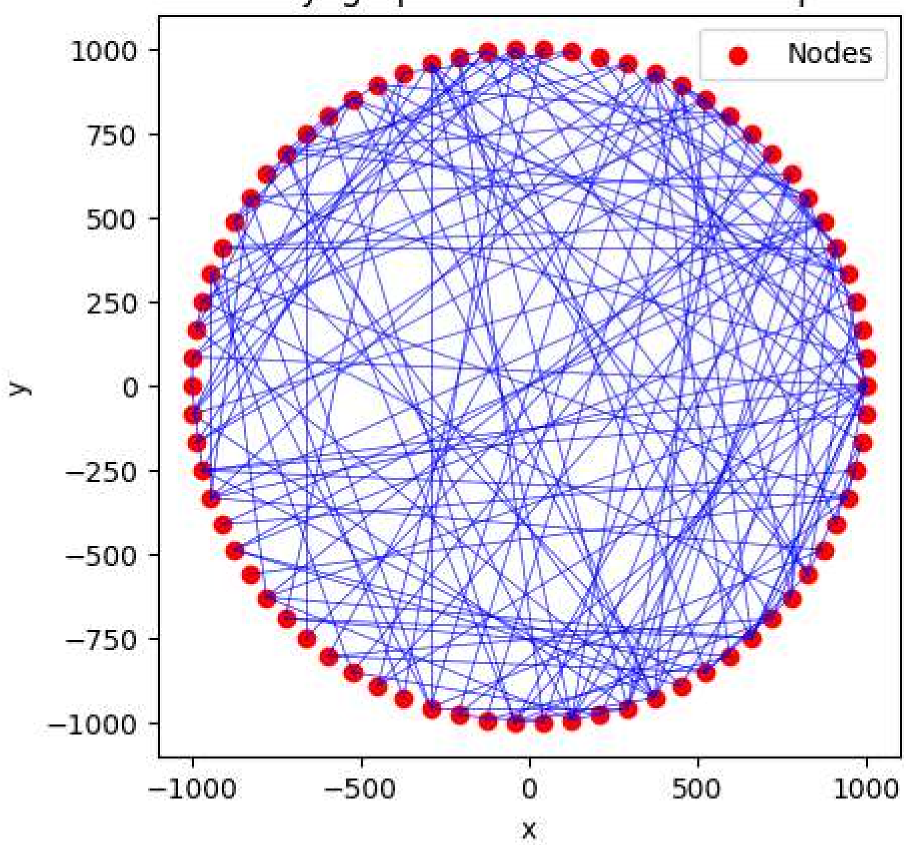
12.1 The Erdős–Rényi random graph

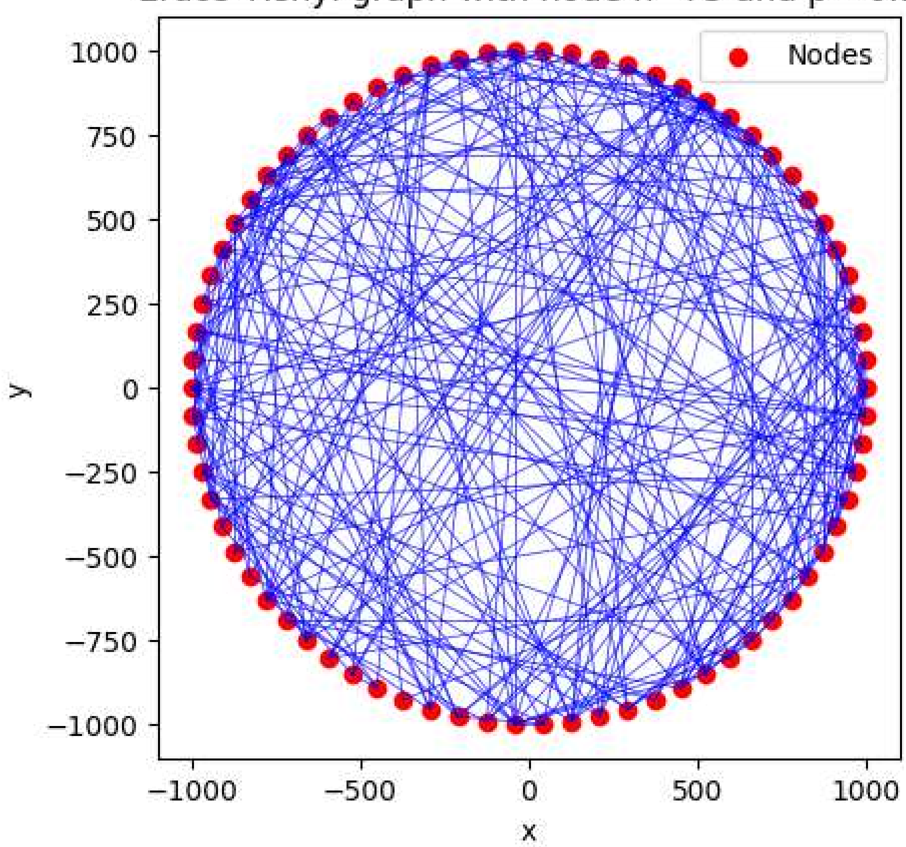
12.1.a

In [ ]:
```
import numpy as np
from matplotlib import pyplot as plt
import random
from scipy.special import comb

n_list= [75,100, 50]
# no_edges = n*(n-1)/2
p_list = [0.05, 0.1]
r = 1000 #radius of circle for plotting
graphnumber =0

for n in n_list:
        for p in p_list:
            graphnumber+=1
            n=n
            p=p
            amatrix = np.zeros((n,n))
            for i in range(n):
                for j in range(n):
                    if i>j:
                        x= random.random()
                        if x< p:
                            amatrix[i][j]= 1
                            amatrix[j][i]=1
            degree= np.zeros((n,1))

            for i in range(n):
                degree[i,:] = np.sum(amatrix[i,:])

            def probability_distribution(n, p, k):
                return comb(n-1, k) * (p**k) * ((1-p)**(n-k-1))

            k_values = np.linspace(0,20,50)
            p_k_list=[]
            for k in k_values:
                p_k_analytical =  probability_distribution(n, p, k)
                p_k_list.append(p_k_analytical)



            theta = np.linspace(0, 2*np.pi, n)
            x = r * np.cos(theta)
            y = r * np.sin(theta)


            # print(amatrix)
            # plt.figure(figsize=(10, 6))

            plt.scatter(x, y, color='red', label='Nodes')
            for i in range(n):
                for j in range(n):
                    if amatrix[i][j] ==1:
                        plt.plot([x[i], x[j]], [y[i], y[j]], color='blue', alpha=0.5, linewidth= 0.5)
            plt.gca().set_aspect('equal', adjustable='box')
            plt.xlabel('x')
            plt.ylabel('y')
            plt.legend()
            plt.title(f'Erdős–Rényi graph with node n={n} and p ={p}')


            plt.show()
```

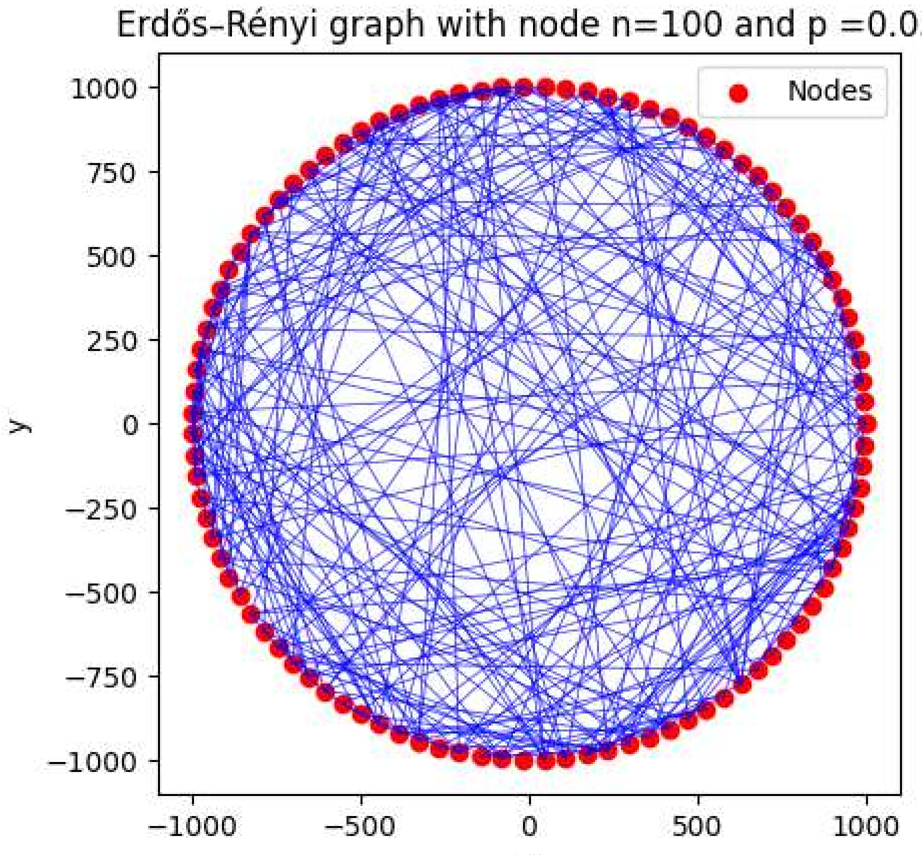
Erdős–Rényi graph with node n=75 and p =0.05

