PHY407
Lab #10: Random numbers, Monte Carlo Integration

Q1 Nicholas Pavanel, Q2 and Q3 Hayley Agler

November 2020

Q1

a)

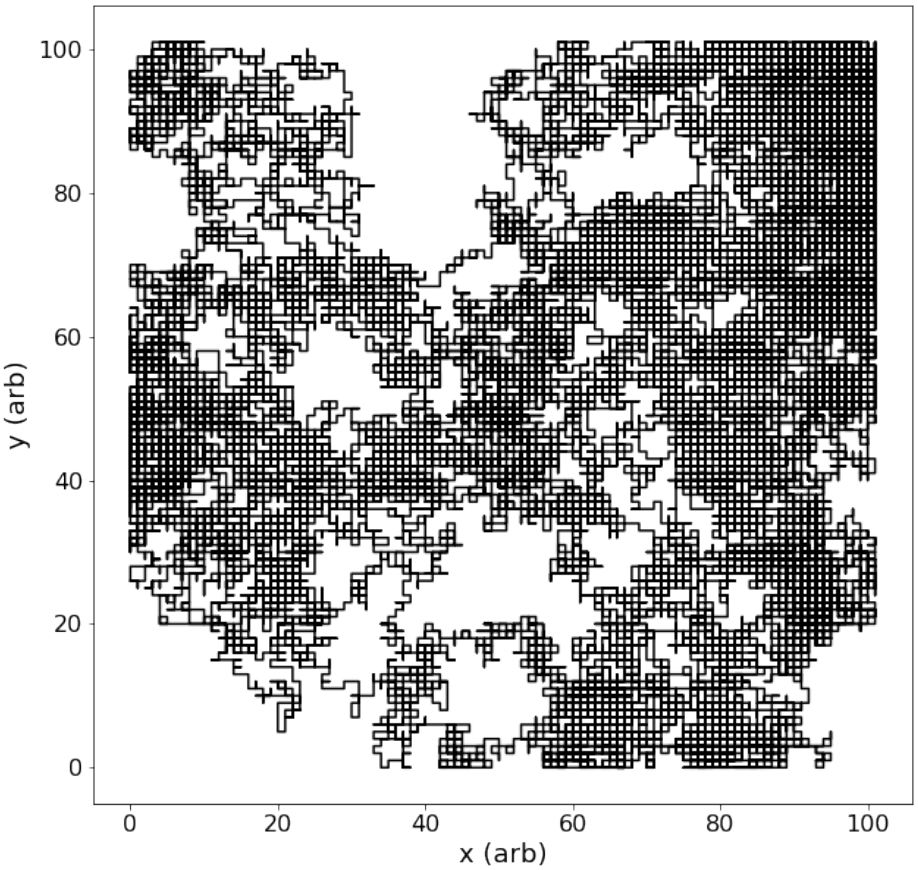

Figure 1: This figure shows the trajectory of a particle that has undergone 50,000 steps of Brownian motion over a two dimensional lattice.

Figure 1 shows the trajectory of a particle undergoing Brownian motion. Brownian motion is random motion. That is, at each step, the particle is randomly assigned to move in one of four directions: up, down, left or right. In the case that the particle is beside a boundary, the boundary direction of motion is removed from the particle's selection of movement directions before a random direction of motion is assigned. The particle in Figure 1 is confined to a two dimensional lattice with size 101x101, initialized at the centre of the lattice, and integrated for 50,000 steps.

