Computational Physics (PHYS6350)

Lecture 18: Random numbers and Monte Carlo methods part II

March 30, 2023

Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

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

Computing integral as the average

From previous lecture

An integral

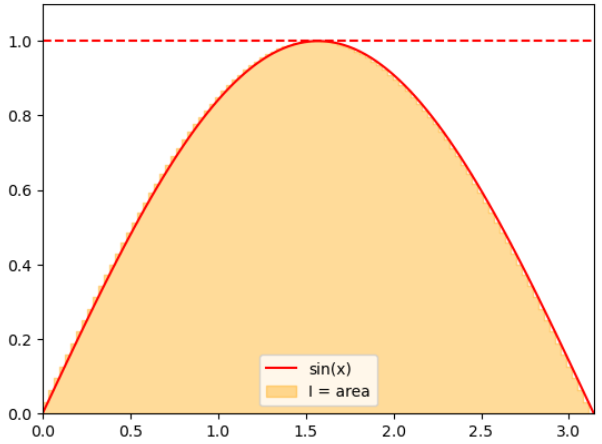
$$I = \int_a^b f(x)dx$$

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 $\langle f^2 \rangle$

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

How about multi-dimensional integrals?

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

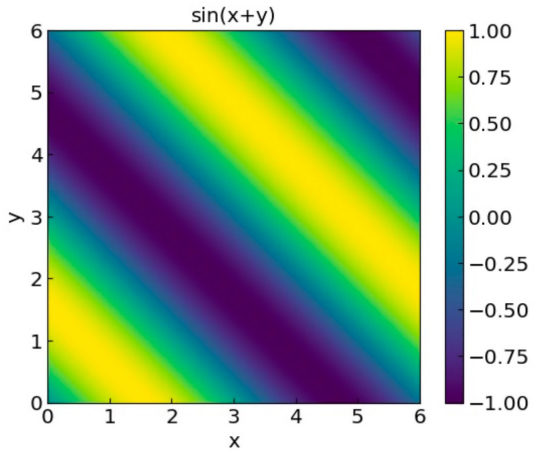
The total number of integrand evaluations is $N_{tot} = \prod_{d=1}^{N_D} N_d$,
e.g. if we use the same number N of points in each dimension, N_{tot} scales exponentially with D

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

curse of dimensionality

Computing multi-dimensional integrals: Example

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



Applying the rectangle rule we get:

```
%%time

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

Ndimmax = 4

for Ndim in range(1,Ndimmax + 1):
    Nrect = [50 for i in range(Ndim)]
    a = [0. for i in range(Ndim)]
    b = [np.pi/2 for i in range(Ndim)]
    I = rectangle_rule_multi(f,Nrect,a,b)
    print("D =",Ndim, " I =",I)
```

```
D = 1  I = 0.9681356137777438
D = 2  I = 1.9962176337747817
D = 3  I = 2.1796094400043926
D = 4  I = 0.5014154076818487
CPU times: user 6.49 s, sys: 58.6 ms, total: 6.55 s
Wall time: 6.55 s
```

Analytic result:

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

Not very accurate if we want reasonable runtime for D $>$ 3

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 $f(\eta)$?

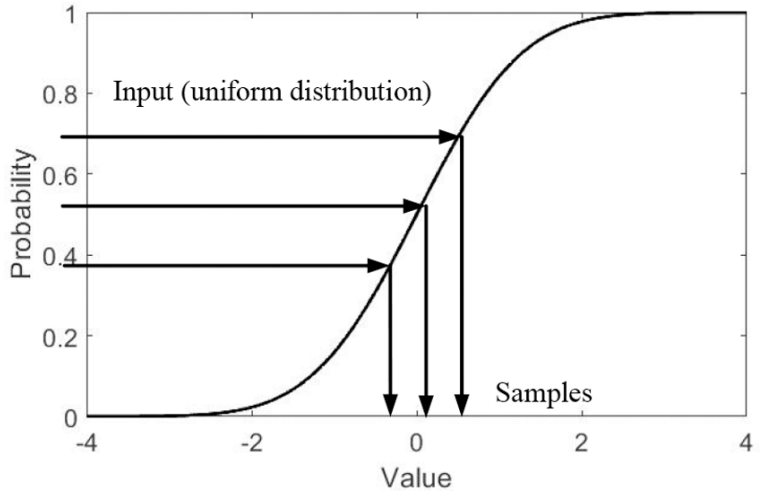
Without the loss of generality assume that $\xi \in (-\infty, \infty)$ and that $f(\eta)$ maps η to ξ such that $f(0) \to -\infty$. Consider now the cumulative distribution function $G(x) = Pr(\xi < x) = \int_{-\infty}^{x} \rho(\xi) d\xi$. It corresponds to the probability that $\eta < y$ where y is 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.

