Network Dynamics and Learning
Homework 3 report

Tommaso Monopoli
s278727

Please refer to the appendix A at the end of this document for all the Python codes used to solve the problems.

1 Preliminary parts

The first part of this assignment is to get familiar with the topics of epidemic simulation on graphs and random graph generation, and serves as a preliminary part to solve later problems.

1.1 Epidemic on a known graph

In this part we need to simulate an epidemic on a given graph $\mathcal{G}$. In particular, the experiment employs a symmetric k-regular undirected graph with n nodes; each node is directly connected to the $k = 4$ nodes whose index is closest to their own modulo n.

The disease propagation model that we use to simulate the epidemic is a discrete-time simplified version of the SIR epidemic model. At any time $t = 0, 1, \dots$ nodes are in a state $X_i(t) \in \mathcal{A} = \{S, I, R\}$, where S indicates a node susceptible to the disease, I an infected one and R one that recovered from the disease.

$X(t)$ can be seen as a discrete-time Markov chain over the configuration space $\mathcal{X} = \{S, I, R\}^{\mathcal{V}}$. The time units $t = 0, 1, \dots$ are weeks: $t = 0$ is the beginning of the first week (week 1), $t = 1$ is the beginning of the second week (week 2) and so on. The transition probabilities P_{XY}, where $X, Y \in \mathcal{X}$, depend on two parameters only, β and ρ, which are described hereafter.

Let $\beta \in [0, 1]$ be the probability that the infection is spread from an I individual to a S one (given that they are connected by a link, i.e. $W_{ij} = 1$, with W being the adjacency matrix of $\mathcal{G}$) during one time step. Assuming that a susceptible node i has m infected neighbors, this means that the probability that individual i does not get infected by any of its neighbors during one time step is $(1 - \beta)^m$. Thus, the probability that individual i becomes infected by any of its neighbors is $1-(1-\beta)^m$. Furthermore, let $\rho \in [0, 1]$ be the probability that an infected individual will recover during one time step. The epidemic is therefore driven by the following transition probabilities:

$$\mathbb{P}(X_i(t+1) = I \,|\, X_i(t) = S, \sum_{j \in \mathcal{V}} W_{ij} \delta^I_{X_j(t)} = m) = 1 - (1 - \beta)^m$$

$$\mathbb{P}(X_i(t+1) = R \,|\, X_i(t) = I) = \rho$$

where $\delta^I_{X_j(t)} = \begin{cases} 1 & \text{if } X_j(t) = I \\ 0 & \text{otherwise} \end{cases}$ and $\sum_{j\in\mathcal{V}} W_{ij}\delta^I_{X_j(t)}$ is the number of infected neighbors of node i. S individuals which do not get infected (I) keep their S state. R individuals keep their state too. I individuals can only keep their state or recover from the infection (R).

We simulate an epidemic on a symmetric k-regular graph with $|\mathcal{V}| = 500$ nodes and $k = 4$, $\beta = 0.3$ and $\rho = 0.7$, for $N = 100$ times. For each simulation, an initial configuration with 10 infected nodes is chosen at random.

At the end of the simulations, we compute:

- the average number of newly infected individuals during each week;
- the average total number of susceptible, infected and recovered individuals at the beginning of each week.[1]

The average is computed over the 100 simulations. The evolution of the required average quantities is plotted in fig. 1.

Note that the plot in fig. 1a should be interpreted as follows: the point of the plot corresponding to week i measures the average number of newly infected individuals *during* week $i-1$, i.e. in a period of time between the beginning of week $i-1$ and the end of week $i-1$ (or, equivalently, the beginning of week i). Moreover, the plot in fig. 1b should be interpreted as follows: the points of the three plots corresponding to week i are a snapshot of the average total number of S/I/R individuals at the beginning of week i.

The Python code used to perform the simulations is reported in appendix A (see function `simulate_epidemics_given_graph`)

1.2 Generate a random graph

In this part we will generate a random graph according to the preferential attachment model, with average degree k (or close to it).

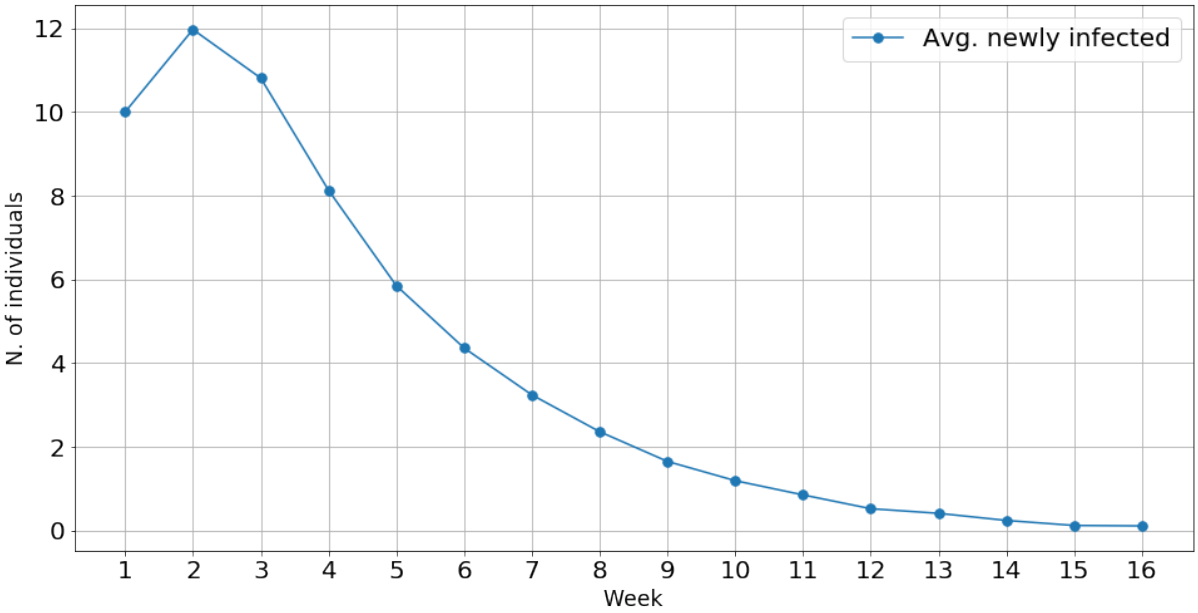
To generate such a graph, at time $t = 1$ we start with an initial complete graph $\mathcal{G}_1$ with $|\mathcal{V}_1| = k+1$ nodes. Then, at every time-step $t \geq 2$ we create a new graph $\mathcal{G}_t$ by adding a new node n_t to $\mathcal{G}_{t-1}$ and connect it to c existing nodes of $\mathcal{G}_{t-1}$, chosen according to the preferential attachment rule. The rule states that the probability that in $\mathcal{G}_t$ there will be a link between node n_t and node $i \in \mathcal{V}_{t-1}$ is:

$$\mathbb{P}(W_{n_t,i}(t) = 1 \,|\, \mathcal{G}_{t-1}) = \frac{w_i(t-1) + a}{\sum_{j\in\mathcal{V}_{t-1}}(w_j(t-1) + a)}$$

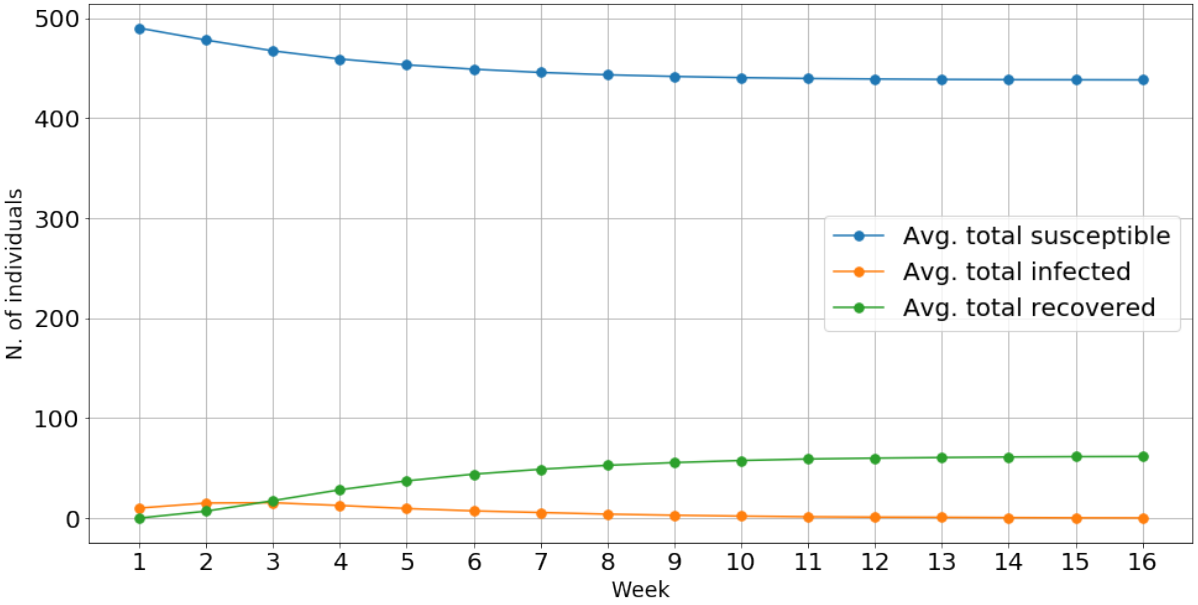
where $w_i(t-1)$ is the degree of node i in $\mathcal{G}_{t-1}$ (i.e. prior to adding the new node) and $W(t)$ is the adjacency matrix for the next time step t.

It can be proved that a preferential attachment random graph with $a = 0$ and fixed c exhibits a degree distribution $p_k(t)$ which converges with probability one to the following

[1] In this problem and the following ones the "avg. tot. infected" is the number of individuals who have state I at the beginning of each week, not the cumulative quantity of people who got infected since week 1; on the other hand, "avg. tot. recovered." is the cumulative quantity of people who recovered from the infection since week 1.



