Computational Physics (PHYS6350)

Lecture 17: Random numbers

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

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

Course materials: https://github.com/vlvovch/PHYS6350-ComputationalPhysics
Online textbook: https://vovchenko.net/computational-physics/

(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

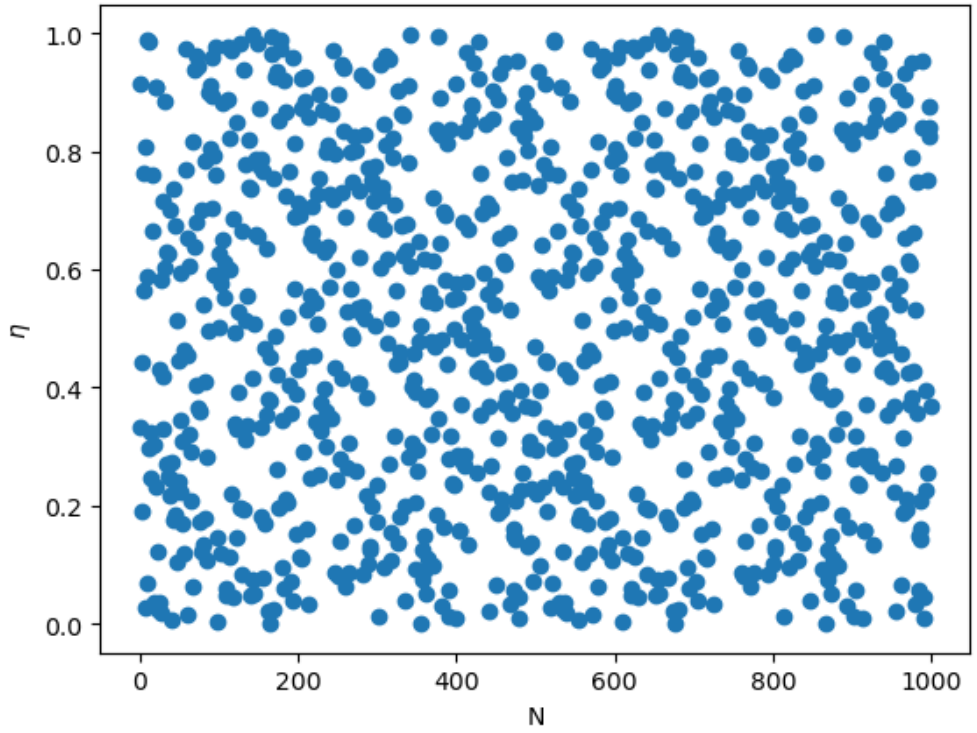
**Do not use LCG in any serious calculation(!)*