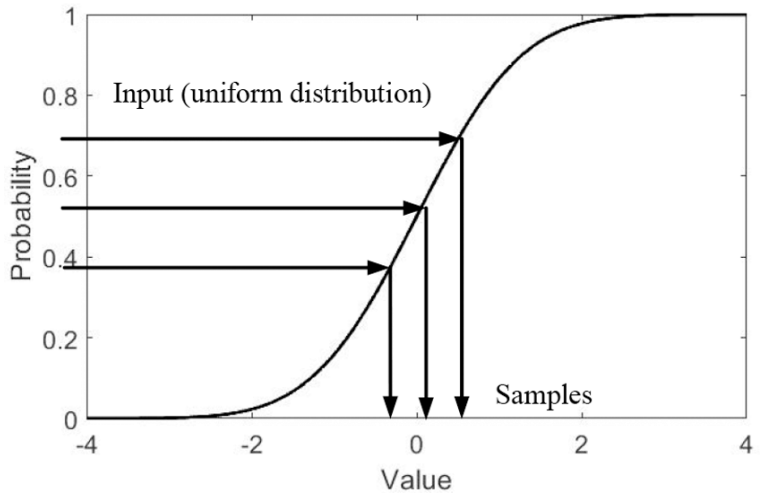


Inverse transform sampling

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

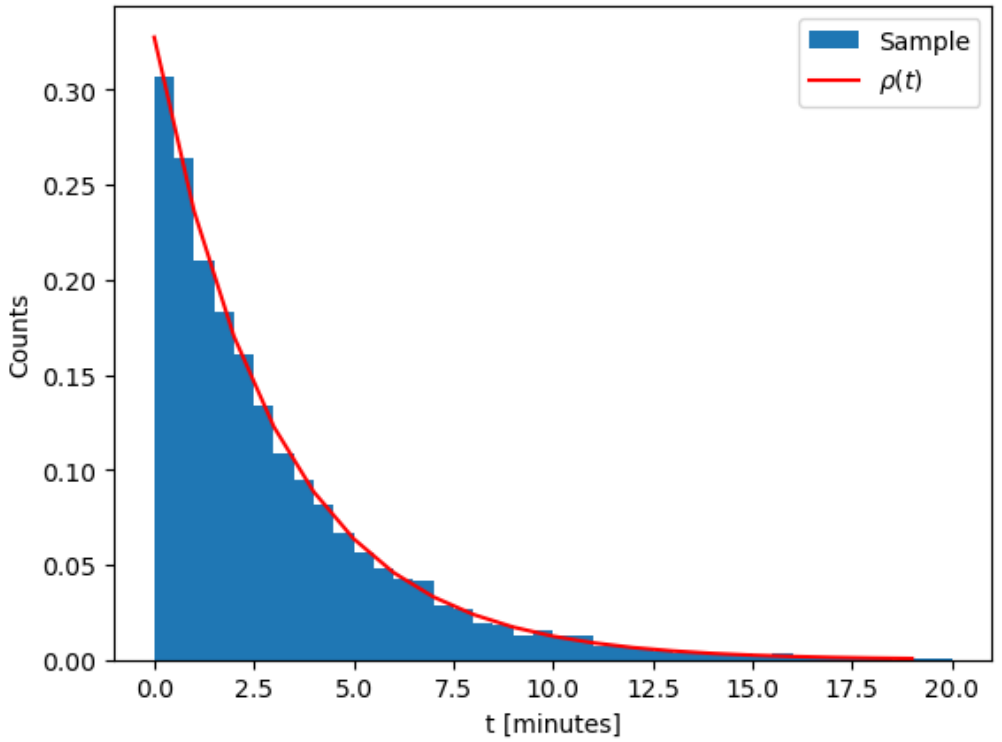
The cumulative distibution function reads
$$F(x) = \int_0^x \frac{1}{\tau} e^{-\frac{t}{\tau}} dt = 1 - e^{-\frac{x}{\tau}}.$$

