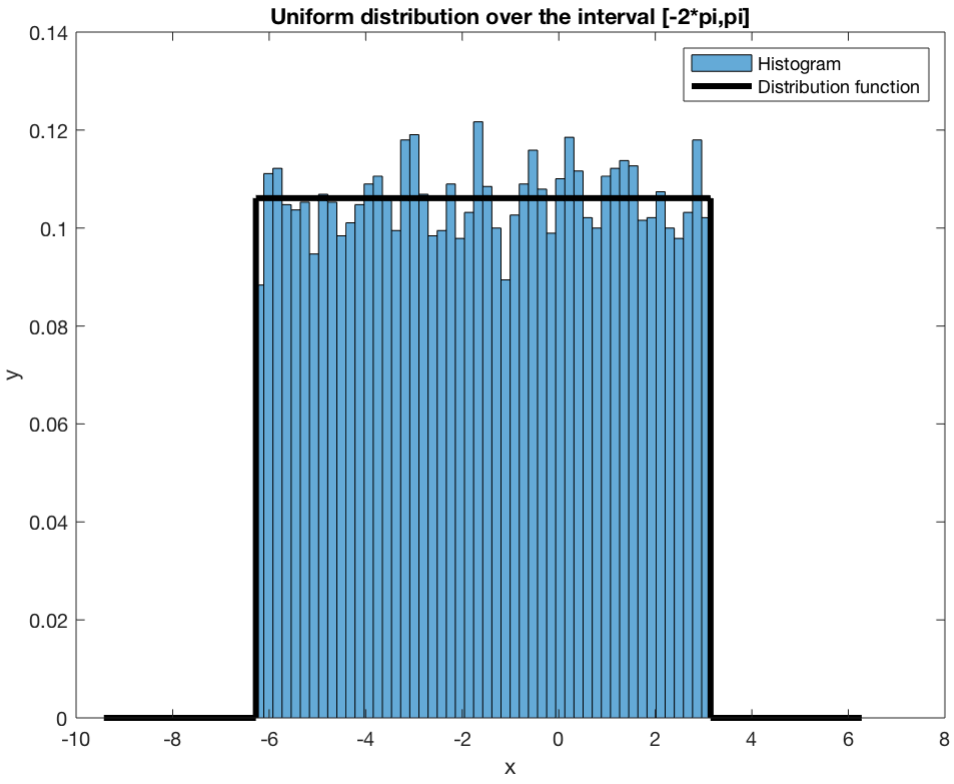
ASSIGNMENT

1.(a) Uniform distribution over the interval $[-2\pi, \pi]$

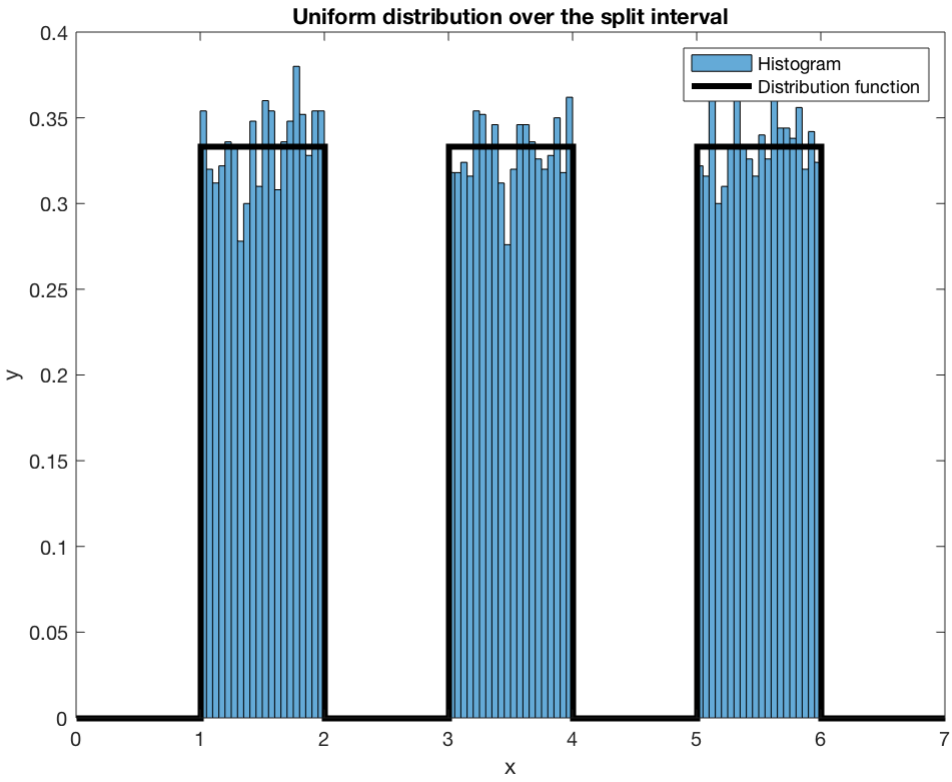
```
% Call the function
t = uniform(-2*pi,pi,10000);
%Plot the histogram and normalize values
histogram(t,50,'Normalization',"pdf")

hold on

%Plot the pdf
x = -3*pi:0.01:2*pi;
pd = makedist('Uniform','lower',-2*pi,'upper',pi);
f = pdf(pd,x);
stairs(x,f,'k',"LineWidth",3)

title('Uniform distribution over the interval [-2*pi,pi]')
xlabel('x')
ylabel('y')
legend('Histogram','Distribution function')

hold off
```



(b) Uniform distribution over the union of the three intervals [1, 2] ∪ [3, 4] ∪ [5, 6]

```
clear
clc
```

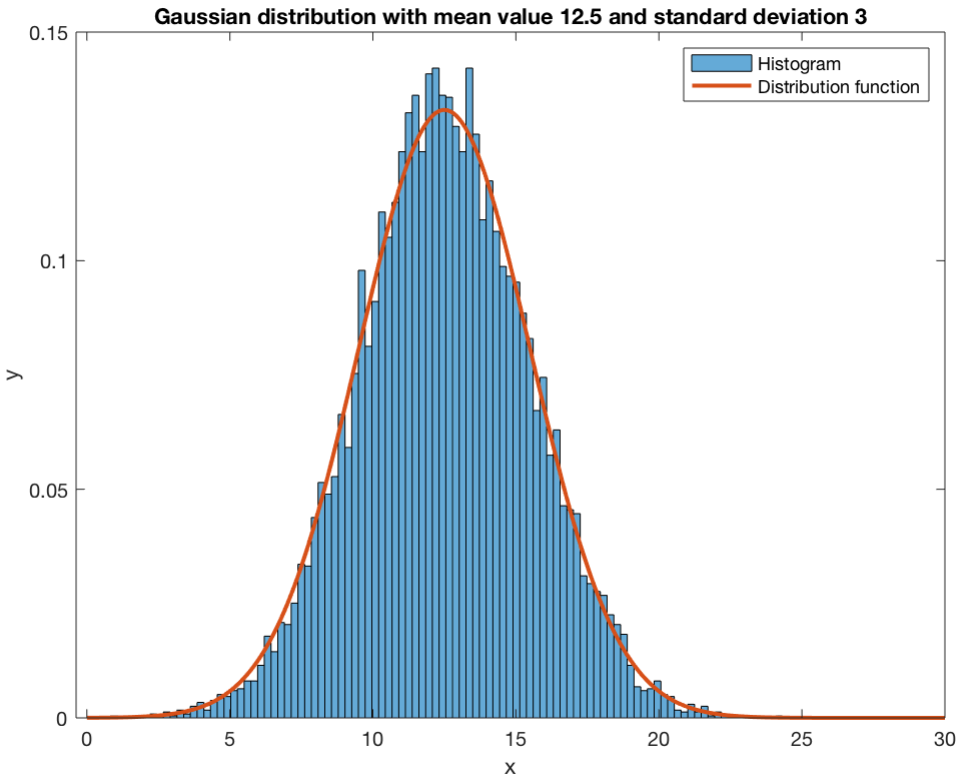
```
%Call the function
t = splituniform(1,2,3,4,5,6,10000);
%Plot the histogram and normalize values
histogram(t,100,'Normalization',"pdf")

hold on

%Plot the pdf
y = [0 1/3 0 1/3 0 1/3 0 0];
stairs(0:7, y, 'k',"LineWidth",3)

title('Uniform distribution over the split interval')
xlabel('x')
ylabel('y')
legend('Histogram','Distribution function')

hold off
```



(c) Gaussian distribution with a given mean value 12.5 and standard deviation 3 using the Box-Muller Method

```
clear
clc

%Call the function
t = normal(12.5,3,10000);
%Plot the histogram and normalize values
histogram(t,100,"Normalization","pdf")
```