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()
```

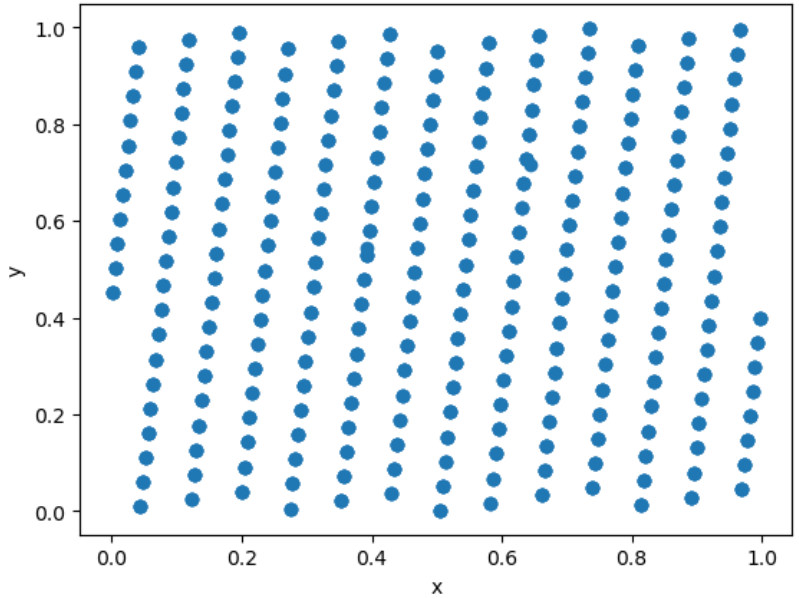
Linear congruential generator

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:

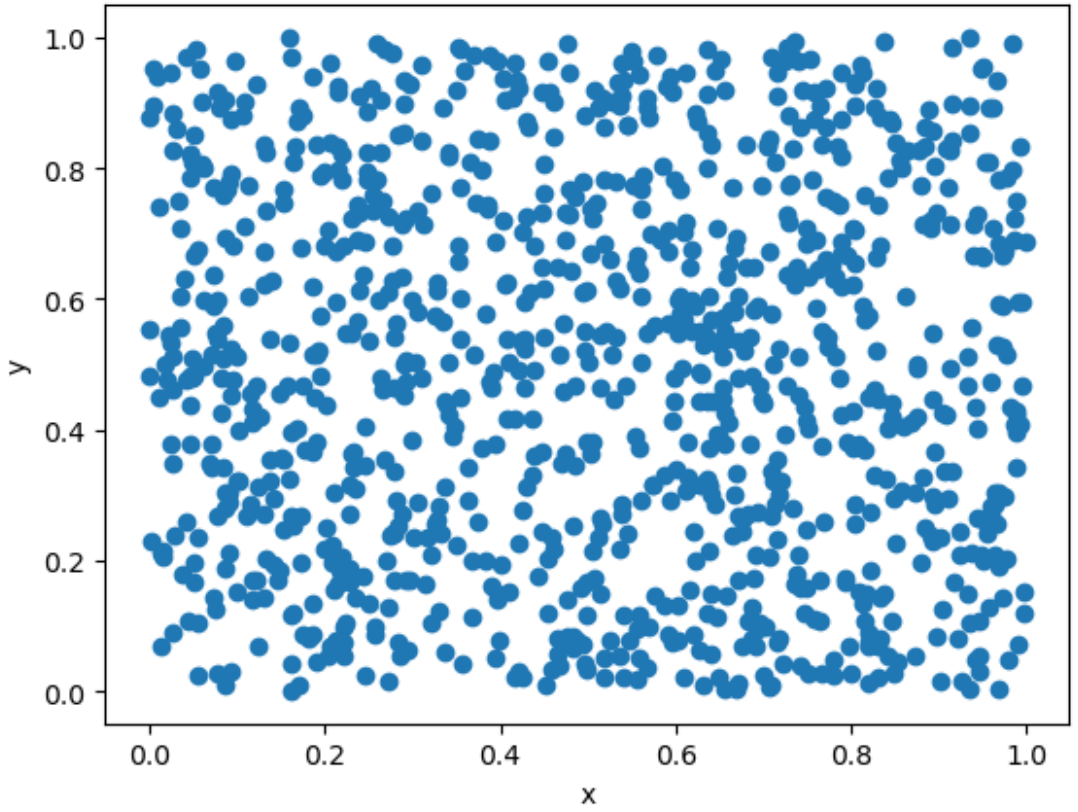
```python
# Use Mersenne Twister
import numpy as np