The functions that produce a Brownian trajectory for a particle defined in a two dimensional lattice of size *domain_size* for a number of steps *num_time_steps* are shown below:

```
# define the function that determines where the next move will be, include boundary conditions
def nextmove(x, y, L):
    """ randomly choose a direction: 0 = up, 1 = down, 2 = right, 3 = left"""

    choices = [0,1,2,3]  # define the choices for movement

    # we take care of the cases when the particle is in a corner
    if x == 0 and y == 0:
        hold = choices.pop(3) # remove the move left option
        hold = choices.pop(1) # remove the move down option
    elif x == 0 and y == L:
        hold = choices.pop(3) # remove the move left option
        hold = choices.pop(0) # remove the move up option
    elif x == L and y == 0:
        hold = choices.pop(2) # remove the move right option
        hold = choices.pop(1) # remove the move down option
    elif x == L and y == L:
        hold = choices.pop(2) # remove the move right option
        hold = choices.pop(0) # remove the move up option

    # we take care of the cases when the particle is on a boundary
    elif x == 0:
        hold = choices.pop(3) # remove the move left option if the particle is on the left boundary
    elif x == L:
        hold = choices.pop(2) # remove the move right option if the particle is on the left boundary
    elif y == 0:
        hold = choices.pop(1) # remove the move down option if the particle is on the bottom boundary
    elif y == L:
        hold = choices.pop(0) # remove the move up option if the particle is on the top boundary

    direction =  np.random.choice(choices)  # choose the direction of movement

    if direction == 0:  # move up
        y += 1
    elif direction == 1:  # move down
        y -= 1
    elif direction == 2:  # move right
        x += 1
    elif direction == 3:  # move left
        x -= 1
    else:
        print("error: direction isn't 0-3")

    return x, y
```

```
# define a function that returns random walk coordinates
def brownian_walk(domain_size, num_time_steps):
    # arrays to record the trajectory of the particle
    X = []
    Y = []
    centre_point = (domain_size-1)//2  # middle point of domain
    xp = centre_point
    yp = centre_point
    for i in range(num_time_steps):
        xp, yp = nextmove(xp, yp, domain_size)
        X.append(xp)
        Y.append(yp)
    return X, Y
```

b)

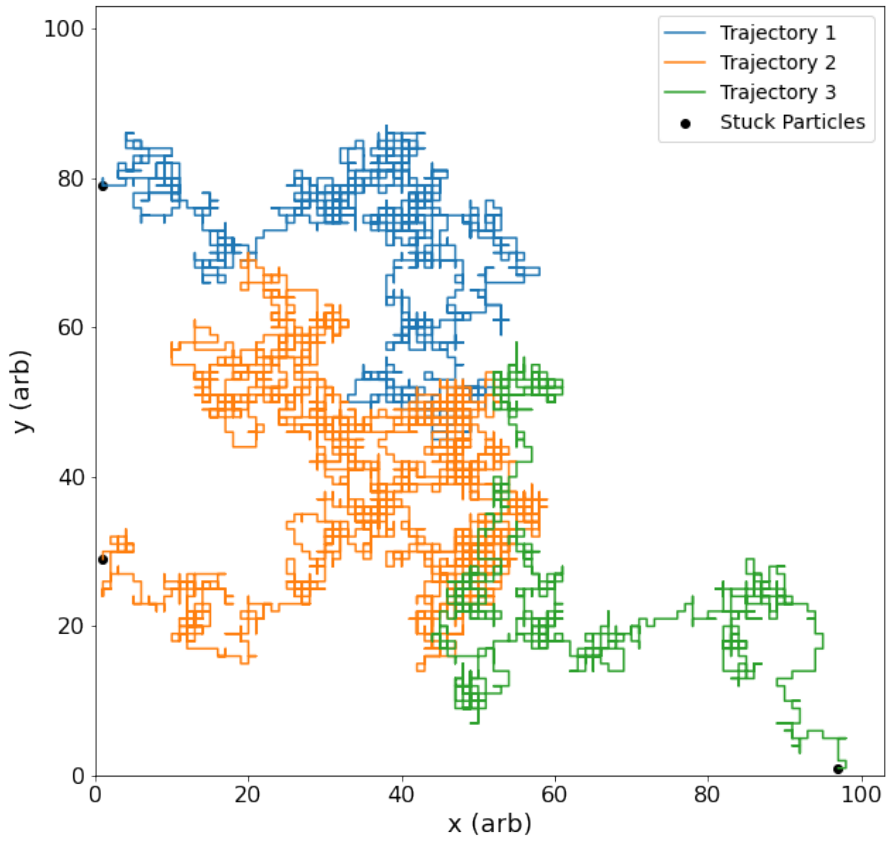

Figure 2: This figure displays the core-workings of the diffusion-limited aggregation (DLA) program that is presented in the next subsection. This figure shows that the DLA program of the next section initializes a particle in the centre of the domain, and evolves it with the Brownian motion functions of the previous section until the particle 'hits' a wall (truthfully it is until the particle hits a wall or another particle). At this point, the particle sticks to its location, and a new particle is initialized.