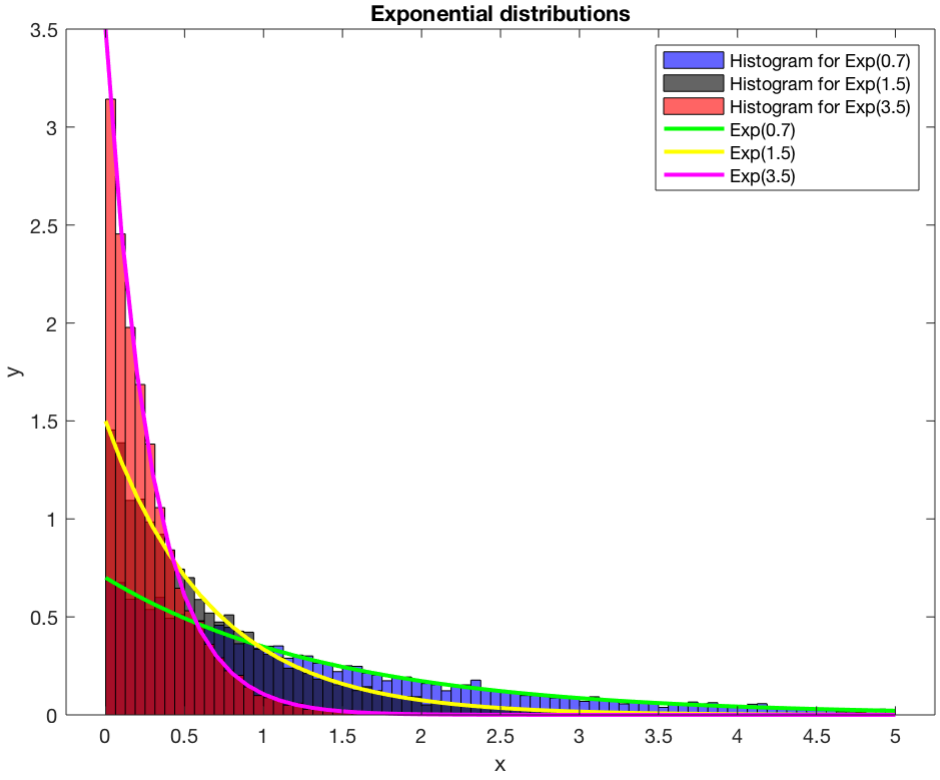
```
hold on

%Plot the pdf
x = 0:0.1:30;
mean = 12.5;
std = 3;
f = exp(-(x-mean).^2./(2*std^2))./(std*sqrt(2*pi));
plot(x,f,'linewidth',2)

title('Gaussian distribution with mean value 12.5 and standard deviation 3')
xlabel('x')
ylabel('y')
legend('Histogram','Distribution function')

hold off
```



(d) Continuous distribution with probability density function $f(x, \lambda) = \lambda e^{-\lambda x}$

```
clear
clc

%Call the function
t1 = exponential(0.7,10000);
%Plot the histograms and normalize values
histogram(t1,80,"Normalization","pdf","FaceColor",'b',"BinLimits",[0,5])

hold on
```

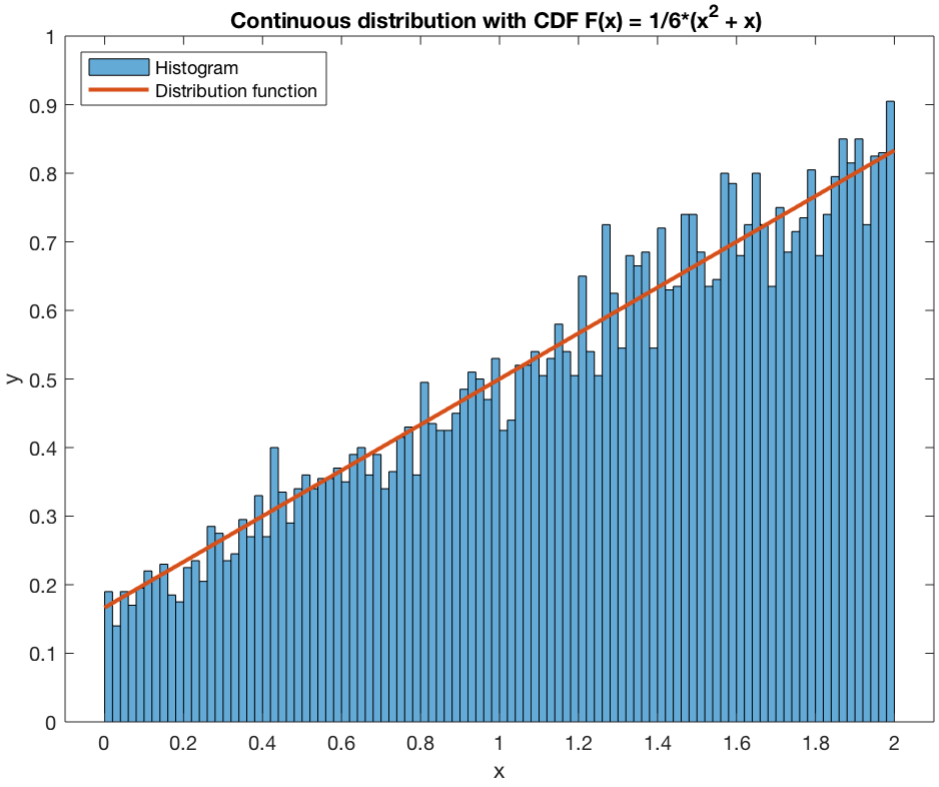
```
%Do the same for the other distributions
t2 = exponential(1.5,10000);
histogram(t2,80,"Normalization","pdf","FaceColor",'k',"BinLimits",[0,5])
t3 = exponential(3.5,10000);
histogram(t3,80,"Normalization","pdf","FaceColor",'r',"BinLimits",[0,5])

%Plot the pdfs
x = 0:0.1:5;
l1 = 0.7;
l2 = 1.5;
l3 = 3.5;
f1 = l1*exp(-l1*x);
f2 = l2*exp(-l2*x);
f3 = l3*exp(-l3*x);
plot(x,f1,'g',x,f2,'y',x,f3,'m',"LineWidth",2)

title('Exponential distributions')
xlabel('x')
ylabel('y')
legend('Histogram for Exp(0.7) ','Histogram for Exp(1.5)', 'Histogram for Exp(3.5)', 'E


hold off
```



(e)

```
clear
```

```
clc

%Call the function
t = cdf(10000);
%Plot the histogram and normalize values
histogram(t,100,"Normalization","pdf")

hold on

%Plot the pdf
x = 0:0.01:2;
f = x/3 + 1/6;
plot(x,f,"LineWidth",2)

title('Continuous distribution with CDF F(x) = 1/6*(x^2 + x) ')
xlabel('x')
ylabel('y')
legend('Histogram','Distribution function','Location',"northwest")

hold off
```



FUNCTIONS

```
%Uniform distribution over [a,b] with N randomly obtain samples
```

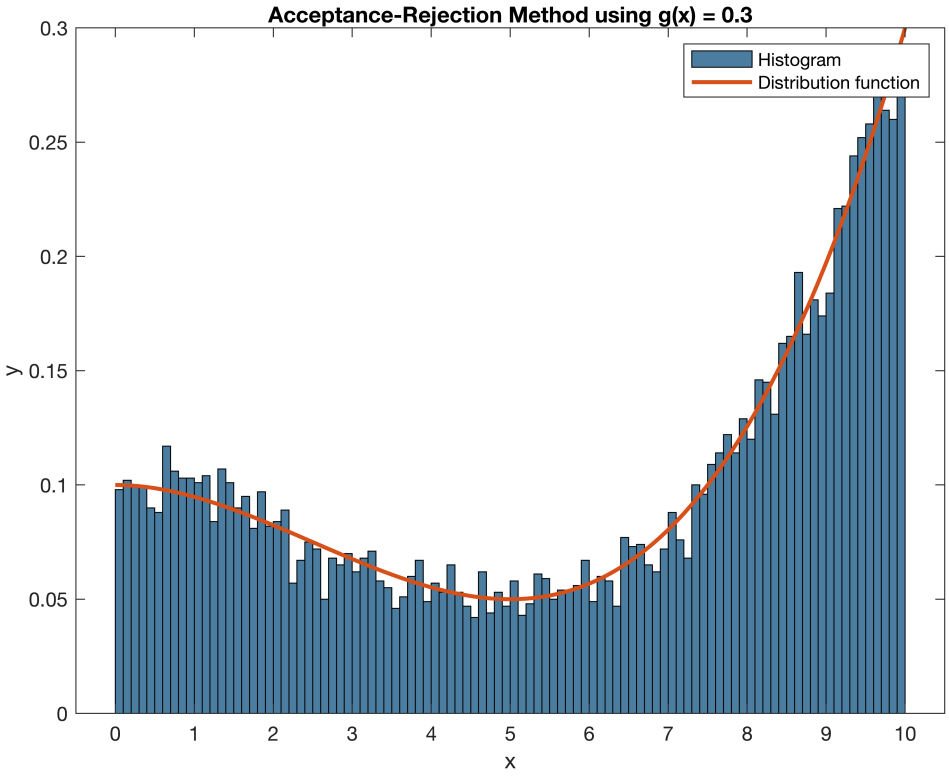
```
function y = uniform(a,b,N)

    y = (b-a)*rand(N,1) + a;

end

% Uniform distribution over the union of [a,b],[c,d] and [e,f] with N N randomly obtain
function y = splituniform(a,b,c,d,e,f,N)

    x = rand(N,1);

    %Split x into 3 parts
    I1 = x <= 1/3;
    I2 = x>=1/3 & x<=2/3;
    I3 = x>=2/3;

    % Values have to be adjust according to which section they belong to
    y = 3*(b-a)*(x.*I1) + a*I1 + 3*(d-c)*(x.*I2 - 1/3*I2) + c*I2 +3*(f-e)*(x.*I3 - 2/3*

end

% Gaussion distribution with given mean and standard deviation with N
% randomly chosen samples (via Box-Muller Method)
function y = normal(mean,std,N)

    x1 = rand(N,1);
    x2 = rand(N,1);
    %Sample r from the Rayleigh Distribution via Inverse Function Method
    r = sqrt(-2*log(1-x1));
    %Sample theta Uniform Distribution over [0,2*pi]
    theta = 2*pi*x2;
    %Standard Normal
    standardnormal = r.*cos(theta);
    %This step will give us our required distribution
    y = std*standardnormal + mean;

end

%Exponential Distribution with parameter l with N randomly chosen samples
function y = exponential(l,N)

    x = rand(N,1);
    y = -log(1-x)/l;

end

%Continuous Distribution with the given CDF
function y = cdf(N)

     x = rand(N,1);
     y = (-1 + sqrt(1+24*x))/2;

end
```

Question 2

(a),(b) Random samples distributed according to $f(x)$ using the acceptance-rejection method with $g(x) = 0.3\ \forall x \in [0, 10]$ where

$$f(x) = \frac{1}{1000}\left(100 + \frac{4}{5}x^3 - 6x^2\right)$$

```
clear
clc

%Call function, save values and plot histogram
[samples, accepted, total] = acceptedsamples(10000);
histogram(samples,100,"Normalization","pdf")

hold on

%Plot pdf
t = 0:0.1:10;
F = (100+4/5*power(t,3)-6*power(t,2))/1000;
plot(t,F,"linewidth",2)

title('Acceptance-Rejection Method using g(x) = 0.3')
xlabel('x')
ylabel('y')
legend('Histogram','Distribution function')

hold off
```




```
rejectionrate = (1- accepted/total)*100
```

```
rejectionrate =
  66.469976866597378
```

(c), (d) Notice $f(x)$ has local minima in the interval $[0, 10]$. To find the local minima, we can calculate the derivative and set it to zero

$$f'(x) = \frac{1}{1000}\left(\frac{4}{5}.3x^2 - 6.2x\right) = 0$$

$$\Rightarrow \frac{12}{5.1000}x^2 - 12x = 0$$

$$\Rightarrow x^2 - 5x = 0$$

$$\Rightarrow x = 0, 5$$

We need to find a function of the form

$$g(x) = a + b(x-5)^2$$

with the appropriate variable *a* and *b* such the $g(x) \leq f(x)\ \forall\ x \in [0, 10]$.