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

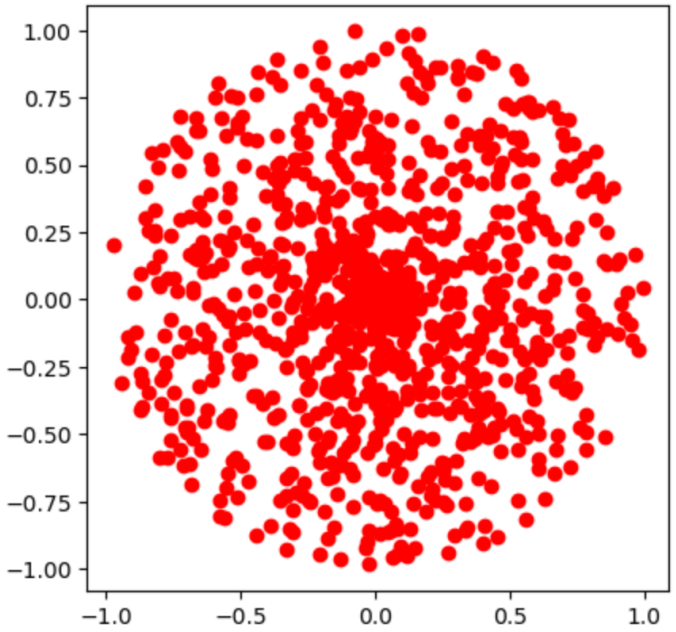
Let us sample points on a plane inside a unit circle. One way to do that is to switch to polar coordinates