Figure 2 shows the beginning of diffusion-limited aggregation (DLA). Figure 2 shows the trajectory of three particles undergoing Brownian motion until they 'hit' a wall or another particle. That is, the particles evolve with Brownian motion until their next step would take them out of the defined lattice, or onto the top of another particle. When the next step of the particle is as such, the particle sticks; it does not move from its current location, its evolution is finished, and the evolution of new particle is initialized in the centre of the domain.

The central idea for the DLA programs defined in this section, and used in the next section, is a 'mirror array'. The 'mirror array' of the programs shown at the end of this section are the arrays referred to as 'anchors'. The main idea is that the mirror array will mimic the integration domain, except that it will be composed of ones and zeros, zeros for free space and ones for boundaries or space held by other stuck particles. After a random direction of motion is chosen, the program will consult the mirror array as to whether the chosen direction is free, and thus will instruct the programs when a particle should move or stick.

The functions that perform this type of DLA integration, and in this case, saves the trajectories of all integrated particles as well as the positions of stuck particles, for a domain size of size *domain_size* and for a number of particles *num_particles* are shown below:

```
# define the function that determines where the next move will be, include boundary conditions
def nextmove_anchor_b(x, y, anchors):
    """ randomly choose a direction: 0 = up, 1 = down, 2 = right, 3 = left"""

    choices = [0,1,2,3]  # define the choices for movement
    direction =  np.random.choice(choices) # choose the direction of movement

    if direction == 0:  # move up
        if anchors[x][y+1] == 0:
            y += 1 # if empty move to it
            moves = True
        else:
            anchors[x][y] = 1      # if a wall or another particle stick in your spot
            moves = False
    elif direction == 1:         # move down
        if anchors[x][y-1] == 0:
            y -= 1
            moves = True
        else:
            anchors[x][y] = 1
            moves = False
    elif direction == 2:  # move right
        if anchors[x+1][y] == 0:
            x += 1
            moves = True
        else:
            anchors[x][y] = 1
            moves = False
    elif direction == 3:  # move left
        if anchors[x-1][y] == 0:
            x -= 1
            moves = True
        else:
            anchors[x][y] = 1
            moves = False
    else:
```

```
        print("error: direction isn't 0-3")

    return x, y, anchors, moves

# define a function that returns random walk coordinates
def DLA_walk_b(domain_size,num_particles):

    # define anchor array, locations that will cause a particle to stick = 1, open = 0
    anchors = np.zeros((domain_size+2,domain_size+2))     # define full array
        (of size +2 as we add two lines of zeros for boundaries)
    x = np.zeros(domain_size+2)   # define list that will act as middle anchor lines
    x[0] = x[-1] = 1              # add middle anchor points
    boundary_anchors = np.ones(domain_size+2)     # create final anchors line
    anchors[0] = anchors[-1] = boundary_anchors   # create the boundary lines
    anchors[1:-1] = x                             # create the middle anchor points

    t0 = time()
    X = []                 # these arrays hold the trajectories of all the particles
    Y = []
    stuck_x = []           # these arrays hold the locations of stuck particles
    stuck_y = []
    centre_point = (len(anchors) - 1) // 2  # middle point of domain
    for i in range(num_particles):
        x = []             # these arrays hold the trajectory of a single particle
        y = []
        xp = yp = centre_point       # start each new particle in the centre
        moves = True                 # initalize the particle with moves open
        while moves is True:
            xp, yp, anchors, moves = nextmove_anchor_b(xp, yp, anchors)
            x.append(xp)
            y.append(yp)
        stuck_x.append(x[-1])
        stuck_y.append(y[-1])
        X.append(x)
        Y.append(y)
    print('Total time taken: ' + str(time()-t0) + ' s or ' + str((time()-t0)/60) + ' min')
    return X, Y, anchors, stuck_x, stuck_y
```

where X, Y hold the trajectories, *stuck_x* and *stuck_y* hold the locations of stuck particles, and anchors holds the information of stuck particles only.

c)

Figure 3 shows an example of diffusion limited aggregation. DLA is described as the following. A particle is initialized in the centre of the domain. The initialized particle randomly steps throughout the domain, until it 'hits' a wall. That is, until its next step would take it out of the domain. When the particle 'hits' a wall, it sticks: it does not make the step out of the domain and instead ends its evolution in the location that it is in. Now a new partcile is initialized in the centre of the domain. The new particle again randomly steps throughout the domain, this time until it either 'hits' a wall, or until it 'hits' another particle (hitting another particle refers to when the next step of the particle would take on top of another particle).