Notice since g is quadratic polynomial, it has a global minima and it's found at $x = 5$. To find the optimal values of a and b for the acceptance-rejection method, we need to minimize the area between f and g. We can do this by

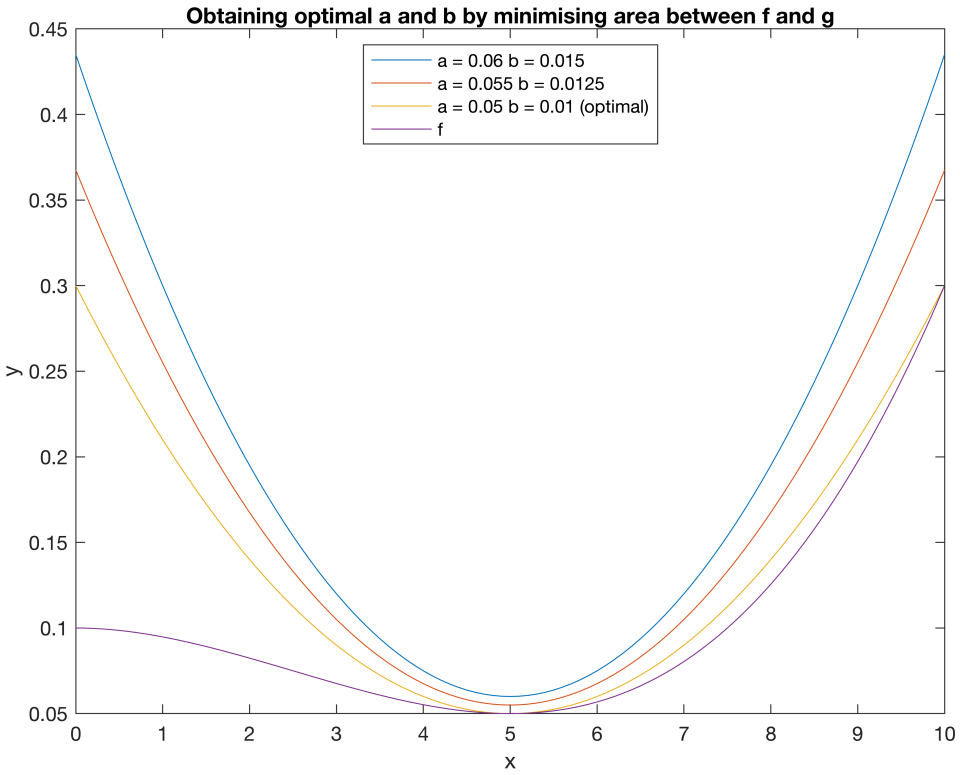
1. Setting $f(5) = g(5)$ to find a. This will make $g(x)$ and $f(x)$ coincide at $x = 5$
2. Setting $f(10) = g(10)$ to find *b* so that we can stretch *g* just enough

By doing this, we get

$$a = f(5) = 0.05 \text{ and } b = 0.01$$

An illustration of this would be:

```
clear
clc

g1 = @(x) 0.05 + 0.01*power(x-5,2);
g2 = @(x) 0.055 + 0.0125*power(x-5,2);
g3 = @(x) 0.06 + 0.015*power(x-5,2);

f = @(x) 0.001*(100 + 0.8*power(x,3) - 6*power(x,2));

x = 0:0.1:10;

plot(x, g3(x), x, g2(x), x, g1(x), x, f(x))
title('Obtaining optimal a and b by minimising area between f and g')
xlabel('x')
ylabel('y')
legend('a = 0.06 b = 0.015','a = 0.055 b = 0.0125', 'a = 0.05 b = 0.01 (optimal)','f','
```



Clearly $g(x) \leq f(x)$ in $[0, 10]$ $a = 0.05$ and $b = 0.01$. Hence now we can go ahead and execute our algorithm. For the algorithm, we need the following

$$g(x) = a + b(x-5)^2$$

$$\Rightarrow A_g = \int_0^{10} g(x) = 10a + \frac{500}{6}b$$

Hence the required CDF is

$$\text{CDF} = \int_0^x g(t)/A_g \, dt$$

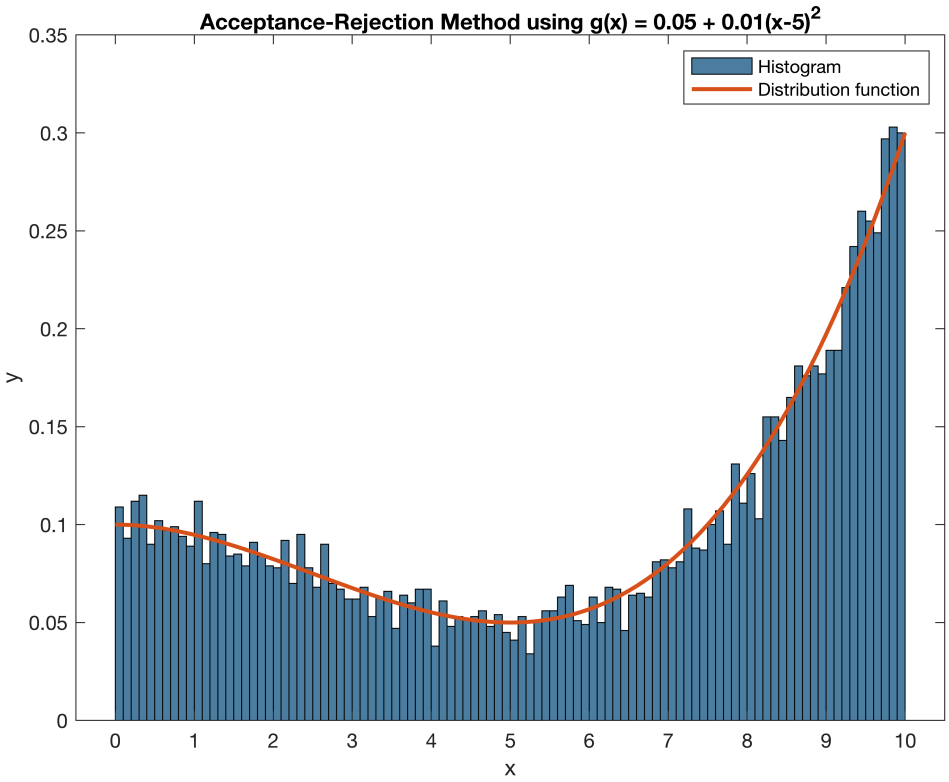
$$\Rightarrow \text{CDF} = \frac{(\text{ax} + \text{bx}^3/2 + 25\text{bx} - 5x^2b)}{10a + \frac{500}{6}b}$$

Using WolframAlpha, we get the inverse of the CDF

$$\text{CDF}^{-1} = -5^{\frac{1}{3}}\left(\sqrt{5}\sqrt{320 * x^2 - 320x + 81} - 40x + 20\right)^{\frac{1}{3}} + \frac{5^{\frac{2}{3}}}{\left(\sqrt{5}\sqrt{320 * x^2 - 320x + 81} - 40x + 20\right)^{\frac{1}{3}}} + 5;$$

Using this, we create the required function

```
[samples, accepted, total] = acceptedsamples2(10000);
histogram(samples,100,"Normalization","pdf")

hold on

%Plot pdf
t = 0:0.1:10;
F = (100+4/5*power(t,3)-6*power(t,2))/1000;
plot(t,F,"linewidth",2)
title('Acceptance-Rejection Method using g(x) = 0.05 + 0.01(x-5)^2')
xlabel('x')
ylabel('y')
legend('Histogram','Distribution function')
```




```
rejectionrate = (1- accepted/total)*100
```

```
rejectionrate =
  23.935199269850926
```

Functions

```
function [samples,accepted,total] = acceptedsamples(N)

    g = 0.3;
    Ag = 3;
    cutoff = g/Ag;

    % Accepted Samples
    accepted = 1;
    %Total Samples
    total = 1;

    while accepted <= N

        % Sample a random variable x from the probability density function g(x)/Ag usin
        xbar = 1/cutoff*rand;
        % Sample a random variable y uniformly in [0, g(x)]
        ybar = g*rand;
        f = (100 + 4/5*power(xbar,3) - 6*power(xbar,2))/1000;

        % If y ≤ f(x) then accept the sample and return x. Otherwise, reject
```

```
        if f >= ybar
            samples(accepted) = xbar;
            accepted=accepted+1;
        end
        total = total + 1;
    end

end

function [samples, accepted, total] = acceptedsamples2(N)

    accepted = 1;
    total= 1;

    while accepted <= N

        % Sample a random variable x from the probability density function g(x)/Ag usin
        x = rand;
        xbar = -5^(1/3)*(sqrt(5)*sqrt(320*x^2 - 320*x + 81) - 40*x + 20)^(1/3) + 5^(2/3
        % Sample a random variable y uniformly in [0, g(x)]
        g = 0.05 + 0.01*power(xbar-5,2);
        ybar = g*rand;
        f = (100 + 4*power(xbar,3)/5 -6*power(xbar,2))/1000;
        % If y ≤ f(x) then accept the sample and return x. Otherwise, reject
        if f >= ybar
            samples(accepted) = xbar;
            accepted=accepted+1;
        end
        total  = total + 1;
    end
end
```

