Scientific Computation Project 1

Andreea Patra — CID 01365348

March 2, 2020

Part 1

1.1)

This function calculates the minimum number of flights required to travel between 2 distinct airports across a network. It will also output the number of distinct routes which have this minimum number of routes. It has as input an adjacency list, a starting node and a destination node. The adjacency list is a list of list and the i^{th} element is a list of nodes that are directly linked to the i^{th} node. The output is a list with first element as the minimum number of flights and the second element as the number of distinct routes.

The algorithm used is Breadth First Search to search for the shortest path. Initially it initialises a list, "explored" to keep track of explored nodes, a queue,"queue", to keep track of all the paths to be checked and a list, "list_flights" to keep track of all the shortest paths found. We will check if the starting node equals the destination node. We will use a while loop and iterate over all the possible paths in the queue. We will use a deque data type which will allow us to have a time complexity of O(1) when we get the first element of the queue. Therefore we will get the first path from the queue and and the last node of the path. We will use an if statement to check if that node has been checked before. This allows us to iterate through the adjacency list at most ones for each node. If the node was not explored, we will iterate through all its neighbours. We will create a copy of the path and attach to it the neighbour. Then, we will attack the new path to the queue which will then be checked. If the neighbour matches the destination and it has a minimum length, we append the new path to the "list_flights". At the end of the if statement, we will add the node to the list of explored nodes. Finally, we will calculate the minimum number of flights and the number of distinct routes with that minimum. If there is no path between the nodes, we return an empty list.

Moving on, we will analyse the time complexity. We assume that V is the total number of nodes in the network and E is the total number of edges. For a complete graph, we have $fracV(V-1)2$, therefore E is at most V^2. Inside the while loop, we have O(1) operation for "popleft" and for getting the last node of the list. We then have a comparison and a for loop. The for loop has copy

operation of the path which has complexity O(n) where n is the length of the path and 2 O(1) appending operations, one O(1) operation of calculating the length,one O(1) assignment operation, one O(1) comparison and potentially one 2 O(1) appending operation to "list_flights". Therefore, knowing that the sum of all the edges of in the network, the for loop has as the worst case, a time complexity of O(E). We iterate through the nodes at most once, thus the time complexity will be O(V). This final time complexity for the while loop is O(V+E). The final operations get the minimum number of flights and calculate the number of distinct routes. These 2 operations have O(1) complexity. Therefore, the function has complexity O(V+E).

1.2i)

This function focuses on analysing journeys on a subway network. For each direct connection between nodes i and j, we have a density d_{ij} which represents the average number of people on the journey. We define a safety factor S_{ab} as the maximum density encountered on the journey from a to b. The function should return the safest journey as th one with the lowest S. As input, we have an adjacency list defined as in 1), a starting node and a destination node. The output is a list whose first element is a list that represents the full journey and the second element is the safety factor.

The function follows the same Breadth First Search algorithm as in the previous part but with some modifications. Our queue is going to be a list of 2 elements with the first element the path and the second as the safety factor. When we iterate through the neighbour we look at the neighbour and the density as well. We have an additional if statement which checks if the density is bigger than the previous one and if it is, the safety factor of the path is updated. The other if statement checks whether the neighbour is the destination node and if the safety factor is smaller. If the requirements are met, then the new path is added to the "list_stations". In the end, we return the safest journey and if there are more safe journey, we return either of them.

As for the time complexity, it remains the same as the if statement which was introduced has operations with O(1) complexity. Thus, the time complexity for the function is O(V+E).

1.2ii)

For this part, we will consider the same subway stations, but we are now provided with durations of direct journeys. We are asked to find the shortest journey. As input we have an adjacency list, a starting node and a destination node. If there are multiple shortest journeys, we return the one with the least amount of stations.

For this function, we use the Dijkstra algorithm. We initialise a set where we assign initially the distance for each node as being infinite. The set "vertices" is a set of the nodes from the network and the previous set will keep track of the previous node in the path from the node i to j. We set the distance of the starting node to be zero. From the distance set, we remove the node with the shortest distance, which initially will be the starting node. We iterate through the adjacency list for this removed node. We set the provisional distance of the neighbours to be the weight of edge

between the removed node and the neighbour plus the shortest path from the removed node and the source which can be found in the dist set. Every time we remove the node with the smallest distance and update the dist set, we recalculate the provisional distances and update them only if they are smaller. When the destination node is reached, we go through our previous set and construct the path. The distance is the one that belongs to the destination node in the dist set. Finally, if there are multiple journey, we return the one with the least amount of nodes.

