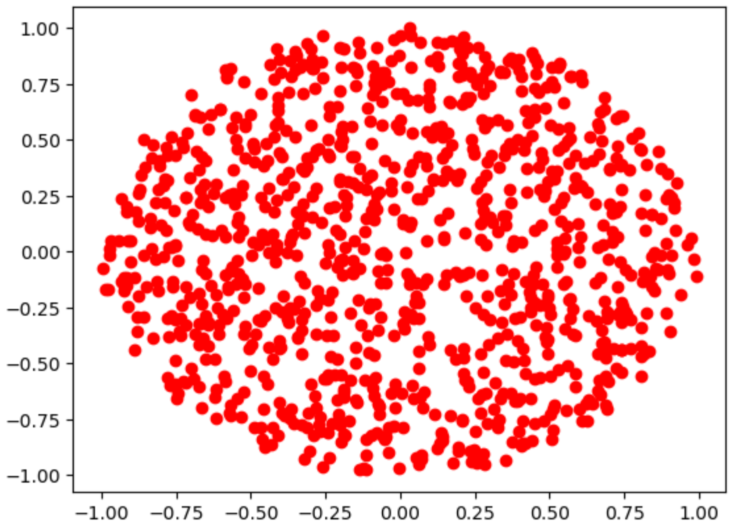
Cumulative distribution function

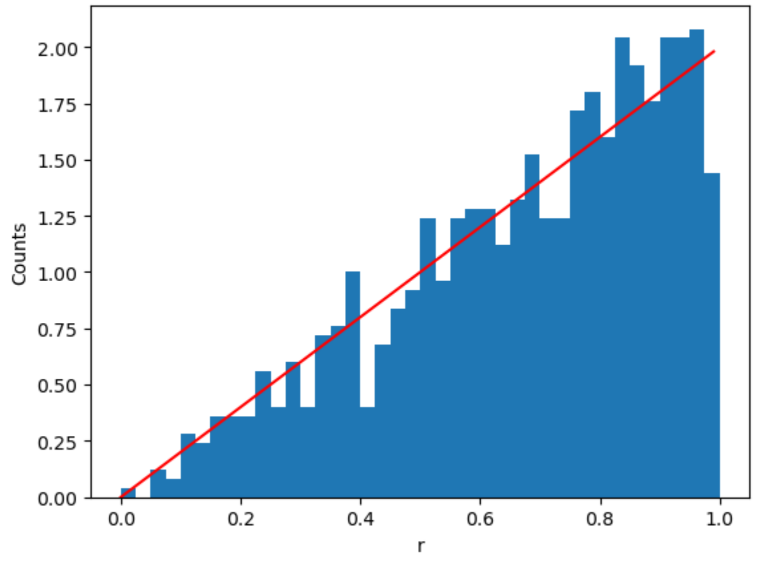
$$F_r(r) = \int_0^r \rho_r(r')dr' = r^2.$$