Question 3.

$$I = \int_0^1 x^3(1-x)^2\,dx = \frac{32}{315}$$

(a) Determine a Hit-Miss Monte-Carlo estimate I_U of *I* using $N = 1000$ uniform variates in $[0, 1]$.

```
montecarlo = weird_hitmiss(1000000)
```

```
montecarlo = 0.1017
```

```
% Absolute value of the error = |actual value - estimate value|
absolute_value = abs(32/315 - montecarlo)
```

```
absolute_value = 8.4698e-05
```

The corresponding Variance (or Mean Squared Error) is given by $Var[I_u] = I(A - I)/N$

```
A = 1;
N = 1;

MSE = montecarlo*(A - montecarlo)/N
```

```
MSE = 0.0913
```

(b) Determine a Monte-Carlo estimate of *I* using importance sampling. Choose $N = 1000$ random variables with a probability density function $f(x) = 5x^4, x \in [0, 1]$ as sampling points.

```
f1 = importancesampling1(100000)
```

```
f1 = 0.1017
```

```
% Absolute value of the error = |actual value - estimate value|
absolute_value = abs(32/315 - f1)
```

```
absolute_value = 1.1994e-04
```

```
f2 = importancesampling2(100000)
```

```
f2 = 0.1018
```

```
% Absolute value of the error = |actual value - estimate value|
absolute_value = abs(32/315 - f2)
```

```
absolute_value = 1.6965e-04
```

```
f3 = importancesampling3(100000)
```

```
f3 = 0.1016
```

```
% Absolute value of the error = |actual value - estimate value|
absolute_value = abs(32/315 - f3)
```

```
absolute_value = 3.2172e-05
```

```
f4 = importancesampling4(100000)
```

```
f4 = 0.1019
```

```
% Absolute value of the error = |actual value - estimate value|
absolute_value = abs(32/315 - f4)
```

```
absolute_value = 2.9094e-04
```

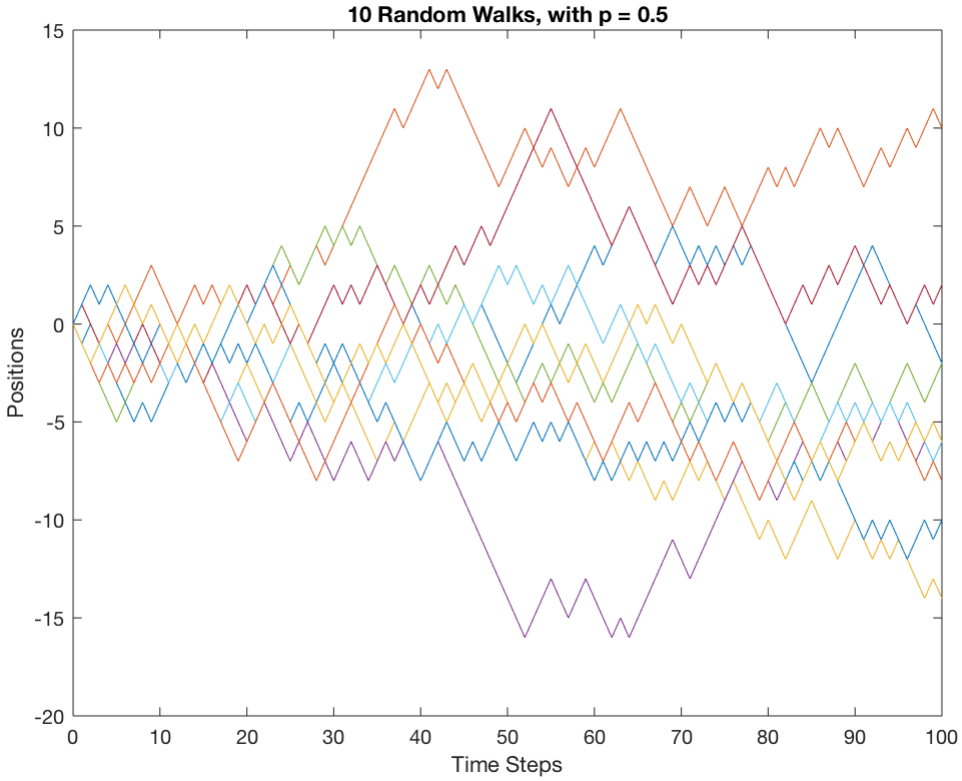
```
function I = weird_hitmiss(N)

    %The area of the sample space is equal to 3
    A = 1;
    %% This is the function of which we want to compute the
    %% integral
    f = @(t) power(t,3)*power(1-t,1/2);
    %% sample N abscissas uniformly in [0,1]
    x = rand(N, 1) * 1;
    %% sample N ordinates uniformly in [0,1]
    y = rand(N,1);

    %% Count only those points whose y is < f(x)
    k = 0;
    for i=1:N
    if y(i) < f(x(i))
        k = k +1;
    end
    end
    %% Get the "Hit-miss" Monte Carlo estimate:
    I=A*k/N;
end

function In = importancesampling1(N)

    x = rand(1,N);
    % Given Integral
    I = @(t)(t.^3).*((1-t).^(1/2));
    % Inverse of the cumulative distribution function
    C = @(t)(t.^(1/5));
    % Inverse sampling for f
    Z = C(x);
    % pdf
    p = @(t)5.*(t.^4);
    In = mean(I(Z)./p(Z));

end

function In = importancesampling2(N)

    x = rand(1,N);
    % Given Integral
    I = @(t)(t.^3).*((1-t).^(1/2));
```

```
    % Inverse of the cumulative distribution function
    C = @(t)(t.^(1/4));
    % Inverse sampling for f
    Z = C(x);
    % pdf
    p = @(t)4.*(t.^3);
    In = mean(I(Z)./p(Z));

end

function In = importancesampling3(N)

    x = rand(1,N);
    % Given Integral
    I = @(t)(t.^3).*((1-t).^(1/2));
    % Inverse of the cumulative distribution function
    C = @(t)(t.^(1/3));
    % Inverse sampling for f
    Z = C(x);
    % pdf
    p = @(t)3.*(t.^2);
    In = mean(I(Z)./p(Z));

end


function In = importancesampling4(N)

    x = rand(1,N);
    % Given Integral
    I = @(t)(t.^3).*((1-t).^(1/2));
    % Inverse of the cumulative distribution function
    C = @(t)(t.^(1/2));
    % Inverse sampling for f
    Z = C(x);
    % pdf
    p = @(t)2.*(t);
    In = mean(I(Z)./p(Z));

end
```

Question 4.

(a)(i) Plot 10 random paths up to $N = 100$ time-steps $(i = 0, 1, \ldots, N)$ using $X_0 = 0$.

```
% Since we can't have indices equal to zero, we will shift our indices one
% step to the right. So X0 = x(1)

% Create an empty matrix
rw = zeros(10,101);

% Replace rows with walks each walk
for i = 1:10
    rw(i,:) = randomwalk(0.5,101);
end

%Now plot the walks
plot(0:1:100,rw(1,:))

hold on

for i = 2:10
    plot(0:1:100,rw(i,:))
end

title('10 Random Walks, with p = 0.5')
xlabel('Time Steps')
ylabel('Positions')
% No point in putting legend

hold off
```



(ii) Construct a histogram of the position reached at time $i = 5$ of 100 sample paths. The probability distribution function is given by

$$P(X = x) = \begin{bmatrix} N-1 \\ \frac{x+N-1}{2} \end{bmatrix} p^{\frac{x+N-1}{2}} (1-p)^{\frac{N-1-x}{2}}$$

Here, x is position, N is the time step (hence, N-1 steps have been made), and p is the probability of going up along y axis.

Let u be the number of steps of taken in the positive direction of y-axis, and let *d* be the number of steps taken in negative direction. Consider the following:

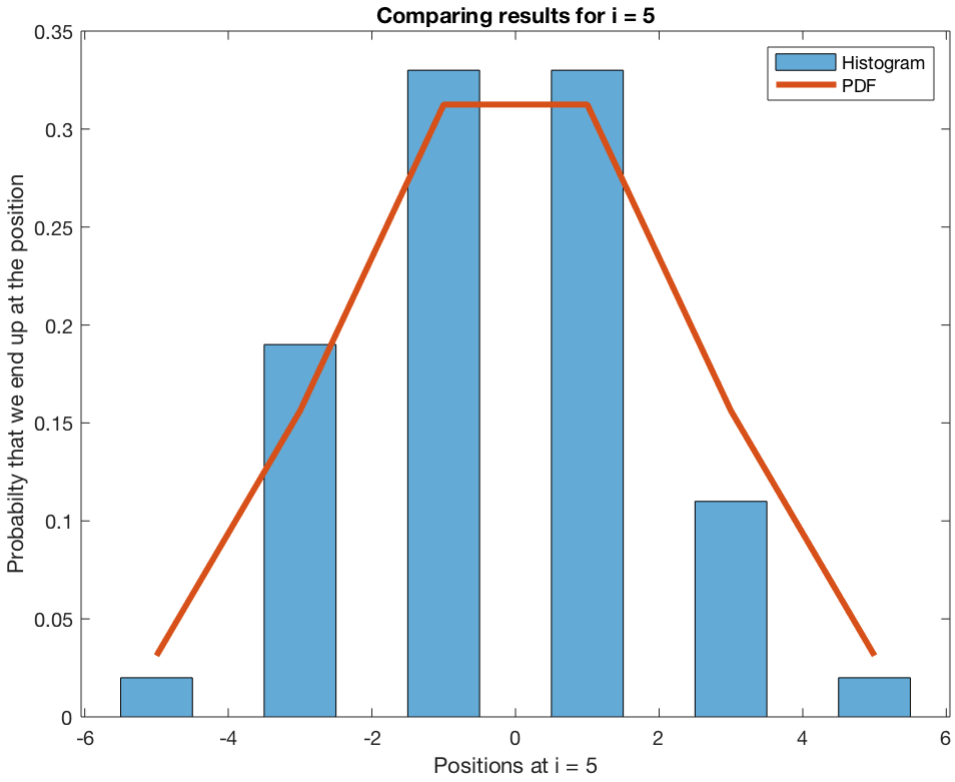
$$x = u - d\,, N - 1 = u + d$$

$$\Rightarrow 2u - N + 1 = x$$

u can take values $0 \text{ to } N-1$. Every position is uniquely determined by the number steps taken in the positive direction (and also in the negative direction, since one determines the other). Therefore, for a given *N*, *x* can take values $-N+1 \text{ to } N-1 \ (= 2(N-1) - N + 1)$.

```
%Record the walks
for i = 1:100
    rw(i,:) = randomwalk(0.5,101);
end
```