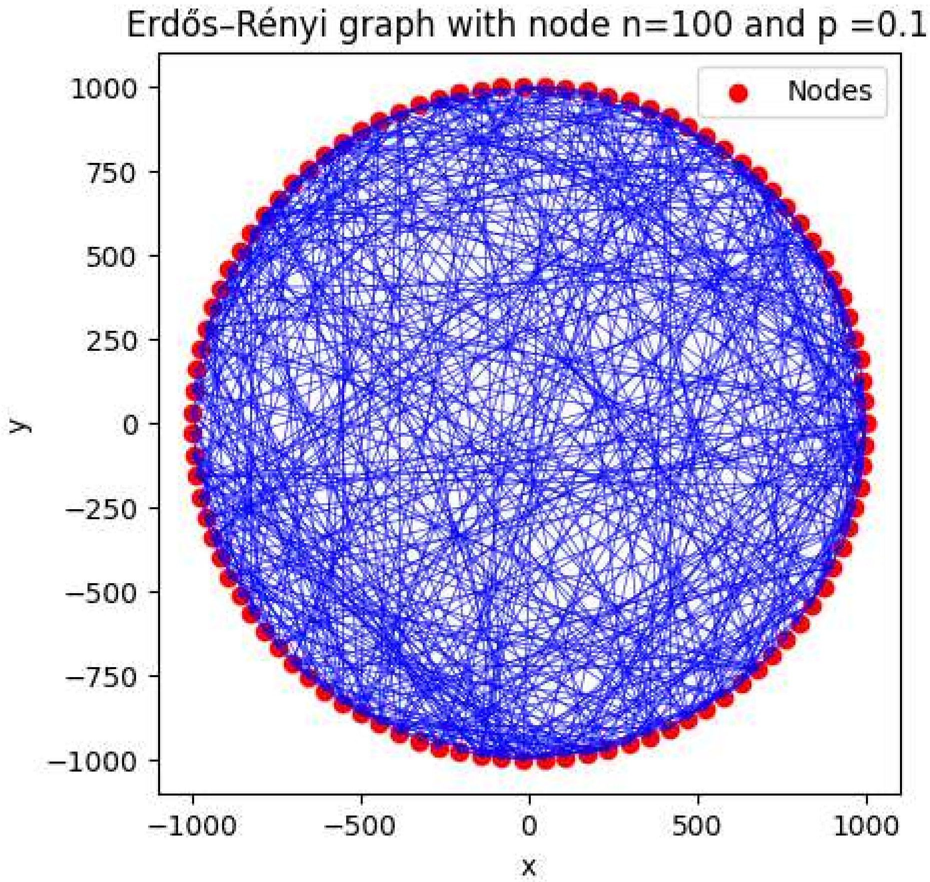


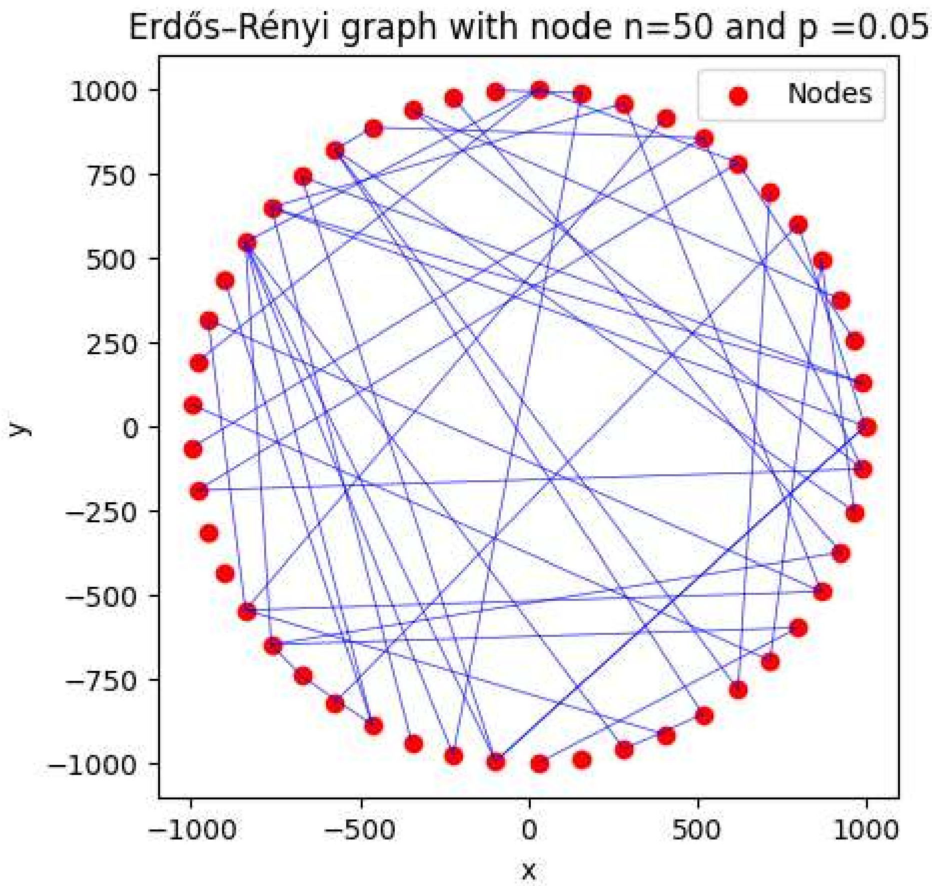
Erdős–Rényi graph with node n=75 and p =0.1

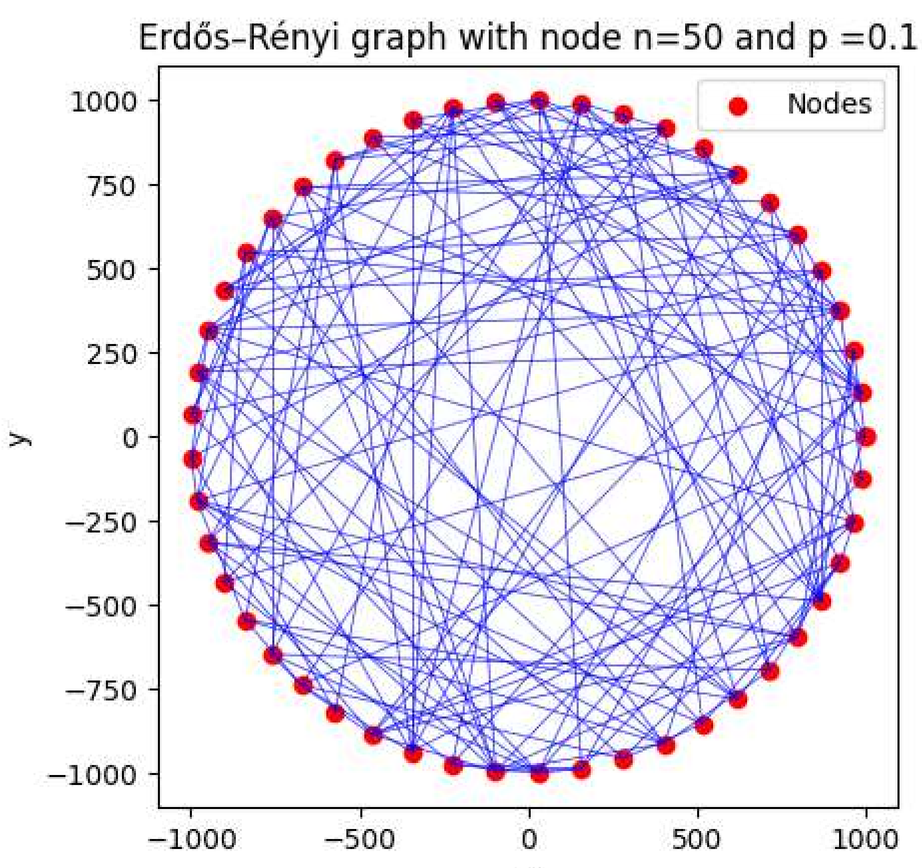


Erdős–Rényi graph with node n=100 and p =0.05





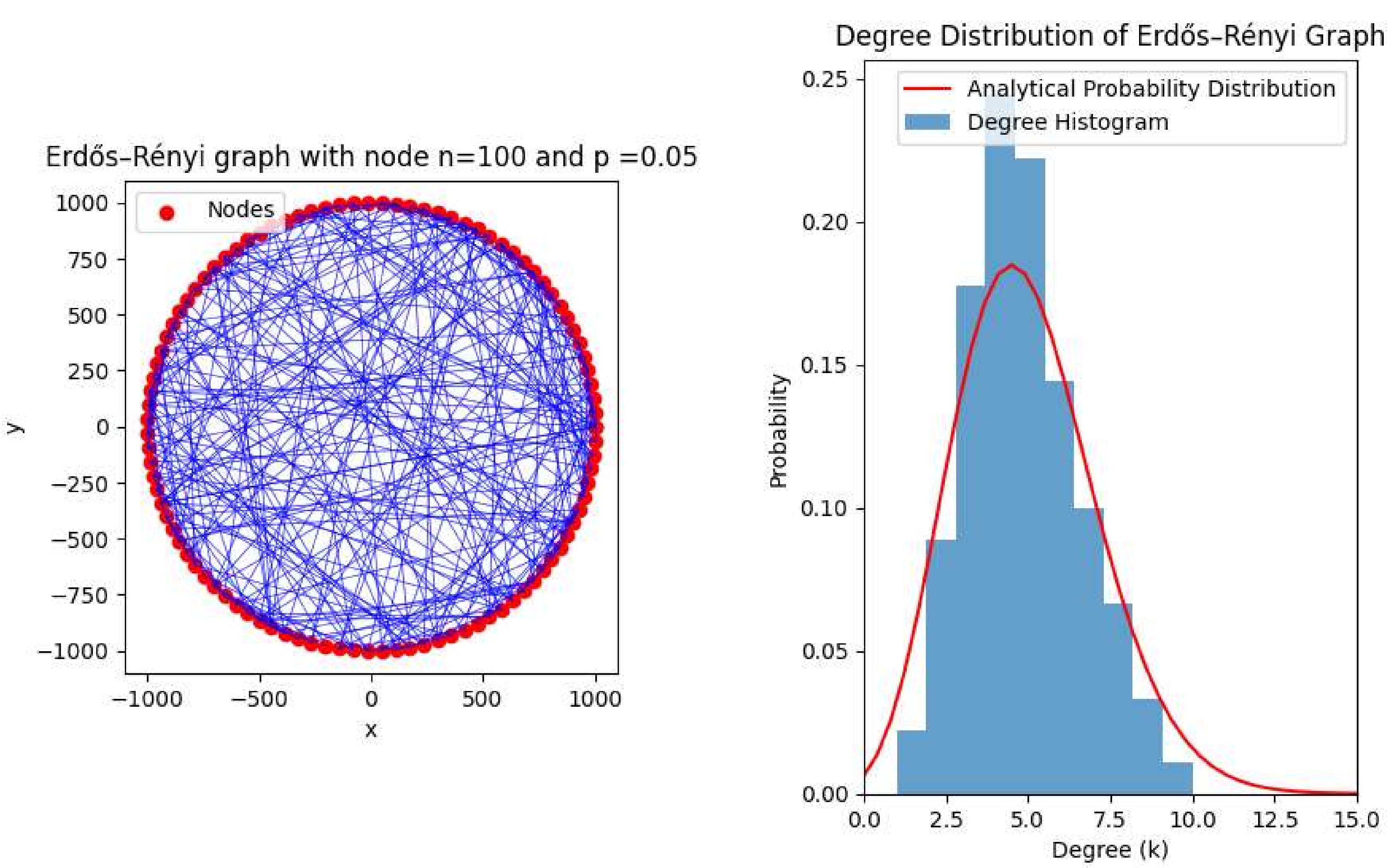




In [ ]:

```
n=100
p=0.05
amatrix = np.zeros((n,n))
for i in range(n):
    for j in range(n):
        if i>j:
            x= random.random()
            if x< p:
```