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

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

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

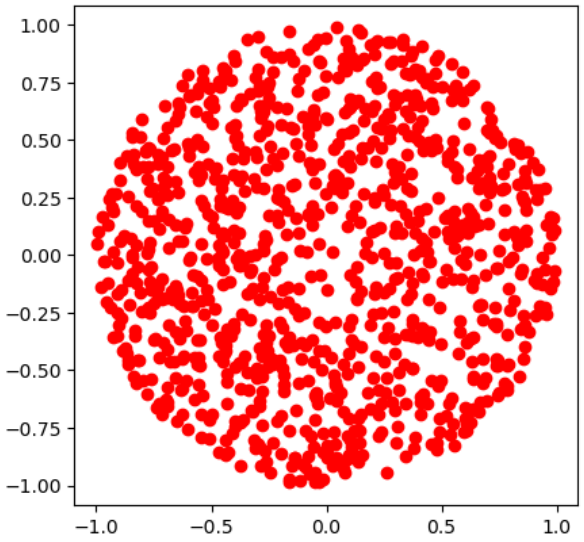
Cumulative distribution function

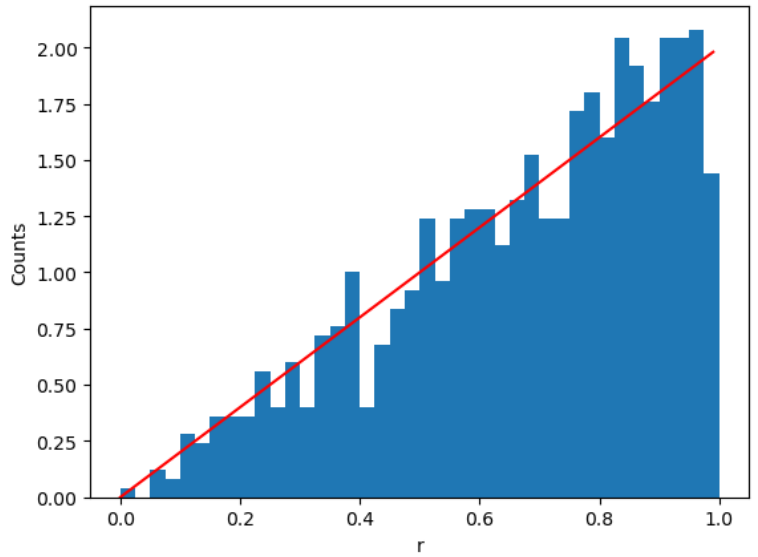
$$F_r(r) = \int_0^r \rho_r(r')dr' = r^2.$$

