Week 4

We simulate processes many times to estimate relevant quantities. This week, we look at how the structure of the network interacts with the dynamics of the spreading process.

Just as how k_i is a simple measure of the structural centrality of a node, $x_i = 1 - s_i$ is a simple measure of dynamical centrality. If x_i is relatively large, node i is relatively central in the network from the perspective of the disease.

In [ ]:
```
import numpy as np
import scipy.stats as scistats
import matplotlib.pyplot as plt
from collections import Counter, defaultdict
```

In [ ]:
```
class Network():
    def __init__(self,num_nodes):
        self.adj = {i:set() for i in range(num_nodes)}
        self.num_edge = 0
        self.num_nodes = num_nodes

    def add_edge(self,i,j):
        self.adj[i].add(j)
        self.adj[j].add(i)
        self.num_edge+=1

    def neighbors (self,i):
        return self.adj[i]

    def edge_list(self):
        return [(i,j) for i in self.adj for j in self.adj[i] if i<j]
```

In [ ]:
```
class Erdos_renyi_Network(Network):

    def __init__(self, num_nodes, mean):
        super().__init__(num_nodes)

        # Parameter p for a Erdos-renyi Graph
        self.p = mean/(num_nodes-1)

        # Construct Erdos-renyi Graph
        for i in range(num_nodes):
            for j in range(i+1,num_nodes):
                if np.random.random()<self.p:
                    self.add_edge(i,j)

class Poisson_Configuration_Network(Network):

    def __init__(self, num_nodes=10000,mn=20):
        super().__init__(num_nodes)

        k = np.random.poisson(lam=mn,size=num_nodes)
```

```
        S = np.array([i for i in range(num_nodes) for _ in range(k[i])])
        S = np.random.permutation(S)

        if len(S)%2:
            S = S[:-1]

        S = S.reshape(-1,2)

        for i,j in S:
            if i!=j:
                self.add_edge(i,j)

class Geometric_Configuration_Network(Network):

    def __init__(self, num_nodes=10000, mn=20):
        super().__init__(num_nodes)

        k = np.random.geometric(1/(mn+1), size=num_nodes)

        S = np.array([i for i in range(num_nodes) for _ in range(k[i]-1)])
        S = np.random.permutation(S)

        if len(S)%2:
            S = S[:-1]

        S = S.reshape(-1,2)

        for i,j in S:
            if i!=j:
                self.add_edge(i,j)
```

In [ ]:

```
class SIR_Model():

    def __init__(self, network, p_infected,p_infect):

        self.p_infected = p_infected
        self.p_infect = p_infect
        self.network = network

        # SIR nodes
        self.S = {node for node in range(self.network.num_nodes)}
        self.I = set()
        self.R = set()

        # Initially infect a small fraction of the population
        self.I.update(np.random.choice(list(self.S), size=int(self.p_infected*self.
        self.S.difference_update(self.I)

    def run(self):
        '''Runs simulation for a cycle'''

        new_I = set()
        for node in self.I:
            for adj in self.network.neighbors(node):
                if adj in self.S and np.random.random()<self.p_infect:
```

```
                    new_I.add(adj)

            self.R.add(node)

        self.I.difference_update(self.R)
        self.I.update(new_I)
        self.S.difference_update(self.I)


    def run_to_extinction(self):
        '''Runs simulation until extinction, then returns time series of SIR number

        S_list, I_list, R_list = [len(self.S)], [len(self.I)], [len(self.R)]

        while self.I:

            self.run()

            S_list.append(len(self.S))
            I_list.append(len(self.I))
            R_list.append(len(self.R))

        return S_list, I_list, R_list
```

In [ ]:

```
class SIRV_Model():

    def __init__(self, network, p_infected,p_infect,p_vaccinated):

        self.p_infected = p_infected
        self.p_infect = p_infect
        self.p_vaccinated = p_vaccinated
        self.network = network

        # SIR nodes
        self.S = {node for node in range(self.network.num_nodes)}
        self.I = set()
        self.R = set()
        self.V = set()

        # Initially vaccinated fraction of the population
        self.V.update(np.random.choice(list(self.S), size=int(self.p_vaccinated*sel

        # Then infect a small fraction of the whole population
        self.I.update(np.random.choice(list(self.S), size=int(self.p_infected*self.

        self.S.difference_update(self.V)
        self.I.difference_update(self.V)
        self.S.difference_update(self.I)


    def run(self):
        '''Runs simulation for a cycle'''

        new_I = set()
        for node in self.I:
            for adj in self.network.neighbors(node):
```

```
                if adj in self.S and np.random.random()<self.p_infect:
                    new_I.add(adj)

            self.R.add(node)

        self.I.difference_update(self.R)
        self.I.update(new_I)
        self.S.difference_update(self.I)


    def run_to_extinction(self):
        '''Runs simulation until extinction, then returns time series of SIR number

        S_list, I_list, R_list = [len(self.S)], [len(self.I)], [len(self.R)]

        while self.I:

            self.run()

            S_list.append(len(self.S))
            I_list.append(len(self.I))
            R_list.append(len(self.R))

        return S_list, I_list, R_list
```

In [ ]:

```
class Efficient_SIRV_Model():

    def __init__(self, network, p_infected,p_infect,p_vaccinated):

        self.p_infected = p_infected
        self.p_infect = p_infect
        self.p_vaccinated = p_vaccinated
        self.network = network

        # SIR nodes
        self.S = {node for node in range(self.network.num_nodes)}
        self.I = set()
        self.R = set()
        self.V = set()

        # Initially vaccinated friend of a fraction of the population
        while len(self.V)<int(self.p_vaccinated*self.network.num_nodes):
            node = np.random.randint(self.network.num_nodes)
            if self.network.adj[node]:
                self.V.add(np.random.choice(list(self.network.adj[node])))

        # Then infect a small fraction of the whole population
        self.I.update(np.random.choice(list(self.S), size=int(self.p_infected*self.

        self.S.difference_update(self.V)
        self.I.difference_update(self.V)
        self.S.difference_update(self.I)


    def run(self):
        '''Runs simulation for a cycle'''
```

```
        new_I = set()
        for node in self.I:
            for adj in self.network.neighbors(node):
                if adj in self.S and np.random.random()<self.p_infect:
                    new_I.add(adj)

            self.R.add(node)

        self.I.difference_update(self.R)
        self.I.update(new_I)
        self.S.difference_update(self.I)


    def run_to_extinction(self):
        '''Runs simulation until extinction, then returns time series of SIR number

        S_list, I_list, R_list = [len(self.S)], [len(self.I)], [len(self.R)]

        while self.I:

            self.run()

            S_list.append(len(self.S))
            I_list.append(len(self.I))
            R_list.append(len(self.R))

        return S_list, I_list, R_list
```

Q1

Write code to estimate vector x_i, the probability that node i becomes infected at some point for given λ.