(a)



(b)

Figure 1: Problem 1.1. Evolution of the required average quantities.

power law distribution:

$$\lim_{t\to\infty} p_k(t) = p_k = \begin{cases} 0 & \text{if } k < c \\ \frac{2c(c+1)}{k(k+1)(k+2)} & \text{if } k \geq c \end{cases}$$

This degree distribution has expected value $2c$. [2]

Our aim is to choose c so that the resulting preferential attachment random graph with $a = 0$ will have a given average degree k, if the number of steps (i.e. of nodes to add) is sufficiently high. Therefore, we can impose $c = k/2$. However, to manage the possibility of k being odd, we can choose the value of c as $c = \begin{cases} \lfloor k/2 \rfloor & \text{if } t \text{ is even} \\ \lceil k/2 \rceil & \text{if } t \text{ is odd} \end{cases}$. It can be verified that this choice of c will still generate a random graph with average degree k, for a sufficiently high number of time steps (see code in appendix A)

I implemented a function `generate_preferential_attachment_graph`, which generates a random preferential attachment graph given k, a, the final number of nodes n and a boolean value `seed`, which if "true" makes sure that the random graph generated is always the same whenever k, a and n are the same (pseudo-random generation).

2 Simulate a pandemic without vaccination

In this part we will generate a preferential attachment random graph and simulate an epidemic on it. The disease propagation model is again the discrete-time version of the SIR epidemic model described and used in section 1.1.

I developed a function `simulate_epidemic_without_vaccination`, very similar to the one presented in section 1.1 except for the fact that it generates and uses a preferential attachment random graph before the simulations, given the attributes `n`, `k`, `a`. The function has three boolean attributes which let us control the "degree of randomness" in the simulations:

- `random_infected_everytime`: if "true", each simulation will start with a different set of initially infected individuals, if "false", the set of initially infected individuals will be randomly chosen once and for all the simulations;
- `gen_rand_graph_everytime`: if "true", a different random graph is generated and used for each simulation; if "false", a single random graph is generated and used for all the simulations;
- `seed`: passed to `generate_preferential_attachment_graph` described in section 1.2; if "true", the random graph generated will be the same whenever `n`, `k`, `a` are the same (pseudo-random generation).

[2] In fact:

$$\mathbb{E}\left[\lim_{t\to\infty} w_i(t)\right] = \sum_{k=0}^{\infty} k p_k = \sum_{k=0}^{c-1} k \overbrace{p_k}^{=0} + \sum_{k=c}^{\infty} k p_k = \sum_{k=c}^{\infty} k \frac{2c(c+1)}{k(k+1)(k+2)} = 2c(c+1) \overbrace{\sum_{k=c}^{\infty} \frac{1}{(k+1)(k+2)}}^{\text{Mengoli's series}} =$$

$$= 2c(c+1)\frac{1}{c+1} = 2c$$

We perform $N = 100$ simulations of an epidemic over a preferential attachment random graph, with $|\mathcal{V}| = 500$ nodes, average degree $k = 6$, $\beta = 0.3$ and $\rho = 0.7$, for 15 weeks.

For each simulation, an initial configuration with 10 infected nodes is chosen at random (`random_infected_everytime` is true). All the simulations are performed on the same random graph (`gen_rand_graph_everytime` is false and `seed` is true).

At the end of the simulations, we compute:

- the average number of newly infected individuals during each week;
- the average total number of susceptible, infected and recovered individuals at the beginning of each week.

The average is computed over the 100 simulations. The evolution of the required average quantities is plotted in fig. 2.

Note that the plots in fig. 2 should be interpreted as explained in section 1.2.

3 Simulate a pandemic with vaccination

In this part we again simulate an epidemic over a preferential attachment random graph, but this time we will also try to to slow down the epidemic by vaccinating some individuals. During each week, some parts of the population will receive vaccination. Once a person is vaccinated, it cannot be infected; if an infected person is vaccinated, it is assumed that it cannot infect other people. The individuals to vaccinate should be selected uniformly at random from the population that has not yet received vaccination; this means that individuals can get vaccinated whether they are S, I or R. Furthermore, the vaccination is assumed to take effect immediately once given.

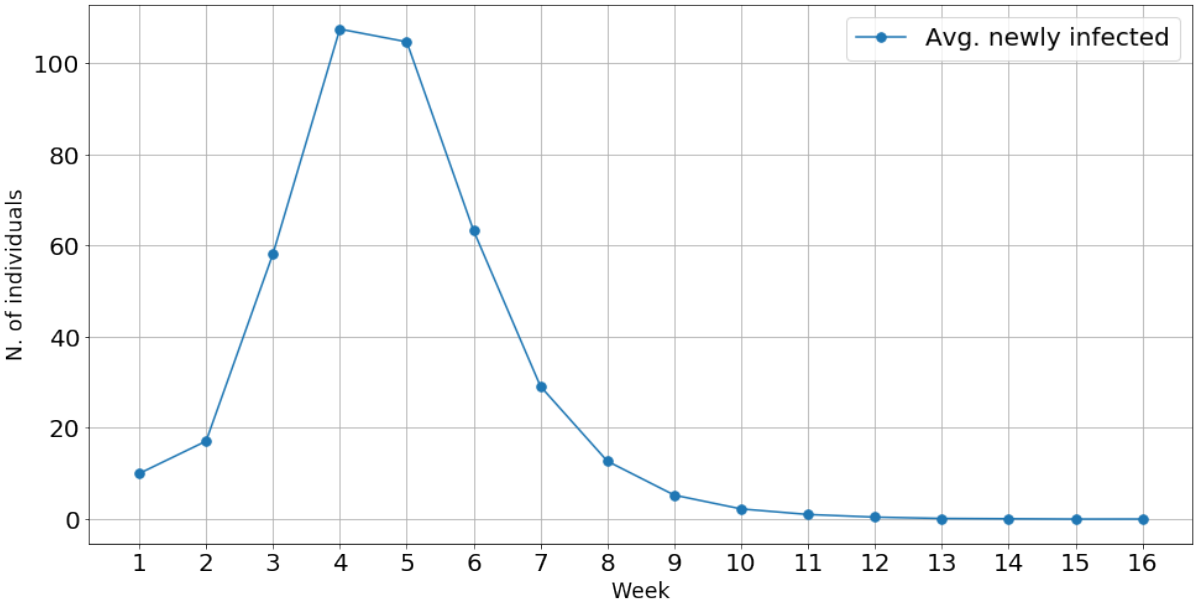
The vaccination campaign to be implemented introduces a new possible state, V, which indicates an individual who received vaccination and therefore cannot get infected nor infect other people.

I developed a function `simulate_epidemic_with_vaccination`, very similar to the one presented in section 2 but which enables us to vaccinate some people at random during each week, choosing among the population that has not yet received vaccination. The amount of people to vaccinate during each week is determined by the vector `vacc` passed as an attribute to the function, in the way explained soon below (see the "vaccination scheme" Vacc(t)).

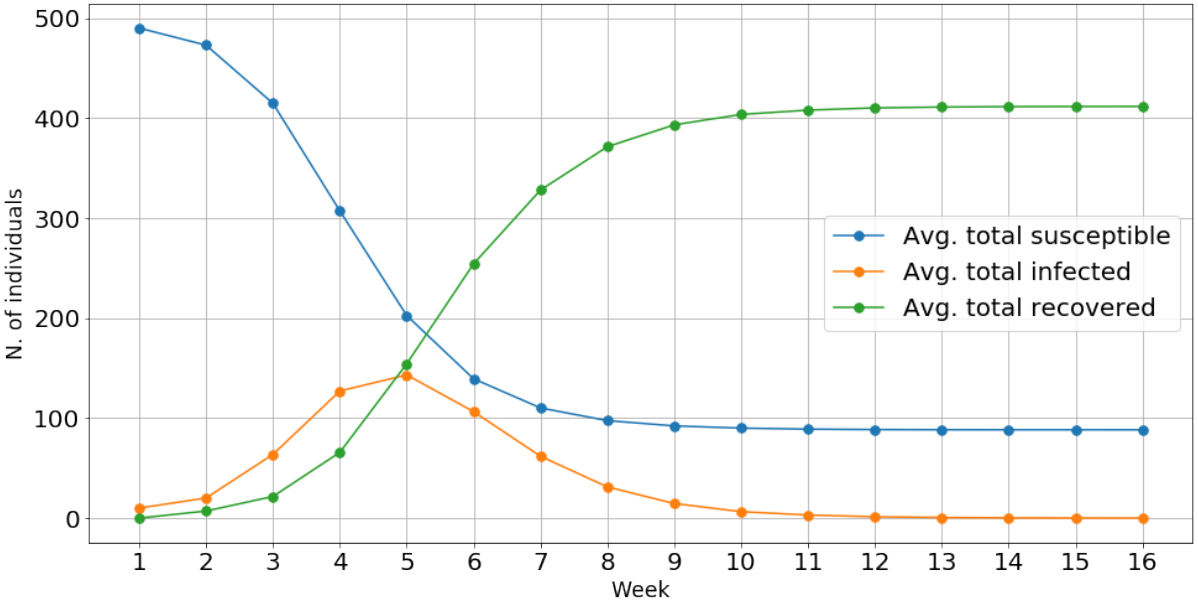
N.B.: for simplicity, a choice for my implementation was that the individuals who should receive vaccination *during the course of week i* are instead all vaccinated *at the beginning of week $i + 1$*, before week $i + 1$ is simulated. If some individuals are theoretically already vaccinated before the simulation starts (i.e. by the beginning of week 1) then in my implementation these individuals are vaccinated at the beginning of week 1, before simulating it.

We perform $N = 100$ simulations of an epidemic over a preferential attachment random graph, with $|\mathcal{V}| = 500$ nodes, average degree $k = 6$, $\beta = 0.3$ and $\rho = 0.7$, for 15 weeks.