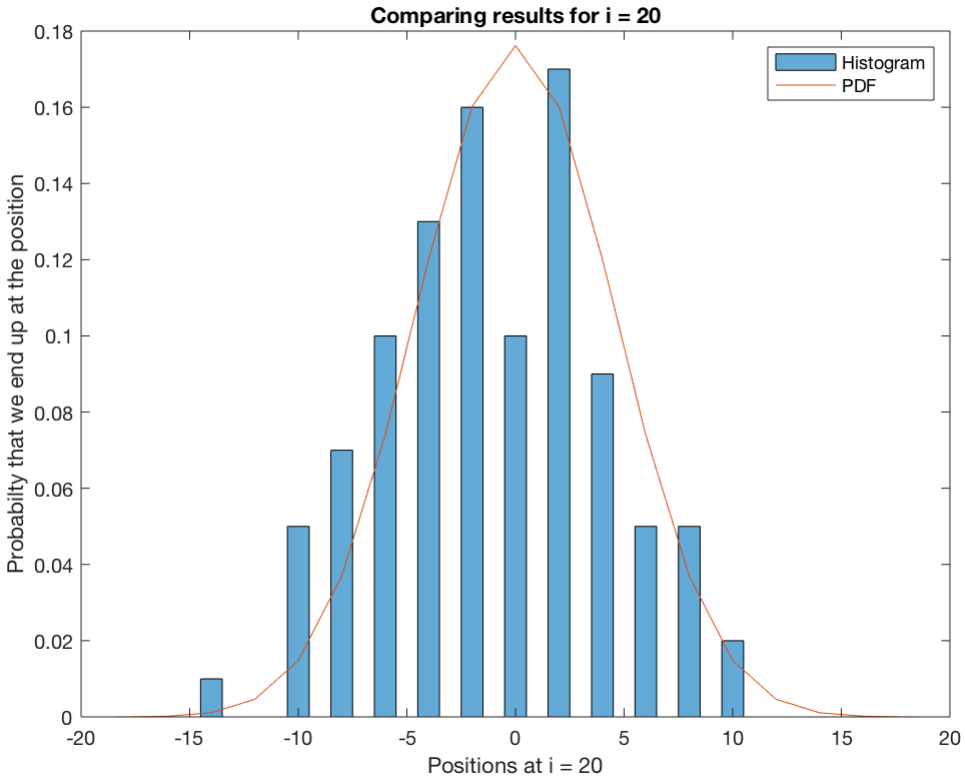
```
% X5 = x(6)
positionsat5 = rw(:,6);
%Plot histogram and normalize values
histogram(positionsat5,'Normalization',"pdf")

hold on

%Plot the pdf
bin5 = @(x) nchoosek(5,(x+5)/2)*power(1/2,5);
for i = 1:6
    bernoulli5(i) = bin5(2*(i-1) - 5 );
end
plot(-5:2:5,bernoulli5,'linewidth',3)

title('Comparing results for i = 5')
xlabel('Positions at i = 5')
ylabel('Probabilty that we end up at the position')
legend('Histogram','PDF')

hold off
```



(iii) Repeat for $i = 20$

```
% X20 = x(21)
positionsat20 = rw(:,21);
%Plot histogram and normalize values
histogram(positionsat20,'Normalization','pdf')
```

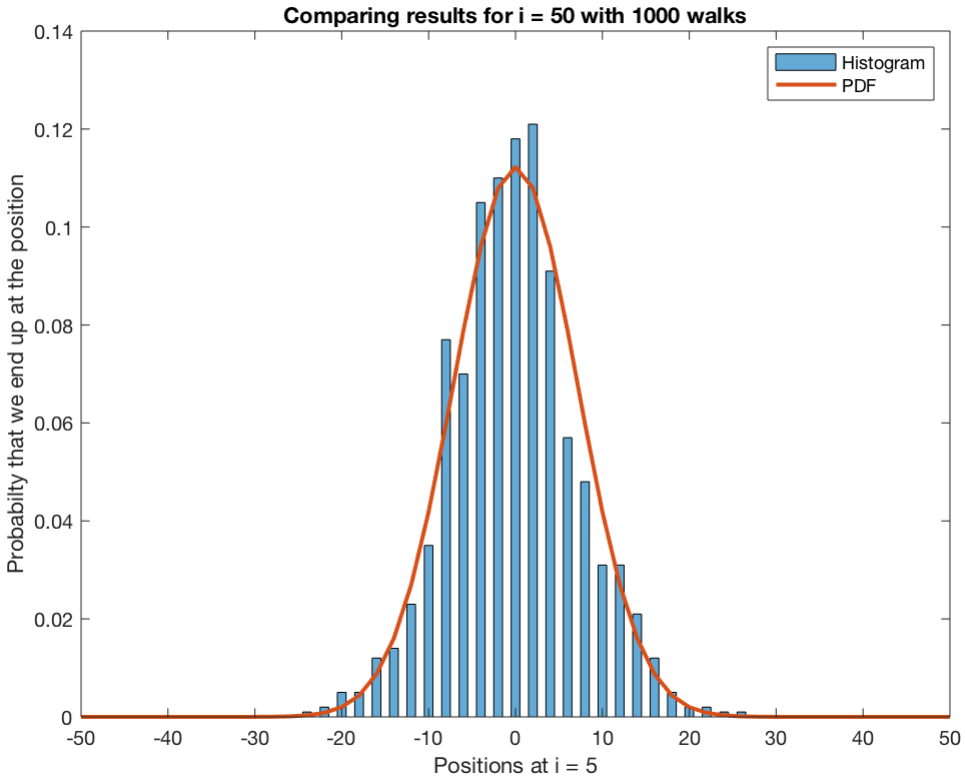
```
hold on

%Plot the pdf
bin20 =@(x) nchoosek(20,(x+20)/2)*power(1/2,20);

for i = 1:21
    bernoulli20(i) = bin20(2*(i-1) - 20);
end

plot(-20:2:20,bernoulli20)

title('Comparing results for i = 20')
xlabel('Positions at i = 20')
ylabel('Probabilty that we end up at the position')
legend('Histogram','PDF')

hold off
```



(iv) Record 1000 random walks. Choose random time step, let's say $i = 50$, plot the histogram and compare results to the pdf.

```
for i = 1:1000
    rw(i,:) = randomwalk(0.5,101);
end

finalpositions = rw(:,51);
%Plot histogram and normalize values
histogram(finalpositions,'Normalization','pdf')
```

```
hold on

bin100 = @(x) nchoosek(50,(x+50)/2)*power(0.5,50);

for i = 1:51;
    bernoulli100(i) = bin100(2*(i-1) - 50);
end

plot(-50:2:50,bernoulli100,"linewidth",2)

title('Comparing results for i = 50 with 1000 walks')
xlabel('Positions at i = 5')
ylabel('Probabilty that we end up at the position')
legend('Histogram','PDF')

hold off
```

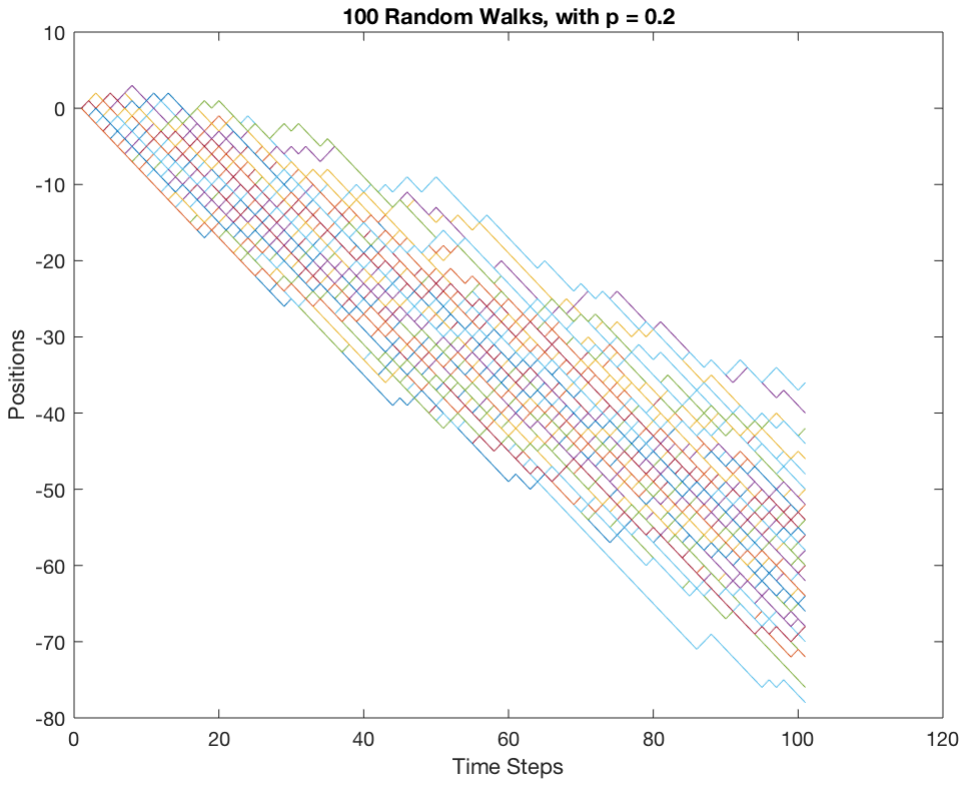


It is easy to see that when we increase *N*, our recordings are more compatible with the probability density function.