Noting that $x_i = 1 - s_i$

In [ ]:
```
def simulate_xi(network,p_infect, p_infected=0.01, runs=100):
    '''Runs simulations for large number of times and return vector of node i being
    R_history = []

    for _ in range(runs):
        simulation = SIR_Model(network,p_infected, p_infect)
        S,I,R = simulation.run_to_extinction()
        R_history.extend(simulation.R)

    # Return vector proportion of times node i has been infected
    return [Counter(R_history)[i]/runs for i in range(network.num_nodes)]
```

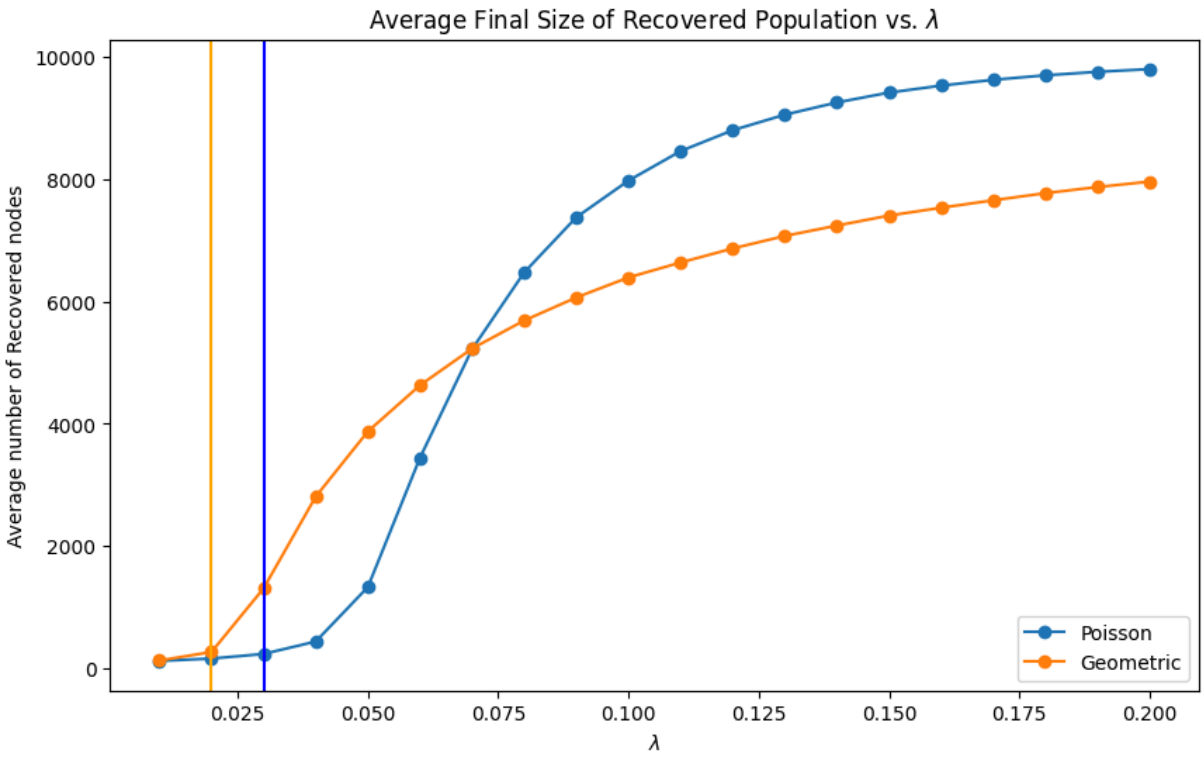
EXTRA

Trying to find critical value for poisson and geometric network

In [ ]:
```
def average_R_vs_lambda(network, p_infect_array, p_infected=0.01, runs=50):
    avg_R = []

    for p_infect in p_infect_array:
        R_values = []
        for _ in range(runs):
            simulation = SIR_Model(network,p_infected, p_infect)
            S,I,R = simulation.run_to_extinction()
            R_values.append(R[-1])
        avg_R.append(np.mean(R_values))

    return avg_R
```

In [ ]:
```
n = 10000
k = 20
p_infected=0.01
p_infect_array = np.linspace(0.01,0.2,20)

p_network = Poisson_Configuration_Network(n,k)
g_network = Geometric_Configuration_Network(n,k)

avg_p_R = average_R_vs_lambda(p_network, p_infect_array)
avg_g_R = average_R_vs_lambda(g_network, p_infect_array)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(p_infect_array,avg_p_R, marker='o',label='Poisson')
plt.plot(p_infect_array, avg_g_R, marker='o', label='Geometric')
plt.axvline(0.02,color='orange')
plt.axvline(0.03,color='blue')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average number of Recovered nodes')
plt.title(r'Average Final Size of Recovered Population vs. $\lambda$')
plt.show()
```



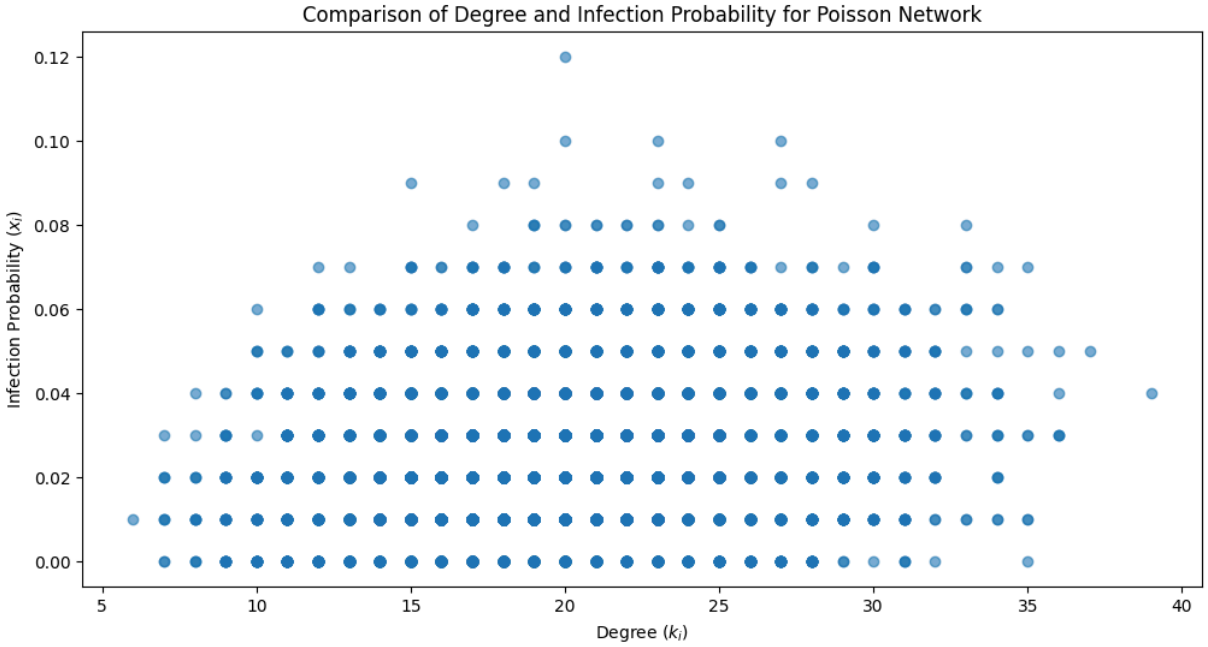
So for Geom $\lambda* \approx 0.02$ and for Poisson $\lambda* \approx 0.03$