The DLA program that produced Figure 3 was created so that it would keep initializing new particles in the centre of the domain until an initialized particle could not move out of the initialization position. Furthermore, the DLA program that produced Figure 3 was created so that it would only save the locations of stuck particles.

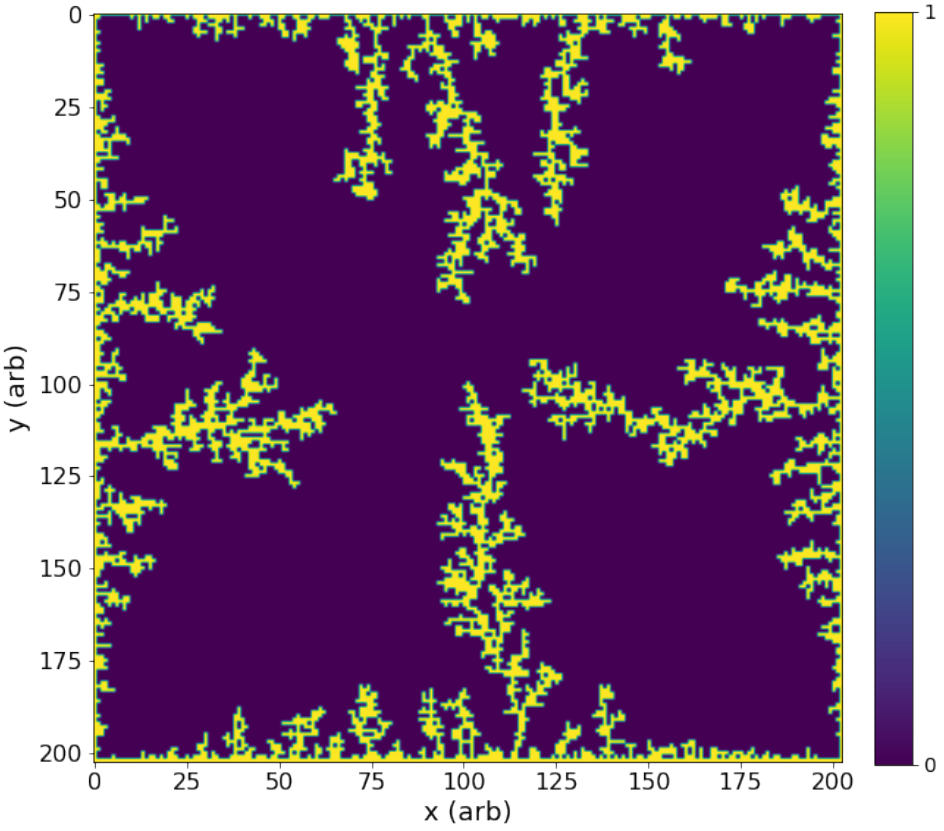

Figure 3: This figure shows an example of two dimensional diffusion-limited aggregation. In this figure areas of dark purple (0) represent empty space, and ares of bright yellow (1) represent 'stuck' particles (or the boundaries). Information about stuck particles and free space was held in a 'mirror' array composed of 1s and 0s of size domain_size + 2 x domain_size + 2 to account for all boundaries. Every particle in the image was initialized in the centre of the domain and evolved under Brownian motion until it was going to step onto another particle or out of the domain.

The example of DLA in Figure 3 was ran over a two dimensional square domain of width 201 steps.