(b) (i) Plot 100 random paths up to N = 100 time-steps $(i = 0, 1, \ldots, N)$ using $X_0 = 0$ and $p = 0.2$

```
clear
clc

rw = zeros(100,101);
```

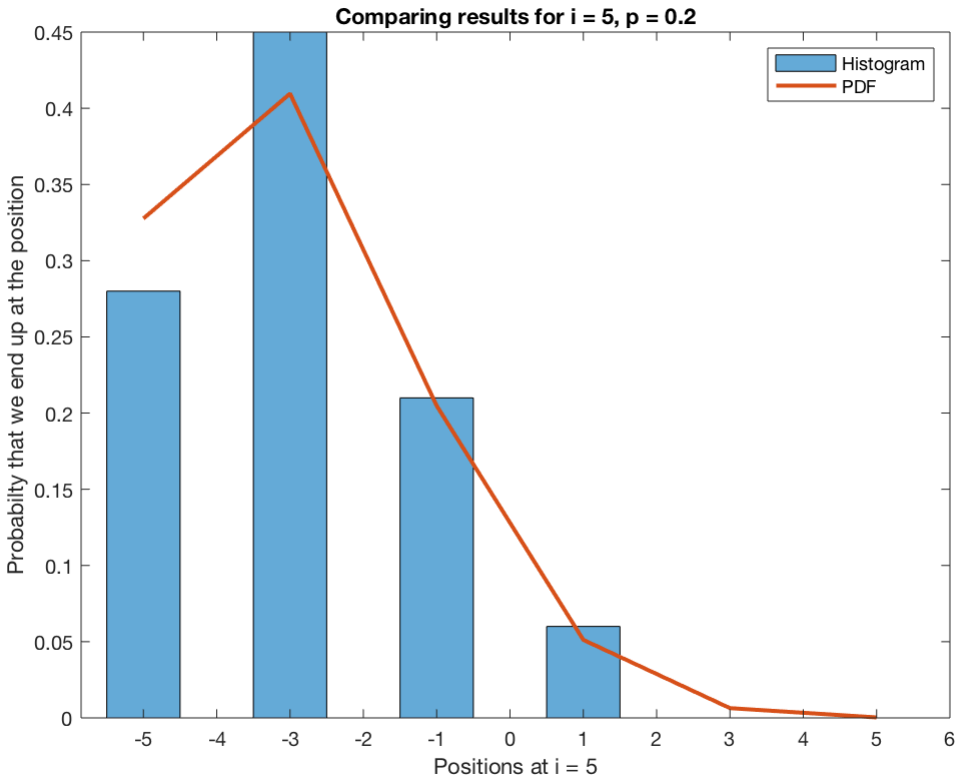
```
%Record the random walks
for i = 1:100
    rw(i,:) = randomwalk(0.2,101);
end

% Plot the random walkswalks

plot(rw(1,:))

hold on

for i = 2:100
    plot(rw(i,:))
end

title('100 Random Walks, with p = 0.2')
xlabel('Time Steps')
ylabel('Positions')
% No point in putting legend

hold off
```



(ii) Construct a histogram of the position reached at time $i = 5$ of 100 sample paths.

```
% X5 = x(6)
newpositionat5 = rw(:,6);

%Plot histogram and normalize values
```

```
histogram(newpositionat5,'Normalization','pdf')

hold on


% Plot pdf to compare
newbin5 = @(x) nchoosek(5,(x+5)/2)*power(0.2,(x+5)/2)*power(0.8,(5-x)/2);

for i = 1:6;
    newbernoulli5(i) = newbin5(2*(i-1) - 5);
end

plot(-5:2:5,newbernoulli5,'linewidth',2)

title('Comparing results for i = 5, p = 0.2')
xlabel('Positions at i = 5')
ylabel('Probabilty that we end up at the position')
legend('Histogram','PDF')
hold off
```



FUNCTION FOR RANDOM WALK

```
function x = randomwalk(p, N)
    x(1) = 0;
    for i = 2:N
        u = rand;
        if u < p
```

```
            x(i) = x(i-1) + 1;
        else
            x(i) = x(i-1) - 1;
        end
    end
end
```

Question5

(a) Write a function $\mathrm{recursiveSAW}(N)$ to recursively enumerate all self-avoiding walksof length $1..N$.Using this function, reproduce the numbers in the sequence given above for lengths up to 10.

The algorithm used in this implementation is taken from [1].

```
clear
clc

global total
global lattice

format long

tic
selfavoidingwalksoflength(1)
```

```
ans =
     4
```

```
selfavoidingwalksoflength(2)
```

```
ans =
    12
```

```
selfavoidingwalksoflength(3)
```

```
ans =
    36
```

```
selfavoidingwalksoflength(4)
```

```
ans =
   100
```

```
selfavoidingwalksoflength(5)
```

```
ans =
   284
```

```
selfavoidingwalksoflength(6)
```

```
ans =
   780
```

```
selfavoidingwalksoflength(7)
```

```
ans =
        2172
```

```
selfavoidingwalksoflength(8)
```

```
ans =
        5916
```

```
selfavoidingwalksoflength(9)
```

```
ans =
       16268
```

```
selfavoidingwalksoflength(10)
```

```
ans =
       44100
```

```
toc
```

```
Elapsed time is 1.016704 seconds.
```

```
tic
selfavoidingwalksoflength(17)
```

```
ans =
    46466676
```

```
toc
```

```
Elapsed time is 183.350565 seconds.
```

FUNCTIONS

Since we are using two functions, they need to have a sort of language which they can use to "speak" to each other. For this reasons, we make the variables 'total' and 'lattice' global variables.

```
function recursivewalk(x,y,j)

    % x and y are the coordinates, and j is the number of steps left in
    % the walk

    % 'Total' keeps a track of the enumerated walks . It is initiated in
    % the second function 'selfavoidingwalksoflength(n)' at 0
    global total
    % 'Lattice' is the matrix where we perform the walk. It is also
    % initiated in the second function with all of its values equal to 0.
    % As we move around in the lattice, we label the occupied spaces 1.
    global lattice

    %When j = 0, there are no more steps left, which means we have found a
    %new walk. Hence, we add 1 to the total
    if j == 0
        total = total + 1;

    % j acts as a checker here. When j isn't equal to 0, we check which neighbors of (x
    %unoccupied (not equal to one), move along those directions and recall
    %this function. This is the recursive part of the algorithm.
    else
        if lattice(x+1,y) == 0
            lattice(x+1,y) = 1;
            recursivewalk(x+1,y,j-1);
        end
        if lattice(x-1,y) == 0
```

```
            lattice(x-1,y) = 1;
            recursivewalk(x-1,y,j-1);
        end
        if lattice(x,y+1) == 0
            lattice(x,y+1) = 1;
            recursivewalk(x,y+1,j-1);
        end
        if lattice(x,y-1) == 0
            lattice(x,y-1) = 1;
            recursivewalk(x,y-1,j-1);
        end
    end
    %This is where reset the lattice. Otherwise the new walks will use
    %the memory of the previous walks
    lattice(x,y) = 0;

end

function x = selfavoidingwalksoflength(n)

    global total
    global lattice

    %Initiate total
    total = 0;
    %Initiate lattice
    lattice = zeros(2*n+1,2*n+1);

    % Start at origin
    lattice(n,n) = 1;


    % This is the part where we save computing time by classifying walks.
    % Note apart from the straight walks (of which there are four, which is
    % why we add add 4 to the total), every walk is rotation/reflection of
    % 7 other walks. This is why we only describe one of them and multiply
    % the total by 8.
    for h = 1:n-1
        lattice(n,n+h) = 1;
        lattice(n+1,n+h) = 1;
        recursivewalk(n+1,n+h,n-h-1)
    end

    % Total number of walks
    x = 8*total + 4;
end
```

Question5

(b) Write a function $\mathrm{simplesamplingSAW}(N)$ to use simple sampling to estimate the number of self-avoiding walks of length $1..N$. Test your function by comparing its output against the exact values for lengths up to 10, and give an estimate of the accuracy of your result.