Q2

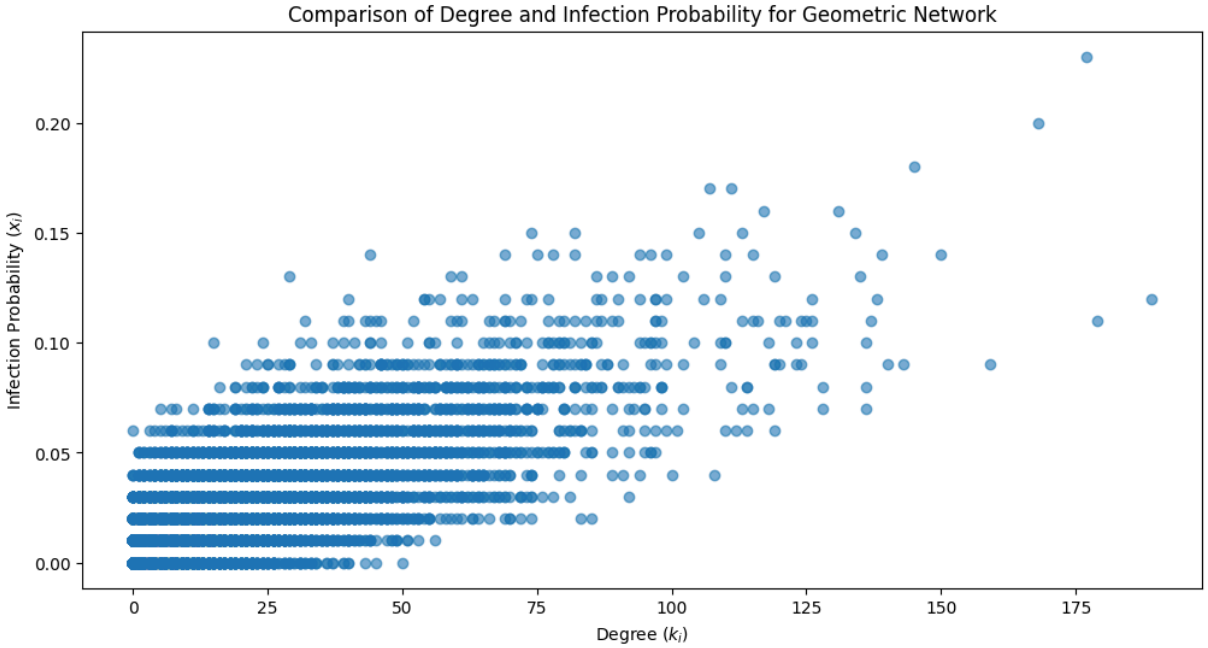
For a couple of large networks - one with Poisson, one with Geometric distribution, compare x_i to the degree k_i.

Keep λ around the epidemic threshold.

In [ ]:
```
n = 10000
k = 20
p_infected=0.01
p_infect=0.03

p_network = Poisson_Configuration_Network(n,k)
p_simulation = SIR_Model(p_network,p_infected,p_infect)

k_i = [len(p_network.adj[i]) for i in range(p_network.num_nodes)]
x_i = simulate_xi(p_network,p_infect,p_infected)

plt.figure(figsize=(12, 6))
plt.scatter(k_i, x_i, alpha=0.6)
plt.xlabel(r'Degree ($k_{i}$)')
plt.ylabel(r'Infection Probability ($x_{i}$)')
plt.title('Comparison of Degree and Infection Probability for Poisson Network')
plt.show()
```



In [ ]:
```
p_infect = 0.02

g_network = Geometric_Configuration_Network(n,k)
simulation = SIR_Model(g_network,p_infected,p_infect)

k_i = [len(g_network.adj[i]) for i in range(g_network.num_nodes)]
x_i = simulate_xi(g_network,p_infect,p_infected)

plt.figure(figsize=(12, 6))
plt.scatter(k_i, x_i, alpha=0.6)
plt.xlabel(r'Degree ($k_{i}$)')
plt.ylabel(r'Infection Probability ($x_{i}$)')
plt.title('Comparison of Degree and Infection Probability for Geometric Network')
plt.show()
```



Q3

A randomly chosen node will become infected with probability

$$< x >= \frac{1}{n}\sum_{i=1}^{n} x_i$$

However, a randomly chosen neighbour will become infected with probability

$$\frac{1}{n}\sum_{i,j}\frac{A_{ij}x_j}{k_i} =< \frac{Ax}{k} >$$

Let us denote the above $< y >$

This is known as the disease paradox: your friends are more likely to have a disease than you do.

1. Combine simulation and analytic arguments as you see fit.
2. Is the issue more or less pronounce on networks with larger variance in the degree distribution ?

Hint : For configuration network, variance = mean = λ for a Poisson Network.

For Geometric Network of parameter p, mean = k = $\frac{1-p}{p}$ and variance = $\frac{1-p}{p^2}$

So for geom $p = \frac{1}{k+1}$ and $var = k(k+1)$

Analytic argument:

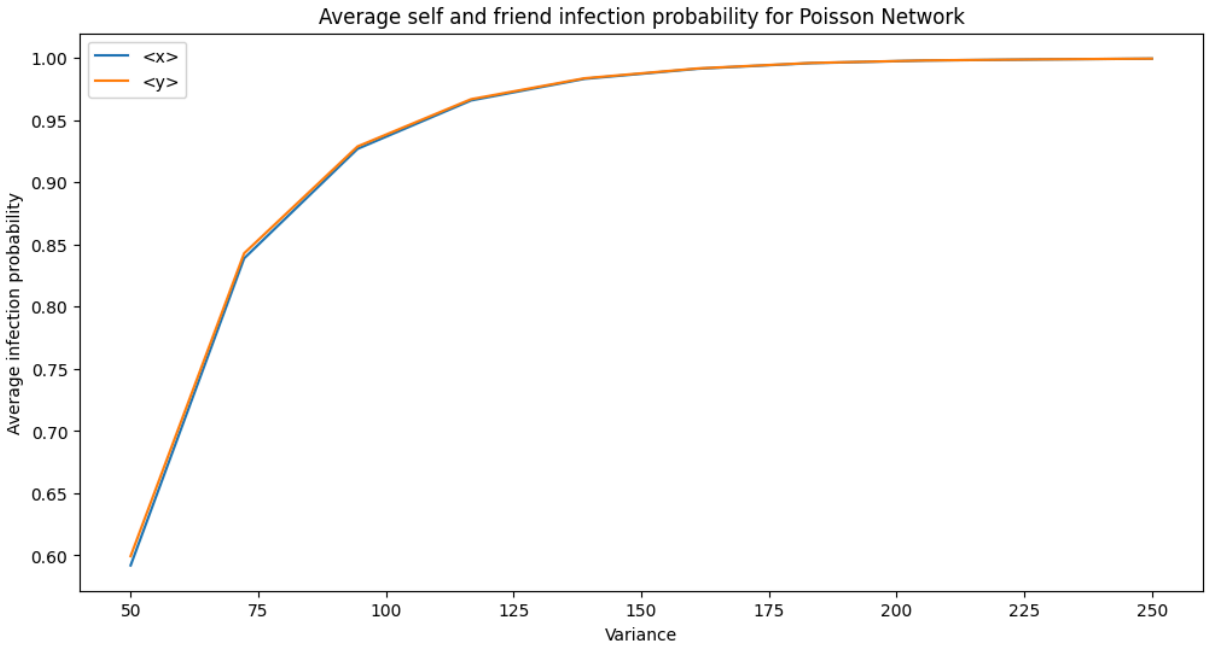
Show that

$$< \frac{Ax}{k} \geq < x >$$

In [ ]:
```
def simulate_x_prob(network,p_infect, p_infected=0.01, runs=50):
    '''Runs simulations and computes mean probality of a node being infected'''
    infection_counts = np.zeros(network.num_nodes)

    for _ in range(runs):
        simulation = SIR_Model(network,p_infected, p_infect)
        simulation.run_to_extinction()
        for node in simulation.R:
            infection_counts[node]+=1

    return infection_counts/runs

def disease_paradox(network,xi_vect):
    '''Takes simulated xi_vect and return <x>,<y>'''
```

```
    Ax_k_vect = []
    for i in range(network.num_nodes):
        if len(network.adj[i])==0:
            continue
        else:
            Ax_k_vect.append(np.mean([xi_vect[j] for j in network.adj[i]]))

    return np.mean(xi_vect),np.mean(Ax_k_vect)
```

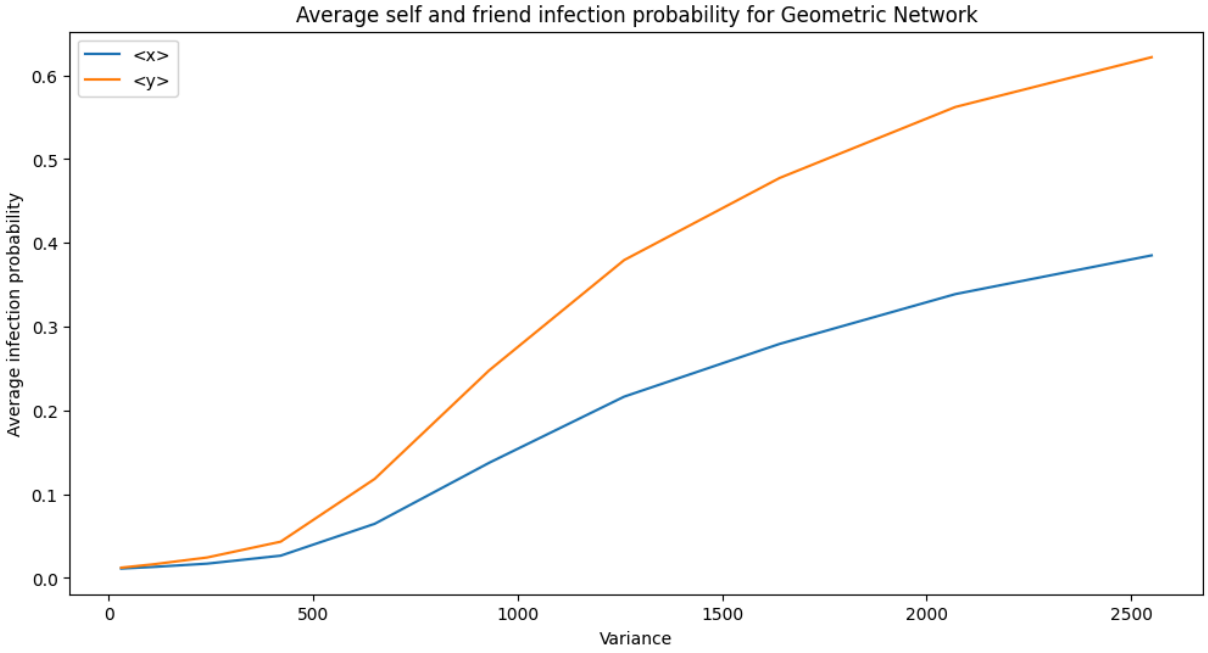
In [ ]:
```
means = np.linspace(50,250,10)

simulated_p_x = []
simulated_p_y = []

for k in means:
    p_network = Poisson_Configuration_Network(n,k)

    xi = simulate_x_prob(p_network,p_infect=0.03)
    x, y = disease_paradox(p_network,xi)
    simulated_p_x.append(x)
    simulated_p_y.append(y)

plt.figure(figsize=(12, 6))
plt.plot(means, simulated_p_x, label='<x>')
plt.plot(means, simulated_p_y, label='<y>')
plt.xlabel('Variance')
plt.ylabel('Average infection probability')
plt.title('Average self and friend infection probability for Poisson Network')
plt.legend()
plt.show()
```



In [ ]:
```
means = np.linspace(5,50,10)

simulated_g_x = []
simulated_g_y = []
```

```
for k in means:
    g_network = Geometric_Configuration_Network(n,k)

    xi = simulate_x_prob(g_network,p_infect=0.02)
    x, y = disease_paradox(g_network,xi)
    simulated_g_x.append(x)
    simulated_g_y.append(y)


var = np.multiply(means , means+1)
```

In [ ]:
```
plt.figure(figsize=(12, 6))
plt.plot(var, simulated_g_x, label='<x>')
plt.plot(var, simulated_g_y, label='<y>')
plt.xlabel('Variance')
plt.ylabel('Average infection probability')
plt.title('Average self and friend infection probability for Geometric Network')
plt.legend()
plt.show()
```



We can observe the disease paradox in both types of network. The issue is more pronounce on networks with larger variance in the degree distribution for the Geometric Network compared to the Poisson case.

Q4 Spreading of disease as a function of λ and vaccination

Let us denote the proportion of vaccination as V.

When nodes are vaccinated, we can simply remove them from the network.

1. Randomly vaccinate proportions of the population, then run the simulation for a range of λ values and plot the outbreak size against λ. Find epidemic threshold for different vaccination rates.

2. Different protocol to vaccinate. First pick nodes randomly as before, but then pick a random neigbour of the nodes to be vaccinated.

Comment : I expect the protocol from 2. to increase the epidemic threshold. Since according to the disease paradox, mean excess infection is greater than mean infection.

Bonus : Vice versa, you can keep λ constant and investigate when the vaccination rate causes the epidemic to be subcritical.