The functions that perform this type of DLA integration, for a domain size of size *domain_size* are shown below:

```
# define the function that determines where the next move will be, include boundary conditions
def nextmove_anchor(x, y, anchors):
    """ randomly choose a direction: 0 = up, 1 = down, 2 = right, 3 = left"""

    choices = [0,1,2,3]  # define the choices for movement
    direction =  np.random.choice(choices) # choose the direction of movement
    centre_point = (len(anchors[0]) - 1) // 2

    if direction == 0:             # move up
        if anchors[x][y+1] == 0:
            y += 1                 # if empty move to it
        else:
            anchors[x][y] = 1      # if a wall or another particle create stuck particle
            x = y = centre_point   # re-initialize to centre of domain
    elif direction == 1:           # move down
        if anchors[x][y-1] == 0:
            y -= 1
        else:
            anchors[x][y] = 1
            x = y = centre_point
    elif direction == 2:           # move right
        if anchors[x+1][y] == 0:
            x += 1
        else:
            anchors[x][y] = 1
            x = y = centre_point
    elif direction == 3:           # move left
        if anchors[x-1][y] == 0:
            x -= 1
        else:
            anchors[x][y] = 1
            x = y = centre_point
    else:
        print("error: direction isn't 0-3")

    # if all areas around the particle are full, stop motion
    if anchors[x][y+1] == 1 and anchors[x][y-1] == 1 and anchors[x+1][y] == 1 and anchors[x-1][y] == 1:
        moves = False
    else:
        moves = True

    return x, y, anchors, moves

# define a function that returns random walk coordinates
def DLA_walk(domain_size):
    t0 = time()

    # define anchor array, locations that will cause a particle to stick = 1, open = 0
    anchors = np.zeros((domain_size+2,domain_size+2))    # define total array
    x = np.zeros(domain_size+2) # define list that will act as middle anchor lines
```

```
x[0] = x[-1] = 1                                        # add middle anchor points
boundary_anchors = np.ones(domain_size+2)               # create final anchors line
anchors[0] = anchors[-1] = boundary_anchors             # create the boundary lines
anchors[1:-1] = x                                       # create the middle anchor point

centre_point = (len(anchors) - 1) // 2  # middle point of domain
print(centre_point)
xp = centre_point
yp = centre_point
moves = True
while moves is True:
    xp, yp, anchors, moves = nextmove_anchor(xp, yp, anchors)
print('Total time taken: ' + str(time()-t0) + ' s or ' + str((time()-t0)/60) + ' min')
return anchors
```

Q2

This question asks us to estimate the volume of a 10-dimensional hypersphere of unit radius using the Monte Carlo mean value method for many dimensions. The integral to be computed is:

$$I = \int_{-1}^{+1} f(\boldsymbol{r}) \mathrm{d}\boldsymbol{r} \tag{1}$$

where $\boldsymbol{r}$ is a 10-dimensional vector. The function $f(\boldsymbol{r})$ is equal to 1 everywhere inside the hypersphere and 0 everywhere outside, in other words:

$$f(\boldsymbol{r}) = \begin{cases} 1 & \text{if } \sum_{i=1}^{10} r_i^2 \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

To compute this integral, we generate a uniformly random 10-dimensional vector $\boldsymbol{r}$ where each r_i is in the range from -1 and 1. Then the integral is computed using the mean value Monte Carlo method:

$$I \simeq \frac{V}{N} \sum_{i=1}^{N} f\left(\mathbf{r}_i\right) \tag{3}$$

where $N = 1,000,000$ and V is the volume of the hypercube enclosing the hypersphere. In this case, the length of a side of the hypercube is $L = 2$ so its volume is $V = 2^{10} = 1024$. The error associated with this calculation was computed by first finding the variance:

$$\operatorname{var} f = \left\langle f^2 \right\rangle - \langle f \rangle^2 = \frac{1}{N} \sum_{i=1}^{N} [f(\mathbf{r}_i)]^2 - \left(\frac{1}{N} \sum_{i=1}^{N} f(\mathbf{r}_i) \right)^2 \tag{4}$$

Then the standard deviation was found using:

$$\sigma = V \sqrt{\frac{\operatorname{var} f}{N}} \tag{5}$$

The following table summarizes the results of the above method:

I	$\operatorname{var} f$	σ
2.551808	$2.485\,790 \times 10^{-3}$	$5.105\,428 \times 10^{-2}$

To 4 significant digits, the integral in equation (1) was calculated to be $I = 2.552$ with a standard deviation of $\sigma = 5.105 \times 10^{-2}$.