Solving $F_r(r) = \eta$ we get

$$r = \sqrt{\eta}.$$

```python
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 an isotropic direction

One common problem that occurs in Monte Carlo simulations is random sampling of an isotropic direction in 3D space. For instance, this issue occurs when sampling a random orientation of some axially symmetric object (such as a rod) or the momentum of a particle.

This problem is equivalent to choosing a random point on a unit sphere. The coordinates x, y, z on a unit sphere can be parametrized by azimuthal and polar angles, $\phi \in [0, 2\pi)$ and $\theta \in [0, \pi]$:

$$\begin{aligned} x &= \sin(\theta)\cos(\phi), \\ y &= \sin(\theta)\sin(\phi), \\ z &= \cos(\theta). \end{aligned}$$

Sampling an isotropic direction

$$x = \sin(\theta)\cos(\phi),$$
$$y = \sin(\theta)\sin(\phi),$$
$$z = \cos(\theta).$$

Recall that

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

thus the random variable ϕ and θ are independent. ϕ is uniformly distributed in $[0, 2\pi)$, thus, its sampling is straightforward. However, the polar angle θ has a weighted probability density

$$\rho_\theta(\theta) = \frac{1}{2}\sin(\theta),$$

thus, its, distribution is non-uniform. The cumulative distribution function reads

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

thus

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

In practice, it can make sense to work directly with $\cos(\theta)$ and $\sin(\theta)$. Indeed, we have

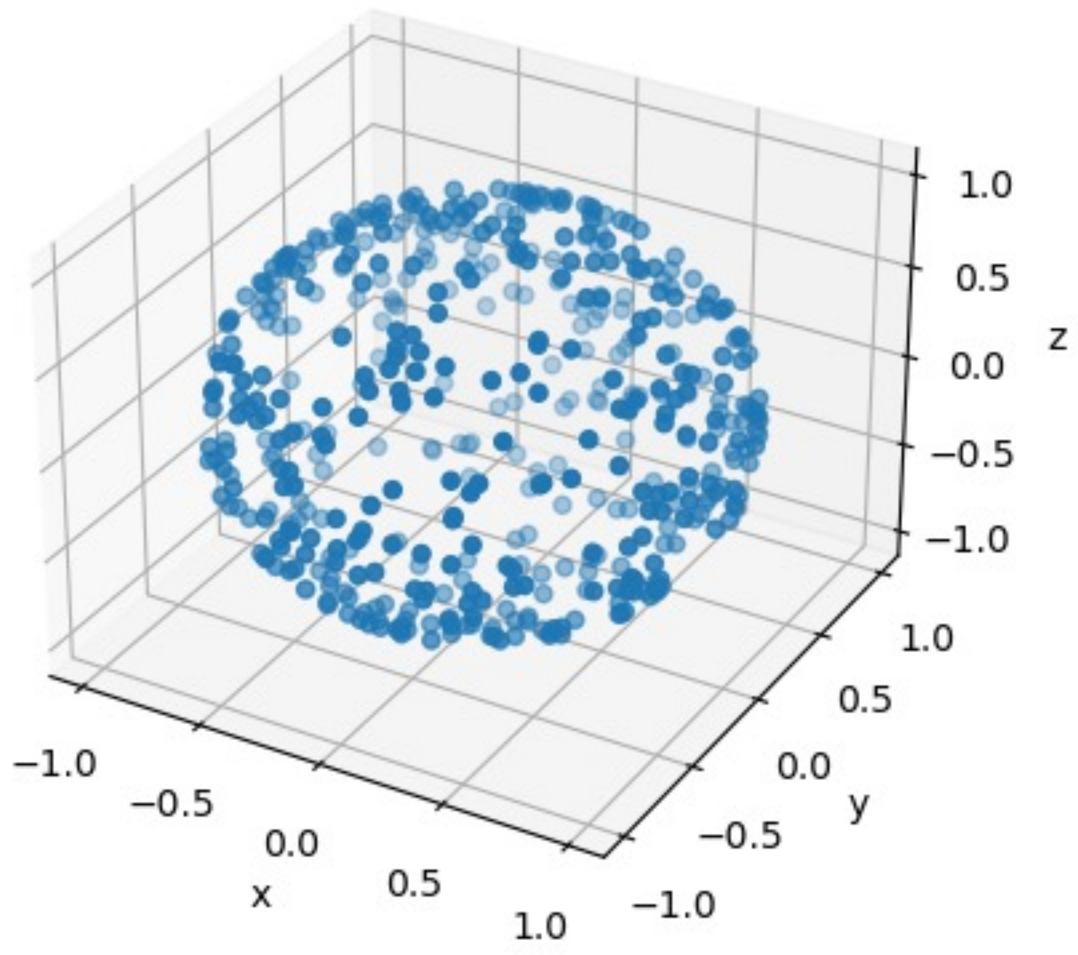
$$\cos(\theta) = 2\eta - 1,$$

and

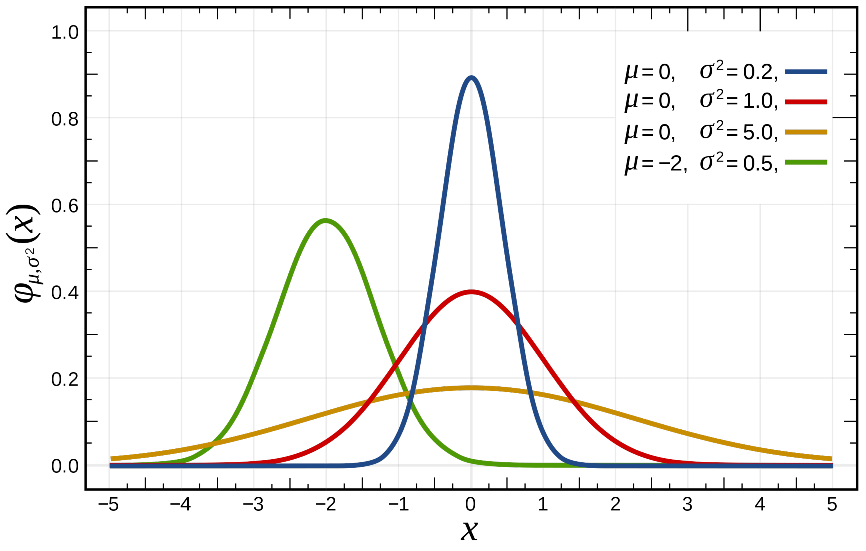
$$\sin(\theta) = \sqrt{1 - [\cos(\theta)]^2}$$

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 distribution is the normal (or Gaussian) distribution

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