The algorithm used in this implementation is taken from [2].

```
clear
clc

%Actual values
N = [ 4  12 36 100 284 780 2172 5916 16268 44100 ];
```

```
%Store Value for max samples = 100000
tic
s(1) = simplesampling(1,100000);
%Repeat for other values of N
s(2) = simplesampling(2,100000);
s(3) = simplesampling(3,100000);
s(4) = simplesampling(4,100000);
s(5) = simplesampling(5,100000);
s(6) = simplesampling(6,100000);
s(7) = simplesampling(7,100000);
s(8) = simplesampling(8,100000);
s(9) = simplesampling(9,100000);
s(10) = simplesampling(10,100000);
toc
```

```
Elapsed time is 59.758439 seconds.
```

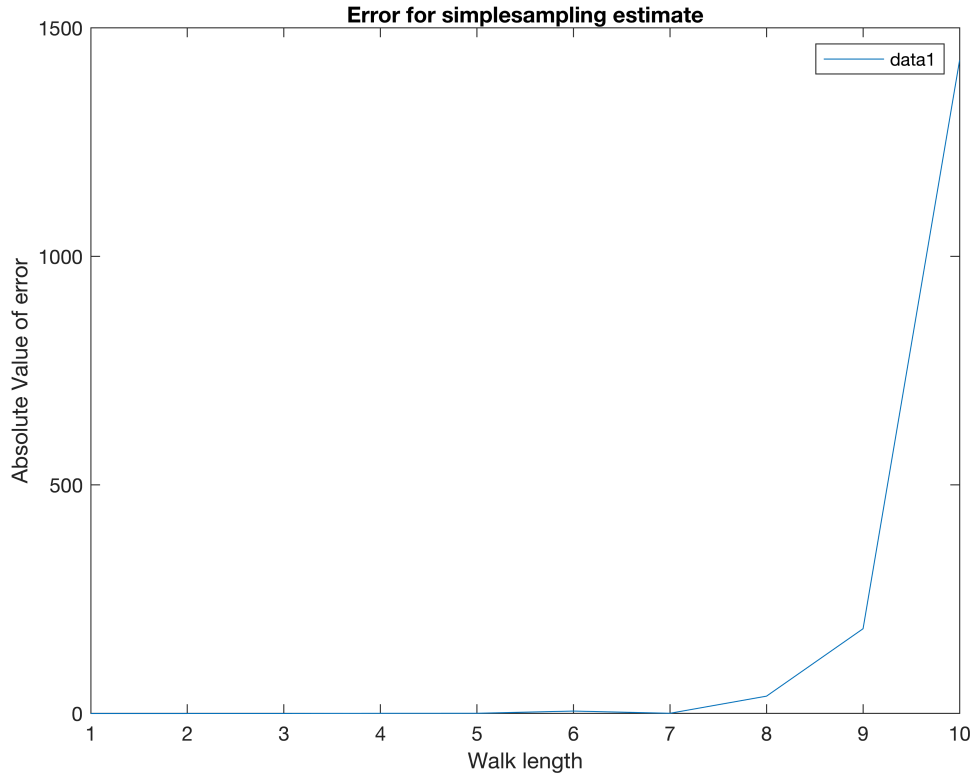
```
%Calculate error
for i = 1:10
    error(i) = abs(N(i)-s(i));
end
% Compare values
T = table(transpose(N),transpose(s),transpose(error));
T.Properties.VariableNames{1} = 'Actual Values';
T.Properties.VariableNames{2} = 'Simple Sampling Estimate';
T.Properties.VariableNames{3} = 'Simple Sampling Error';
T
```

```
T = 10×3 table
```

	Actual Values	Simple Sampling Estimate	Simple Sampling Error
1	4	4	0
2	12	11.989120000000000	0.010880000000000
3	36	36.047359999999998	0.047359999999998
4	100	99.617279999999994	0.382720000000006
5	284	2.844876800000000e+02	0.487680000000012

	Actual Values	Simple Sampling Estimate	Simple Sampling Error
6	780	7.852851200000000e+02	5.285120000000006
7	2172	2.171535360000000e+03	0.464640000000145
8	5916	5.953945600000000e+03	37.945600000000013
9	16268	1.608253440000000e+04	1.854655999999995e+02
10	44100	4.552916992000000e+04	1.429169920000000e+03

```
plot(error)
title('Error for simplesampling estimate')
xlabel('Walk length')
ylabel('Absolute Value of error')
legend()
```



For a managable number of trials (i.e N <= 10^7, which is a value I have tested,), we can't give an estimate for the number of self-avoiding random walks of length 50 with simple sampling because the directions are chosen randomly at every step which decreases the number of successful trials. Given max samples = N, the number of successful trials decreases as length of the walk increases. Hence, this method is not suitable for computing estimates for long walks. If we execute $simplesampling(50, 10000000)$ we will get 0 as our result

```
function estimate = simplesampling(l,maxsamples)

    % l is length of the walk
    % maxsamples is the number of trials that we want to run
```

```
% Count of accepted trials (the ones that reach length l)
s = 0;
% Count of total number of trials performed
samples = 0;

while samples < maxsamples

    % Initiate lattice where we perform our walks before every trial to
    % 0. Every time we step to a new coordinate we mark 1.
    lattice = zeros(2*l + 1 ,2*l + 1);
    % Increase trial count by 1
    samples = samples + 1;
    % Initiate steps performed in walk to 0 before every walk
    n = 0;
    % Start at origin
    x = l+1;
    y = l+1;
    % Mark origin
    lattice(x,y) = 1;

    % This is the while loop we use to walk around in the lattice. If
    % we reach the required walk length we break out of the loop
    while n < l

        % Pick random direction and move there
        z = randi(4,1);

        if z == 1
            x = x+1;
        elseif z == 2
            x = x-1;
        elseif z == 3
            y = y+1;
        elseif z == 4
            y = y-1;
        end

        % If the walk self intersects (i.e we step to a position we have
        % visited before), we dispose the trial and start a new one by
        % skipping to the end of this while loop.
        if lattice(x,y) == 1

            break

        % If we don't self intersect, we update the coordinate's value
        % to 1 and increase the number of steps taken
        else

            lattice(x,y) = 1;
            n = n+1;

        end
```

```
        end
        % When we n = l, it means we had successful. Hence we increase our
        % count of s by 1
        if n == l
            s = s+1;
        end
    end
    %This is estimate of the total number of walks
    estimate = power(4,l)*s/maxsamples;
end
```

Question5

(c) Write a function $\mathrm{rosenbluthsamplingSAW}(N)$ to use Rosenbluth sampling to estimate the number of self-avoiding walks of length $1..N$. Test your function bycomparing its output against the exact values for lengths up to 10, and give an estimate of the accuracy of your result.

The algorithm used in this implementation is taken from [2].

```
clear
clc

%Actual values
N = [ 4  12 36 100 284 780 2172 5916 16268 44100 ]
```

```
N = 1×10
           4          12          36         100         284         780 ···
```

```
tic
%Rosenbluth estimate for max samples =100000
r(1) = rosenbluthsampling(1,100000);
r(2) = rosenbluthsampling(2,100000);
r(3) = rosenbluthsampling(3,100000);
r(4) = rosenbluthsampling(4,100000);
r(5) = rosenbluthsampling(5,100000);
r(6) = rosenbluthsampling(6,100000);
r(7) = rosenbluthsampling(7,100000);
r(8) = rosenbluthsampling(8,100000);
r(9) = rosenbluthsampling(9,100000);
r(10) = rosenbluthsampling(10,100000);
toc
```

```
Elapsed time is 287.004615 seconds.
```

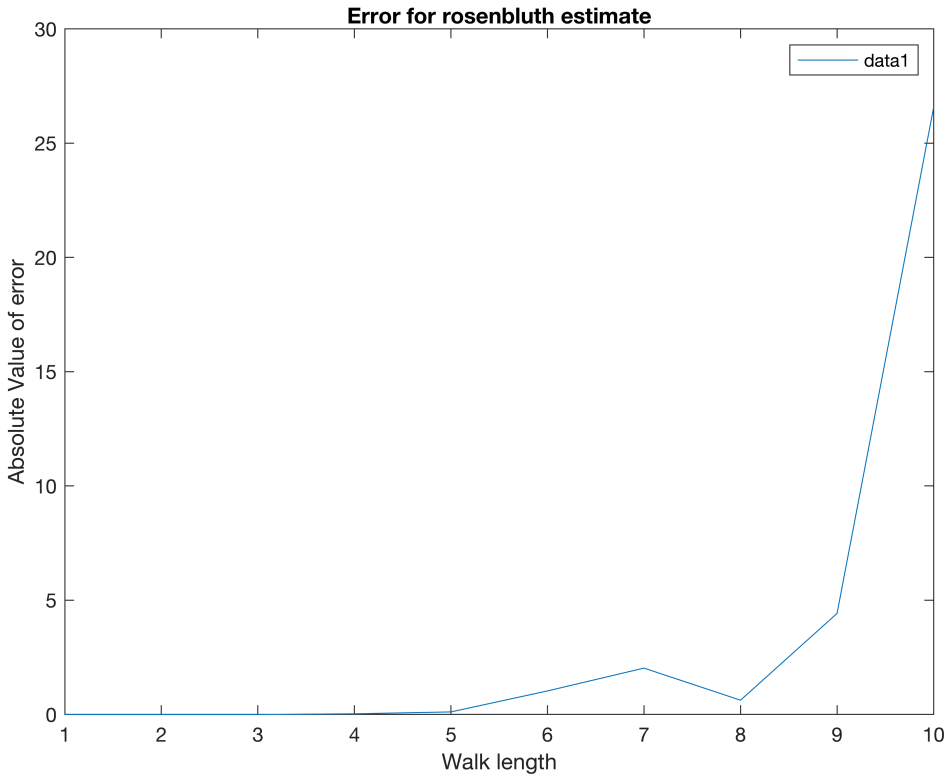
```
format long
%Calculate error
for i = 1:10
    errorforrosenbluth(i) = abs(N(i) - r(i));
end
%Make Table
T = table(transpose(N),transpose(r),transpose(errorforrosenbluth));
T.Properties.VariableNames{1} = 'Actual Values';
T.Properties.VariableNames{2} = 'Rosenbluth Sampling Estimate';
T.Properties.VariableNames{3} = 'RosenbluthSampling Error';
T
```

```
T = 10×3 table
```

	Actual Values	Rosenbluth Sampling Estimate	RosenbluthSampling Error
1	4	4	0
2	12	12	0
3	36	36	0

	Actual Values	Rosenbluth Sampling Estimate	RosenbluthSampling Error
4	100	1.000299600000000e+02	0.029960000000003
5	284	2.841123600000000e+02	0.112360000000024
6	780	7.789694400000000e+02	1.030560000000037
7	2172	2.174030280000000e+03	2.030279999999948
8	5916	5.915374920000000e+03	0.625079999999798
9	16268	1.627242372000000e+04	4.423720000000685
10	44100	4.407344460000000e+04	26.555399999997462

```
plot(errorforrosenbluth)
title('Error for rosenbluth estimate')
xlabel('Walk length')
ylabel('Absolute Value of error')
legend()
```