We can compare this result to the volume of the 10-dimensional hypersphere calculated using the following equation:

$$V_n(R) = \frac{R^n \pi^{n/2}}{\Gamma\left(\frac{n}{2}+1\right)} \tag{6}$$

where

$$\forall x \in R_{\geq 0}, \Gamma(x+1) = x\Gamma(x) \quad \text{and} \quad \forall n \in N^*, \Gamma(n) = (n-1)! \tag{7}$$

In our case, $R = 1$, $n = 10$, and this equation gives a value of $V = 2.550$. This tells us that the volume found using the Monte Carlo method is very close to the actual volume, and is accurate up to about 3 significant digits.

Q3

a)

This question asks us to evaluate the integral:

$$\int_0^1 \frac{x^{-1/2}}{1+\mathrm{e}^x}\,\mathrm{d}x \tag{8}$$

using the regular mean value and importance sampling methods. To compute the integral with the mean value method, the following equation was used:

$$I \simeq \frac{b-a}{N}\sum_{i=1}^{N} f\left(x_i\right) \tag{9}$$

where the x_i are randomly chosen points between the bounds of integration (a=0, b=1), and the number of sample points is $N = 10000$. The integral in equation (8) was calculated again using the importance sampling method, using the equation:

$$I \simeq \frac{1}{N}\sum_{i=1}^{N} \frac{f\left(x_i\right)}{w\left(x_i\right)} \int_a^b w(x)\mathrm{d}x \tag{10}$$

with $N = 10000$. The weighting function $w(x) = x^{-1/2}$ removes the singularity in the function $f(x)$ at $x = 0$. In order to generate non-uniformly random points, the transformation method was used with the probability distribution:

$$p(x) = \frac{1}{2\sqrt{x}} \tag{11}$$

The transformation method works by computing:

$$\int_{-\infty}^{x(z)} p\left(x'\right)\mathrm{d}x' = \int_0^z \mathrm{d}z' = z \tag{12}$$

then solving for x. In this case, we get $x(z) = z^2$. By feeding uniform random numbers z in the interval from 0 to 1 into $x(z)$, we get nonuniform random values x with the distribution p(x). These nonuniform random values were then used to compute equation (10). Both the mean value and importance sampling calculations were repeated 100 times.

The resulting values of the integral I were plotted as histograms in figures 4 and 5 using 10 bins from 0.80 to 0.88. From figure 5 we can see that all of the values of the integral (8) computed using importance sampling fall between 0.83 and 0.85, whereas in figure 4 we see that the values computed using the regular mean value method fall approximately between 0.81 and 0.88. Thus it is clear from figures 4 and 5 that the importance sampling method has superior precision over the mean value method.

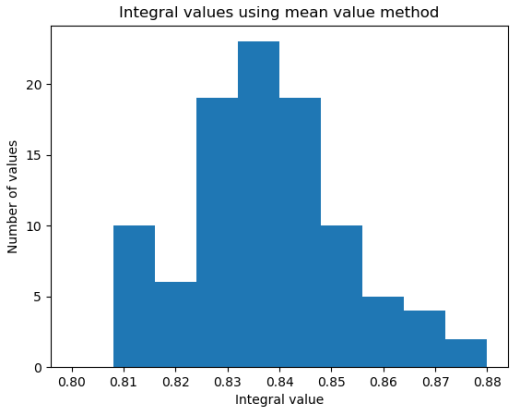

Figure 4: Values of the integral (8) computed using the regular mean value method.