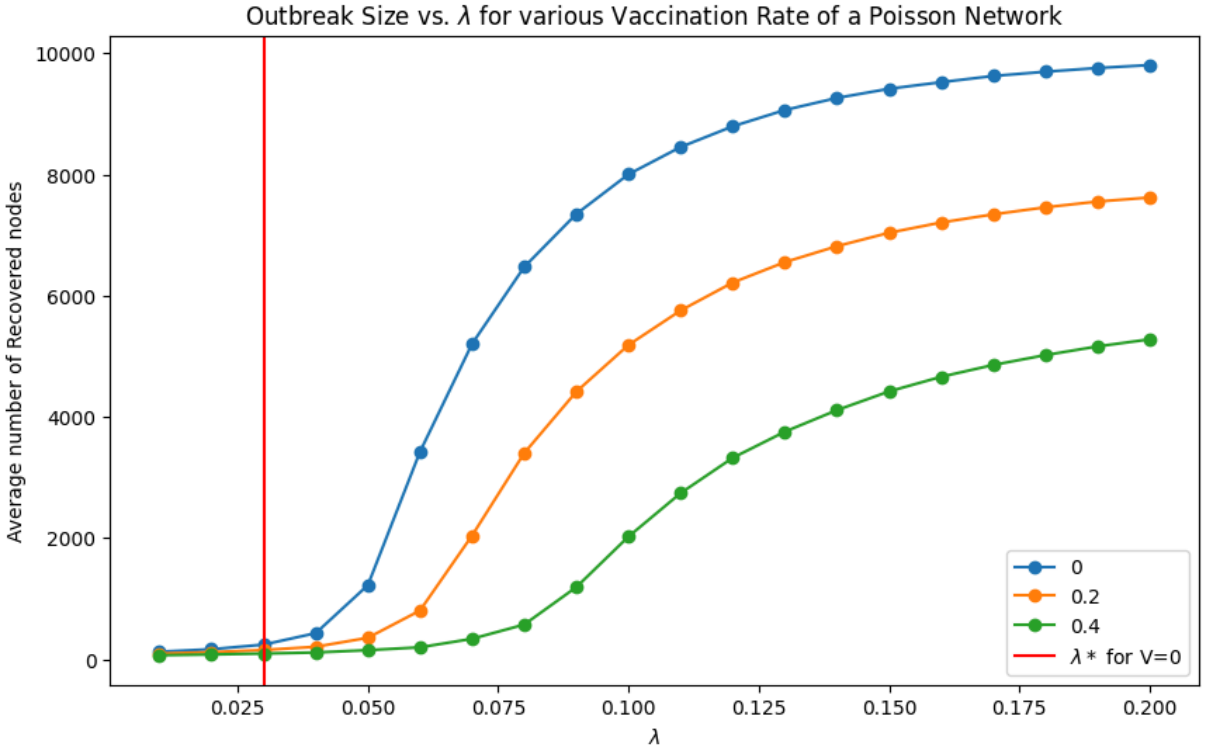
Questions:

1. Do we vaccinate first ? Then cause initial infection?
2. When forming initial infection, do I pick a vaccinated person as part of the random choice? Because if I only pick susceptible subset then proportion is no longer 0.01 of population.

In [ ]:
```
def average_R_vs_lambda_II(network, p_infect_array, p_vaccinated, p_infected=0.01,
    '''Modified to take into account of vaccinations'''
    avg_R = []

    for p_infect in p_infect_array:
        R_values = []
        for _ in range(runs):
            simulation = SIRV_Model(network,p_infected, p_infect,p_vaccinated)
            S,I,R = simulation.run_to_extinction()
            R_values.append(R[-1])
        avg_R.append(np.mean(R_values))

    return avg_R
```

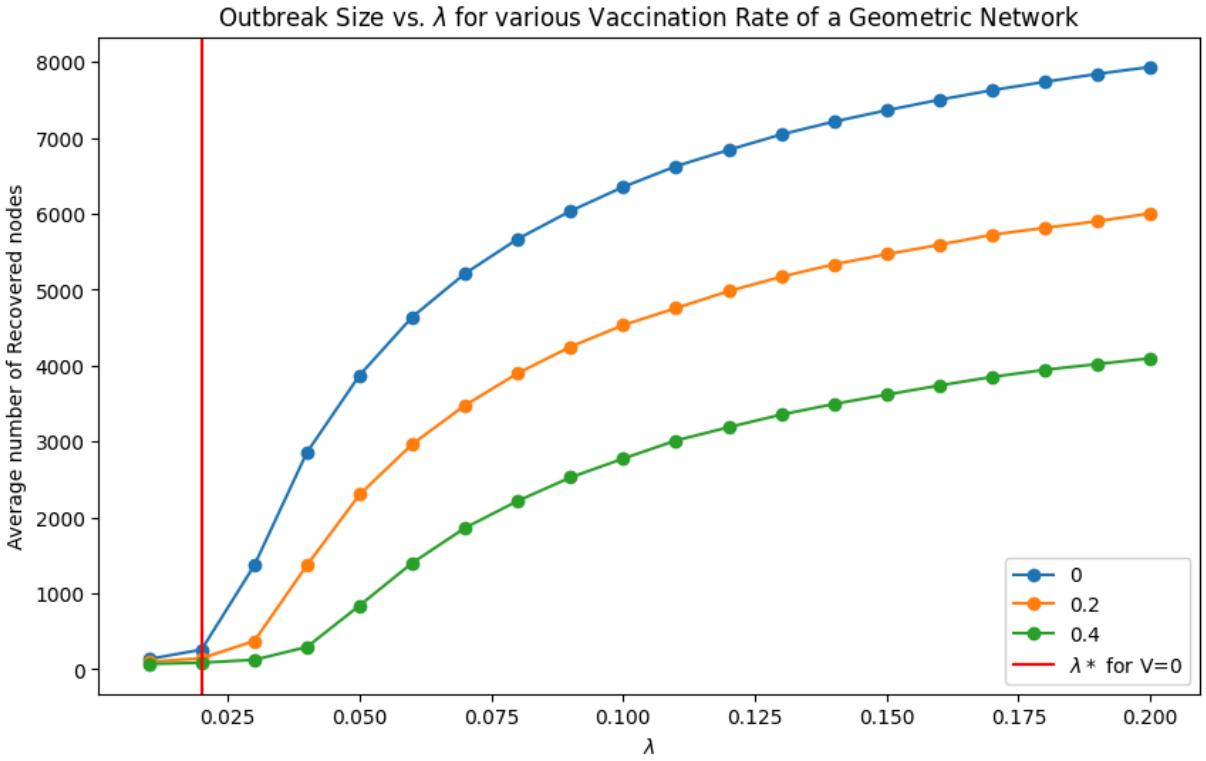
In [ ]:
```
n = 10000
k = 20
vaccinations = [0,0.2,0.4]
p_infect_array=np.linspace(0.01,0.2,20)

p_network = Poisson_Configuration_Network(n,k)
plt.figure(figsize=(10,6))

for V in vaccinations:
    avg_p_R = average_R_vs_lambda_II(p_network, p_infect_array,V)
    plt.plot(p_infect_array,avg_p_R, marker='o',label='{}'.format(V))

plt.axvline(0.03,color='r',label=r'$\lambda*$ for V=0')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average number of Recovered nodes')
plt.title(r'Outbreak Size vs. $\lambda$ for various Vaccination Rate of a Poisson N
```