Moving on, we will investigate the time complexity. The while loop has V iterations. Calculating the minimum, has a time complexity of O(n) where n is the size of the set. The operations inside the for loop have time complexity O(1). Considering the fact that we calculate the minimum on each iteration, the time complexity would be of the form $O(V^2 + E)$. The next while loop will not affect the time complexity as it does not depend on the number of nodes or edges. Implementing a Dijkstra algorithm with a binary heap using heapq will reduce the time complexity to $O((E + V) \log_2 V)$ as extracting, rearranging the tree and adding a new node are done in $O(\log_2 V)$.

1.3)

For this part, we will consider a network of stations and an outside station. For the outside station x, we know the fare for reaching each in-network station from x and the returning fare from the in-network station to x. We are also given an adjacency list which has cycling routes between the stations from inside the network. We need to calculate the cheapest journey between 2 distinct in-network stations.

Firstly, we calculate all the possible combinations of stations and taking note of the fare. For each node, we check all the other nodes and calculate the fare. Then, we sort the list of the all possible combinations. Then, using a Breadth First Search algorithm we check if there is actually a path between those nodes.

The first part of the function with the nested loop has time complexity of $O(V^2)$ where V is the number of in-network stations. Inside the loop, we have O(1) complexity operations which do not affect the final complexity. The sorting operation is done in $O(V \log V)$. The next for loop has V iterations and each iteration has a time complexity of O(V+E). Therefore, the final complexity is $O(V^2 + V^2 + EV + VlogV)$=$O(V^2 + EV + VlogV)$=$O(V^2 + VlogV)$.

Part 2

2.1i)

The function "rwgraph" simulates M Nt-step random walks on a graph given as input. All walkers start at the same node which is given as input. The probability of going from node i to node j is $\frac{A_{ij}}{q_i}$ where q_i is the degree of the node i.

As we have an unweighted, undirected graph, the nodes that have a direct connection to node i

have an equal probability of being chose. X is a matrix of size (M,Nt+1). We have a nested loop. For each simulation, we have Nt iterations where we get the neighbours of the the node and we choose a random neighbour to continue our walk. Therefore, the time complexity is O($M \times Nt$).

2.1ii)

We will now analyse the simulation results as M and Nt increases using Barabasi-Alber graph with 2000 nodes and 4 edges to attach from a new node to an existing one. Initially, we can define a stationary distribution for an undirected graph. It is known that the stationary distribution for node y, is $\frac{q_y}{2|E|}$ where q_y is the degree of node y. That means that if we have a Nt-steps random walk, we would expect to visit node y, $Nt \times \frac{q_y}{2|E|}$. This can easily be seen from Figure 1 where we have plotted the the degree distribution for large Nt. We can see that the nodes with lower degree are visited more often as it is expected from our formulation. As expected, along the simulation the distribution of degrees is kept due to the fact that the transition matrix is a Markov Chain matrix and it has its properties.

Secondly, we have increased M and Nt and plotted the histogram of last node in all simulations, figure 2. We can see that for lower values of M and Nt around 250, the nodes appear at most twice in the simulations. As we increase, we have more nodes appearing more times. When M and Nt are the highest, the trend is an exponential decay with the peak for a lower node number. Therefore we can see that eventually we oscillate around the node labels 0-25.