There are a lot of standard implementations of sampling this distribution. Let us go through one such method. First, we can make a change of variable $x \to \mu + \sigma x$. The new variable then has a normal distribution with zero mean and standard deviation of unity

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

Calculating the cumulative distribution function $F(x) = \int_{-\infty}^{x} \rho(x)$ is not entirely trivial.

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

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

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

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

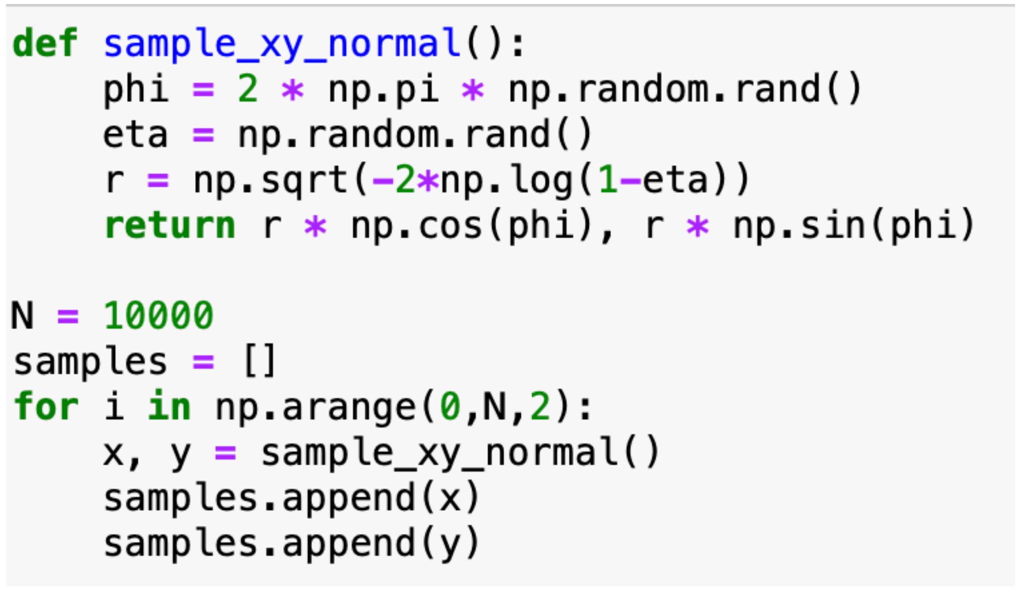
and the cumulative distribution function

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

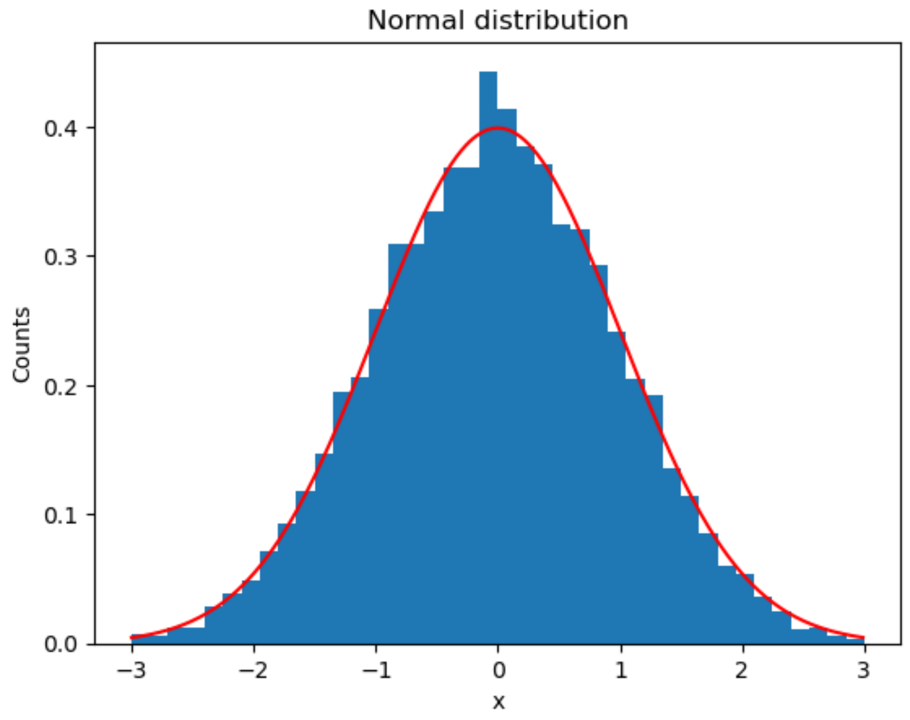
therefore

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

Sampling normally distributed variables