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()
```



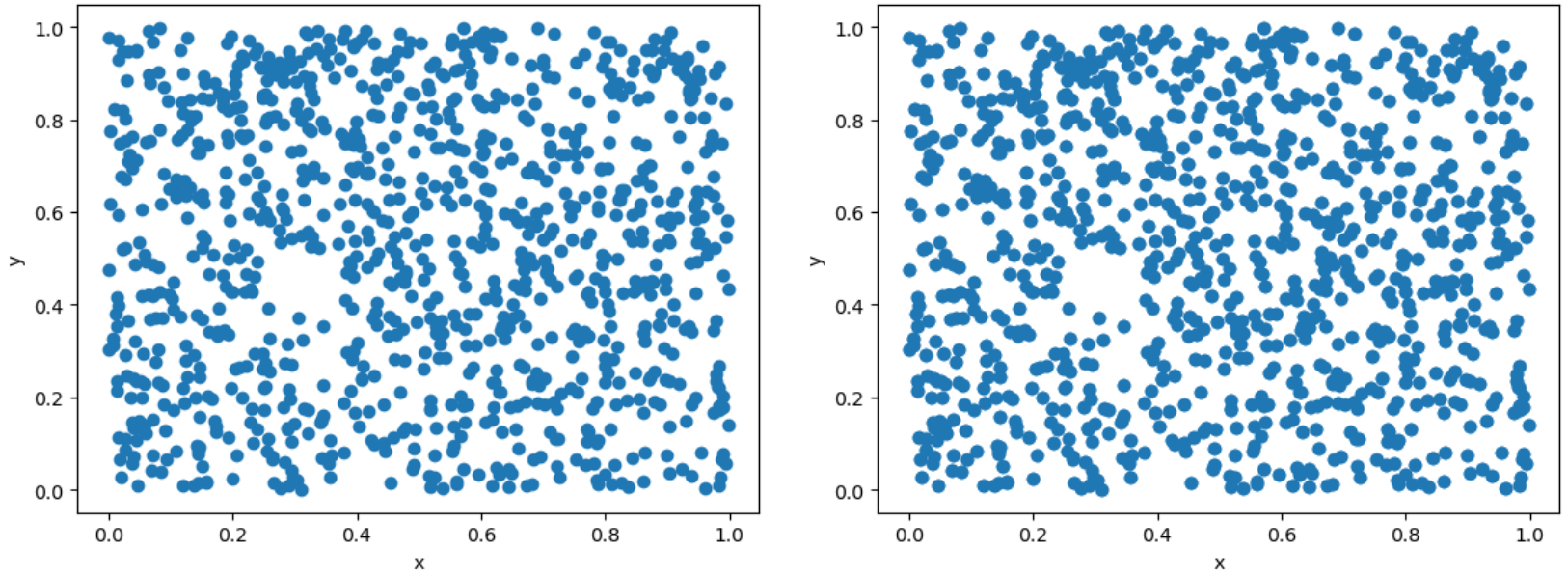
Random seed

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