Out[ ]: Text(0.5, 1.0, 'Outbreak Size vs. $\\lambda$ for various Vaccination Rate of a Poi
sson Network')



In [ ]:
```
g_network = Geometric_Configuration_Network(n,k)
plt.figure(figsize=(10,6))

for V in vaccinations:
    avg_g_R = average_R_vs_lambda_II(g_network, p_infect_array,V)
    plt.plot(p_infect_array,avg_g_R, marker='o',label='{}'.format(V))

plt.axvline(0.02,color='r',label=r'$\lambda*$ for V=0')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average number of Recovered nodes')
plt.title(r'Outbreak Size vs. $\lambda$ for various Vaccination Rate of a Geometric

p_network = Poisson_Configuration_Network(n,k)
plt.figure(figsize=(10,6))
```

Out[ ]: <Figure size 1000x600 with 0 Axes>



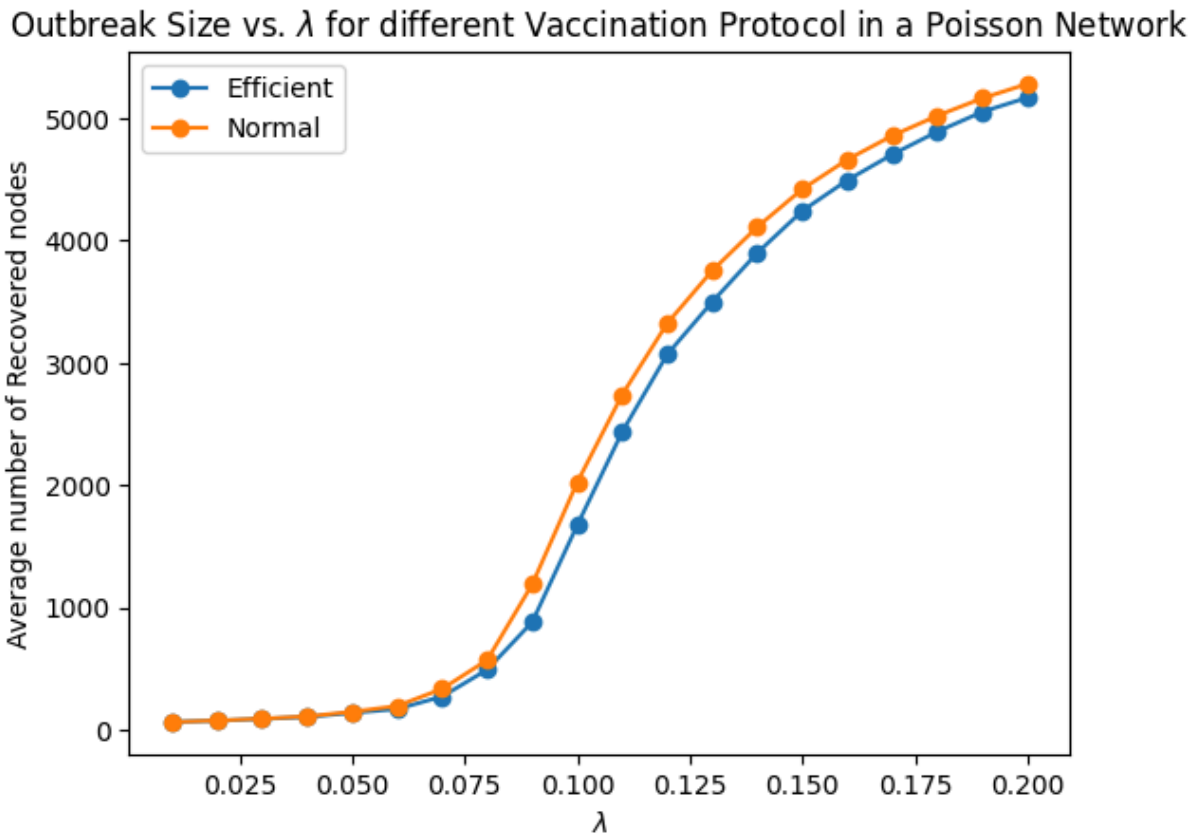

```
<Figure size 1000x600 with 0 Axes>
```

As expected, the onset of a critical dynamic is delayed for larger rates of vaccination.

In [ ]:
```
def average_R_vs_lambda_III(network, p_infect_array, p_vaccinated, p_infected=0.01,
    '''Modified to use the efficient vaccination protocol'''
    avg_R = []

    for p_infect in p_infect_array:
        R_values = []
        for _ in range(runs):
            simulation = Efficient_SIRV_Model(network,p_infected, p_infect,p_vaccin
            S,I,R = simulation.run_to_extinction()
            R_values.append(R[-1])
        avg_R.append(np.mean(R_values))

    return avg_R
```