Throughout these 15 weeks, we now distribute vaccination to the population. This is done such that the total percentage of population that has received vaccination *by* each



(a)



(b)

Figure 2: Problem 2. Evolution of the required average quantities.

week is according to the following "vaccination scheme":

$$\text{Vacc}(t) = [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60, 60]$$

For example, $\text{Vacc}(3) = 15$ means that by week 3 (i.e. before week 3 is simulated) 15% of the population has been vaccinated. The percentage of population to vaccinate *during* week 3 is given by $\text{Vacc}(4) - \text{Vacc}(3) = 10$. Note that we simulate for 15 weeks, but $\text{Vacc}(t)$ has 16 elements: this is to give meaning to the computation of the percentage of population that receives vaccination during week 15, $\text{Vacc}(16) - \text{Vacc}(15)$; the total percentage of population which received vaccination during the 15 weeks of the simulation is therefore $\text{Vacc}(16)$.

For each simulation, an initial configuration with 10 infected nodes is chosen at random (`random_infected_everytime` is true). All the simulations are performed on the same random graph (`gen_rand_graph_everytime` is false and `seed` is true).

At the end of the simulations, we compute:

- the average number of newly infected and newly vaccinated individuals during each week;
- the average total number of susceptible, infected, recovered and vaccinated individuals at the beginning of each week.

The average is computed over the 100 simulations. The evolution of the required average quantities is plotted in fig. 3.

Note that the plots in fig. 3 should be interpreted as explained in section 1.2. In particular, the point of the "avg. newly vaccinated" line corresponding to week i measures the average number of newly vaccinated individuals *during* week $i - 1$ (these individuals are vaccinated just before simulating week i, as previously explained). Moreover, the point of the "avg. total vaccinated" line corresponding to week i is a snapshot of the average cumulative number of vaccinated (V) individuals at the beginning of week i.

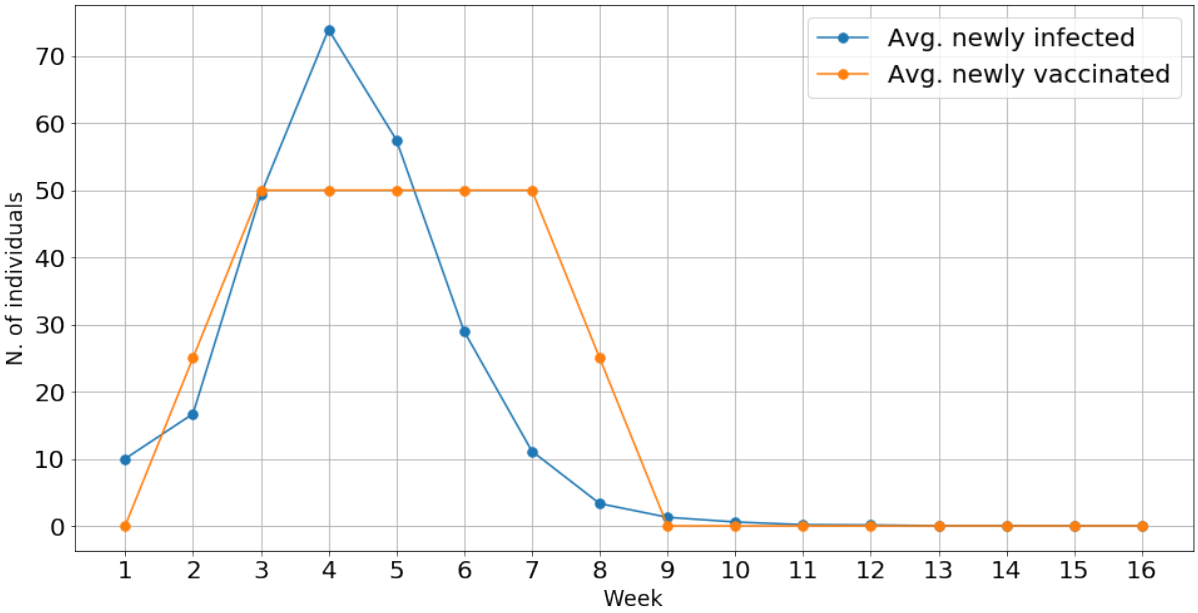
We can see that the vaccination campaign was effective in containing the average number of new infections and total infections: on average, a little more than 200 people got the disease since week 1 (fig. 3b), versus more than 400 without the vaccination (fig. 2b); moreover, vaccination "relaxed" the peak of the epidemic: on average, the peak value of infected people is a little less than 100 (recorded at the beginning of week 4), versus almost 150 (at the beginning of week 5) without vaccination.

4 The H1N1 pandemic in Sweden 2009

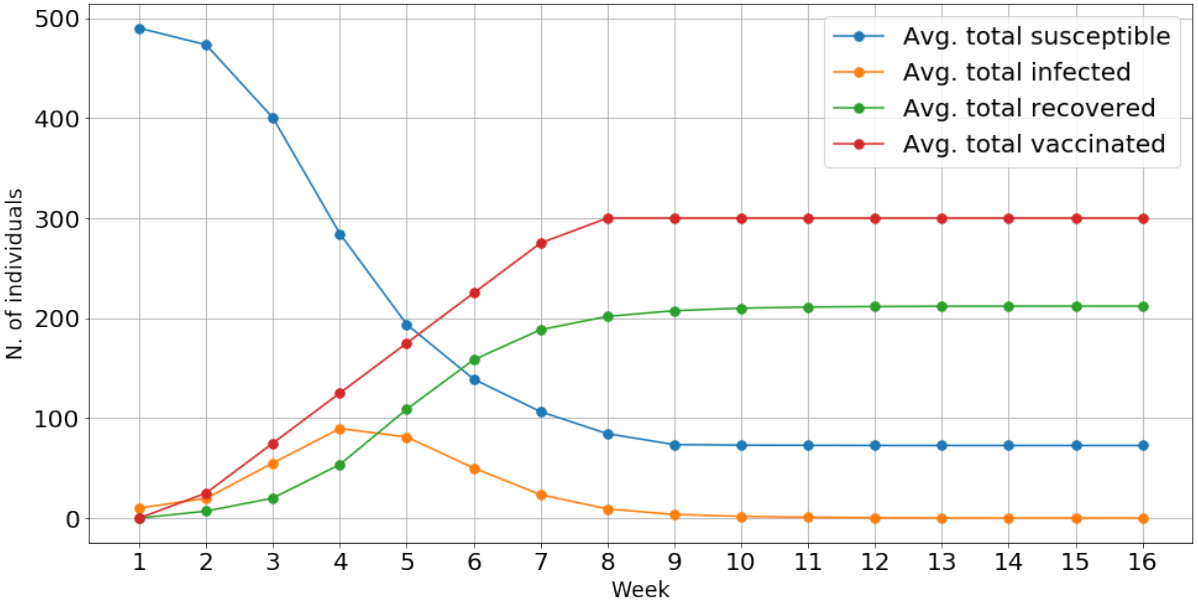
In this part we will use all the previous parts in order to estimate the social structure of the Swedish population and the disease-spread parameters during the H1N1 pandemic. During the fall of 2009 about 1.5 million people out of a total population of 9 million were infected with H1N1, and about 60% of the population received vaccination.

We want to simulate the pandemic between week 42, 2009 and week 5, 2010. During those 15 weeks, the fraction of population that had received vaccination was:

$$\text{Vacc}(t) = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]$$



(a)



(b)

Figure 3: Problem 3. Evolution of the required average quantities.

In order not to spend too much time running simulations, we will scale down the population of Sweden by a factor of 10^4. This means that the population during the simulation will be $n = |\mathcal{V}| = 934$. For the scaled version, the number of newly infected individuals each week in the period between week 42, 2009 and week 5, 2010 was:

$$I_0(t) = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]$$

The following algorithm will do a gradient-based search over the parameter space generated by (k, β, ρ) to find the set of parameters that best matches the real pandemic. Specifically, we want to find a triplet $(\bar{k}, \bar{\beta}, \bar{\rho})$ for which, when generating the corresponding preferential attachment random graph and simulating an epidemic on it a certain number of times, the resulting avg. number of newly infected individuals $I(t)$ approximates well the real $I_0(t)$ of the H1N1 pandemic. In other words, we want to minimize the root-mean-square error (RMSE) between the model and the real pandemic:

$$\text{RMSE}(I(t),\, I_0(t)) = \sqrt{\frac{1}{16}\sum_{t=1}^{16}(I(t) - I_0(t))^2}$$

Algorithm: Start with an initial guess of the parameters k_0, β_0 and ρ_0 along with some Δk, $\Delta\beta$ and $\Delta\rho$.

1. For each set of parameters (k, β, ρ) in the parameter space $k \in \{k_0 - \Delta k, k_0, k_0 + \Delta k\}$, $\beta \in \{\beta_0 - \Delta\beta, \beta_0, \beta_0 + \Delta k\}$ and $\rho \in \{\rho_0 - \Delta\rho, \rho_0, \rho_0 + \Delta\rho\}$:
   (a) Generate a preferential attachment random graph $\mathcal{G}$ with $n = |\mathcal{V}| = 934$ nodes, average degree k, $a = 0$. This is generated once and for all simulations (`gen_rand_graph_everytime` is false and `seed` is true), to keep the "degree of randomness" sufficiently low.[3]
   (b) Starting from week 42, simulate the pandemic for 15 weeks on $\mathcal{G}$, with vaccination scheme Vacc(t) defined above. At the beginning of each simulation, Vacc$(0) = 1$ individual is randomly chosen, and given state I (`random_infected_everytime` is true). Perform this simulation N times and compute the average number of newly infected individuals during each week, $I(t)$.
   (c) Compute RMSE$(I(t),\, I_0(t))$
2. Update k_0, β_0 and ρ_0 to the set of parameters yielding the lowest RMSE. If the result was the same set of parameters, do the following:
   (a) if it hasn't already been performed in a previous iteration of the algorithm, perform a refinement of the deltas: $\Delta\beta \leftarrow \Delta\beta/2$ and $\Delta\rho \leftarrow \Delta\rho/2$.
   (b) if such refinement has already been performed previously, then the algorithm stops and returns the set of parameters found.