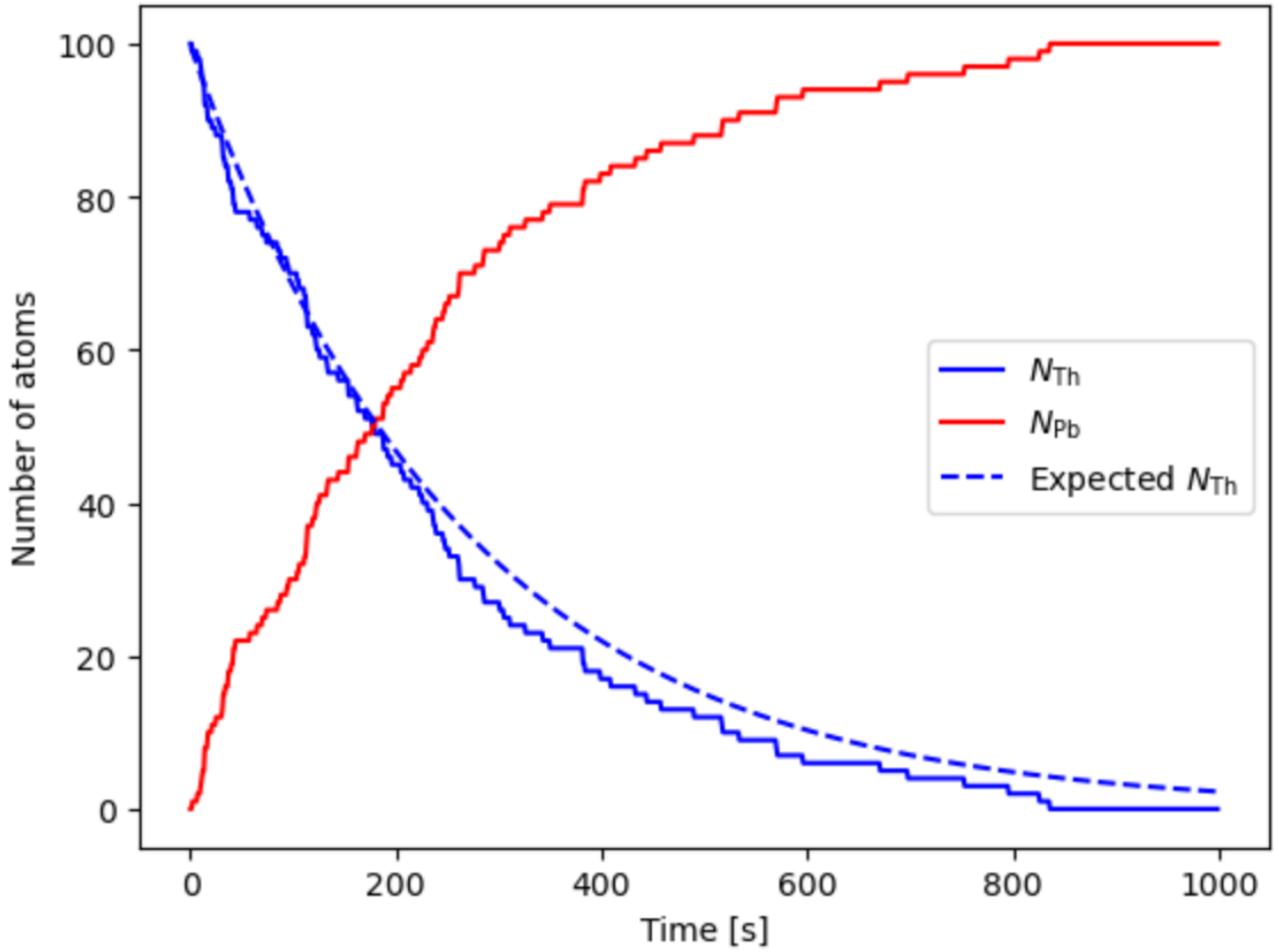
$$p(t) = 1 - 2^{-t/\tau}.$$

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
```