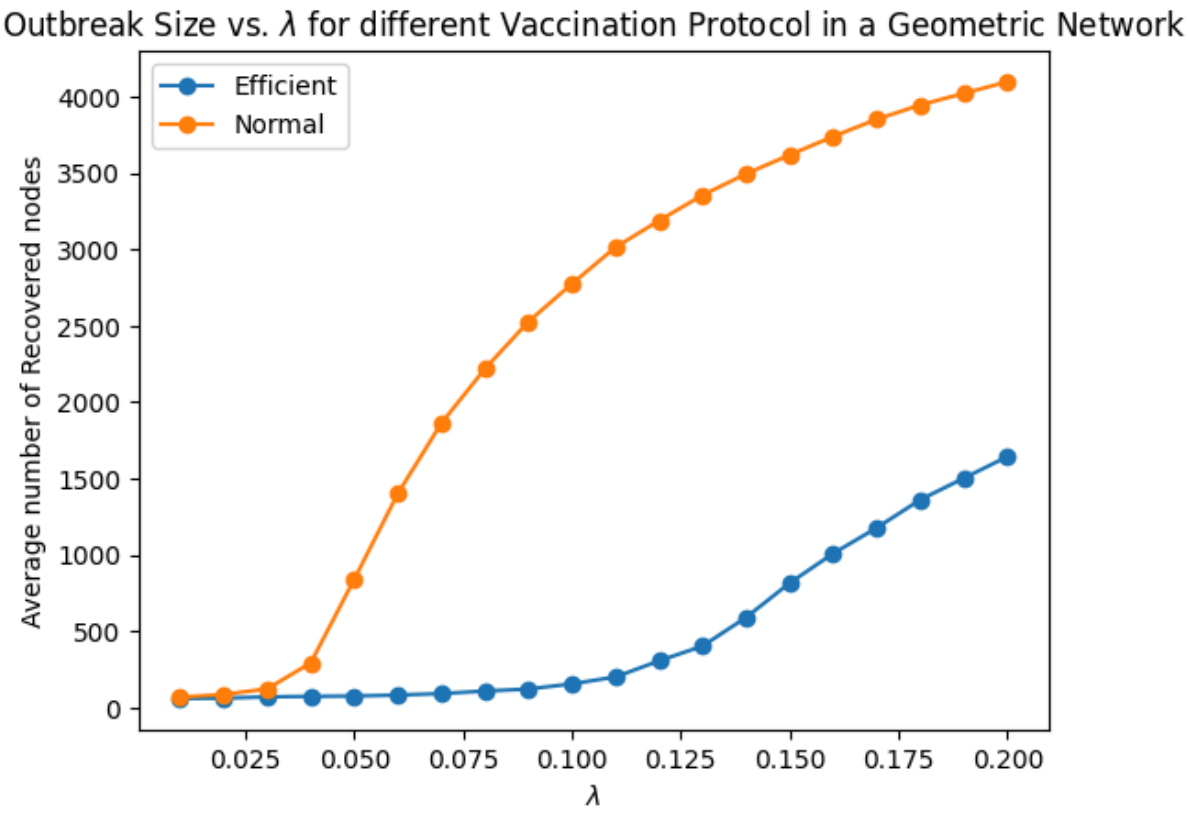
In [ ]:
```
V = 0.4

eff_avg_p_R = average_R_vs_lambda_III(p_network, p_infect_array,V)
plt.plot(p_infect_array,eff_avg_p_R, marker='o',label='Efficient')
plt.plot(p_infect_array,avg_p_R, marker='o', label='Normal')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average number of Recovered nodes')
plt.title(r'Outbreak Size vs. $\lambda$ for different Vaccination Protocol in a Poi
```

Out[ ]:
```
Text(0.5, 1.0, 'Outbreak Size vs. $\\lambda$ for different Vaccination Protocol in
a Poisson Network')
```



In [ ]:
```
eff_avg_g_R = average_R_vs_lambda_III(g_network, p_infect_array,V)
plt.plot(p_infect_array,eff_avg_g_R, marker='o',label='Efficient')
plt.plot(p_infect_array,avg_g_R, marker='o', label='Normal')
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average number of Recovered nodes')
plt.title(r'Outbreak Size vs. $\lambda$ for different Vaccination Protocol in a Geo
```

Out[ ]:
```
Text(0.5, 1.0, 'Outbreak Size vs. $\\lambda$ for different Vaccination Protocol in
a Geometric Network')
```



The more efficient vaccination protocol proved to be very effective for the Geometric Network and not much so for the Poisson Distribution.

How does this relate to the disease paradox?

The disease paradox states that our friends are on average, more likely to be infected than us. Therefore, if we choose to vaccinate the friend of a node, we are targeting the nodes that have higher probability of infecting other nodes as well. Which is why the second protocol is more efficient at stopping the spread of a disease.

TL;DR : Higher xi, higher prob of infecting others, so if we vaccinate those of high xi, disease spread less.

How does this relate to the friendship paradox?

Friendship paradox states that on average, your friends have a higher degree than you. At least for the Geometric distribution, we saw that node degree and infection probability had a positive correlation. There was ~0 correlation between node degree and infection probaility for Poisson Network. This can be observed above, where choosing to vaccinate nodes with higher degrees didn't do much for the Poisson Network.

TL;DR : If higher ki leads to higher xi, then if we vaccinate those of higher ki, disease spread less, else no effect.

Q5

In an unvaccinated population, s_i is the probability that node i is never infected.

$$s_i = \prod_j (1 - \lambda + s_j\lambda)^{A_{ij}}$$

Derive the above for the two vaccination strategies in Q4 then test the approximation against simulations

Simple Vaccination Protocol :

The probability that a node is never infected is either because it is vaccinated, or it is unvaccinated but none of its neighbours ever infect it.

$$s_i = v + (1 - v)\prod_j (1 - \lambda + s_j\lambda)^{A_{ij}}$$

Efficient Vaccination Protocol :

Here the equation is similar to above but v is no longer a constant. Let $v'(i)$ be the probability node i is vaccinated.

A node i is vaccinated if a friend j of the node is chosen and the node itself is chosen from that friend.

$$v'(i) = \sum_j (\frac{v'(j)}{k_j})^{A_{ij}}$$

The equation above can possibly be solved by iteration as well - first, initialise $v_i = v$ where v is the vaccination rate.

Then calculate $< v >$ and we see what happens idk. Sub in v for $< v >$ instead for the equation above to calculate s_i. I expect $< s >$ to be smallest for unvaccinated and greatest for efficient vaccination protocol.

In [ ]:
```
def approx_R_prob_vs_lambda_unvax(network, p_infect_array, tol=1e-6, max_iter=20):
    """Returns approximate outbreak size for unvaccinated population by iterative m
    approx_R = []

    for p_infect in p_infect_array:
        s = np.random.uniform(size=n)

        for _ in range(max_iter):
            s_new = np.ones(network.num_nodes)
            for i in range(n):
                s_new[i]  = np.prod([1+p_infect*(-1+s[j]) for j in network.adj[i]])
            if np.linalg.norm(s_new-s)<tol:
                break
```

```
            s = s_new

        approx_R.append(1-np.mean(s))

    return approx_R

def approx_R_prob_vs_lambda_vax(network, p_infect_array, v, tol=1e-6, max_iter=20):
    """Returns approximate outbreak size for simpled vaccinated population by itera
    approx_R = []

    for p_infect in p_infect_array:
        s = np.random.uniform(size=n)

        for _ in range(max_iter):
            s_new = np.ones(network.num_nodes)
            for i in range(n):
                s_new[i]  = v + (1-v)*np.prod([1+p_infect*(-1+s[j]) for j in networ
                #s_new[i] = np.prod([(1+p_infect*(-1+s[j]))*(1-v)**2 + 2*v - v**2 f

            if np.linalg.norm(s_new-s)<tol:
                break
            s = s_new

        approx_R.append(1-np.mean(s))

    return approx_R

def approx_R_prob_vs_lambda_vax_eff(network, p_infect_array, vi, tol=1e-6, max_iter
    """Returns approximate outbreak size for efficient vaccinated population by ite
    approx_R = []

    for p_infect in p_infect_array:
        s = np.random.uniform(size=n)

        for _ in range(max_iter):
            s_new = np.ones(network.num_nodes)
            for i in range(n):
                #s_new[i] = np.prod([(1+p_infect*(-1+s[j]))*(1-vi[i])**2 + 2*vi[i]
                s_new[i]  = vi[i] + (1-vi[i])*np.prod([1+p_infect*(-1+s[j]) for j i

            if np.linalg.norm(s_new-s)<tol:
                break
            s = s_new

        approx_R.append(1-np.mean(s))

    return approx_R

def approx_vi(network, o_v, tol=1e-6, max_iter=30):
    v = np.ones(network.num_nodes)*o_v

    for _ in range(max_iter):
        v_new = np.ones(network.num_nodes)
        for i in range(network.num_nodes):
            if network.adj[i]:
                v_new[i] = np.sum([v[j]/len(network.adj[j]) for j in network.adj[i]
```