I implemented the above algorithm in the Python function `find_best_k_beta_rho` (see appendix A). This function prints on screen the best set of parameters found, and returns

[3]This algorithm is very slow to execute, as in each iteration $3 \times 3 \times 3 = 27$ triplets of parameters should be tested (i.e. $27 \times N$ simulations must be performed). Therefore, keeping the "degree of randomness" somewhat low enables us to run the algorithm with a relatively low number N of simulations per triplet.

4 vectors containing (1) the avg. number of newly infected individuals during each week $I(t)$, the avg. total number of (2) susceptible, (3) infected and (4) recovered individuals at the beginning of each week.

I made a few trials with number of simulations $N = 30$ and starting from different values of k_0, β_0 and ρ_0. My best result was the set of parameters $\bar{k} = 16$, $\bar{\beta} = 0.1$, $\bar{\rho} = 0.6$ (obtained by starting from $k_0 = 15$, $\beta_0 = 0.3$, $\rho_0 = 0.6$ with $\Delta k = 1$, $\Delta\beta = 0.1$, $\Delta\rho = 0.1$), which yielded a RMSE of 3.864. The evolution of the required average quantities is plotted in fig. 4.

Despite keeping the "degree of randomness" pretty low by imposing `gen_rand_graph_everytime` equal to false and `seed` equal to true, the algorithm still shows a moderately high variance when computing the RMSE for the same set of parameters. This problem could be solved by increasing the number of simulations N. In fact, by running the code written for problem 3 multiple times (with the default $N = 100$) I verified that roughly the same plots are generated every time, i.e. the evolution of the average quantities is pretty much the same every time. However, imposing $N = 100$ to run the algorithm above will require $27 \times 100 = 2700$ simulations to be performed at each iteration, taking a huge amount of time.

Another aspect of this algorithm is that it is very sensitive to the initial setting of k_0, β_0 and ρ_0. Starting from different parameters will make the algorithm stop in different local minima of the parameter space generated by (k, β, ρ). By running the algorithm with different initial settings of the three parameters we can possibly find better local minima (i.e. a set of parameters $(\bar{k}, \bar{\beta}, \bar{\rho})$ yielding a lower RMSE than the best one obtained in my experiments). A strategy to tackle this problem could be performing a grid search or a random search to estimate the parameters, instead of using the gradient-descent algorithm described (see section 5).

5 Different random graphs and algorithms

In this final section we repeat the experiment performed in problem 4 but we try simulating the epidemic on different random graphs and introducing different algorithms to estimate the social structure of the population (described by k) and the disease-spread parameters β and ρ.

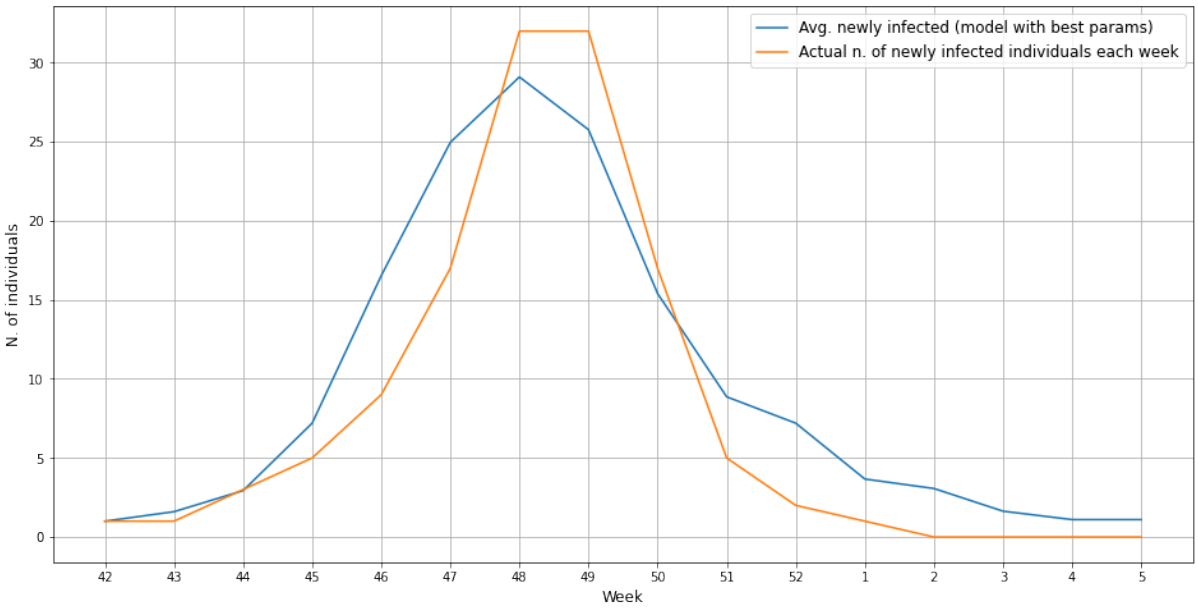
5.1 Erdos-Renyi random graph

We repeat the experiment 4 generating and using an Erdos-Renyi random graph $\mathcal{G}_{n,p}$, with given number of nodes n and given probability p.

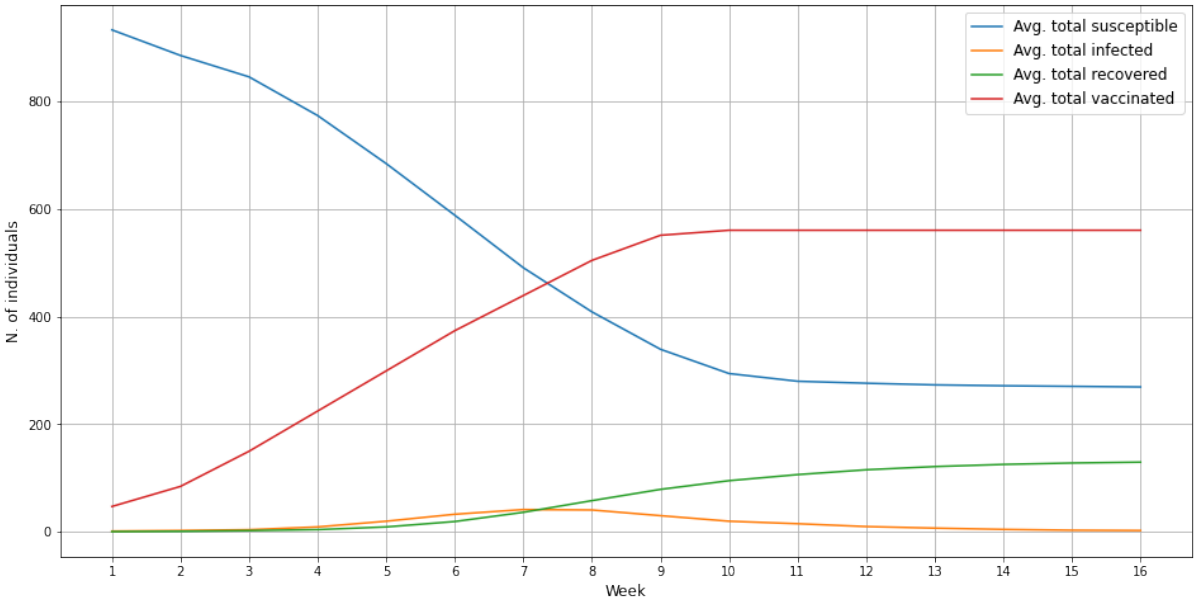
To generate such a graph, we connect every pair of nodes (i, j), $i \neq j$ by an undirected and unweighted link with probability p:

$$\mathbb{P}(W_{ij} = W_{ji} = 1 \,|\, i \neq j) = p$$

The degree distribution of $\mathcal{G}_{n,p}$ is a binomial distribution $\mathrm{Bin}(n-1, p)$. Therefore, the average degree of the Erdos-Renyi random graph $\mathcal{G}_{n,p}$ is $\mathbb{E}[w_i] = k = (n-1)p$ for all nodes i.



(a)



(b)

Figure 4: Problem 4. Evolution of the required average quantities. $\bar{k} = 16$, $\bar{\beta} = 0.1$, $\bar{\rho} = 0.6$, RMSE $= 3.864$.

I implemented a function `generate_erdos_renyi_graph`, which generates an Erdos-Renyi random graph given the desired average degree k, the number of nodes n and a boolean value `seed`, which if "true" makes sure that the random graph generated is always the same whenever k and n are the same (pseudo-random generation). The probability p is computed internally to the function as $p = k/(n-1)$.

I also implemented the methods `simulate_epidemic_with_vaccination_ER` and `find_best_k_beta_rho_ER`, which work in the same way of `simulate_epidemic_with_vaccination` and `find_best_k_beta_rho` except for the fact that they employ Erdos-Renyi random graphs instead of preferential attachment random graphs with $a = 0$.

As before, I made a few trials with number of simulations $N = 30$ and starting from different values of k_0, β_0 and ρ_0. My best result was the set of parameters $\bar{k} = 12$, $\bar{\beta} = 0.2$, $\bar{\rho} = 0.9$ (obtained by starting from $k_0 = 15$, $\beta_0 = 0.3$, $\rho_0 = 0.6$ with $\Delta k = 1$, $\Delta\beta = 0.1$, $\Delta\rho = 0.1$), which yielded a RMSE of 4.634 (worse than the best one achieved using the preferential attachment random graph with $a = 0$).

5.2 Preferential attachment with $a = 1$

The problem 4 is solved again, this time by generating and using a preferential attachment random graph with $a = 1$. We can verify numerically that imposing $a = 1$ still yields a random graph with average degree k, when choosing c in the same way as before (as written in section 1.2) and when the number of nodes n is sufficiently high.

Again, I made a few trials with number of simulations $N = 30$ and starting from different values of k_0, β_0 and ρ_0. My best result was the set of parameters $\bar{k} = 14$, $\bar{\beta} = 0.1$, $\bar{\rho} = 0.5$ (obtained by starting from $k_0 = 15$, $\beta_0 = 0.3$, $\rho_0 = 0.6$ with $\Delta k = 1$, $\Delta\beta = 0.1$, $\Delta\rho = 0.1$), which yielded a RMSE of 4.837 (still worse than the best one achieved using the preferential attachment random graph with $a = 0$).