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

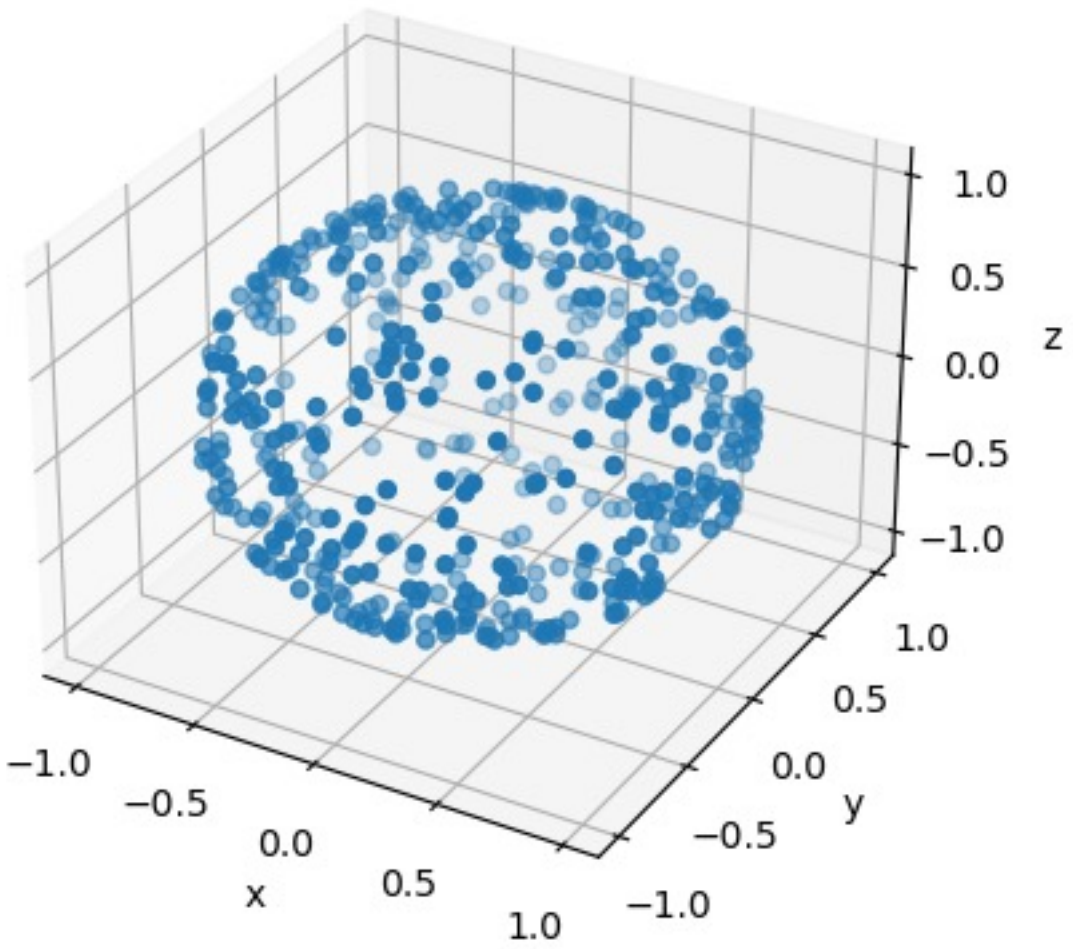
$$\cos(\theta) = 2\eta - 1,$$

and

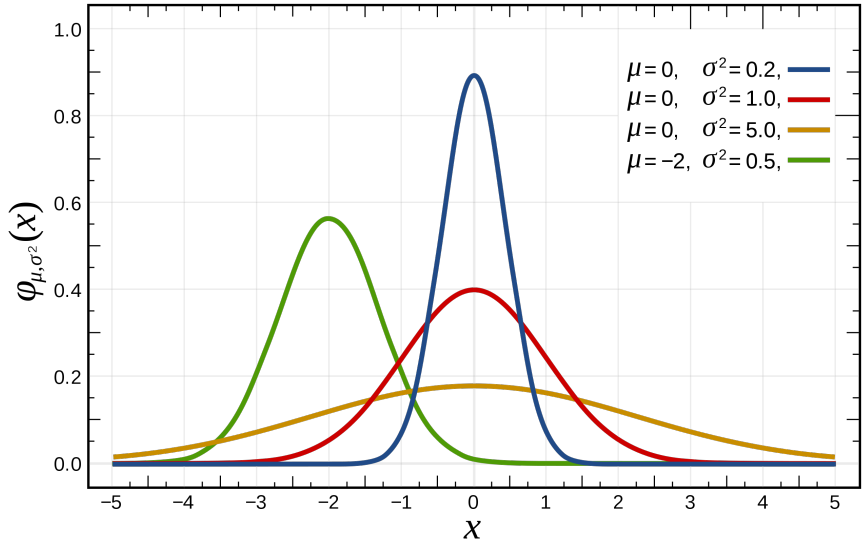
$$\sin(\theta) = \sqrt{1 - [\cos(\theta)]^2}$$

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 indepdent 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

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

Therefor 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

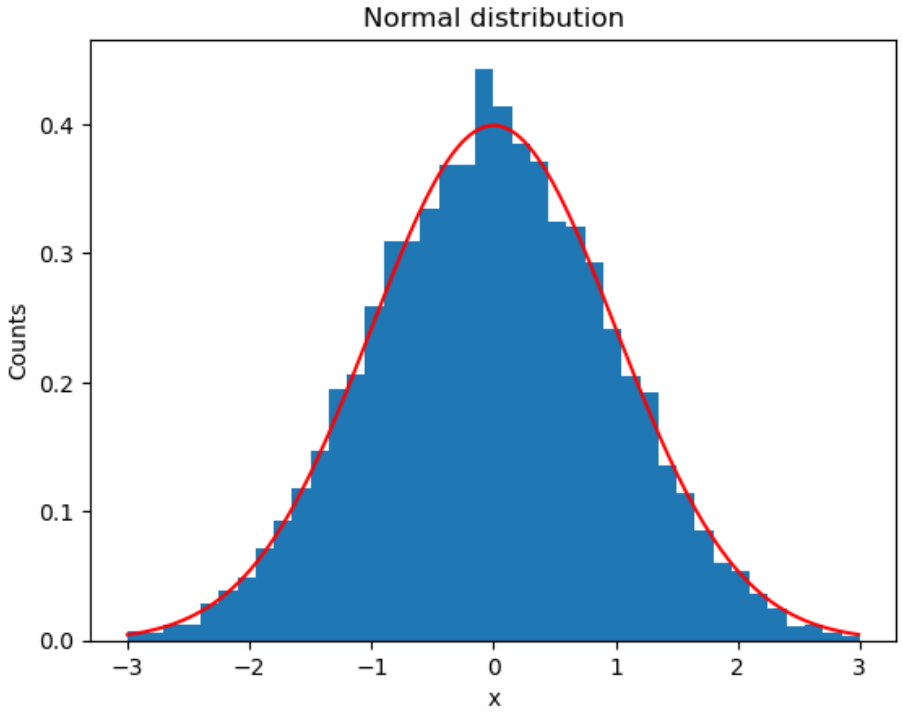
$$F_r(r) = \int_0^r r' e^{-r'^2/2} dr' = 1 - e^{-r^2/2},$$