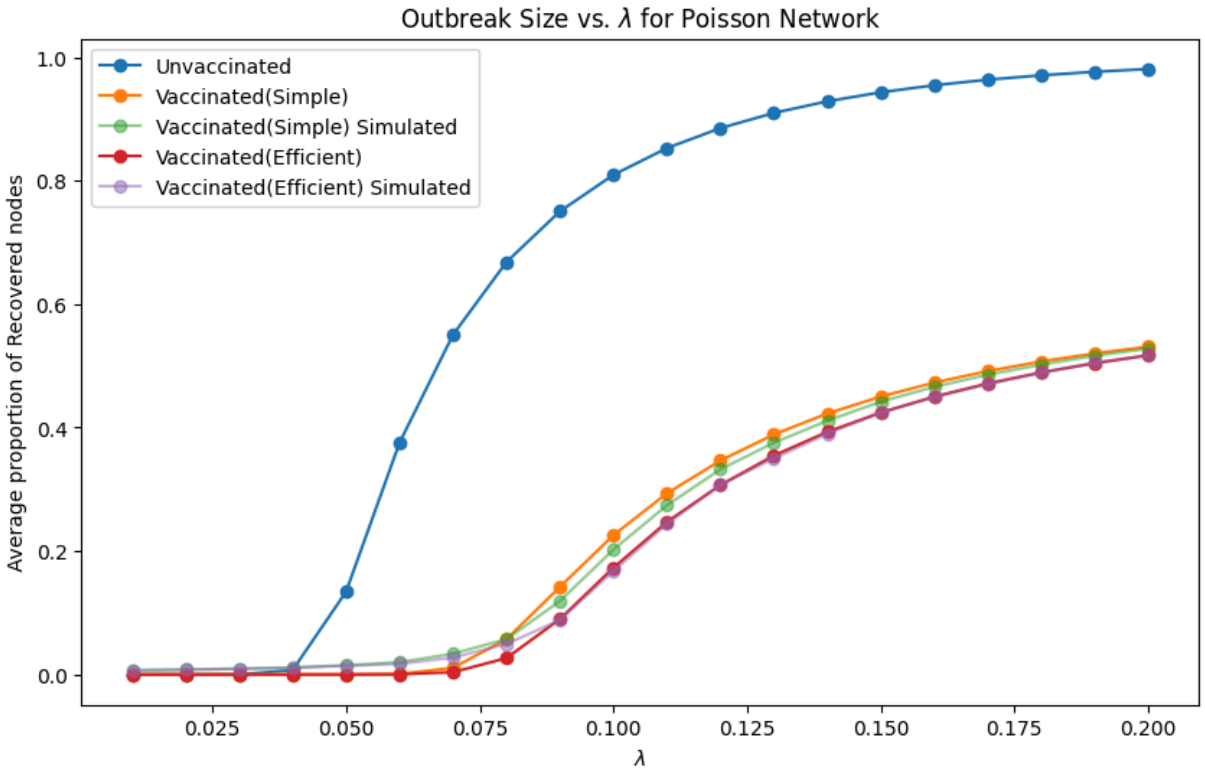
```
            else:
                v_new[i] = 0

        if np.linalg.norm(v_new-v)<tol:
                break
        v = v_new

    return v
```

In [ ]:
```
p_infect_array = np.linspace(0.01,0.2,20)

approx_R_prob_unvax = approx_R_prob_vs_lambda_unvax(p_network,p_infect_array)
```

In [ ]:
```
v=0.4
approx_R_prob_vax = approx_R_prob_vs_lambda_vax(p_network, p_infect_array, v)
```

In [ ]:
```
vi = approx_vi(p_network,v)

approx_R_prob_vax_eff = approx_R_prob_vs_lambda_vax_eff(p_network, p_infect_array,v
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(p_infect_array,approx_R_prob_unvax, marker='o',label='Unvaccinated')
plt.plot(p_infect_array,approx_R_prob_vax, marker='o',label='Vaccinated(Simple)')
plt.plot(p_infect_array,np.array(avg_p_R)/n, marker='o', label='Vaccinated(Simple)
plt.plot(p_infect_array,approx_R_prob_vax_eff, marker='o',label='Vaccinated(Efficie
plt.plot(p_infect_array,np.array(eff_avg_p_R)/n, marker='o',label='Vaccinated(Effic
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average proportion of Recovered nodes')
plt.title(r'Outbreak Size vs. $\lambda$ for Poisson Network')
plt.show()
```



In [ ]:
```
p_infect_array = np.linspace(0.01,0.2,20)

approx_R_prob_unvax = approx_R_prob_vs_lambda_unvax(g_network,p_infect_array)
```

In [ ]:
```
v=0.4
approx_R_prob_vax = approx_R_prob_vs_lambda_vax(g_network, p_infect_array, v)
```

In [ ]:
```
vi = approx_vi(g_network,v)

approx_R_prob_vax_eff = approx_R_prob_vs_lambda_vax_eff(g_network, p_infect_array,v
```

```
c:\Users\Hsin\AppData\Local\Programs\Python\Python39\lib\site-packages\numpy\core\fr
omnumeric.py:86: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
C:\Users\Hsin\AppData\Local\Temp\ipykernel_13360\3827329537.py:54: RuntimeWarning: i
nvalid value encountered in subtract
  if np.linalg.norm(s_new-s)<tol:
c:\Users\Hsin\AppData\Local\Programs\Python\Python39\lib\site-packages\numpy\core\_m
ethods.py:181: RuntimeWarning: invalid value encountered in reduce
  ret = umr_sum(arr, axis, dtype, out, keepdims, where=where)
```

In [ ]:
```
plt.figure(figsize=(10,6))
plt.plot(p_infect_array,approx_R_prob_unvax, marker='o',label='Unvaccinated')
plt.plot(p_infect_array,approx_R_prob_vax, marker='o',label='Vaccinated(Simple)')
plt.plot(p_infect_array,np.array(avg_g_R)/n, marker='o', label='Vaccinated(Simple)
plt.plot(p_infect_array,approx_R_prob_vax_eff, marker='o',label='Vaccinated(Efficie
plt.plot(p_infect_array,np.array(eff_avg_g_R)/n, marker='o',label='Vaccinated(Effic
plt.legend()
plt.xlabel(r'$\lambda$')
plt.ylabel('Average proportion of Recovered nodes')
```

```
plt.title(r'Outbreak Size vs. $\lambda$ for Geometric Network')
plt.show()
```