5.3 Grid search and random search

The gradient-based algorithm described in section 4 to estimate the social structure of the Swedish population (described by the parameter k) and the disease-spread parameters β and ρ is not the only possible one.

We repeat the experiment 4, but this time we perform both a grid search and a random search over the parameter space generated by (k, β, ρ).

Grid search: the sets of parameters to be tested are all the triplets (k, β, ρ) such that $k \in \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$, $\beta \in \{0.05, 0.1, 0.15, 0.2\}$, $\rho \in \{0.6, 0.7, 0.8, 0.9\}$. Therefore, the total number of triplets to be tested is $10 \times 4 \times 4 = 160$. For each triplet, $N = 30$ simulations are performed.

Random search: we test 100 triplets of parameters sampled uniformly at random from the parameter space generated by (k, β, ρ) where $k \in \{7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$, $\beta \in [0.03, 0.25]$, $\rho \in [0.6, 0.95]$. For each triplet, $N = 30$ simulations are performed.

The epidemic is again simulated on a preferential attachment random graph with $a = 0$, as in problem 4 (`gen_rand_graph_everytime` is false and `seed` is true).

My best results were:

- Grid search: $\bar{k} = 15$, $\bar{\beta} = 0.1$, $\bar{\rho} = 0.6$, RMSE $= 4.659$;
- Random search: $\bar{k} = 15$, $\bar{\beta} = 0.112$, $\bar{\rho} = 0.611$, RMSE $= 4.757$.

A Python code

In this section is reported (starting from the next page) the Python code used to solve the problems.

Python code

HW3

[ ]:
```
import networkx as nx
import numpy as np
np.set_printoptions(suppress=True)
import scipy as sp
import matplotlib.pyplot as plt
from numpy.random import choice

from time import time
```

1 1. Preliminary parts

1.1 1.1 Epidemic on a known graph

[ ]:
```
def simulate_epidemic_fixed_graph(G, beta=0.3, rho=0.7, n_weeks=15,␣
↪n_initial_infected=10, n_simulations=100, random_infected_everytime=True):
    n = len(G)    # n. of people

    new_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_S = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_R = np.zeros((n_simulations, n_weeks+1), dtype=int)

    new_I[:,0] = n_initial_infected
    tot_S[:,0] = n-n_initial_infected
    tot_I[:,0] = n_initial_infected
    tot_R[:,0] = 0

    ### Week 1 (the epidemics breaks out)
    # Choose at random n_initial_infected people which are initially infected
    # (here or inside the simulations loop if random_infected_everytime)
    if not random_infected_everytime:
        initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
        initial_config = np.array(["I" if i in initial_infected else "S" for i␣
↪in range(n)])
```

```
    for s in range(n_simulations):
        if random_infected_everytime:
            initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
            initial_config = np.array(["I" if i in initial_infected else "S"␣
↪for i in range(n)])

        old_config = initial_config

        ### Weeks 2-(n_weeks-1) (the epidemics spreads)
        for w in range(0, n_weeks):

            new_config = []

            # Compute m for each node
            for i in range(n):

                if old_config[i] == "S":
                    #neighbors = [(i-2)%n, (i-1)%n, (i+1)%n, (i+2)%n]    # in␣
↪exercise 1.1
                    neighbors = list(G.neighbors(i))
                    neighbors_state = old_config[neighbors]
                    m = sum([1 for st in neighbors_state if st=="I"])
                    p_infection = 1-(1-beta)**m
                    new_state = choice(("I","S"), p=(p_infection,␣
↪1-p_infection))
                    new_config.append(new_state)

                elif old_config[i] == "I":
                    new_state = choice(("R","I"), p=(rho, 1-rho))
                    new_config.append(new_state)

                else: #old_config[i] == "R"
                    new_config.append("R")

            # Compute the required quantities
            # Newly infected individuals
            new_I[s,w+1] = sum([1 for i in range(n) if (new_config[i]=="I" and␣
↪old_config[i]!="I")])
            # Total n. of S,I,R individuals
            tot_S[s,w+1] = sum([1 for i in new_config if i=="S"])
            tot_I[s,w+1] = sum([1 for i in new_config if i=="I"])
            tot_R[s,w+1] = sum([1 for i in new_config if i=="R"])

            old_config = np.array(new_config)

    # Compute the required averages
```

```
    avg_new_I = np.mean(new_I, axis=0)
    avg_tot_S = np.mean(tot_S, axis=0)
    avg_tot_I = np.mean(tot_I, axis=0)
    avg_tot_R = np.mean(tot_R, axis=0)

    return (avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R)
```

[ ]:
```
### Create 4-regular graph with n=500 nodes
GG = nx.Graph()
n_nodes = 500
nx.add_cycle(GG, range(n_nodes)) # C_n

for n in range(n_nodes):
    GG.add_edge(n,(n+1)%n_nodes)
    GG.add_edge(n,(n+2)%n_nodes)

#nx.draw_circular(GG, with_labels=False)


### Simulate epidemics
avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R = simulate_epidemic_fixed_graph(GG,␣
 ↪beta=0.3, rho=0.7, n_weeks=15, n_initial_infected=10, n_simulations=100,␣
 ↪random_infected_everytime=True)
print(avg_new_I)
print(avg_tot_S)
print(avg_tot_I)
print(avg_tot_R)


### Plots
n_weeks = len(avg_new_I)
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected', marker='.',␣
 ↪markersize=15)
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
plt.tick_params(axis='both', labelsize=20)
ax.set_xlabel('Week', size='xx-large')
ax.set_ylabel('N. of individuals', size='xx-large')
ax.legend(prop={'size': 20})
plt.grid()

fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible', marker='.',␣
 ↪markersize=15)
```

```
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected', marker='.',␣
↪markersize=15)
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered', marker='.',␣
↪markersize=15)
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
plt.tick_params(axis='both', labelsize=20)
ax.set_xlabel('Week', size='xx-large')
ax.set_ylabel('N. of individuals', size='xx-large')
ax.legend(prop={'size': 20})
plt.grid()
```

1.2 1.2 Generate a random graph

[ ]:
```
def generate_preferential_attachment_graph(k=4, a=0, n_nodes=1000, seed=False):
    # a = intrinsic probability of a node to be selected as a neighbor from new␣
↪nodes
    # k -> (k+1)-complete graph and floor(k/2) or ceil(k/2) links added at each␣
↪step (for the node that is added on that step)
    # n_nodes = number of nodes
    # seed = the seed to set np.random.seed to, and in this case returns the␣
↪same graph (fixed k and a);
    #        if False, then do not set it, and so returns always a different␣
↪graph

    np.random.seed()
    if seed:
        np.random.seed(42)    # the same graph will be generated for the same␣
↪k, a, n_nodes

    # PAG is initialized to G1, a complete graph with k+1 nodes (nodes: 0,1,...
↪,k)
    PAG = nx.complete_graph(k+1)

    # Compute n_iterations
    n_iterations = n_nodes - len(PAG)

    node_to_add = k+1

    for t in range(2,n_iterations+2):
        ### step t is beginning

        # Add a new node to PAG
        PAG.add_node(node_to_add)

        # Connect the new node to c=k/2 existing nodes
        degrees = np.array([d for _, d in PAG.degree()])
```

```
        deg_distr = (degrees+a)/np.sum(degrees+a)

        # Choose the value of c, i.e. how many existing nodes the new node must␣
↪be connected to
        if t%2 == 0:
            c = int(np.floor(k/2))
        else:
            c = int(np.ceil(k/2))

        neighbors = choice(np.arange(len(PAG)), size=c, p=deg_distr,␣
↪replace=False)    # replace=False guarantees no neighbor is chosen twice
        for neigh in neighbors:
            PAG.add_edge(node_to_add,neigh)

        node_to_add += 1

    return PAG
```

[ ]:
```
GPA = generate_preferential_attachment_graph(k=4, a=0, n_nodes=1000, seed=True)
avg_degree = np.mean([d for _,d in GPA.degree()])
print(avg_degree)
```