```
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)
```

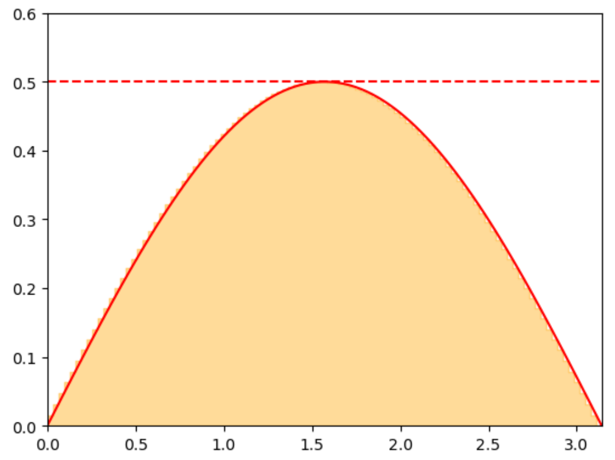


Rejection sampling

In the rejection sampling method one samples a variable ξ from an envelope distribution and accepts this value with a certain probability.

Consider the distribution function for the polar angle again:

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



Note that ρ_θ is bounded from above $\rho_\theta < \rho_\theta^{\max} = 1/2$. The rejection sampling method proceeds by

1. Sampling a candidate value θ_{cand} from a uniform distribution over $(0, \pi)$.
2. Accepting the value θ_{cand} with a probability $p = \rho_\theta(\theta_{\text{cand}})/\rho_\theta^{\max}$.

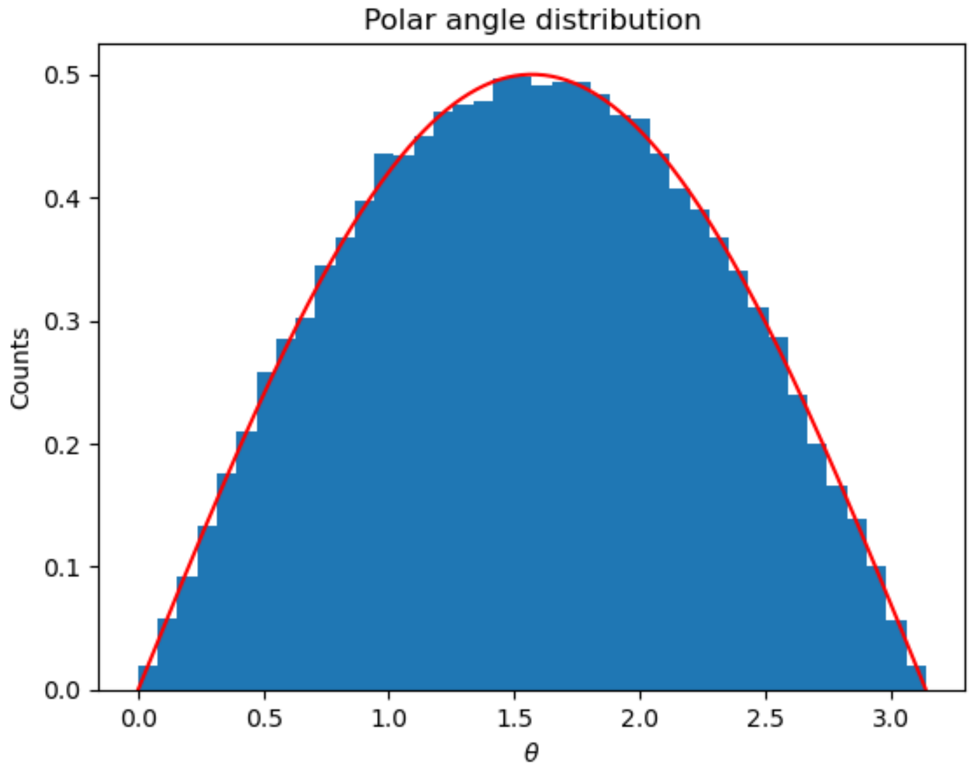
The second step can be performed by sampling y as a uniform distribution over $(0, \rho_\theta^{\max})$ and accepting θ_{cand} is $y < \rho_\theta(\theta_{\text{cand}})$.

The procedure has simple geometrical interpretation. Considering $\theta_{\text{cand}} \equiv x$ and y to be the coordinates of a point on a plane, we accept θ_{cand} for all points that lie below the curve given by the probability density $\rho_\theta(x)$. This ensures that the θ_{cand} are accepted with a rate proprotional to $\rho_\theta(\theta)$, as desired.

One advantage of rejection sampling is that $\rho_\theta(\theta)$ need not 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)
```



Rejection sampling

Rejection sampling has the following pros and cons

Pros:

- Does not need the distribution to be normalized
- Will also work if $y_{\max}$ is larger than the true maximum of $\rho(x)$
- Works for generic distributions and does not require the evaluation of cumulative distribution function

Cons:

- Can be inefficient if the rejection rate is very high (highly peaked distribution)
- Not directly applicable to distributions over infinite ranges

Generalization of rejection sampling can take care of some of the deficiencies. These include:

- Adaptive rejection sampling by considering several enveloping rectangles
- Variable transformation to map infinite interval into a finite one
- Sampling from a non-uniform enveloping distribution