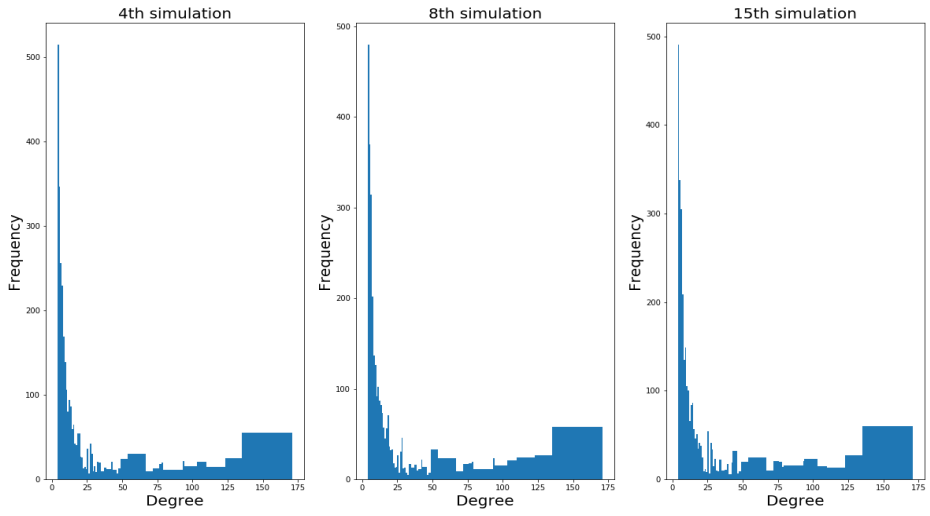

Figure 1

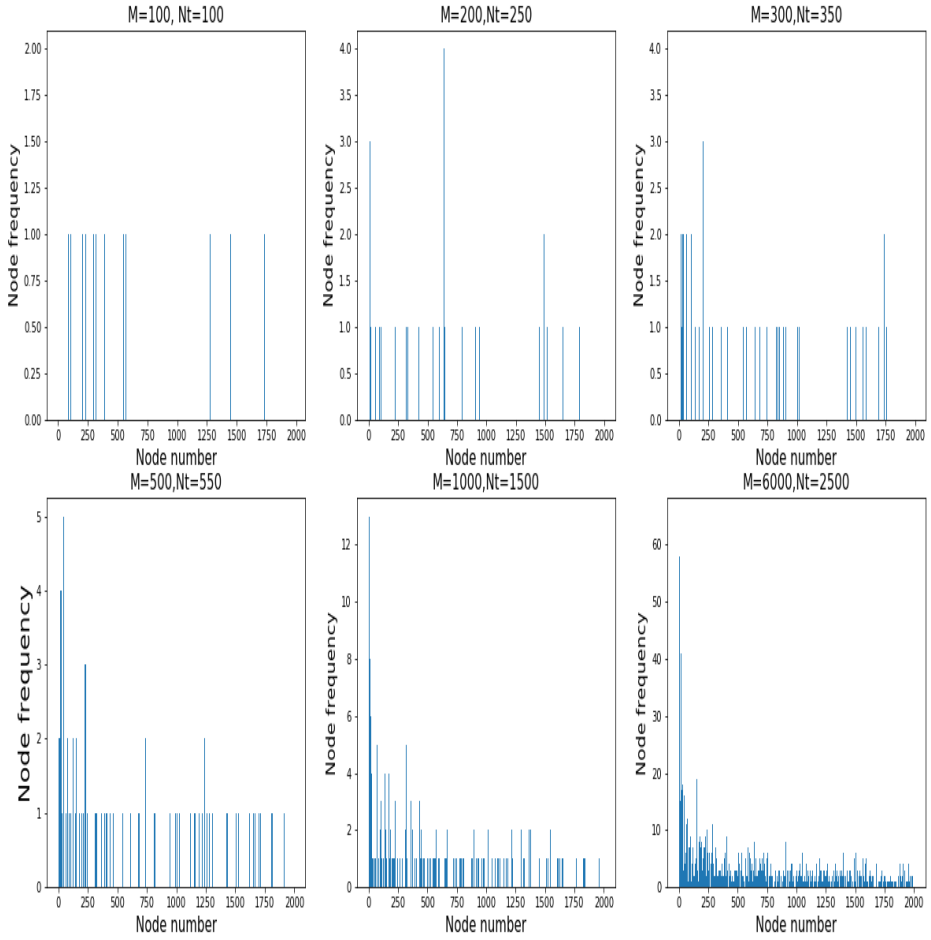
M=100, Nt=100
M=200,Nt=250
M=300,Nt=350
M=500,Nt=550
M=1000,Nt=1500
M=6000,Nt=2500
Node frequency
Node number

Figure 2

2.1iii)