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)
```

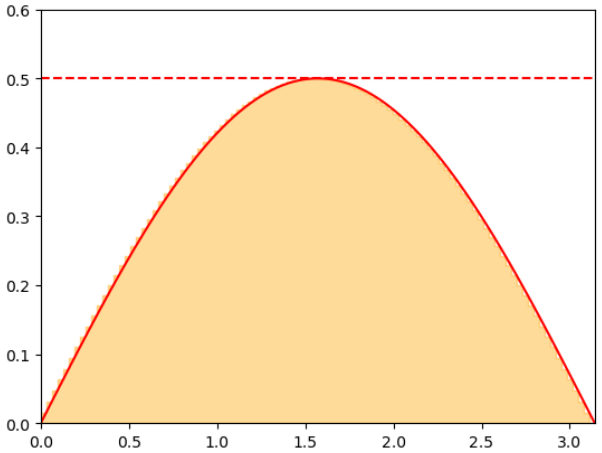


Rejection sampling

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 $p = \rho_\theta(\theta_{\text{cand}})/\rho_\theta^{\max}$.

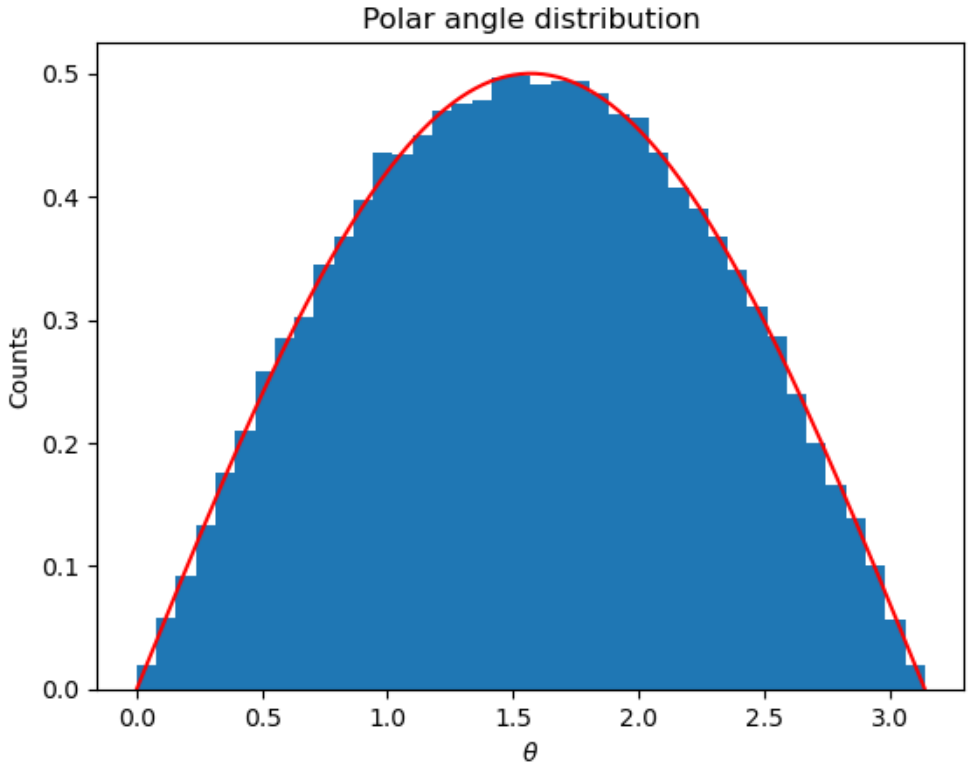
The second step can be performed by sampling y as a uniform distribution over $(0, \rho_\theta^{\max})$ and accepting θ_{cand} is $y < \rho_\theta(\theta_{\text{cand}})$.

The procedure has simple geometrical interpration. Considering $\theta_{\text{cand}} \equiv x$ and y to be x and y 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 adavantage 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 evaluation of cumulative distribution function