2 2. Simulate a pandemic without vaccination

[ ]:
```
def simulate_epidemic_without_vaccination(n, k, a, beta=0.3, rho=0.7,␣
↪n_weeks=15, n_initial_infected=10, n_simulations=100,␣
↪random_infected_everytime=True, gen_rand_graph_everytime=True, seed=False):
    # random_infected_everytime: if True, each simulation will start with a␣
↪different set of initially infected nodes
    # gen_rand_graph_everytime: if True, each simulation will be on a different␣
↪random graph
    # seed: if True, generate a single random graph for all the simulations␣
↪(moreover, same k and a always implies same graph)

    np.random.seed()

    new_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_S = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_R = np.zeros((n_simulations, n_weeks+1), dtype=int)

    ### Beginning of week 1
    # Initial situation
    new_I[:,0] = n_initial_infected
    tot_S[:,0] = n-n_initial_infected
    tot_I[:,0] = n_initial_infected
```

```
    tot_R[:,0] = 0

    # Choose at random n_initial_infected people which are initially infected
    # (here or inside the simulations loop if random_infected_everytime)
    if not random_infected_everytime:
        initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
        initial_config = np.array(["I" if i in initial_infected else "S" for i␣
↪in range(n)])

    if not gen_rand_graph_everytime:
        # Generate a unique random graph (either with seed or not)
        # moreover, if seed=True, the unique random graph is always the same,␣
↪under the same k and a
        # else, the unique random graph is always different
        G = generate_preferential_attachment_graph(k=k, a=a, n_nodes=n,␣
↪seed=seed)
        np.random.seed()


    for s in range(n_simulations):

        if gen_rand_graph_everytime:
            # Generate the i-th random graph for the i-th simulation (max␣
↪degree of randomness)
            G = generate_preferential_attachment_graph(k=k, a=a, n_nodes=n,␣
↪seed=False)

        if random_infected_everytime:
            initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
            initial_config = np.array(["I" if i in initial_infected else "S"␣
↪for i in range(n)])

        old_config = initial_config

        ### Weeks 1-(n_weeks-1) (the epidemics spreads)
        for w in range(0, n_weeks):

            new_config = []

            count_new_I = 0
            count_tot_S = 0
            count_tot_I = 0
            count_tot_R = 0
```

```
            # Compute m for each node
            for i in range(n):

                if old_config[i] == "S":
                    #neighbors = [(i-2)%n, (i-1)%n, (i+1)%n, (i+2)%n]    # in␣
↪exercise 1.1
                    neighbors = list(G.neighbors(i))
                    neighbors_state = old_config[neighbors]
                    m = sum([1 for st in neighbors_state if st=="I"])
                    p_infection = 1-(1-beta)**m
                    new_state = choice(("I","S"), p=(p_infection,␣
↪1-p_infection))
                    new_config.append(new_state)
                    if new_state == "I":    # update counter
                        count_new_I += 1
                        count_tot_I += 1
                    else:
                        count_tot_S += 1

                elif old_config[i] == "I":
                    new_state = choice(("R","I"), p=(rho, 1-rho))
                    new_config.append(new_state)
                    if new_state == "R":    # update counter
                        count_tot_R += 1
                    else:
                        count_tot_I += 1

                else: #old_config[i] == "R"
                    new_config.append("R")
                    count_tot_R += 1

            # Compute the required quantities
            # Newly infected individuals
            new_I[s,w+1] = count_new_I    # i.e. during week w, count_new_I␣
↪people were infected
            # Total n. of S,I,R individuals
            tot_S[s,w+1] = count_tot_S    # i.e. at the beginning of week w+1,␣
↪count_tot_S people were "S"
            tot_I[s,w+1] = count_tot_I    # i.e. at the beginning of week w+1,␣
↪count_tot_I people were "I"
            tot_R[s,w+1] = count_tot_R    # i.e. at the beginning of week w+1,␣
↪count_tot_R people were "R" (equivalent to say that a total of count_tot_R␣
↪people recovered)

            old_config = np.array(new_config)
```

```
    # Compute the required averages
    avg_new_I = np.mean(new_I, axis=0)
    avg_tot_S = np.mean(tot_S, axis=0)
    avg_tot_I = np.mean(tot_I, axis=0)
    avg_tot_R = np.mean(tot_R, axis=0)

    return (avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R)
```

[ ]:
```
### Simulate epidemics
avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R = ␣
 ↪simulate_epidemic_without_vaccination(500, 6, 0, beta=0.3, rho=0.7,␣
 ↪n_weeks=15, n_initial_infected=10, n_simulations=100,␣
 ↪random_infected_everytime=True, gen_rand_graph_everytime=False, seed=True)

print(avg_new_I)
print(avg_tot_S)
print(avg_tot_I)
print(avg_tot_R)

### Plots
n_weeks = len(avg_new_I)
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected', marker='.',␣
 ↪markersize=15)
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
plt.tick_params(axis='both', labelsize=20)
ax.set_xlabel('Week', size='xx-large')
ax.set_ylabel('N. of individuals', size='xx-large')
ax.legend(prop={'size': 20})
plt.grid()

fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible', marker='.',␣
 ↪markersize=15)
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected', marker='.',␣
 ↪markersize=15)
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered', marker='.',␣
 ↪markersize=15)
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
plt.tick_params(axis='both', labelsize=20)
ax.set_xlabel('Week', size='xx-large')
ax.set_ylabel('N. of individuals', size='xx-large')
ax.legend(prop={'size': 20})
plt.grid()
```

3 3. Simulate a pandemic with vaccination

[ ]:
```
def simulate_epidemic_with_vaccination(n, k, a, vacc, beta=0.3, rho=0.7,␣
↪n_weeks=15, n_initial_infected=10, n_simulations=100,␣
↪random_infected_everytime=True, gen_rand_graph_everytime=False, seed=True):
    # random_infected_everytime: if True, each simulation will start with a␣
↪different set of initially infected nodes
    # gen_rand_graph_everytime: if True, each simulation will be on a different␣
↪random graph
    # seed: if True, generate a single random graph for all the simulations␣
↪(moreover, same k and a always implies same graph)

    np.random.seed()

    new_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_S = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_R = np.zeros((n_simulations, n_weeks+1), dtype=int)

    # Compute how many people get vaccinated by each week
    # new_V[i] is the n. of people who get vaccinated during week i-1 (i=0 =>␣
↪during the previous weeks)
    # (N.B: for simplicity, we pretend to vaccinate them all at the beginning␣
↪of week i)
    # tot_V[i] is the n. of people who are "V" at the beginning of week i
    tot_V = ((np.array(vacc) / 100) * n)
    new_V = np.array([tot_V[i]-tot_V[i-1] if i!=0 else tot_V[0] for i in␣
↪range(len(vacc))])

    ### Beginning of week 1
    # Initial situation
    new_I[:,0] = n_initial_infected    # i.e. in the weeks before the starting␣
↪week (week 1), a total of n_initial_infected people were "I"
    tot_S[:,0] = n-n_initial_infected    # i.e. in the weeks before the␣
↪starting week (week 1), a total of n-n_initial_infected people were "S"
    tot_I[:,0] = n_initial_infected    # i.e. in the weeks before the starting␣
↪week (week 1), a total of n_initial_infected people were "I"
    tot_R[:,0] = 0    # i.e. in the weeks before the starting week (week 1), no␣
↪one was "R"

    # Choose at random n_initial_infected people which are initially infected
    # (here or inside the simulations loop if random_infected_everytime)
    if not random_infected_everytime:
        initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
```

```
        initial_config = np.array(["I" if i in initial_infected else "S" for i␣
↪in range(n)])

    if not gen_rand_graph_everytime:
        # Generate a unique random graph (either with seed or not)
        # moreover, if seed=True, the unique random graph is always the same,␣
↪under the same k and a
        # else, the unique random graph is always different
        G = generate_preferential_attachment_graph(k=k, a=a, n_nodes=n,␣
↪seed=seed)
        np.random.seed()


    for s in range(n_simulations):

        count_tot_R = 0     # count_tot_R is increased by 1 only when an I␣
↪becomes R

        if gen_rand_graph_everytime:
            # Generate the i-th random graph for the i-th simulation (max␣
↪degree of randomness)
            G = generate_preferential_attachment_graph(k=k, a=a, n_nodes=n,␣
↪seed=False)

        if random_infected_everytime:
            initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
            initial_config = np.array(["I" if i in initial_infected else "S"␣
↪for i in range(n)])

        # Vaccination: initial situation
        non_vacc_people = set(range(n))

        old_config = initial_config

        ### Weeks 1-n_weeks (the epidemics spreads)
        for w in range(0, n_weeks):     # week w=0 means the 1st week and so on␣
↪(mapping from 0...15 to 1...16)
            # We want to simulate week w; the final configuration is the␣
↪starting configuration of week w+1

            new_config = []

            count_new_I = 0
            count_tot_S = 0
            count_tot_I = 0
```

```
            # At the beginning of week w, we take into account the people␣
↪vaccinated during week w-1
            # (for simplicity, we pretend to vaccinate them all at the␣
↪beginning of week w)
            n_to_vacc = int(new_V[w])
            new_vaccinated = choice(list(non_vacc_people), size=n_to_vacc,␣
↪replace=False)
            non_vacc_people -= set(new_vaccinated)
            old_config[new_vaccinated] = "V"
            count_new_V = n_to_vacc

            for i in range(n):

                if old_config[i] == "V":
                    new_config.append("V")

                elif old_config[i] == "S":
                    neighbors = list(G.neighbors(i))
                    neighbors_state = old_config[neighbors]
                    m = np.sum(neighbors_state == "I")
                    p_infection = 1-(1-beta)**m
                    new_state = choice(("I","S"), p=(p_infection,␣
↪1-p_infection))
                    new_config.append(new_state)
                    if new_state == "I":    # update counter
                        count_new_I += 1
                        count_tot_I += 1
                    else:
                        count_tot_S += 1

                elif old_config[i] == "I":
                    new_state = choice(("R","I"), p=(rho, 1-rho))
                    new_config.append(new_state)
                    if new_state == "R":    # update counter
                        count_tot_R += 1
                    else:
                        count_tot_I += 1

                else: #old_config[i] == "R"
                    new_config.append("R")

            # Compute the required quantities
            # Newly infected individuals
            new_I[s,w+1] = count_new_I    # i.e. during week w, count_new_I␣
↪people were infected
            # Total n. of S,I,R individuals
```

```
            tot_S[s,w+1] = count_tot_S    # i.e. at the beginning of week w+1,␣
↪count_tot_S people were "S"
            tot_I[s,w+1] = count_tot_I    # i.e. at the beginning of week w+1,␣
↪count_tot_I people were "I"
            tot_R[s,w+1] = count_tot_R    # i.e. at the beginning of week w+1,␣
↪a total of count_tot_R people recovered

            old_config = np.array(new_config)

    # Compute the required averages
    avg_new_I = np.mean(new_I, axis=0)
    avg_new_V = new_V
    avg_tot_S = np.mean(tot_S, axis=0)
    avg_tot_I = np.mean(tot_I, axis=0)
    avg_tot_R = np.mean(tot_R, axis=0)
    avg_tot_V = tot_V

    return (avg_new_I, avg_new_V, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V)
```