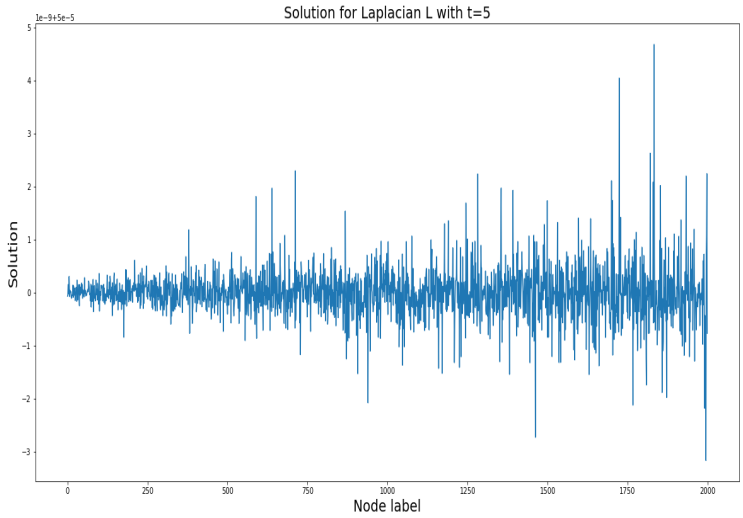

Figure 3

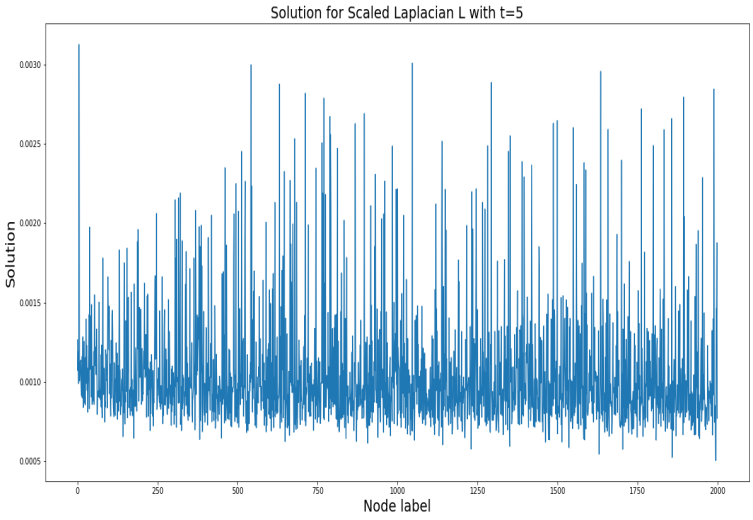

Figure 4

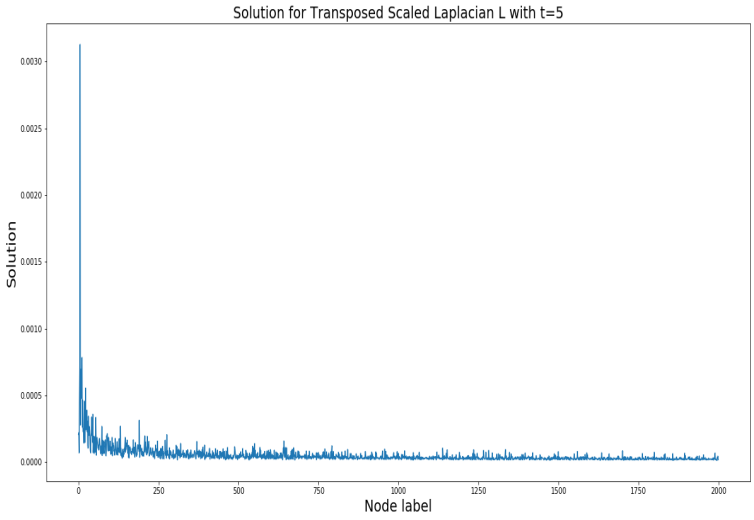

Figure 5

In this part we will look at the similarities and differences of linear diffusion and random walk. Diffusion described the flux inside the network, how the particles flow inside. As stated in the lectures, the linear diffusion for node i can be expressed as the following PDE

$$\frac{\partial f_i}{\partial t} = -\mathrm{C}\Big(\sum_{j=1}^{N} L_{ij} f_j\Big)$$