Cons:

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

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

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

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

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

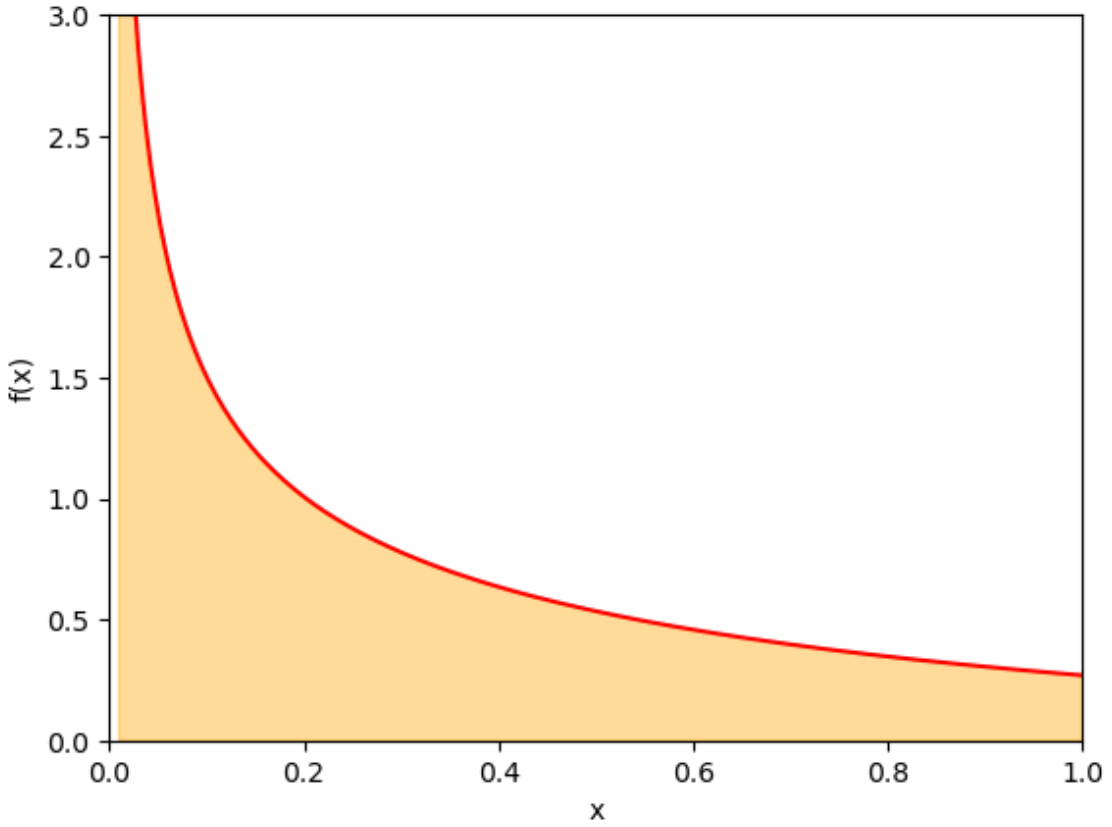
- For w(x) ∝ f(x) one has $\left\langle \frac{f(x)}{w(x)} \right\rangle = \text{const} = I$ and $\delta I = 0$

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

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

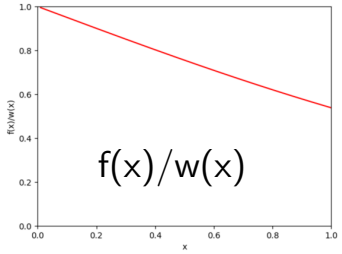
```
def uniform_sample():
    eta = np.random.rand()
    return eta

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.