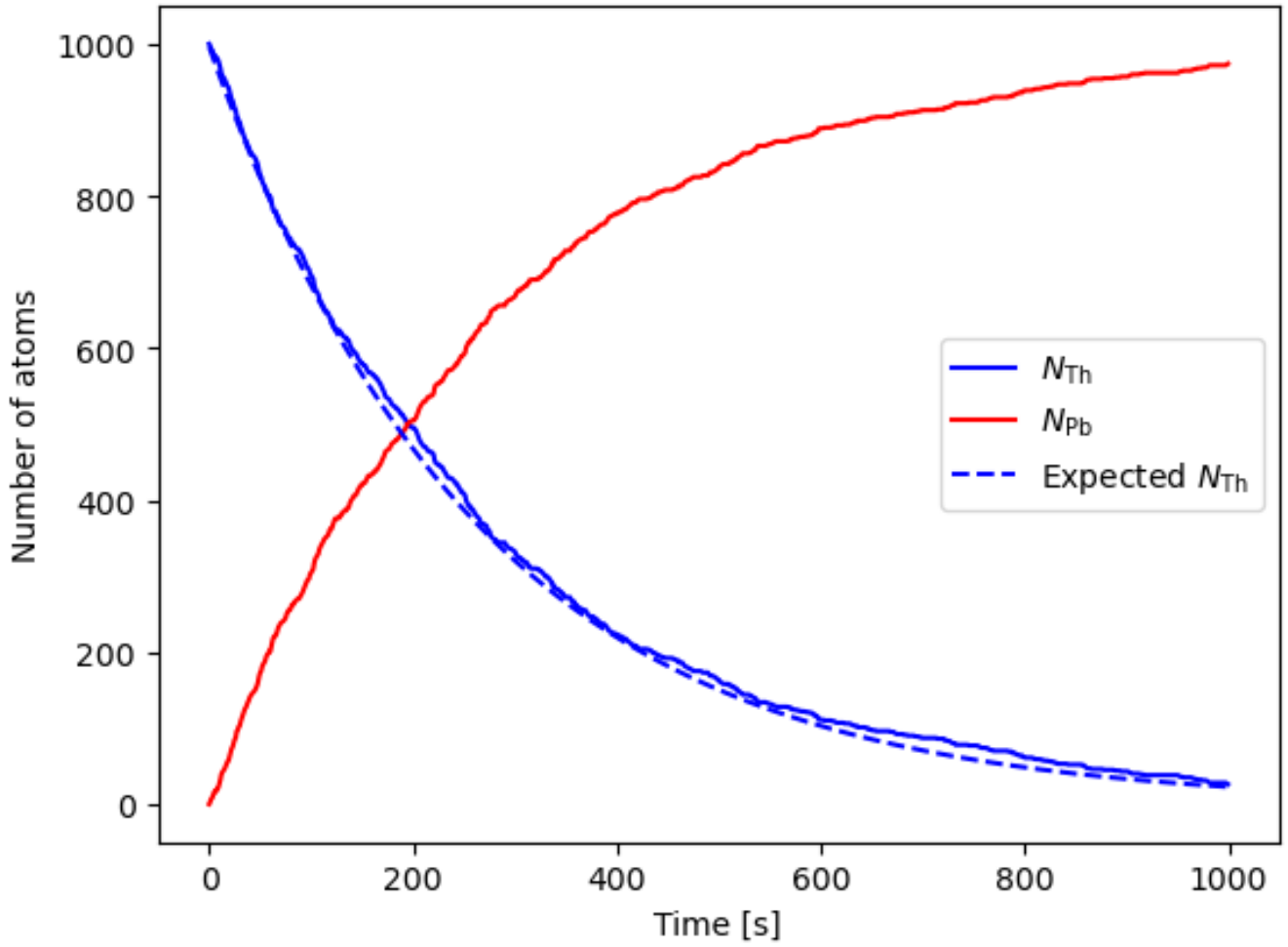
Simulation example: Radioactive decay



Expected:

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

Simulation example: Radioactive decay



Expected:

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

Simulation example: Brownian motion

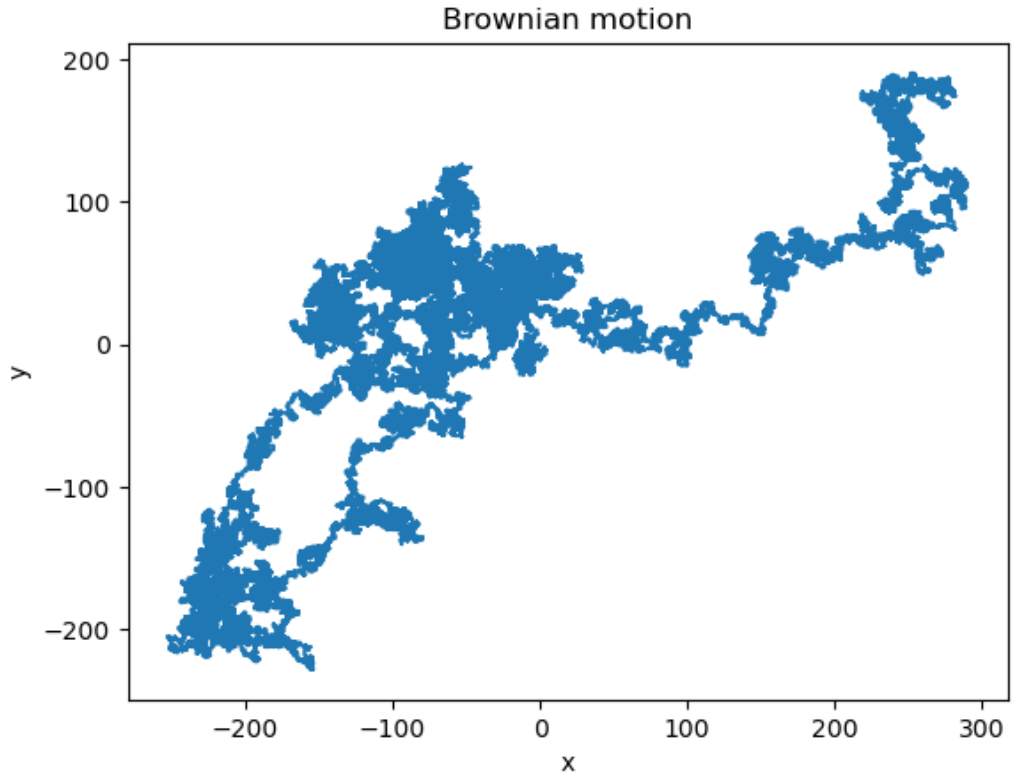
Brownian motion is a motion of a heavy particle in a gas colliding with the lighter gas particles.
We can consider a simplified 2D motion of particle by randomly making
a small step at each iteration in one of the four directions.

```
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)
```