We will consider 3 types of the Laplacian matrix L. The first one is $\mathrm{D} - \mathrm{A}$, the second is $\mathrm{I} - \mathrm{D}^{-1}\mathrm{A}$ and the last one is the transpose of the second one. In order to calculate the solution of the linear diffusion, we make use of the exponential of a matrix. The solution is of the form $f_0 \exp^{Mt}$. If we look at final time 5, and compare how the flux evolves over the nodes. We can see that the distribution of the transponsed scaled Laplacian , Figure 5, matches, the one from the last plot in Figure 1 when M and Nt are large. We know that the random walk can also be computed from the transition matrix P which is incorporated in the transposed scaled Laplacian, thus the resemblance in the plot. When we investigate the eigenvalues, we can see that the magnitude of the minimum eigenvalue for $\mathrm{D} - \mathrm{A}$ is very large compared to the magnitude of the eigenvalue for the transition matrix from the random walk is around 1. This is important as it decides the random node to be chose and it describes the stationary distribution. Both the scaled Laplacian and its transpose have a similar maximum value for the eigenvalue, but they differ in the minimum value. The transpose of the scaled Laplacian has a similar value to the transition matrix. Due to the form of the solution of the linear diffusion, when t goes to infinity, we would only have a contribution from the zero eigenvalue and in general the most contribution comes from the second smallest eigenvalue.

2.2i)

At this part, we will look to model epidemics based on 2 models. We have i_j as the fact that node j is infected. We can model the function with the following PDE,

$$\frac{di_j}{dt} = -\beta i_j + \sum_{k=0}^{N-1} \gamma A_{jk} i_k (1 - i_j)$$

For the second model we have,

$$\frac{di_j}{dt} = \sum_{k=0}^{N-1} \alpha L_{jk} (s_k - s_j)$$

$$\frac{ds_j}{dt} = i_j$$

The function "modelA" simulates model A Nt steps and "modelB" simulates model B. Initially we set N as being the number of nodes of the graph. Then we have iarray as the fraction of infected at a specific node with respect to time and tarray the span of time. We calculate the adjacency matrix and set the initial condition. Then, the function RHS computes the right hand side of the differential equation. We use a vectorised form of the right hand side equation. The first operation is done by multiplying β with i_j. Another operation is the difference between 1 and i_j. Another operation is the multiplication betweeen A and i_k. The next 2 operations are the transpose and multiplication with gamma. To compute $\frac{di_j}{dt}$, we need 7 operations.

Model B has similar approach but with different RHS expressed as a system of equations using the concatenating method.

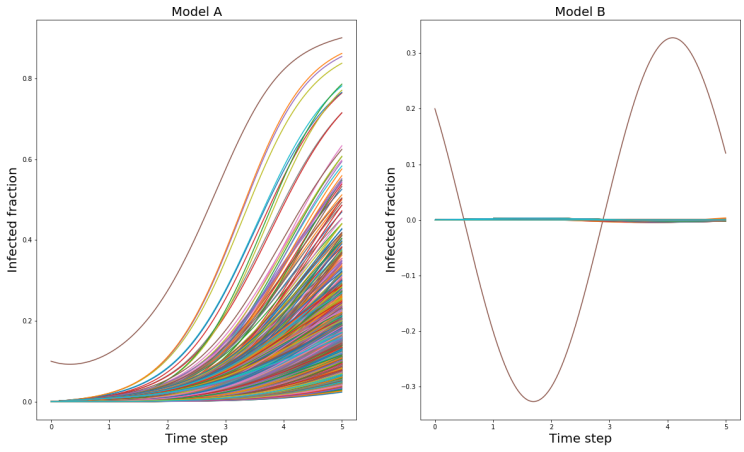

Figure 6

2.2ii)