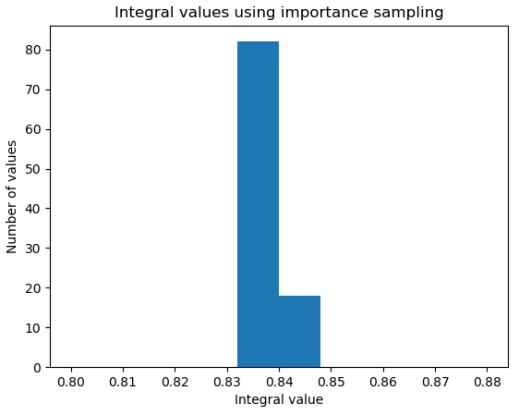

Figure 5: Values of the integral (8) computed using the importance sampling method.

b)

Part a) was repeated again for the integral with a rapidly varying integrand:

$$\int_0^{10} \exp(-2|x-5|)dx \tag{13}$$

Values of this integral were computed using the regular mean value method and importance sampling method as outlined in 3a). However, this time for the importance sampling method, we chose an importance function of the form:

$$w(x) = \frac{1}{\sqrt{2\pi}} e^{-(x-5)^2/2} \tag{14}$$

and a normal probability distribution:

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-(x-5)^2/2} \tag{15}$$

with mean of 5 and standard deviation of 1. The random numbers this time were computed using numpy.random.normal to perform the transformation method for us.

The histograms in figures 6 and 7 show the spread of the values of integral (13) computed using both mean value and importance sampling methods.

From figure 6 we can see that all of the values of the integral (13) computed using the regular mean value method fall between 0.96 and 1.06, whereas the values computed with importance sampling fall between the much smaller range of 0.98 and 1.02. Thus, the importance sampling method evaluates the integral (13) with greater precision. This is expected since the mean value method uses points uniformly distributed on the interval [0,10], but since the function we are integrating over is sharply peaked at x=5, this method uses a lot of points that don't contribute to the expectation value. However, the importance sampling method allows us to more specifically determine how the random points are generated, so we chose a probability density that returns more random points near x=5 thereby increasing the precision of the sampling method.

Code

Q1

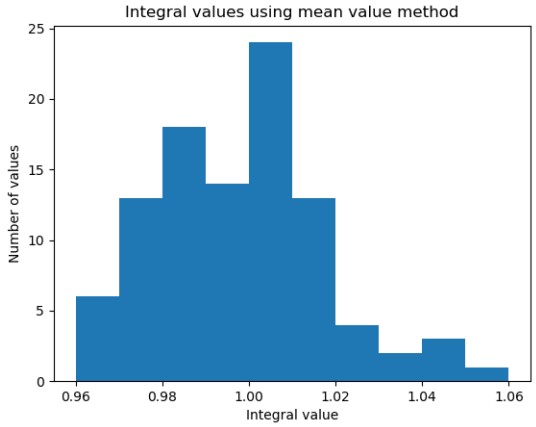

Figure 6: Values of the integral (13) computed using the regular mean value method.

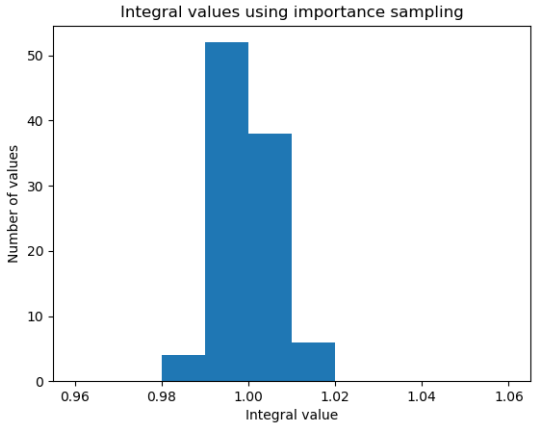

Figure 7: Values of the integral (13) computed using the importance sampling method.

Q2

```
#from numpy.random import random
import numpy as np
import math

dim=10 #n
N=1000000

#define the functon that returns 1 if the points are inside the hypersphere, 0 otherwise
def f(r):
    r_squared = 0
    for i in range(len(r)):
        r_squared += r[i]**2
    if r_squared > 1:
        return 0
    else:
        return 1

#compute the volume using the equation in the handout for comparison
V=(1**(dim)*np.pi**(dim/2))/((dim/2)*math.factorial((dim/2)-1))

#%%
#mean value method
f_avg=0
f_stored=np.zeros(N)
for i in range(N):
    r=np.random.uniform(-1,1, size=(dim,1)) #choses 10 random values between -1 and 1
    for j in range(dim):
        f_stored[i]= f(r) #stores the value of f(r) for later
        f_avg = 2**(dim)*np.sum(f_stored)/N  #computes I
print(f_avg)

#%%
#compute the error
var=(np.sum(f_stored**2))/N-((np.sum(f_stored))/N)**2
```

```
sigma=2**(dim)*np.sqrt(var/N)
```