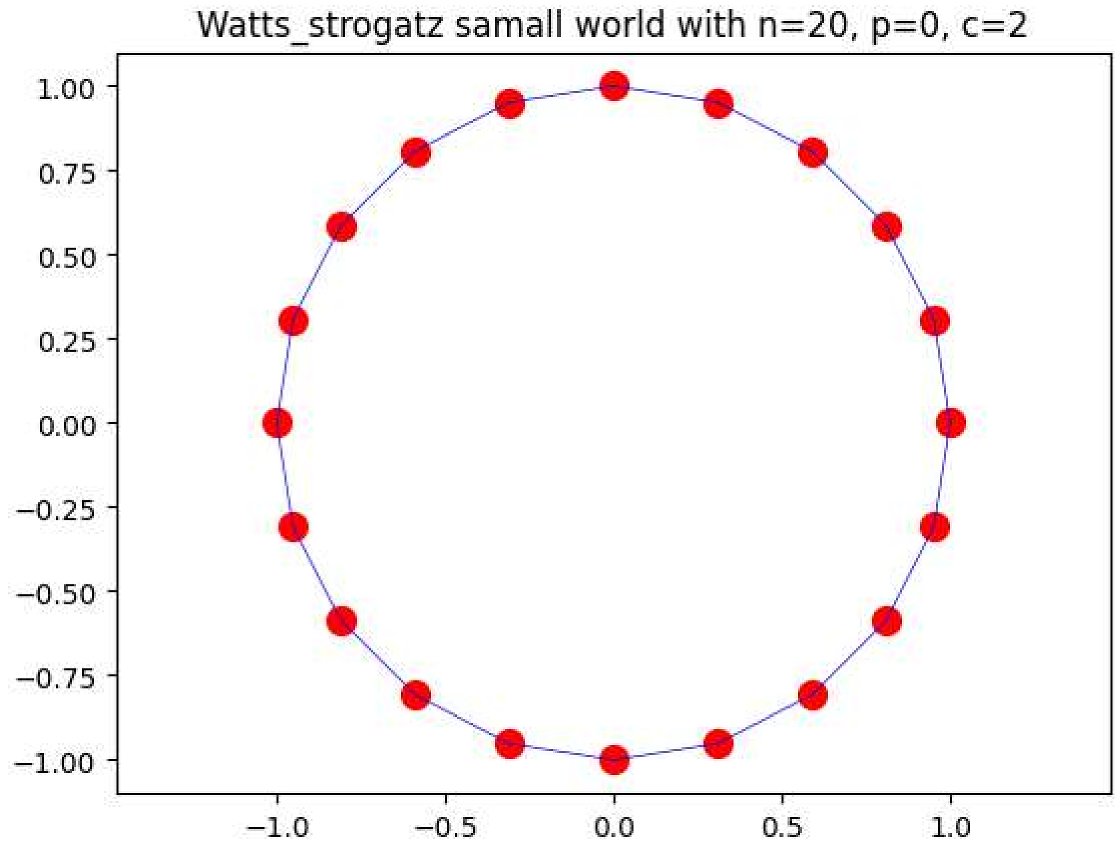
```
                amatrix[i][j]= 1
                amatrix[j][i]=1
degree= np.zeros((n,1))
for i in range(n):
    degree[i,:] = np.sum(amatrix[i,:])

def probability_distribution(n, p, k):
    return comb(n-1, k) * (p**k) * ((1-p)**(n-k-1))

theta = np.linspace(0, 2*np.pi, n)
x = r * np.cos(theta)
y = r * np.sin(theta)

k_values = np.linspace(0,20,50)
p_k_list=[]
for k in k_values:
    p_k_analytical =  probability_distribution(n, p, k)
    p_k_list.append(p_k_analytical)

# plt.plot(k_values, p_k_list)
# # plt.show()
# plt.hist(degree, bins=12)
# plt.show()



plt.figure(figsize=(10, 6))

plt.subplot(1,2,1)
plt.scatter(x, y, color='red', label='Nodes')
for i in range(n):
    for j in range(n):
        if amatrix[i][j] ==1:
            plt.plot([x[i], x[j]], [y[i], y[j]], color='blue', alpha=0.5, linewidth= 0.5)
plt.gca().set_aspect('equal', adjustable='box')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title(f'Erdős-Rényi graph with node n={n} and p ={p}')

plt.subplot(1,2,2)
plt.plot(k_values, p_k_list, label='Analytical Probability Distribution', color='red')
plt.hist(degree, bins=10, density=True, alpha=0.7, label='Degree Histogram')
plt.xlabel('Degree (k)')
plt.ylabel('Probability')
plt.title('Degree Distribution of Erdős-Rényi Graph')
plt.xlim(0,15)
plt.legend()
plt.subplots_adjust(wspace=0.5)

plt.show()
```

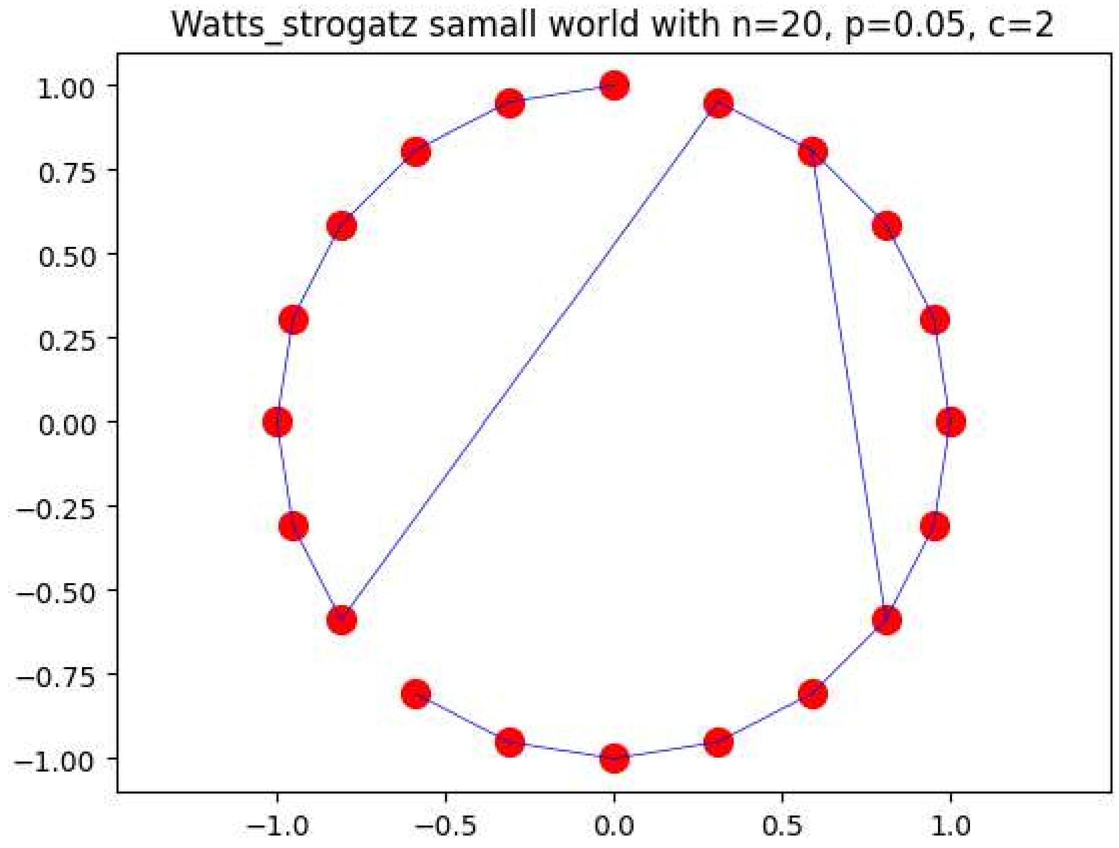


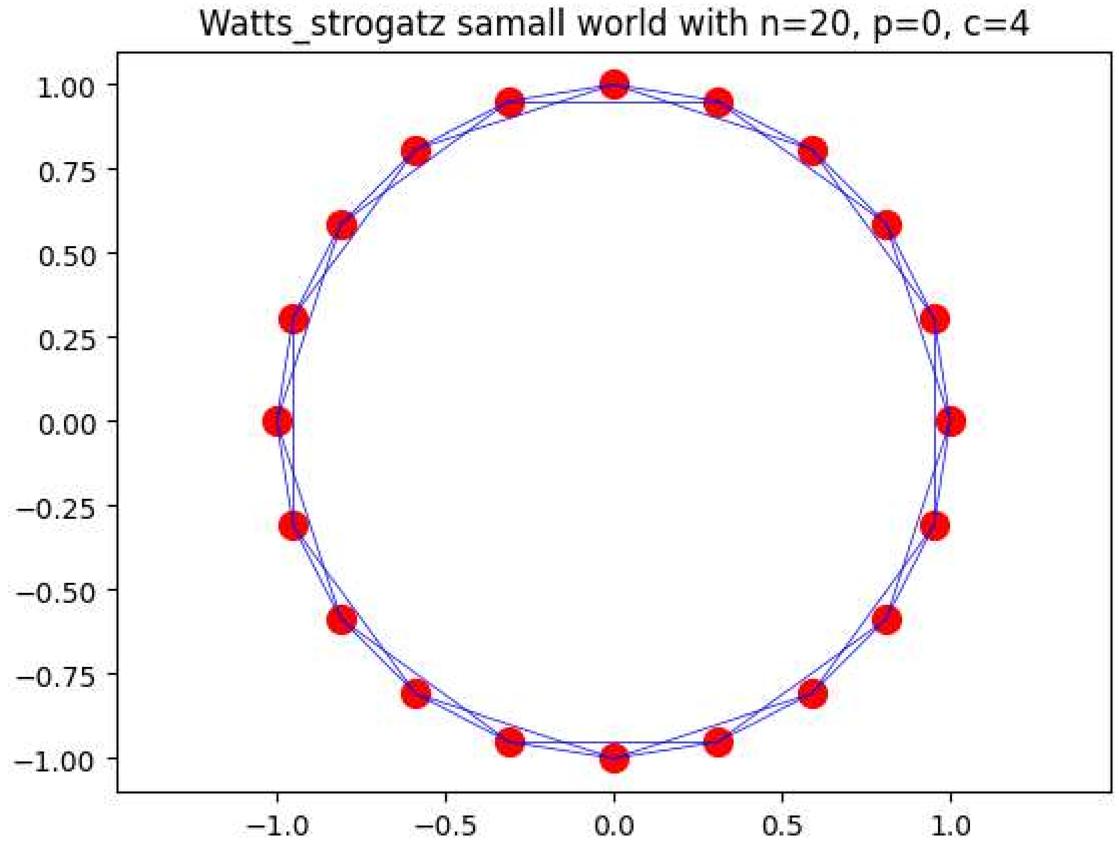
In [ ]:

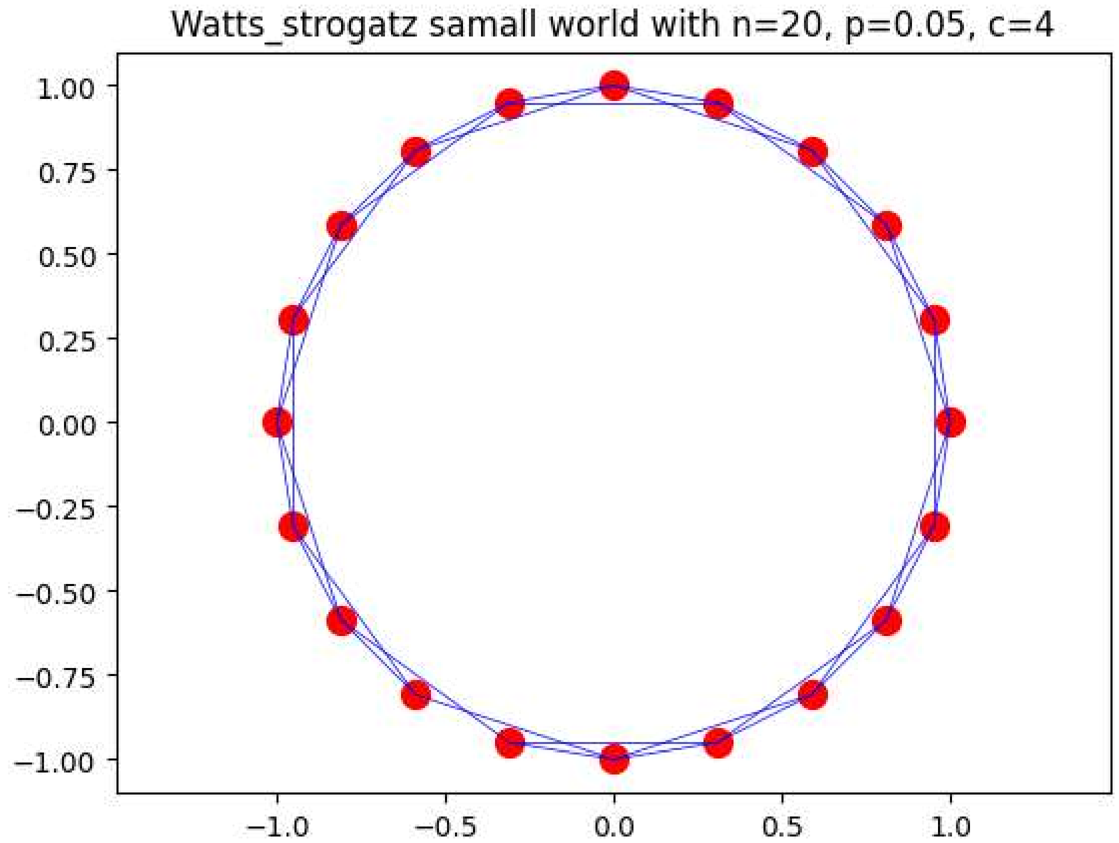
12.2.a

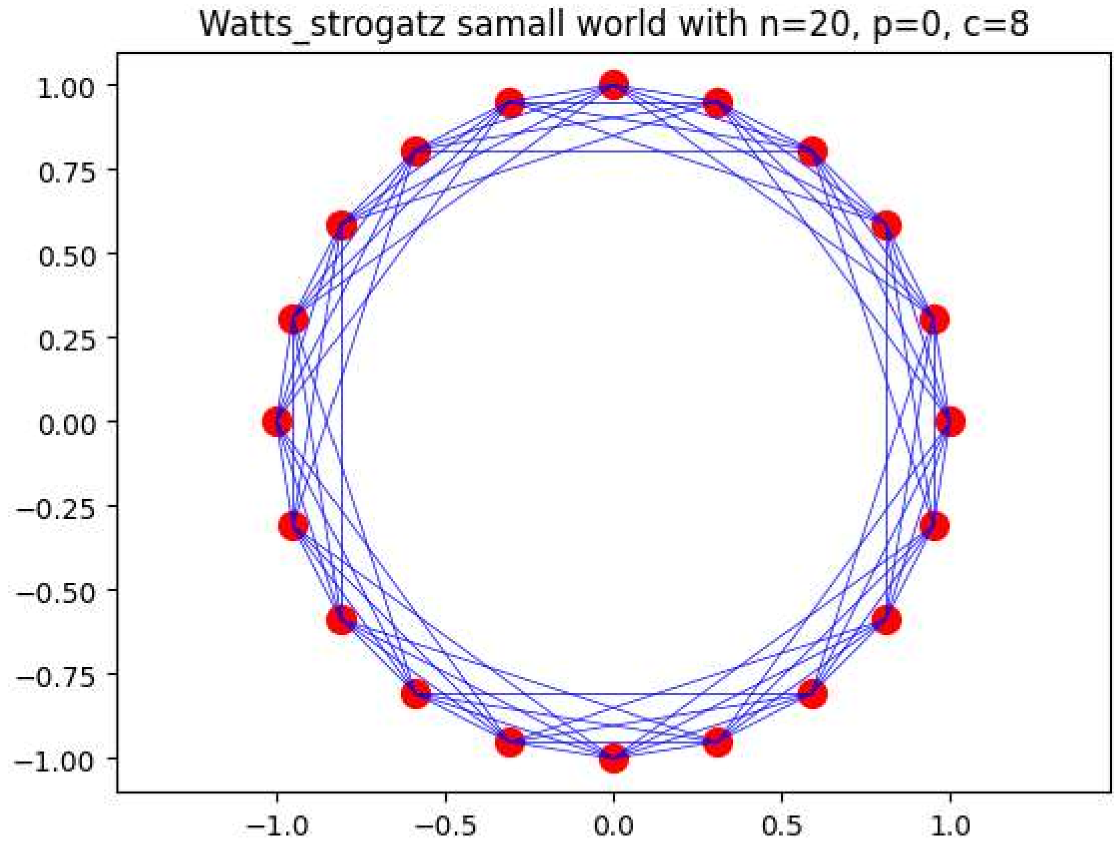
```
In [ ]: import numpy as np
import matplotlib.pyplot as plt

def watts_strogatz_graph(n, k, p):
    G = np.zeros((n, n), dtype=int)
    for i in range(n):
        for j in range(1, k // 2 + 1):
            G[i, (i + j) % n] = 1
            G[i, (i - j) % n] = 1

    for i in range(n):
        for j in range(n):
            if G[i, j] == 1 and np.random.rand() < p:
                G[i, j] = 0
                rand_node = np.random.randint(0, n - 1)
                while rand_node == i or G[i, rand_node] == 1:
                    rand_node = np.random.randint(0, n - 1)
                G[i, rand_node] = 1

    return G

def plot_graph_circular(G):
    angles = np.linspace(0, 2 * np.pi, G.shape[0], endpoint=False)
    positions = np.column_stack([np.cos(angles), np.sin(angles)])

    plt.figure()
    for i in range(G.shape[0]):
        for j in range(i + 1, G.shape[0]):
            if G[i, j] == 1:
                plt.plot([positions[i, 0], positions[j, 0]], [positions[i, 1], positions[j, 1]], color='blue', linew:

    plt.scatter(positions[:, 0], positions[:, 1], c='red', s=100)
    plt.axis('equal')
    plt.title(f'Watts_strogatz samall world with n={n}, p={p}, c={c}')
    plt.show()


n = 20
c_list = [2,4,8]
p_list = [0,0.05]

for c in c_list:
    for p in p_list:
        ws_graph = watts_strogatz_graph(n, c, p)

        plot_graph_circular(ws_graph)
```

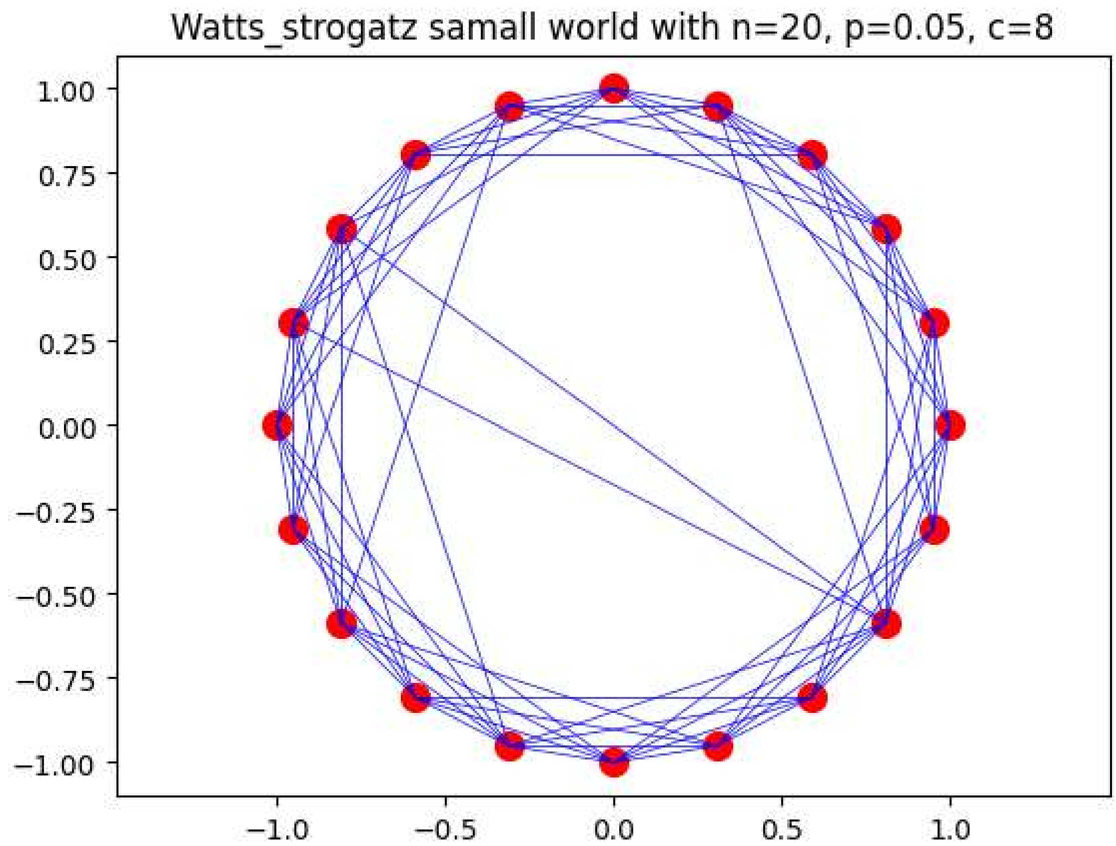



Watts_strogatz samall world with n=20, p=0.05, c=2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


Watts_strogatz samall world with n=20, p=0, c=4
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