[ ]:
```
### Simulate epidemics
vacc = [0, 5, 15, 25, 35, 45, 55, 60, 60, 60, 60, 60, 60, 60, 60, 60]
avg_new_I, avg_new_V, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪simulate_epidemic_with_vaccination(500, 6, 0, vacc, beta=0.3, rho=0.7,␣
↪n_weeks=len(vacc)-1, n_initial_infected=10, n_simulations=100,␣
↪random_infected_everytime=True, gen_rand_graph_everytime=False, seed=True)

print(avg_new_I)
print(avg_new_V)
print(avg_tot_S)
print(avg_tot_I)
print(avg_tot_R)
print(avg_tot_V)

### Plots
n_weeks = len(avg_new_I)
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected', marker='.',␣
↪markersize=15)
ax.plot(range(n_weeks), avg_new_V, label='Avg. newly vaccinated', marker='.',␣
↪markersize=15)
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
plt.tick_params(axis='both', labelsize=20)
ax.set_xlabel('Week', size='xx-large')
ax.set_ylabel('N. of individuals', size='xx-large')
ax.legend(prop={'size': 20})
plt.grid()
```

```
fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible', marker='.',␣
 ↪markersize=15)
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected', marker='.',␣
 ↪markersize=15)
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered', marker='.',␣
 ↪markersize=15)
ax.plot(range(n_weeks), avg_tot_V, label='Avg. total vaccinated', marker='.',␣
 ↪markersize=15)
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
plt.tick_params(axis='both', labelsize=20)
ax.set_xlabel('Week', size='xx-large')
ax.set_ylabel('N. of individuals', size='xx-large')
ax.legend(prop={'size': 20})
plt.grid()
```

4 4. The H1N1 pandemic in Sweden 2009

[ ]:
```
def find_best_k_beta_rho(weekly_infected, vacc, n_nodes, k0, beta0, rho0, dk,␣
 ↪dbeta, drho, a=0, n_simulations=10, seed=True):

    initial_beta0 = beta0
    initial_rho0 = rho0

    refinement_done = False

    print("Initial parameters:", k0, beta0, rho0)
    n_weeks = len(weekly_infected)-1
    n_initial_infected = weekly_infected[0]

    while True:
        parameter_space = np.array(np.meshgrid([k0-dk,k0,k0+dk],␣
 ↪[beta0-dbeta,beta0,beta0+dbeta], [rho0-drho,rho0,rho0+drho])).T.reshape(-1,3)

        RMSE = []
        avg_new_Is = []
        avg_tot_Ss = []
        avg_tot_Is = []
        avg_tot_Rs = []
        avg_tot_V = np.array(vacc)

        for param in parameter_space:
            k = int(param[0])
            beta = param[1]
```

```
            rho = param[2]

            avg_new_I, _, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪simulate_epidemic_with_vaccination(n_nodes, k, a, vacc, beta=beta, rho=rho,␣
↪n_weeks=n_weeks, n_initial_infected=n_initial_infected,␣
↪n_simulations=n_simulations, random_infected_everytime=True,␣
↪gen_rand_graph_everytime=False, seed=seed)
            avg_new_Is.append(avg_new_I)
            avg_tot_Ss.append(avg_tot_S)
            avg_tot_Is.append(avg_tot_I)
            avg_tot_Rs.append(avg_tot_R)

            RMSE.append( np.sqrt((1/(n_weeks+1))*np.
↪sum((avg_new_I-weekly_infected)**2)) )

        # Now that all the 27 RMSEs have been computed
        new_param_min_RMSE = parameter_space[np.argmin(RMSE)]

        best_avg_new_I = avg_new_Is[np.argmin(RMSE)]
        best_avg_tot_S = avg_tot_Ss[np.argmin(RMSE)]
        best_avg_tot_I = avg_tot_Is[np.argmin(RMSE)]
        best_avg_tot_R = avg_tot_Rs[np.argmin(RMSE)]

        ### Stop criterion
        if np.all( new_param_min_RMSE == (k0, beta0, rho0) ):
            # Do a refinement of the deltas
            if not refinement_done:
                dbeta /= 2
                drho /= 2
                refinement_done = True
                continue
            print(f"A local minimum has been reached! RMSE = {np.min(RMSE)},␣
↪params = {new_param_min_RMSE}")
            return (best_avg_new_I, best_avg_tot_S, best_avg_tot_I,␣
↪best_avg_tot_R, avg_tot_V)
        else:
            print(f"Better params found. Best RMSE = {np.min(RMSE)}, new params␣
↪= {new_param_min_RMSE}")
            k0, beta0, rho0 = new_param_min_RMSE
```

[ ]:
```
vacc = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
weekly_infected = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
n_simulations = 30
avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪find_best_k_beta_rho(weekly_infected, vacc, a=0, n_nodes=934, k0=10, beta0=0.
↪3, rho0=0.6, dk=1, dbeta=0.1, drho=0.1, n_simulations=n_simulations,␣
↪seed=True)
```

```
### Plots
n_weeks = len(vacc)
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected (model with best␣
↪params)')
ax.plot(range(n_weeks), weekly_infected, label='Actual n. of newly infected␣
↪individuals each week')
plt.setp( ax, xticks=range(n_weeks),␣
↪xticklabels=list(range(42,53))+list(range(1,6)) )
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()

fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible')
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected')
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered')
ax.plot(range(n_weeks), avg_tot_V, label='Avg. total vaccinated')
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()
```

5 5. Challenge

5.1 Erdos-Renyi

[ ]:
```
def generate_erdos_renyi_graph(n, p, seed=False):

    # Add links between couple of different nodes with probability p
    np.random.seed()
    if seed:
        np.random.seed(42)    # the same graph will be generated for the same␣
↪k, a, n_nodes

    WER = np.random.choice([0,1], size=(n, n), p=[1-p,p])    # random weight␣
↪(adjacency) matrix

    # GER has no self loops and is undirected (so WER must be symmetric)

    for i in range(n):
```

```
        WER[i,i] = 0    # no self loops
        WER[i+1:,i] = WER[i,i+1:]    # to make the WER symmetric
    GER = nx.from_numpy_array(WER, create_using=nx.Graph)
    return GER
```

[ ]:
```
def simulate_epidemic_with_vaccination_ER(n, p, vacc, beta=0.3, rho=0.7,␣
↪n_weeks=15, n_initial_infected=10, n_simulations=100,␣
↪random_infected_everytime=True, gen_rand_graph_everytime=False, seed=True):

    np.random.seed()

    new_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_S = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_I = np.zeros((n_simulations, n_weeks+1), dtype=int)
    tot_R = np.zeros((n_simulations, n_weeks+1), dtype=int)

    tot_V = ((np.array(vacc) / 100) * n)
    new_V = np.array([tot_V[i]-tot_V[i-1] if i!=0 else tot_V[0] for i in␣
↪range(len(vacc))])

    new_I[:,0] = n_initial_infected
    tot_S[:,0] = n-n_initial_infected
    tot_I[:,0] = n_initial_infected
    tot_R[:,0] = 0

    if not random_infected_everytime:
        initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
        initial_config = np.array(["I" if i in initial_infected else "S" for i␣
↪in range(n)])

    if not gen_rand_graph_everytime:
        G = generate_erdos_renyi_graph(n=n, p=p, seed=seed)


    for s in range(n_simulations):

        count_tot_R = 0

        if gen_rand_graph_everytime:
            G = generate_erdos_renyi_graph(n, p, seed=False)
            np.random.seed()

        if random_infected_everytime:
            initial_infected = choice(range(n), size=n_initial_infected,␣
↪replace=False)
```

```
            initial_config = np.array(["I" if i in initial_infected else "S"␣
↪for i in range(n)])

        non_vacc_people = set(range(n))

        old_config = initial_config

        for w in range(0, n_weeks):

            new_config = []

            count_new_I = 0
            count_tot_S = 0
            count_tot_I = 0

            n_to_vacc = int(new_V[w])
            new_vaccinated = choice(list(non_vacc_people), size=n_to_vacc,␣
↪replace=False)
            non_vacc_people -= set(new_vaccinated)
            old_config[new_vaccinated] = "V"
            count_new_V = n_to_vacc

            for i in range(n):

                if old_config[i] == "V":
                    new_config.append("V")

                elif old_config[i] == "S":
                    neighbors = list(G.neighbors(i))
                    neighbors_state = old_config[neighbors]
                    m = np.sum(neighbors_state == "I")
                    p_infection = 1-(1-beta)**m
                    new_state = choice(("I","S"), p=(p_infection,␣
↪1-p_infection))
                    new_config.append(new_state)
                    if new_state == "I":
                        count_new_I += 1
                        count_tot_I += 1
                    else:
                        count_tot_S += 1

                elif old_config[i] == "I":
                    new_state = choice(("R","I"), p=(rho, 1-rho))
                    new_config.append(new_state)
                    if new_state == "R":
                        count_tot_R += 1
                    else:
```

```
                        count_tot_I += 1

                else:
                    new_config.append("R")

            new_I[s,w+1] = count_new_I
            tot_S[s,w+1] = count_tot_S
            tot_I[s,w+1] = count_tot_I
            tot_R[s,w+1] = count_tot_R

            old_config = np.array(new_config)

    avg_new_I = np.mean(new_I, axis=0)
    avg_new_V = new_V
    avg_tot_S = np.mean(tot_S, axis=0)
    avg_tot_I = np.mean(tot_I, axis=0)
    avg_tot_R = np.mean(tot_R, axis=0)
    avg_tot_V = tot_V

    return (avg_new_I, avg_new_V, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V)
```