Q3a

```
import numpy as np
import matplotlib.pyplot as plt

N=10000 #sampling points

#define integrand
def f(x):
    return x**(-1/2)/(1+np.exp(x))

#bounds of integration
a=0
b=1

repeat=100

#mean val method
f_avg=np.zeros(repeat)
f_stored=np.zeros(N)
f_avg_stored=np.zeros(repeat)
for j in range(repeat):
    for i in range(N):
        x=np.random.uniform(a,b) #random points between a and b
        f_stored[i]= f(x)
        f_avg = (b-a)*np.sum(f_stored)/N
        f_avg_stored[j]=f_avg
print(f_avg_stored)


#%%
#importance sampling

#define function for transformation method of sampling
def rand(z):
    return z**2

#define weighting function
def w(x):
    return x**(-1/2)

#define function g(x)=f(x)/w(x) for convenience
def g(x):
    return 1/(1+np.exp(x))

#empty arrays to store values in
g_stored=np.zeros(N)
g_avg_stored=np.zeros(repeat)
for j in range(repeat):
    for i in range(N):
        x=np.random.uniform(a,b) #generate random value
        xi=rand(x) #plug random value into rand to generate nonuniform random value
```

```
        g_stored[i]=g(xi) #compute g(x) and store for later
        g_avg=2*np.sum(g_stored)/N #factor of 2 comes from integral of w(x)
        g_avg_stored[j]=g_avg
print(g_avg_stored)

#%%
#make histograms

plt.figure(1)
plt.title("Integral values using importance sampling")
plt.xlabel("Integral value")
plt.ylabel("Number of values")
plt.hist(g_avg_stored, 10, range=[0.8, 0.88])
plt.show()

plt.figure(2)
plt.title("Integral values using mean value method")
plt.hist(f_avg_stored, 10, range=[0.8, 0.88])
plt.xlabel("Integral value")
plt.ylabel("Number of values")
plt.show()
```

Q3b

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import special

N=10000 #sampling points

#define integrand
def f(x):
    return np.exp(-2*np.abs(x-5))

#bounds of integration
a=0
b=10

repeat=100

#mean val method
f_avg=np.zeros(repeat)
f_stored=np.zeros(N)
f_avg_stored=np.zeros(repeat)
for j in range(repeat):
    for i in range(N):
        x=np.random.uniform(a,b) #random points between a and b
        f_stored[i]= f(x)
        f_avg = (b-a)*np.sum(f_stored)/N
        f_avg_stored[j]=f_avg
print(f_avg_stored)


#%%
```

```
#importance sampling method
#define weighting function
def w(x):
    return 1/(np.sqrt(2*np.pi))*np.exp((-(x-5)**2)/2)

#define mu and sigma for use in the erf
mu=5
sigma=1

#define empty arrays to store values in
fw_stored=np.zeros(N)
fw_avg_stored=np.zeros(repeat)
for j in range(repeat): #loop over 100 times
    for i in range(N): #loop over 10000 times
        xi=np.random.normal(mu,sigma) #compute nonuniformly random points
        fw_stored[i]=f(xi)/w(xi)
        fw_avg=special.erf(5/np.sqrt(2))*np.sum(fw_stored)/N #compute I
        fw_avg_stored[j]=fw_avg #store for later
print(fw_avg_stored)

#%%
#make histograms

plt.figure(1)
plt.title("Integral values using importance sampling")
plt.xlabel("Integral value")
plt.ylabel("Number of values")
plt.hist(fw_avg_stored, 10, range=[0.96,1.06])
plt.show()

plt.figure(2)
plt.title("Integral values using mean value method")
plt.hist(f_avg_stored, 10, range=[0.96,1.06])
plt.xlabel("Integral value")
plt.ylabel("Number of values")
plt.show()
```