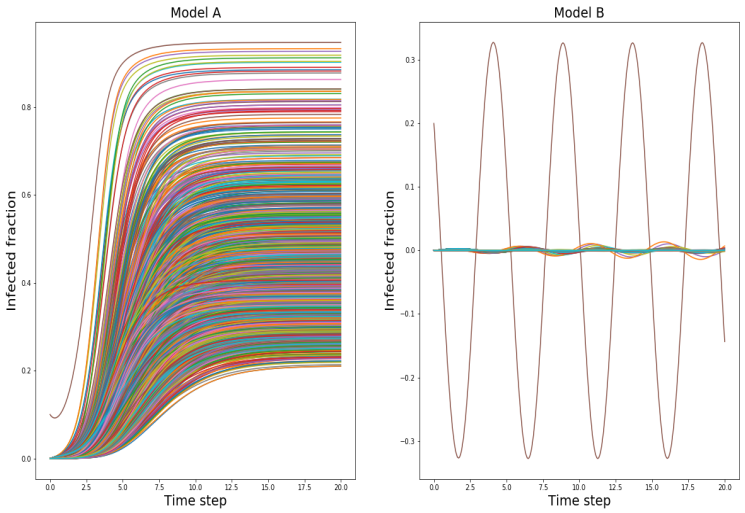

Figure 7

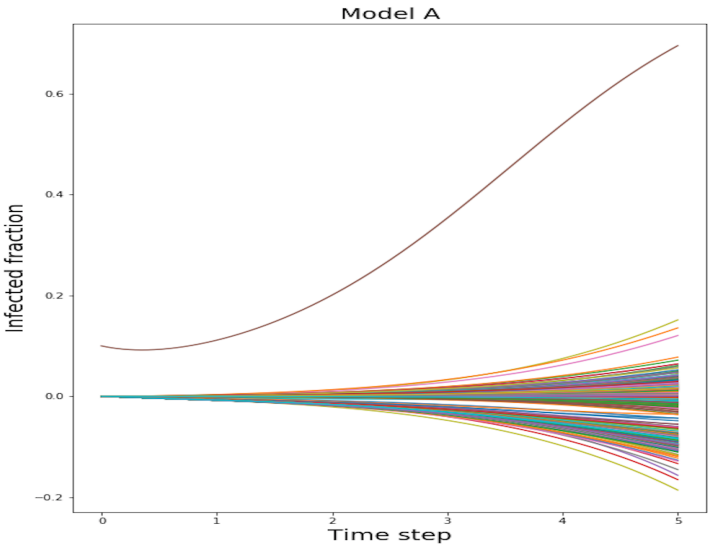

Figure 8

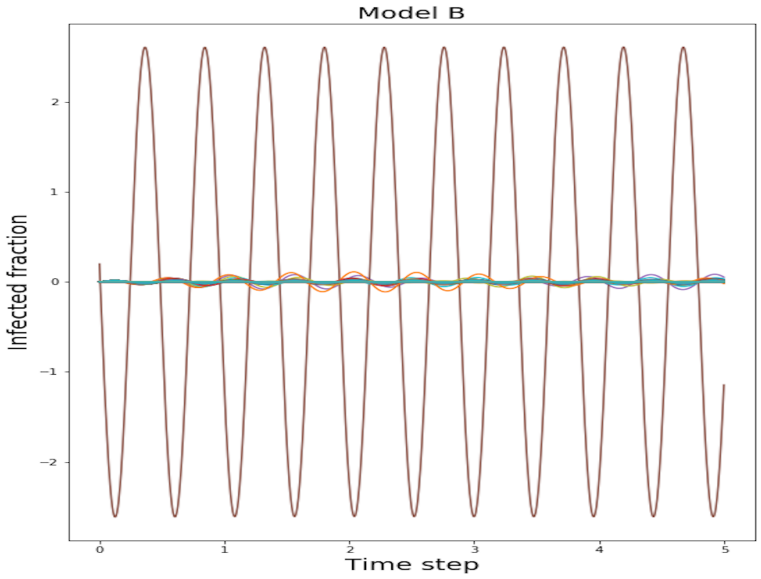

Figure 9

This subsection focuses on the similarities and the differences between the dynamics presented by model A, B and the linear diffusion. Looking at both the plots for model A and model B, we can see that the initial condition influences the dynamics in time. It sets the starting point. Even if in model A, the impact is not considerable, for model B it influences the oscillations considerably. The maximum node to which we assigned a starting value has big oscillations, whereas the other nodes are gravitating really close to 0 for tf=5 as we can see from figure nn. If we increase tf, we can see that the other nodes start having a sine wave behaviour as well. As well, we can see that the dynamics of model A lies entirely above 0 and it has an exponential behaviour. If we increase tf, we can see that it converges somewhere around 1. At first it has a high peak, fast and then it converges. The flux spreads fast until it reaches a stationary distribution and becomes steady as we have discussed in random walk and liner diffusion. If we compare with linear diffusion, we can see that both of them have positive values and one has negatives as well.

Our model B dynamics has a period of 4. Varying the parameters $\alpha, \beta and \gamma$, we can see how they influence the dynamics. If γ and β remain positive we get the the image from figure x, with the exponential growth and converging to 1. On the other hand, if we make γ negative, but we keep β positive, we have a beginning of a saddle node behaviour. Increasing α in model B, will increase the magnitude of the oscillations as well as the period of them. If we increase the tf to 20 for the Transposed Scaled Laplacian, we get a similar distribution to the figur xx which has as the x axis the node label and on the y axis the infected ratio on the last iteration.