[ ]:
```
def find_best_k_beta_rho_ER(weekly_infected, vacc, n_nodes, k0, beta0, rho0,␣
↪dk, dbeta, drho, n_simulations=10, seed=True):

    initial_beta0 = beta0
    initial_rho0 = rho0

    refinement_done = False

    print("Initial parameters:", k0, beta0, rho0)
    counter = 0
    n_weeks = len(weekly_infected)-1
    n_initial_infected = weekly_infected[0]

    while True:
        parameter_space = np.array(np.meshgrid([k0-dk,k0,k0+dk],␣
↪[beta0-dbeta,beta0,beta0+dbeta], [rho0-drho,rho0,rho0+drho])).T.reshape(-1,3)

        RMSE = []
        avg_new_Is = []
        avg_tot_Ss = []
        avg_tot_Is = []
        avg_tot_Rs = []
        avg_tot_V = np.array(vacc)
        counter += 1

        for param in parameter_space:
```

```
            k = int(param[0])
            beta = param[1]
            rho = param[2]

            p = k/(n_nodes-1)    # this value of p ensures a mean degree of k

            avg_new_I, _, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪simulate_epidemic_with_vaccination_ER(n_nodes, p, vacc, beta=beta, rho=rho,␣
↪n_weeks=n_weeks, n_initial_infected=n_initial_infected,␣
↪n_simulations=n_simulations, random_infected_everytime=True,␣
↪gen_rand_graph_everytime=False, seed=seed)
            avg_new_Is.append(avg_new_I)
            avg_tot_Ss.append(avg_tot_S)
            avg_tot_Is.append(avg_tot_I)
            avg_tot_Rs.append(avg_tot_R)

            RMSE.append( np.sqrt((1/(n_weeks+1))*np.
↪sum((avg_new_I-weekly_infected)**2)) )

        # Now that all the 27 RMSEs have been computed
        new_param_min_RMSE = parameter_space[np.argmin(RMSE)]

        best_avg_new_I = avg_new_Is[np.argmin(RMSE)]
        best_avg_tot_S = avg_tot_Ss[np.argmin(RMSE)]
        best_avg_tot_I = avg_tot_Is[np.argmin(RMSE)]
        best_avg_tot_R = avg_tot_Rs[np.argmin(RMSE)]

        ### Stop criterion
        if np.all( new_param_min_RMSE == (k0, beta0, rho0) ):
            # Do a refinement of the deltas
            if not refinement_done:
                dbeta /= 2
                drho /= 2
                refinement_done = True
                continue
            print(f"A local minimum has been reached! RMSE = {np.min(RMSE)},␣
↪params = {new_param_min_RMSE}")
            return (best_avg_new_I, best_avg_tot_S, best_avg_tot_I,␣
↪best_avg_tot_R, avg_tot_V)
        else:
            print(f"Better params found. Best RMSE = {np.min(RMSE)}, new params␣
↪= {new_param_min_RMSE}")
            k0, beta0, rho0 = new_param_min_RMSE
```

[ ]:
```
vacc = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
weekly_infected = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
n_simulations = 30
```

```
avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪find_best_k_beta_rho_ER(weekly_infected, vacc, n_nodes=934, k0=15, beta0=0.
↪3, rho0=0.6, dk=1, dbeta=0.1, drho=0.1, n_simulations=n_simulations,␣
↪seed=True)

### Plots
n_weeks = len(vacc)
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected (model with best␣
↪params)')
ax.plot(range(n_weeks), weekly_infected, label='Actual n. of newly infected␣
↪individuals each week')
plt.setp( ax, xticks=range(n_weeks),␣
↪xticklabels=list(range(42,53))+list(range(1,6)) )
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()

fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible')
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected')
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered')
ax.plot(range(n_weeks), avg_tot_V, label='Avg. total vaccinated')
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()
```

5.2 Grid search

[ ]:
```
def grid_search_k_beta_rho(weekly_infected, vacc, n_nodes, k_list, beta_list,␣
↪rho_list, a=0, n_simulations=10, seed=True):

    n_weeks = len(weekly_infected)-1
    n_initial_infected = weekly_infected[0]

    RMSE = []
    avg_new_Is = []
    avg_tot_Ss = []
    avg_tot_Is = []
    avg_tot_Rs = []
    avg_tot_V = np.array(vacc)
```

```
    parameter_space = np.array(np.meshgrid(k_list, beta_list, rho_list)).T.
↪reshape(-1,3)

    for i,param in enumerate(parameter_space):
        k = int(param[0])
        beta = param[1]
        rho = param[2]
        print(f"Now testing: ({k},{beta},{rho}), trial {i+1} out of␣
↪{len(parameter_space)}")

        avg_new_I, _, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪simulate_epidemic_with_vaccination(n_nodes, k, a, vacc, beta=beta, rho=rho,␣
↪n_weeks=n_weeks, n_initial_infected=n_initial_infected,␣
↪n_simulations=n_simulations, random_infected_everytime=True,␣
↪gen_rand_graph_everytime=False, seed=seed)
        avg_new_Is.append(avg_new_I)
        avg_tot_Ss.append(avg_tot_S)
        avg_tot_Is.append(avg_tot_I)
        avg_tot_Rs.append(avg_tot_R)

        RMSE.append( np.sqrt((1/(n_weeks+1))*np.
↪sum((avg_new_I-weekly_infected)**2)) )

    # Now that all the RMSEs have been computed
    new_param_min_RMSE = parameter_space[np.argmin(RMSE)]

    best_avg_new_I = avg_new_Is[np.argmin(RMSE)]
    best_avg_tot_S = avg_tot_Ss[np.argmin(RMSE)]
    best_avg_tot_I = avg_tot_Is[np.argmin(RMSE)]
    best_avg_tot_R = avg_tot_Rs[np.argmin(RMSE)]

    print(f"All the sets have been tested! RMSE = {np.min(RMSE)}, params =␣
↪{new_param_min_RMSE}")
    return (best_avg_new_I, best_avg_tot_S, best_avg_tot_I, best_avg_tot_R,␣
↪avg_tot_V)
```

[ ]:
```
vacc = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
weekly_infected = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
n_simulations = 30

k_list = [7,8,9,10,11,12,13,14,15,16]
beta_list = [0.05,0.1,0.15,0.2]
rho_list = [0.6,0.7,0.8,0.9]
avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪grid_search_k_beta_rho(weekly_infected, vacc, a=0, n_nodes=934,␣
↪k_list=k_list, beta_list=beta_list, rho_list=rho_list,␣
↪n_simulations=n_simulations, seed=True)
```

```
### Plots
n_weeks = len(vacc)
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected (model with best␣
 ↪params)')
ax.plot(range(n_weeks), weekly_infected, label='Actual n. of newly infected␣
 ↪individuals each week')
plt.setp( ax, xticks=range(n_weeks),␣
 ↪xticklabels=list(range(42,53))+list(range(1,6)) )
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()

fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible')
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected')
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered')
ax.plot(range(n_weeks), avg_tot_V, label='Avg. total vaccinated')
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()
```

5.3 Random search

[ ]:
```
def random_search_k_beta_rho(weekly_infected, vacc, n_nodes,␣
 ↪k_range=list(range(7,17)), beta_range=(0.03,0.25), rho_range=(0.6,0.95),␣
 ↪n_rand_trials=100, a=0, n_simulations=10, seed=True):

    n_weeks = len(weekly_infected)-1
    n_initial_infected = weekly_infected[0]

    RMSE = []
    parameters = []
    avg_new_Is = []
    avg_tot_Ss = []
    avg_tot_Is = []
    avg_tot_Rs = []
    avg_tot_V = np.array(vacc)

    for i in range(n_rand_trials):
        k = int(np.random.choice(k_range, size=1))
```

```
        beta = np.random.uniform(beta_range[0],beta_range[1],1)[0]
        rho = np.random.uniform(rho_range[0],rho_range[1],1)[0]
        print(f"Now testing: ({k},{beta},{rho}), trial {i+1} out of␣
↪{n_rand_trials}")

        avg_new_I, _, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪simulate_epidemic_with_vaccination(n_nodes, k, a, vacc, beta=beta, rho=rho,␣
↪n_weeks=n_weeks, n_initial_infected=n_initial_infected,␣
↪n_simulations=n_simulations, random_infected_everytime=True,␣
↪gen_rand_graph_everytime=False, seed=seed)
        avg_new_Is.append(avg_new_I)
        avg_tot_Ss.append(avg_tot_S)
        avg_tot_Is.append(avg_tot_I)
        avg_tot_Rs.append(avg_tot_R)

        RMSE.append( np.sqrt((1/(n_weeks+1))*np.
↪sum((avg_new_I-weekly_infected)**2)) )
        parameters.append((k,beta,rho))

    # Now that all the RMSEs have been computed
    param_min_RMSE = parameters[np.argmin(RMSE)]

    best_avg_new_I = avg_new_Is[np.argmin(RMSE)]
    best_avg_tot_S = avg_tot_Ss[np.argmin(RMSE)]
    best_avg_tot_I = avg_tot_Is[np.argmin(RMSE)]
    best_avg_tot_R = avg_tot_Rs[np.argmin(RMSE)]

    print(f"All the sets have been tested! RMSE = {np.min(RMSE)}, params =␣
↪{param_min_RMSE}")
    return (best_avg_new_I, best_avg_tot_S, best_avg_tot_I, best_avg_tot_R,␣
↪avg_tot_V)
```

[ ]:
```
vacc = [5, 9, 16, 24, 32, 40, 47, 54, 59, 60, 60, 60, 60, 60, 60, 60]
weekly_infected = [1, 1, 3, 5, 9, 17, 32, 32, 17, 5, 2, 1, 0, 0, 0, 0]
n_simulations = 30

k_range = list(range(7,17))
beta_range = (0.03,0.25)
rho_range = (0.6,0.95)
n_rand_trials = 100
avg_new_I, avg_tot_S, avg_tot_I, avg_tot_R, avg_tot_V =␣
↪random_search_k_beta_rho(weekly_infected, vacc, a=0, k_range=k_range,␣
↪beta_range=beta_range, rho_range=rho_range, n_nodes=934,␣
↪n_rand_trials=n_rand_trials, n_simulations=n_simulations, seed=True)

### Plots
n_weeks = len(vacc)
```

```
fig = plt.figure(1, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_new_I, label='Avg. newly infected (model with best␣
 ↪params)')
ax.plot(range(n_weeks), weekly_infected, label='Actual n. of newly infected␣
 ↪individuals each week')
plt.setp( ax, xticks=range(n_weeks),␣
 ↪xticklabels=list(range(42,53))+list(range(1,6)) )
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()

fig = plt.figure(2, figsize=(16,8))
ax = plt.subplot()
ax.plot(range(n_weeks), avg_tot_S, label='Avg. total susceptible')
ax.plot(range(n_weeks), avg_tot_I, label='Avg. total infected')
ax.plot(range(n_weeks), avg_tot_R, label='Avg. total recovered')
ax.plot(range(n_weeks), avg_tot_V, label='Avg. total vaccinated')
plt.setp(ax, xticks=range(n_weeks), xticklabels=range(1,n_weeks+1))
ax.set_xlabel('Week', size='large')
ax.set_ylabel('N. of individuals', size='large')
ax.legend(prop={'size': 12})
plt.grid()
```