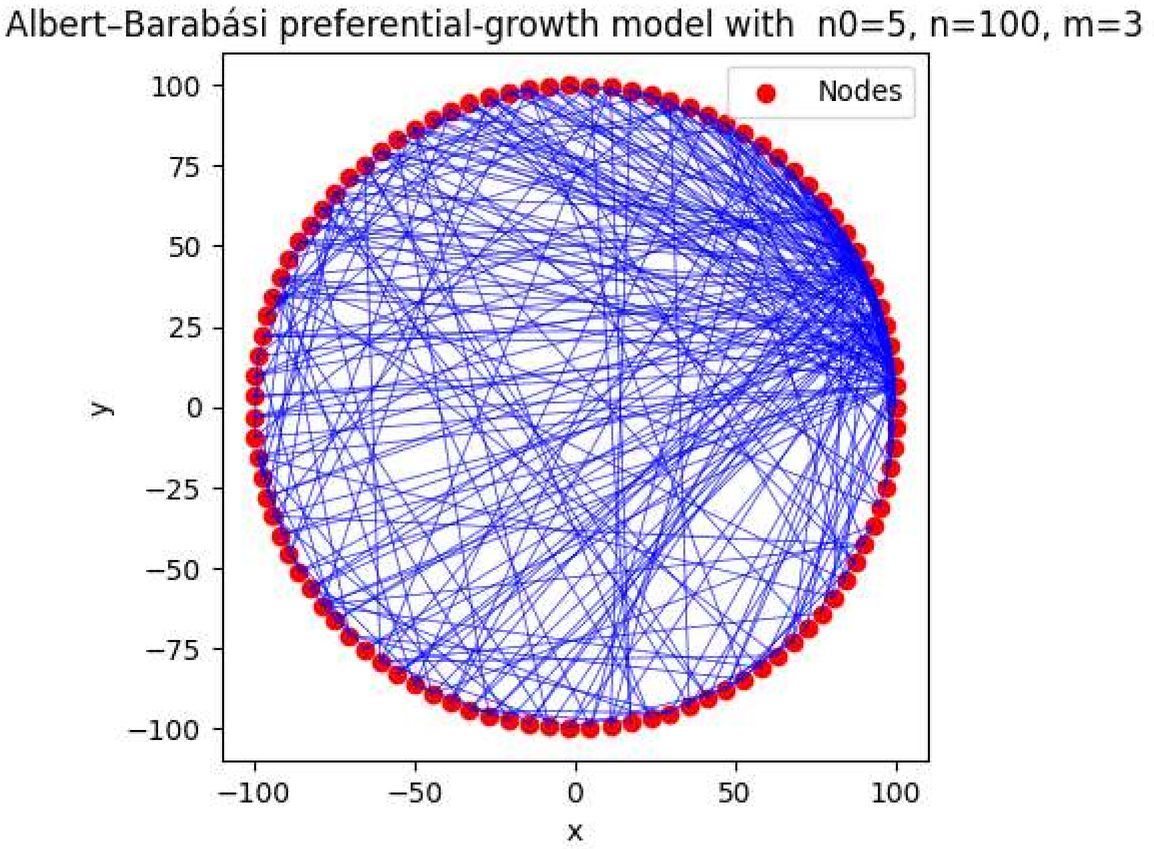

Watts_strogatz samall world with n=20, p=0.05, c=4
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


Watts_strogatz samall world with n=20, p=0, c=8
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0


Watts_strogatz samall world with n=20, p=0.05, c=8
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0

12.3a

```
In [ ]: import numpy as np
import matplotlib.pyplot as plt
import random

n0=5
n=100
m=3
r=100

amatrix= np.ones((n0,n0))
np.fill_diagonal(amatrix, 0)

def find_degree (matrix):
    degree= np.zeros((len(matrix),1))
    for i in range(len(matrix)):
        degree[i,:] = np.sum(matrix[i,:])
    total_degree = np.sum(degree)
    cumulative_sum = np.cumsum(degree)
    prob = cumulative_sum/total_degree
    return (prob)

for i in range(1,n-n0+1):
    new_matrix = np.zeros((n0+i, n0+i))
    new_matrix[:-1,:-1] = amatrix
    new_edge =0
    previous_list=[]
    while new_edge<m:
        prob = find_degree(amatrix)
        random_number = np.random.uniform(0,1)
        for a in range(len(prob)):
            if prob[a]>=random_number and a not in previous_list:
                # toss = np.random.uniform(0,1)
                # if toss >=0.5:
                new_matrix[n0+i-1][a]=1
                # else:
                new_matrix[a][n0+i-1]=1
                new_edge+=1
                previous_list.append(a)
                break
                # print(new_edge)
    amatrix = new_matrix



theta = np.linspace(0, 2*np.pi, len(amatrix))
x = r * np.cos(theta)
y = r * np.sin(theta)




plt.figure(figsize=(10, 6))

plt.subplot(1,2,1)
plt.scatter(x, y, color='red', label='Nodes')
for i in range(n):
    for j in range(n):
        if amatrix[i][j] ==1:
            plt.plot([x[i], x[j]], [y[i], y[j]], color='blue', alpha=0.5, linewidth= 0.5)
plt.gca().set_aspect('equal', adjustable='box')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.title(f'Albert-Barabási preferential-growth model with  n0={n0}, n={n}, m={m}')
plt.show()
```

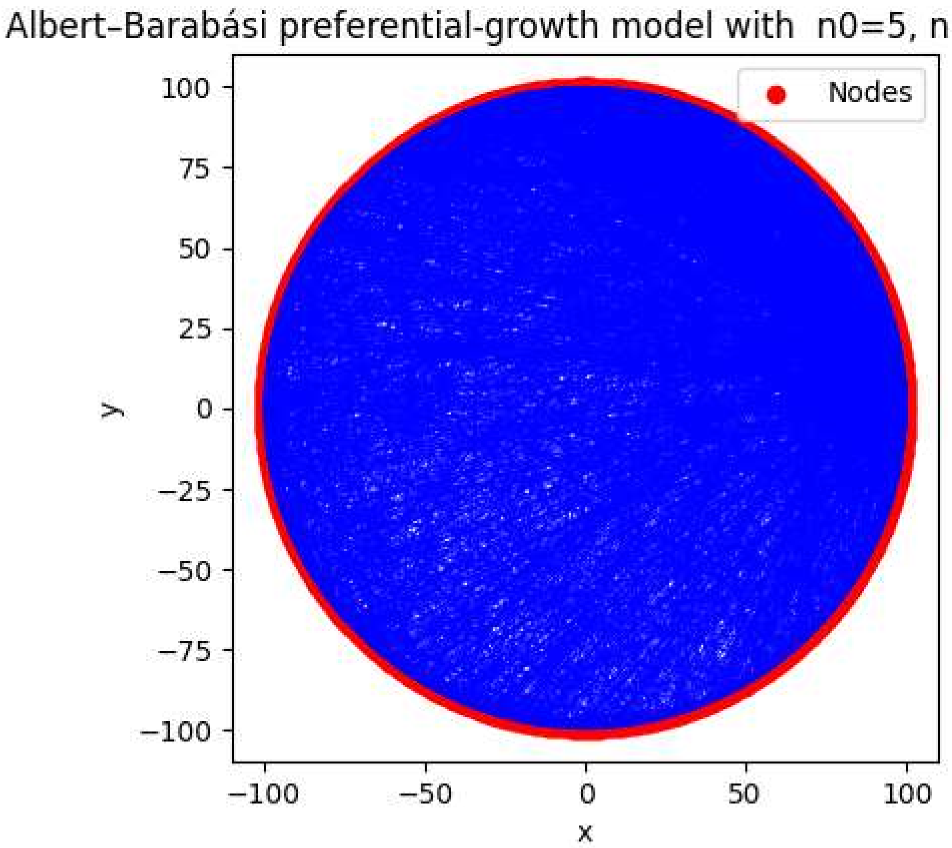


b

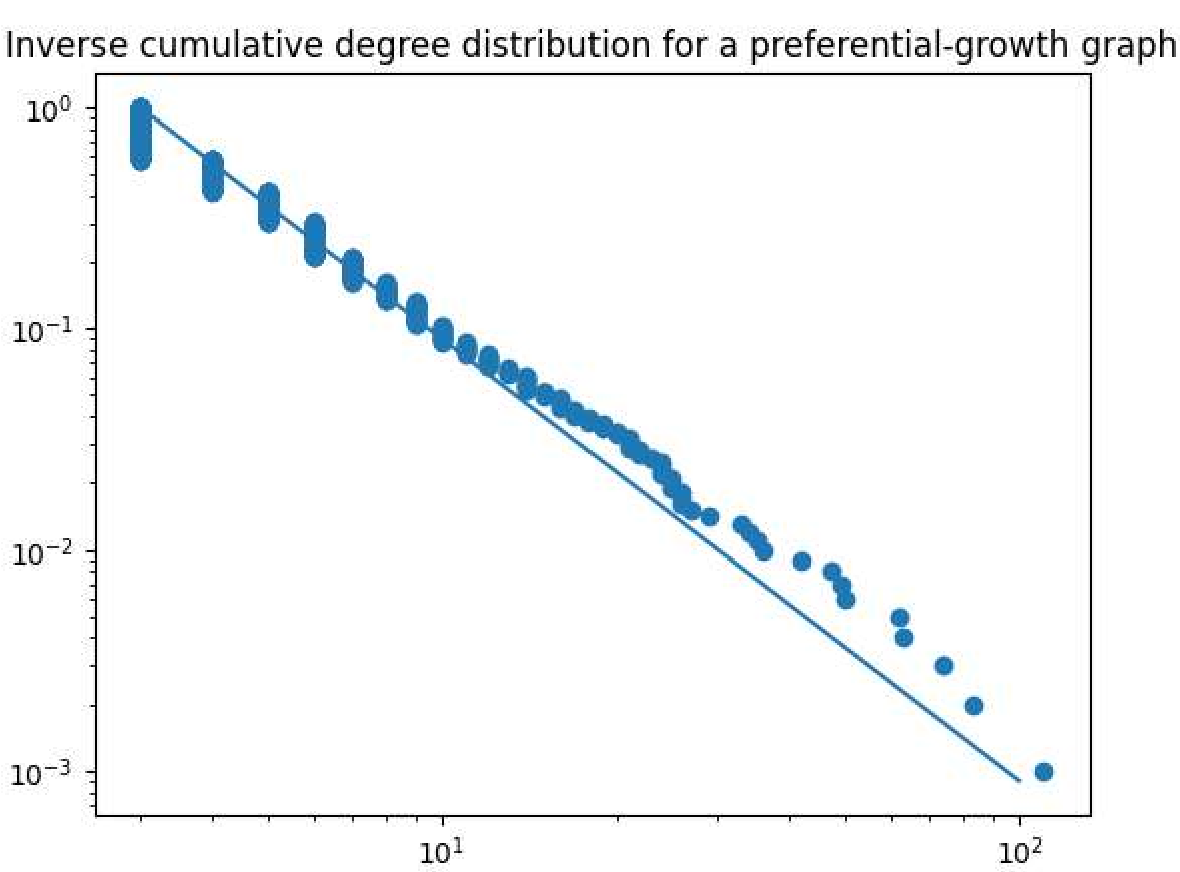
```
import numpy as np
import matplotlib.pyplot as plt
import random

n0=5
n=1000
m=3
r=100

amatrix= np.ones((n0,n0))
np.fill_diagonal(amatrix, 0)

def find_degree (matrix):
    degree= np.zeros((len(matrix),1))
    for i in range(len(matrix)):
        degree[i,:] = np.sum(matrix[i,:])
    total_degree = np.sum(degree)
    cumulative_sum = np.cumsum(degree)
    prob = cumulative_sum/total_degree
    return (prob)

for i in range(1,n-n0+1):
    new_matrix = np.zeros((n0+i, n0+i))
    new_matrix[:-1,:-1] = amatrix
    new_edge =0
    previous_list=[]
    while new_edge<m:
        prob = find_degree(new_matrix)
        random_number = np.random.uniform(0,1)
        for a in range(len(prob)):
            if prob[a]>=random_number and a not in previous_list:
                # toss = np.random.uniform(0,1)
                # if toss >0.5:
                new_matrix[n0+i-1][a]=1
                # else:
                new_matrix[a][n0+i-1]=1
                new_edge+=1
                previous_list.append(a)
                break

    amatrix = new_matrix


print('new_edge', new_edge)
theta = np.linspace(0, 2*np.pi, len(amatrix))
x = r * np.cos(theta)
y = r * np.sin(theta)



plt.figure(figsize=(10, 6))

plt.subplot(1,2,1)
plt.scatter(x, y, color='red', label='Nodes')
for i in range(n):
    for j in range(n):
        if amatrix[i][j] ==1:
            plt.plot([x[i], x[j]], [y[i], y[j]], color='blue', alpha=0.5, linewidth= 0.5)
plt.gca().set_aspect('equal', adjustable='box')
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.legend()
plt.title(f'Albert–Barabási preferential-growth model with  n0={n0}, n={n}, m={m}')
plt.show()
```