Yes we can estimate the number of self-avoiding walks of length 50. This is possible because we put a limit on the number of directions we can move along at each step by looking at which spaces are unoccupied.

```
rosenbluthsampling(50,100000)
```

```
ans =
     5.221923007772319e+21
```

```
function estimate = rosenbluthsampling(l, maxsamples)

    % Count of total number of trials performed
    samples = 0;
    % Since we need the sum of all weights of successful trials, we will
    % store the sum in this variable
    sum = 0;


    while samples < maxsamples

        % Initiate lattice where we perform our walks before every trial to
        % 0. Every time we step to a new coordinate we mark 1.
        lattice = zeros(2*l + 1 ,2*l + 1);
        % Increase trial count by 1
        samples = samples + 1;
        % Initiate steps performed in walk to 0 before every walk
        n = 0;
        % Start at origin
        x = l+1;
        y = l+1;
        % Mark origin
        lattice(x,y) = 1;
        % Initiate weight to 1 before every trial
        weight = 1;

        % This is the while loop we use to walk around in the lattice. If
        % we reach the required walk length we break out of the loop
        while n < l

            % Make a list of the neighbors at every step and update the
            % neighborcount at every step. We first initate these variable
            % before every step and then we update them
            neighborlist = zeros(4,1);
            % This variable also counts as the atmosphere of the coordinate
            neighorcount = 0;

            % When the neighors are unoccupied(== 0), we add them to the
            % list and increase the neighborcount by 1.
            % We have 'U' for up. It always goes in the first position of
            % neighborlist
            % We have 'D' for down. It always goes in the second position of
            % neighborlist
            % We have 'R' for right. It always goes in the third position of
            % neighborlist
            % We have 'L' for left. It always goes in the fourth position of
            % neighborlist
            if lattice(x+1,y) == 0
                neighborlist(1) = 'U';
                neighorcount = neighorcount + 1;
            end
            if lattice(x-1,y) == 0
                neighborlist(2) = 'D';
```

```
            neighorcount = neighorcount + 1;
        end
        if lattice(x,y+1) == 0
            neighborlist(3) = 'R';
            neighorcount = neighorcount + 1;
        end
        if lattice(x,y-1) == 0
            neighborlist(4) = 'L';
            neighorcount = neighorcount + 1;
        end

        % When there are no unoccupied neighbors and we haven't
        % completed our walk, we start a new trial
        if neighorcount == 0

            break

        % Otherwise, we choose an unoccupied neighbor randomly and then
        % move there
        else

            % This is just a technical step. We convert the elements so
            % that they are one of 'U', 'D', 'R', 'L'
            neighborlist = char(neighborlist);
            % We delete the empty places
            neighborlist = nonzeros(neighborlist);
            % Permutate the list randomly and pick the first element. This is how w
            neighborlist = neighborlist(randperm(length(neighborlist)));

            % Move to the new position by updating the coordinate
            if neighborlist(1) == 'U'
                x = x+1;
            end
            if neighborlist(1) == 'D'
                x = x-1;
            end
            if neighborlist(1) == 'R'
                y = y+1;
            end
            if neighborlist(1) == 'L'
                y = y-1;
            end

            % Update the new position
            lattice(x,y) = 1;
            % Increase number of steps taken
            n = n+1;
            % Update weight
            weight = weight*neighorcount;
        end
    end

    % If a walk is completed i.e. n reaches l, we add the final weight
    % of the walk to the running total
```

```
        if n == l
          sum = sum + weight;
        end
    end
    % This is our final estimate!
    estimate = sum/maxsamples;
end
```

Question5

(d) Make table to represent data.

Since there are only 27 terms available in this sequences on OEIS [3], I will only make make a table for the corresponding 27 values for now.

```
clear
clc
% Values from Online Encyclopedia of Integer Sequences
OEIS = [4, 12, 36, 100, 284, 780, 2172, 5916, 16268, 44100, 120292, 324932, 881500, 237
% Values from our recursive enumeration algorithm. Our algorithm takes to
% much time compute walks of length 18 and longer
SAW = [4, 12, 36, 100, 284, 780, 2172, 5916, 16268, 44100, 120292, 324932, 881500, 2374
% Values for simplesampling
SS= [4 , 11.993919999999999, 36.014080000000000, 99.955200000000005, 2.810675200000000e
% Simple sampling absolute error
SSE = abs(OEIS - SS);
% Values for rosenbluthsampling
RS= [4, 12, 36, 100.05840000000, 283.9428, 779.3604, 2173.068, 5941.9008, 16275.21984,
% Rosenbluth sampling absolute error
RSE = abs(OEIS - RS);
format short
T = table(transpose(OEIS),transpose(SAW),transpose(SS),transpose(SSE),transpose(RS),tra
T.Properties.VariableNames{1} = 'OEIS';
T.Properties.VariableNames{2} = 'SAW';
T.Properties.VariableNames{3} = 'Simple Sampling Estimate';
T.Properties.VariableNames{4} = 'Simple Sampling Error';
T.Properties.VariableNames{5} = 'Rosenbluth Sampling';
T.Properties.VariableNames{6} = 'Rosenbluth Sampling Error';
T
```

T = 27×6 table

	OEIS	SAW	Simple Sampl...	Simple Sam...	Rosenbluth ...	Rosenbluth...
1	4	4	4	0	4	0
2	12	12	11.9939	0.0061	12	0
3	36	36	36.0141	0.0141	36	0
4	100	100	99.9552	0.0448	100.0584	0.0584
5	284	284	281.0675	2.9325	283.9428	0.0572
6	780	780	769.6794	10.3206	779.3604	0.6396
7	2172	2172	2.1889e+03	16.9024	2.1731e+03	1.0680
8	5916	5916	5.9192e+03	3.2115	5.9419e+03	25.9008
9	16268	16268	1.6290e+04	21.6282	1.6275e+04	7.2198
10	44100	44100	4.3621e+04	479.2384	4.4207e+04	106.7641
11	120292	120292	1.2436e+05	4.0691e+03	1.2019e+05	100.8272
12	324932	324932	3.0585e+05	1.9083e+04	3.2417e+05	758.2971

	OEIS	SAW	Simple Sampl...	Simple Sam...	Rosenbluth ...	Rosenbluth...
13	881500	881500	8.7242e+05	9.0848e+03	8.8052e+05	984.4650
14	2374444	2374444	2.3220e+06	5.2477e+04	2.3805e+06	6.0505e+03
15	6416596	6416596	6.8719e+06	4.5535e+05	6.3859e+06	3.0721e+04
16	17245332	17245332	1.7352e+07	1.0634e+05	1.7217e+07	2.8360e+04
17	46466676	46466676	4.5870e+07	5.9643e+05	4.6787e+07	3.2050e+05
18	124658732	0	1.2576e+08	1.0979e+06	1.2401e+08	6.5067e+05
19	335116620	0	3.5184e+08	1.6727e+07	3.3080e+08	4.3205e+06
20	897697164	0	8.1364e+08	8.4059e+07	9.0787e+08	1.0171e+07
21	2.4088e+09	0	2.6388e+09	2.3002e+08	2.4116e+09	2.8414e+06
22	6.4446e+09	0	6.5091e+09	6.4548e+07	6.4097e+09	3.4813e+07
23	1.7267e+10	0	2.1814e+10	4.5477e+09	1.7343e+10	7.6875e+07
24	4.6146e+10	0	5.6295e+10	1.0149e+10	4.6542e+10	3.9529e+08
25	1.2348e+11	0	1.4637e+11	2.2886e+10	1.2591e+11	2.4299e+09
26	3.2971e+11	0	2.7022e+11	5.9497e+10	3.3173e+11	2.0186e+09
27	8.8132e+11	0	5.4043e+11	3.4089e+11	8.7228e+11	9.0353e+09

(e) Due to time restrictions, I will draw conclusion from our current results. The implementation is explained with the code in the corresponding file.

SelfAvoidingWalks (Recursive enumeration)

1. Accuracy - 100%
2. Time efficiency - Very quick calculations for short walks. The first 10 terms of the sequence were calculated in `1.016704 seconds. But as the walks get longer, calculation time increases drastically. The 17 term was calculated in 183.350565 seconds`
3. `Computability - Because the computing time increases drastically, larger terms become impossible to compute.`

Simple Sampling

1. Accuracy - Least accurate out of the three algorithms
2. Time efficiency - The first 10 estimates were calculated in `59.758439 seconds. Compared to Rosenbluth Sampling, this is more efficient.`
3. `Computability - As we noticed, simplesampling(50) could not be computed with a managable N. Offers a higher computability than SelfAvoidingWalks, but not as much as RosenbluthSampling.`

Rosenbluth Sampling

1. Accuracy - Better accuracy than simplesampling but obviously not as good as SelfAvoidingWalks

2. Time efficiency - The first 10 estimates were calculated in 287.004615 seconds. The least efficient out of the three.
3. Calculation limits - As we noticed, rosenbluthsampling(50) could be computed. This method offer the most computability.

REFERENCES

1. Dool, W.V.D (2016) "Counting Self-Avoiding Walks on the 2D Square Lattice" (Thesis)
2. Prellberg, T., (2012) "From Rosenbluth Sampling to PERM - rare event sampling with stochastic growth algorithms," in R. Leidl and A. K. Hartmann (eds), Modern Computational Science 12: Lecture Notes from the 4th International Oldenburg Summer School, pages 311-334, BIS-Verlag der Carl von Ossietzky Universita¨t Oldenburg (http://www.maths.qmul.ac.uk/ ˜tp/papers/pub084pre.pdf)
3. *Number of n-step self-avoiding walks on square lattice, in The Online Encyclopedia of Integer Sequences (http://oeis.org/A001411)*