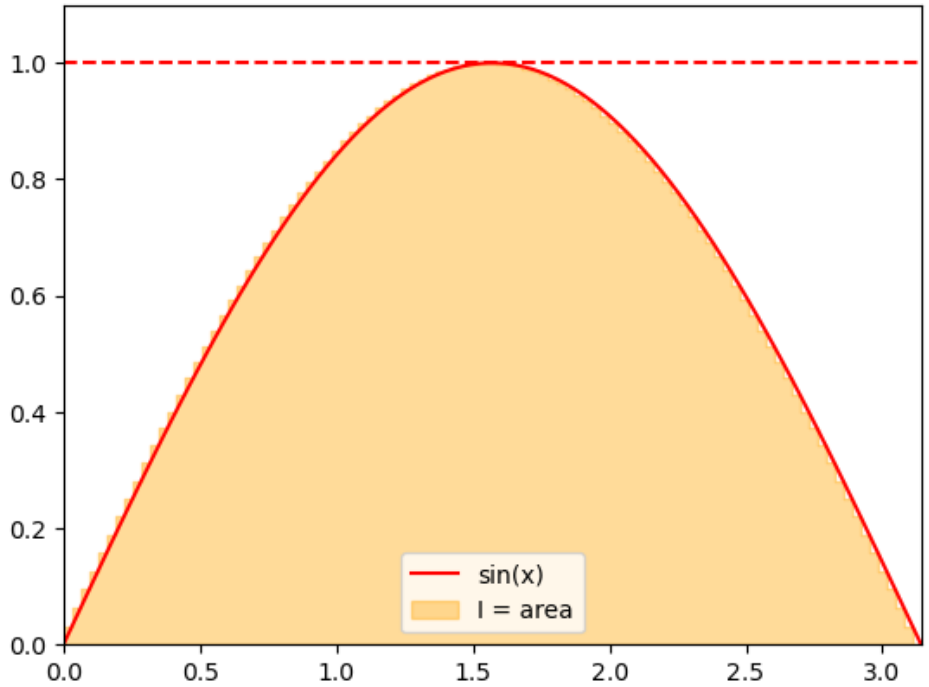


Computing integrals: Estimating the area under the curve

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 $N^{-1/2}$

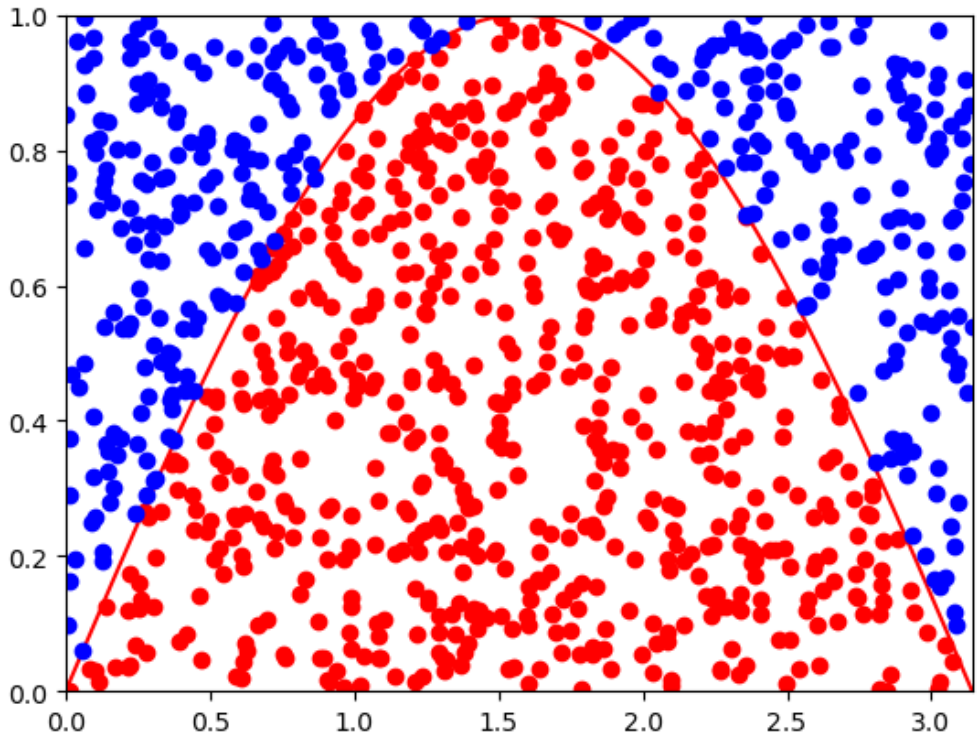
To reduce the error by factor x2 we need to sample x4 more numbers – true for most Monte Carlo methods.

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:

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