new_edge 3



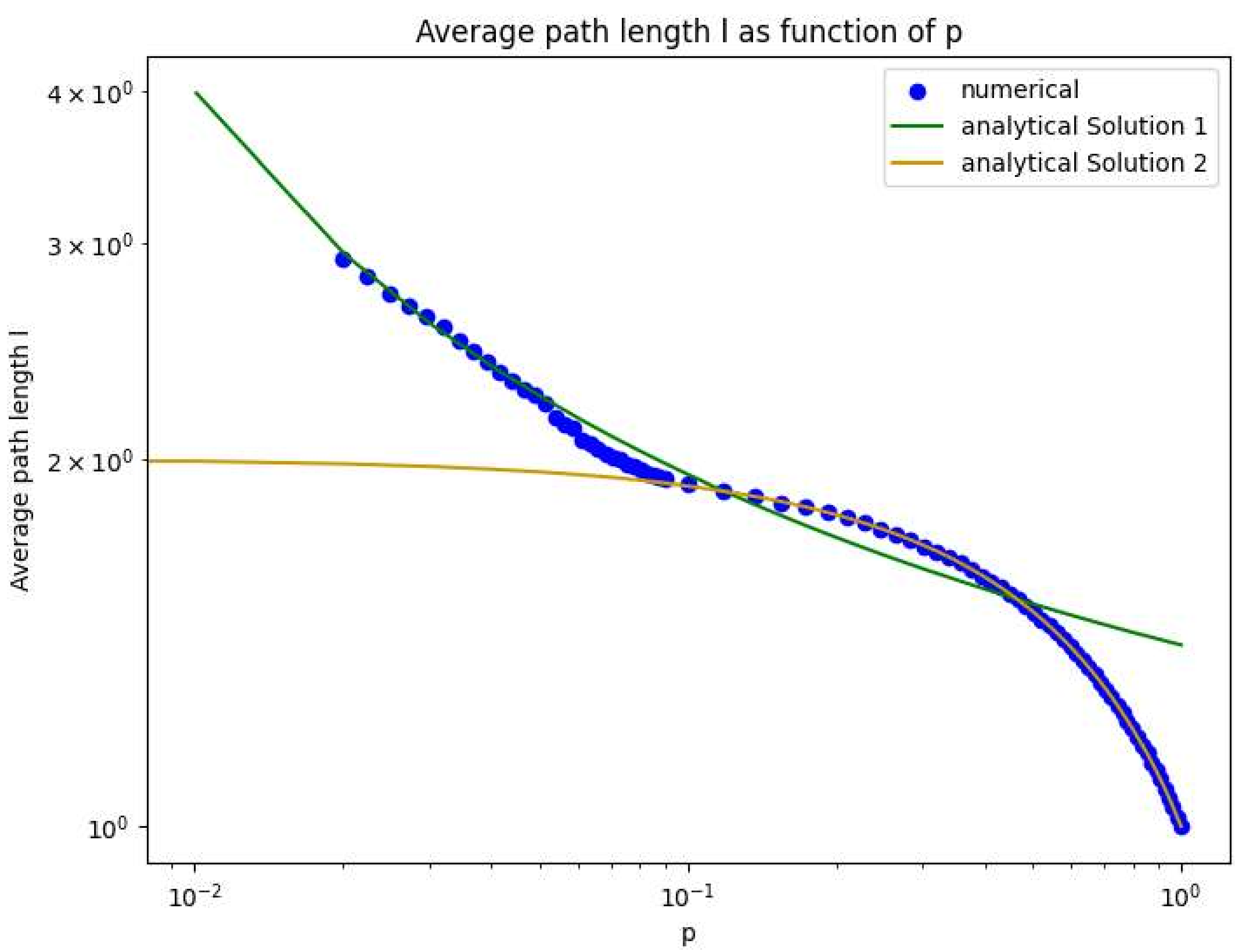
In [ ]:
```
degree =[]
for i in range(len(amatrix)):
    x = np.sum(amatrix[i,:])
    degree.append(x)
ds = np.sort(degree)[::-1]


u= []
for i in range(1,len(amatrix)+1):
    u.append(i/n)
k_valus = x = np.logspace(0.5,2,100)
 #np.linspace(np.max(degree), np.min(degree))
def theoretical_degree_distribution(k, m):
    return (m**2) * (k**(-2))


degree_theortical = theoretical_degree_distribution(k_valus,m)
plt.scatter(ds,u)
plt.loglog(k_valus,degree_theortical)
plt.xscale('log')
plt.yscale('log')
plt.title('Inverse cumulative degree distribution for a preferential-growth graph')
plt.show()
```

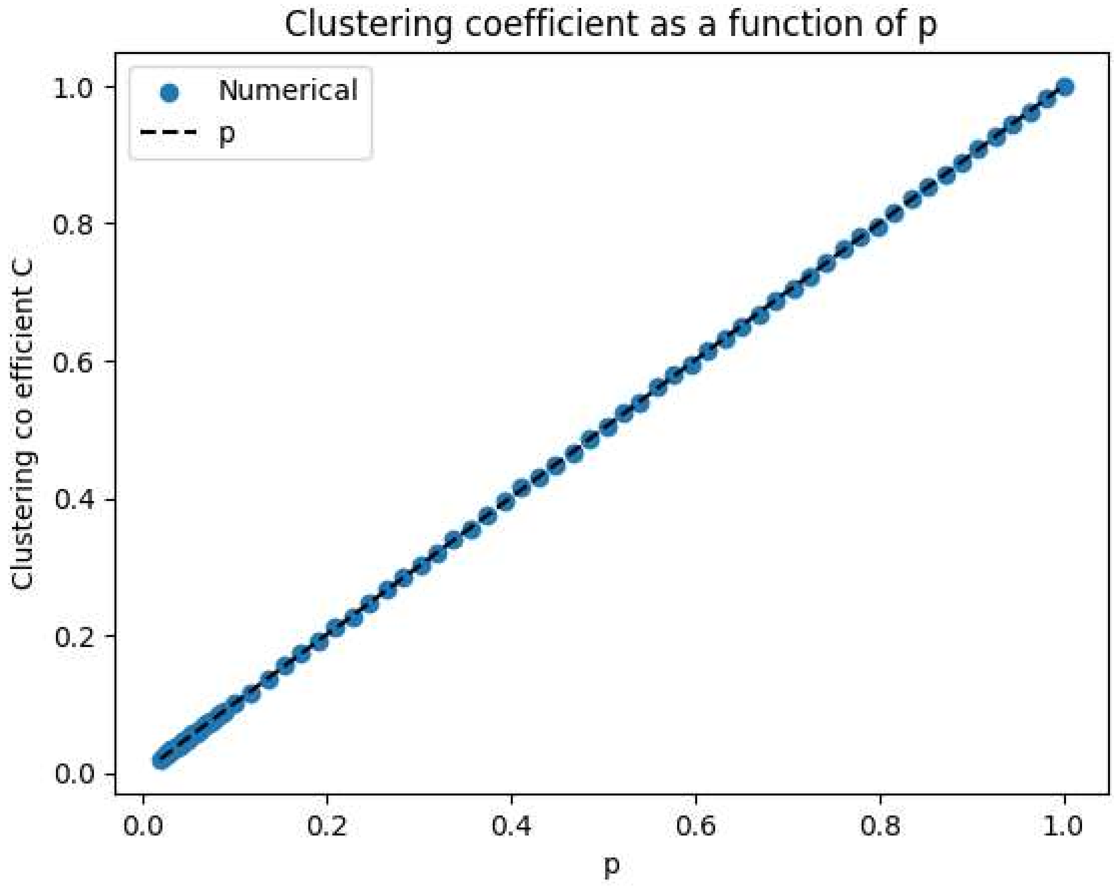


12.4a,b

In [ ]:
```
import numpy as np
from matplotlib import pyplot as plt
import random
import matplotlib.ticker as ticker


n=500
p_list1= np.linspace(0.02,0.09,30)
p_list2= np.linspace(0.1, 1, 50)

p_list = np.concatenate((p_list1, p_list2))

# p_list=[0.05,0.5,0.6]
def compute_average_path_l (a):
    length =0
    for i in range(n):
        for j in range(n):
            if i!=j:
                length+= a[i][j]
    length= length/(n**2-n)
    return length


def analyticalvalue_1(n, p):
    gamma = 0.57722
    numerator = np.log(n) - gamma

    denominator = np.log(p * (n - 1)) if p * (n - 1) > 0 else np.nan

    l = (numerator / denominator) + 0.5
    return l

def analyticalvalue_2(p):
    return 2-p

def check_for_off_diagonal_terms1(a):
    n = len(a)  # Assuming 'n' is defined somewhere before this function is called
    for i in range(n):
        for j in range(n):
            if i != j and a[i][j] == -1:
                return True  # Return True if any off-diagonal element is -1
    return False

def calculate_clustering_coefficient(adjacency_matrix):
    n = len(adjacency_matrix)

    a_cube = np.matmul(np.matmul(adjacency_matrix, adjacency_matrix), adjacency_matrix)
    closed_triangles = np.trace(a_cube)

    degrees = np.sum(adjacency_matrix, axis=0)

    all_triangles = np.sum(np.square(degrees) - degrees)

    clustering_coefficient = closed_triangles / all_triangles if all_triangles > 0 else 0.0

    return clustering_coefficient


average_length_list=[]
c_list=[]
for p in p_list:

    amatrix = np.zeros((n,n))
    for i in range(n):
        for j in range(n):
            if i>j:
                x= random.random()
                if x< p:
                    amatrix[i][j]= 1
                    amatrix[j][i]=1
    degree= np.zeros((n,1))
    c_list.append(calculate_clustering_coefficient(amatrix))
    for i in range(n):
        degree[i,:] = np.sum(amatrix[i,:])

    l = np.full((n,n),-1)
    t=1
    int_a = amatrix

    while check_for_off_diagonal_terms1(l):
        for i in range(n):
            for j in range(i+1,n):
                if amatrix[i,j]!=0:
                    if l[i,j]==l[j,i]==-1:
                        l[i,j]=t
                        l[j,i]=t
```

```
        amatrix=np.dot(amatrix,int_a)
        t+=1
        # print(p, t)
    average_length_list.append(compute_average_path_l(l))
dark_yellow = (0.8, 0.6, 0)

p_ana= np.linspace(0,1,100)
analyticalvalue_1_list=[]
analyticalvalue_2_list=[]
for p in p_ana:
    analyticalvalue_1_list.append(analyticalvalue_1(n,p))
    analyticalvalue_2_list.append(analyticalvalue_2(p))

plt.figure(figsize=(8, 6))
plt.scatter(p_list, average_length_list, label='numerical', color='blue')
plt.loglog(p_ana,analyticalvalue_1_list, label='analytical Solution 1', color= 'green')
plt.loglog(p_ana, analyticalvalue_2_list, label= 'analytical Solution 2', color=dark_yellow)
plt.legend()
# plt.xlim(0.01, 1)
# plt.ylim(0.01,3.9)
plt.xlabel('p')
plt.ylabel('Average path length l')
plt.title('Average path length l as function of p')
plt.show()

plt.scatter(p_list,c_list, label='Numerical')
plt.plot(p_list, p_list, label='p', linestyle='dashed', color='black')
plt.legend()
plt.xlabel('p')
plt.ylabel('Clustering co efficient C')
plt.title('Clustering coefficient as a function of p')
plt.show()
```



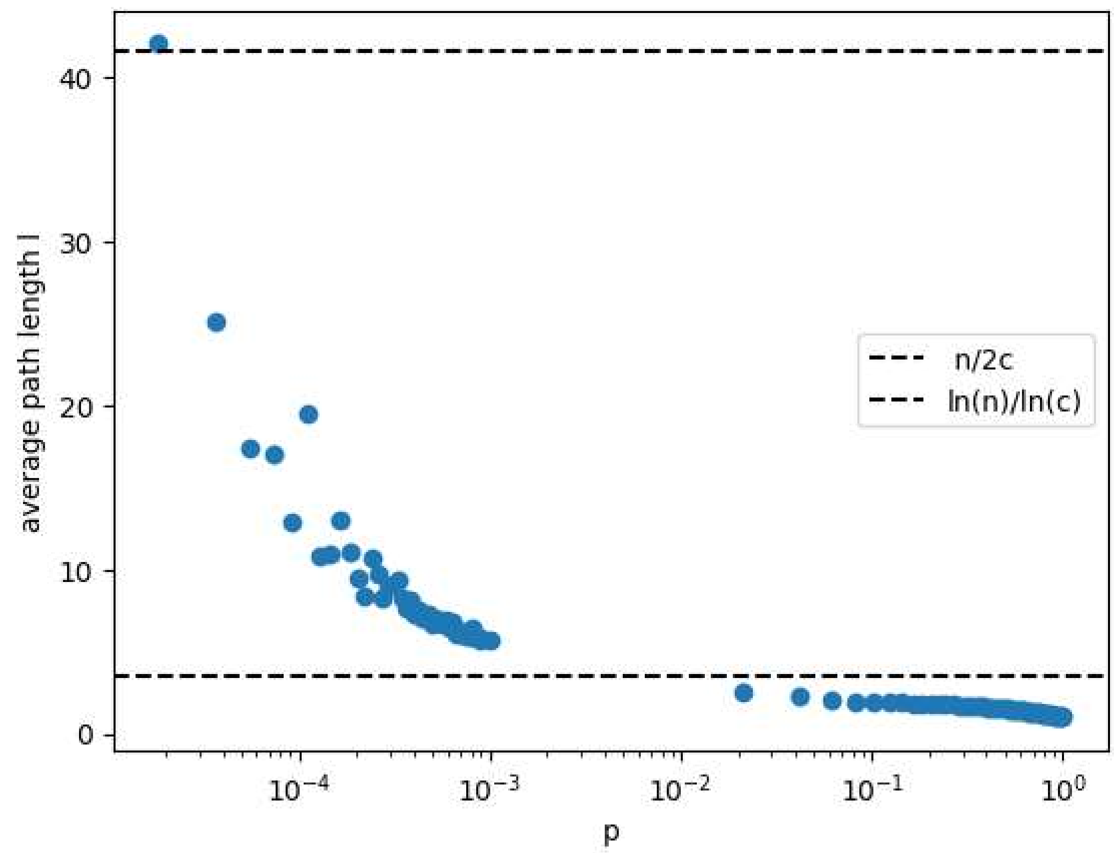

Clustering coefficient as a function of p
Numerical
p
Clustering co efficient C
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
p

```
In [ ]:
```

12.5 a Average path length and clustering coefficient of Watts–Strogatz small-world graphs

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

p_list1= np.linspace(0,0.0009,50)
p_list2= np.linspace(0.001, 1, 50)

p_list = np.concatenate((p_list1, p_list2))
# p_list = np.linspace(0.1,1,10)

def check_for_off_diagonal_terms1(a):
    n = len(a)
    for i in range(n):
        for j in range(n):
            if i != j and a[i][j] == -1:
                return True
    return False

def compute_average_path_l (a):
    length =0
    for i in range(n):
        for j in range(n):
            if i!=j:
                length+= a[i][j]
    length= length/(n**2-n)
    return length

# def analyticalvalue_1()
def watts_strogatz_graph(n, k, p):
    A = np.zeros((n,n))
    for i in range(n):
        for j in range(i+1,n):
            if np.random.rand() < p:
                A[i,j] = 1
                A[j,i] = 1
        # do the nearest neighbour connections
        for b in range(c):
            # c describes how many connections in total, so we need to divide by 2 to get the number of connections p
            A[i,(i+int(b/2)+1)%n] = 1
            A[(i+int(b/2)+1)%n,i] = 1

    return A

def calculate_clustering_coefficient(adjacency_matrix):
    n = len(adjacency_matrix)

    a_cube = np.matmul(np.matmul(adjacency_matrix, adjacency_matrix), adjacency_matrix)
    closed_triangles = np.trace(a_cube)

    degrees = np.sum(adjacency_matrix, axis=0)

    all_triangles = np.sum(np.square(degrees) - degrees)

    clustering_coefficient = closed_triangles / all_triangles if all_triangles > 0 else 0.0

    return clustering_coefficient


n=500
c=6
average_length_list=[]
c_list=[]
for p in p_list:
    amatrix = watts_strogatz_graph(n, c, p)
    c_list.append(calculate_clustering_coefficient(amatrix))
    # print(t,p)
    l = np.full((n,n),-1)
    t=1
    int_a = amatrix

    while check_for_off_diagonal_terms1(l):
        for i in range(n):
            for j in range(i,n):
                if amatrix[i,j]!=0:
                    if l[i,j]==l[j,i]==-1:
                        l[i,j]=t
                        l[j,i]=t


        amatrix=np.matmul(amatrix,int_a)
        t+=1

        # print(p, t , compute_average_path_l(l))
    average_length_list.append(compute_average_path_l(l))
# print(average_length_list)
# print(c_list)
plt.scatter(p_list,average_length_list)
```

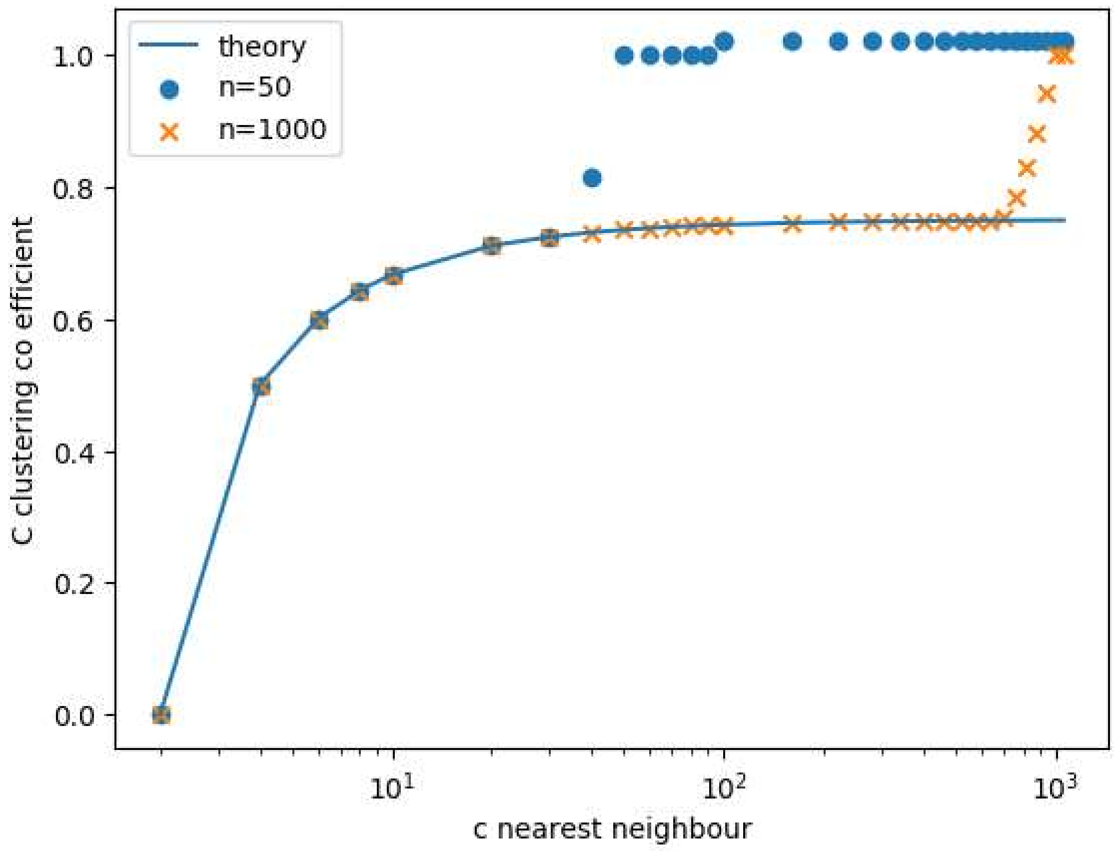
```
plt.xscale('log')
# plt.yscale('log')
plt.axhline(n/(2*c), label =' n/2c', color= 'black', linestyle ='dashed')
plt.axhline(np.log(n)/np.log(c), label= 'ln(n)/ln(c)' , color='black', linestyle ='dashed')
plt.legend()
plt.xlabel('p')
plt.ylabel('average path length l')
plt.show()


# analyticalvalue_1_list=[]
# analyticalvalue_2_list=[]
# for p in p_ana:
#     analyticalvalue_1_list.append(analyticalvalue_1(n,p))
#     analyticalvalue_2_list.append(analyticalvalue_2(p))
```



In [ ]:

```
n_list=[50,1000]
c=6
# c_values= np.linspace(0,1000,10000)

cs_1 = np.array([2,4,6,8,10,20,30,40,50,60,70,80,90,100])

cs_2 = np.arange(start=100, stop=1100, step=60, dtype=int)
c_values = np.concatenate((cs_1,cs_2))
# for n in n_list:
c_list=[]
p=0
for n in n_list:
    temp=[]
    for c in c_values:
        amatrix = watts_strogatz_graph(n, c, p)
        temp.append(calculate_clustering_coefficient(amatrix))
    c_list.append(temp)

c_theory=[]
for c in c_values:
    numerator =3*(c-2)
    denominator = 4*(c-1)
    c_theory.append(numerator/denominator)
    # c_theory.append(c/(n-1))

print(len(c_values))
print(len(c_list[0]))

plt.semilogx(c_values,c_theory, label='theory')
plt.scatter(c_values, c_list[0], label= f'n={n_list[0]}' )
plt.scatter(c_values, c_list[1], label= f'n={n_list[1]}' ,marker='x')
# plt.xlim(1,1000)
plt.legend()
plt.xlabel('c nearest neighbour')
plt.ylabel('C clustering co efficient')
plt.show()
```

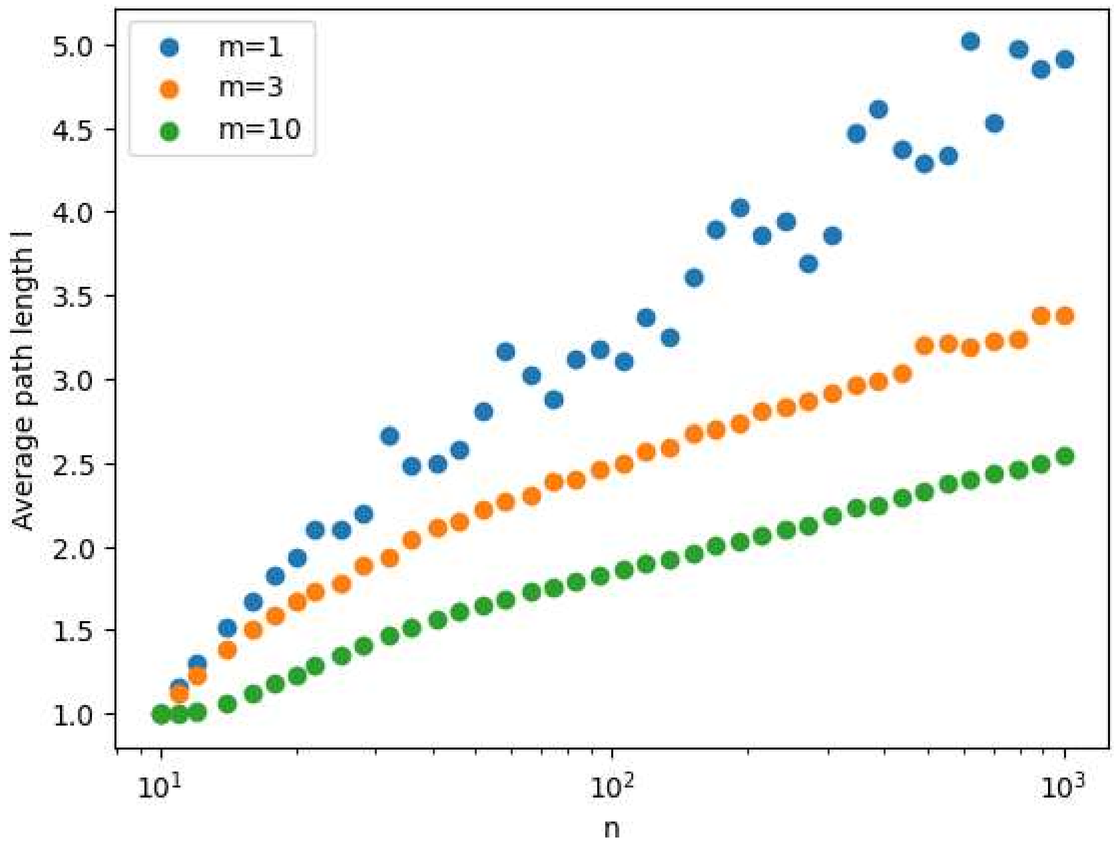
```
31
31
```


theory
n=50
n=1000
1.0
0.8
0.6
0.4
0.2
0.0
C clustering co efficient
10^1
10^2
10^3
c nearest neighbour

12.6 Average path length Albert–Barabási preferential-growth graphs

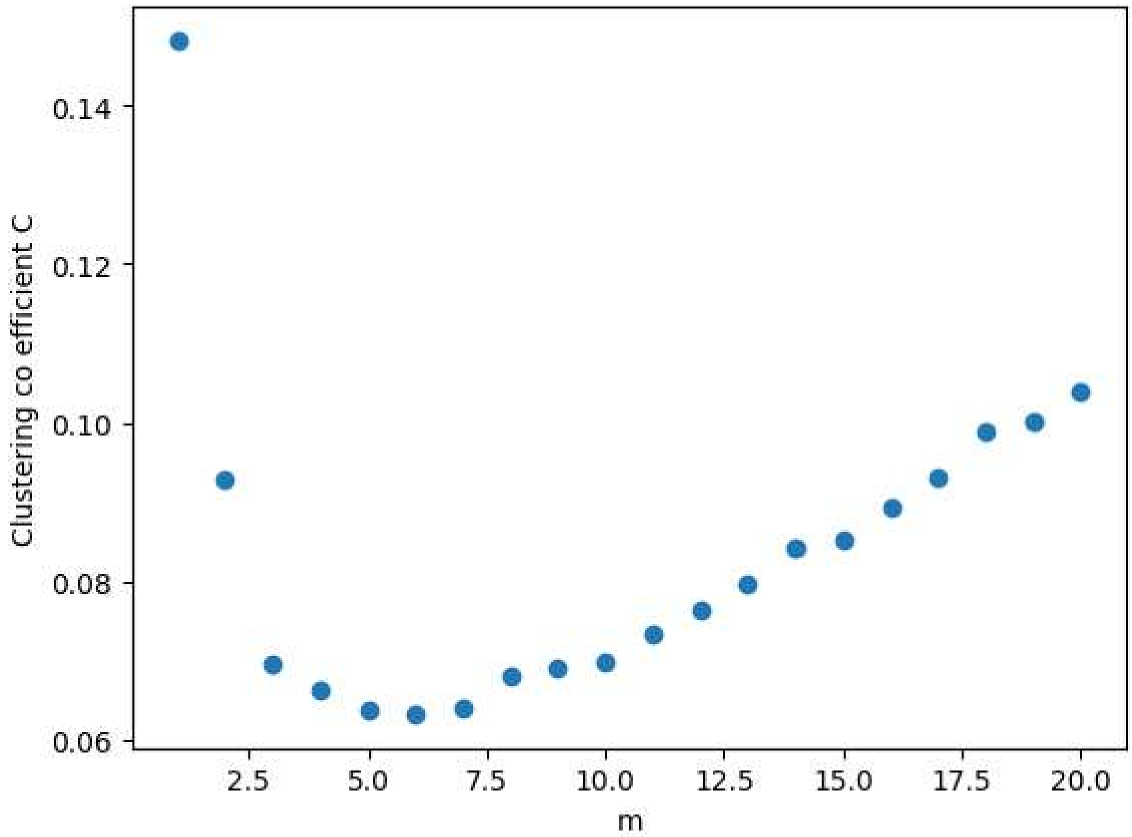
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import random

n0=10
# n=1000
n_list = np.logspace(1,3,40, dtype=int)

m_list=[1,3,10]
r=100


amatrix= np.ones((n0,n0))
np.fill_diagonal(amatrix, 0)

def find_degree (matrix):
    degree= np.zeros((len(matrix),1))
    for i in range(len(matrix)):
        degree[i,:] = np.sum(matrix[i,:])
    total_degree = np.sum(degree)
    cumulative_sum = np.cumsum(degree)
    prob = cumulative_sum/total_degree
    return (prob)
def compute_average_path_l (a):
    length =0
    for i in range(n):
        for j in range(n):
            if i!=j:
                length+= a[i][j]
    length= length/(n**2-n)
    return length

def check_for_off_diagonal_terms1(a):
    n = len(a)
    for i in range(n):
        for j in range(n):
            if i != j and a[i][j] == -1:
                return True
    return False
for m in m_list:
    average_length_list=[]
    for n in n_list:
        amatrix= np.ones((n0,n0))
        np.fill_diagonal(amatrix, 0)
        for i in range(1,n-n0+1):
            new_matrix = np.zeros((n0+i, n0+i))
            new_matrix[:-1,:-1] = amatrix
            new_edge =0
            previous_list=[]
            while new_edge<m:
                prob = find_degree(amatrix)
                random_number = np.random.uniform(0,1)
                for a in range(len(prob)):
                    if prob[a]>=random_number and a not in previous_list:

                        new_matrix[n0+i-1][a]=1
                        new_matrix[a][n0+i-1]=1
                        new_edge+=1
                        previous_list.append(a)
                        break

            amatrix = new_matrix
        l = np.full((n,n),-1)
        t=1
        int_a = amatrix

        while check_for_off_diagonal_terms1(l):
            for i in range(n):
                for j in range(i+1,n):
                    if amatrix[i,j]!=0:
                        if l[i,j]==l[j,i]==-1:
                            l[i,j]=t
                            l[j,i]=t


            amatrix=np.dot(amatrix,int_a)
            t+=1

        average_length_list.append(compute_average_path_l(l))

    plt.scatter(n_list, average_length_list, label=f'm={m}')
plt.xscale('log')
plt.xlabel('n')
plt.ylabel('Average path length l')
plt.legend()
plt.show()
```

```
# print(amatrix)

# print(compute_average_path_l)
```



12.6.b Clustering co efficient for Albert–Barabási graphs

In [ ]:
```
n0=20
m_values = np.arange(1, 21)
n =1000
c_list=[]
def calculate_clustering_coefficient(adjacency_matrix):
    n = len(adjacency_matrix)

    a_cube = np.matmul(np.matmul(adjacency_matrix, adjacency_matrix), adjacency_matrix)
    closed_triangles = np.trace(a_cube)

    degrees = np.sum(adjacency_matrix, axis=0)

    all_triangles = np.sum(np.square(degrees) - degrees)

    clustering_coefficient = closed_triangles / all_triangles if all_triangles > 0 else 0.0

    return clustering_coefficient

for m in m_values:
    # print(m)
    amatrix= np.ones((n0,n0))
    np.fill_diagonal(amatrix, 0)
    for i in range(1,n-n0+1):
        new_matrix = np.zeros((n0+i, n0+i))
        new_matrix[:-1,:-1] = amatrix
        new_edge =0
        previous_list=[]
        while new_edge<m:
            prob = find_degree(amatrix)
            random_number = np.random.uniform(0,1)
            for a in range(len(prob)):
                if prob[a]>=random_number and a not in previous_list:

                    new_matrix[n0+i-1][a]=1
                    new_matrix[a][n0+i-1]=1
                    new_edge+=1
                    previous_list.append(a)
                    break

        amatrix = new_matrix
    c_list.append(calculate_clustering_coefficient(amatrix))

plt.scatter(m_values, c_list)
plt.xlabel('m')
plt.ylabel('Clustering co efficient C')
plt.show()

# print(amatrix)
```


0.14
0.12
0.10
0.08
0.06
Clustering co efficient C
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
m