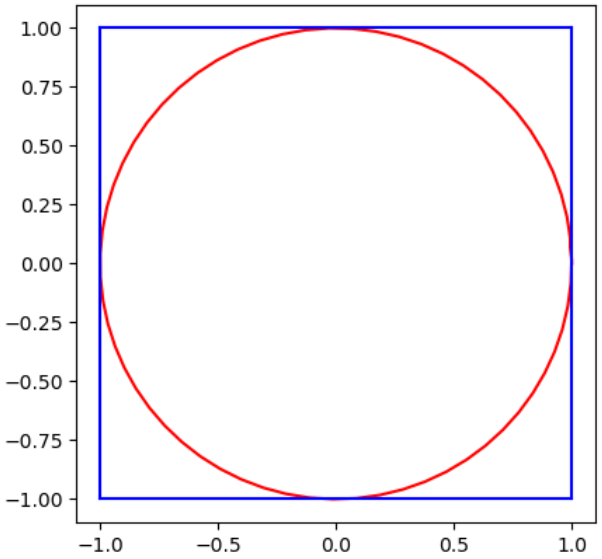
The circle can be embedded into a square with a side length of two. The area of the square is: $A_{sq} = 2^2 = 4$

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: $A_{sq} = 2^2 = 4$

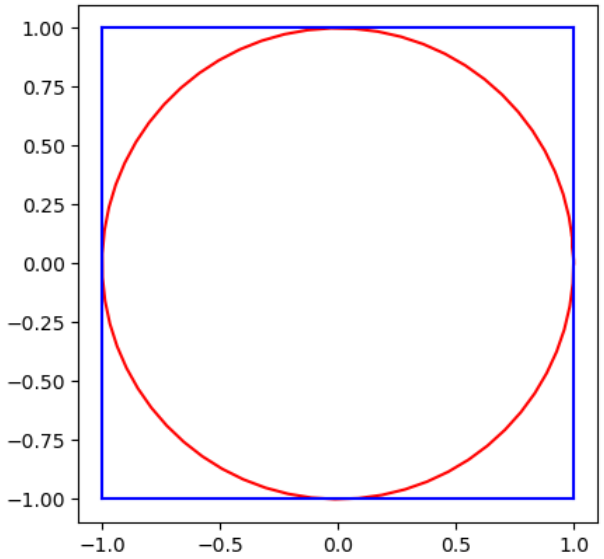
Consider now a random point anywhere inside the square. The probability that it is also inside the circle is the ratio of their areas:

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

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)
```

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



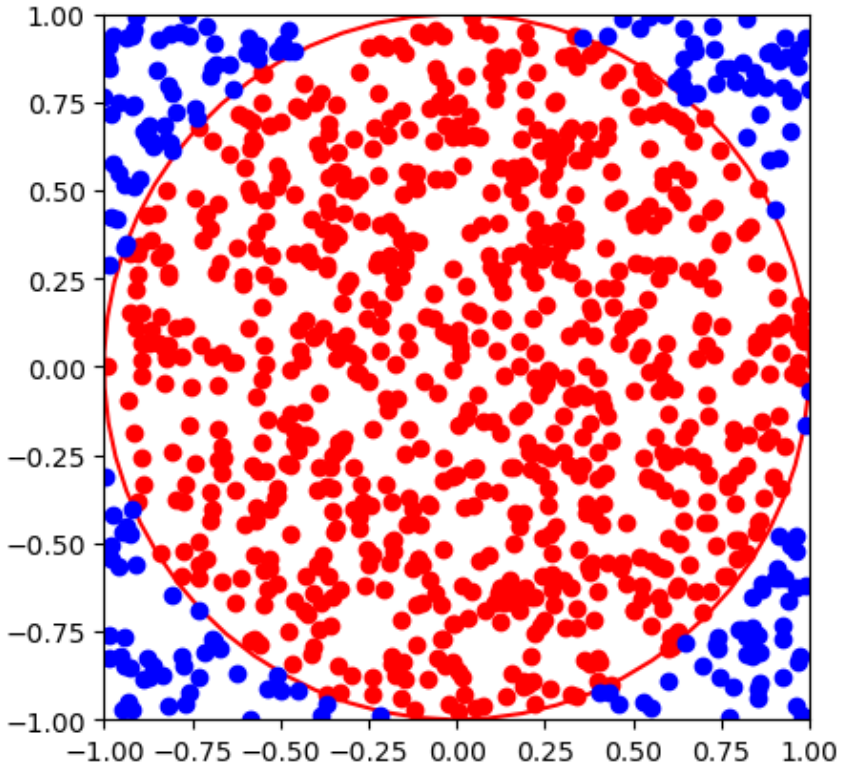
This method is known as the **Monte Carlo estimation of π**.

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):

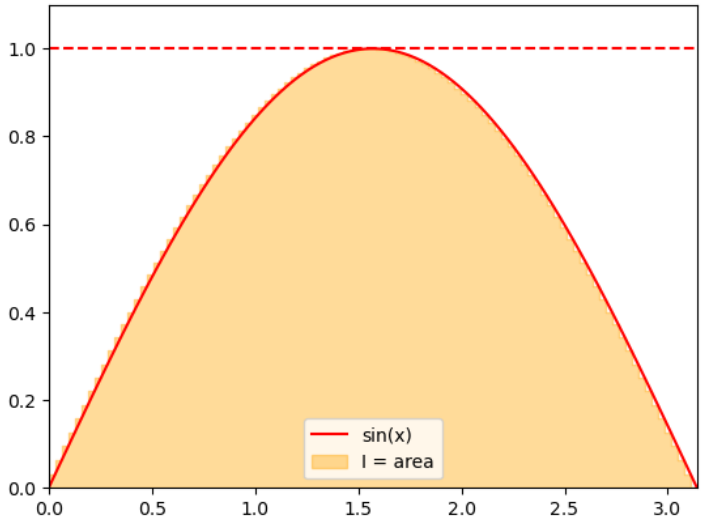
$$\langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i).$$

- 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
```