Scientific Computation Project 2

01724711

March 6, 2023

Part 1

1.

part1q1 is a function that implements a priority queue using the Python heapq module. It can be used to keep track of the nodes in the graph that have not yet been visited by Dijkstra's algorithm and to select the node with the minimum tentative distance as the next node to visit. The function takes a list Hlist of 2-element lists, where the first element is the tentative distance of a node from the source node, and the second element is the node's identifier. The option argument controls the behavior of the function.

- When option is 0, the function creates a heap from the input list Hlist using heapq.heapify and returns the sorted Hlist and a dictionary Hdict that maps node identifiers to their corresponding 2-element lists. This is an initilisation step that then allows for the function to be called with option 1 or 2.
- When option is 1, the function removes the node with the minimum tentative distance from Hlist and returns the new Hlist, Hdict, the removed tentative distance wpop, and the removed node identifier npop.
- When option is 2, the function adds a new node with tentative distance x[0] and node identifier x[1] to Hlist and Hdict. If a node with the same identifier already exists in Hdict, its tentative distance is updated to x[0].

The function returns the updated Hlist and Hdict after each operation.
The heappop method of the heapq module has a running time of $\mathcal{O}(log_2 n)$, where n is the number of elements in the heap. This is because the heap is maintained as a binary tree, and each time an element is removed, the tree is restructured to maintain the heap property, which takes log n time. The heappush operation has a time complexity of $\mathcal{O}(log_2 n)$, where n is the size of the heap. It involves inserting a new element at the end of the heap and then adjusting its position by comparing it with its parent and swapping if necessary, until it is in the correct position. The worst-case scenario occurs when the new element is the maximum element in the heap, and it needs to be moved to the root, which would require log n comparisons and swaps. However, in practice, the average case is much faster, and the worst case is rare. Since lookup and assignment in dictionaries are $\mathcal{O}(1)$ this results in option 2 having $\mathcal{O}(log_2 n)$ asymptotic running time.

2. (a)

The function part1q2 is an implementation of Dijkstra's algorithm that takes a weighted NetworkX graph G, and two integers s and x corresponding to nodes in the graph. The problem that the code is trying to solve is finding the shortest path from node s to node x in the graph G.

The strategy used by the function to solve this problem is as follows:

1. Initialize Mdict with the start node s and its distance 1, Fdict as an empty dictionary, Plist as a list of empty lists with n elements (where n is the number of nodes in G).
2. While Mdict is not empty:
   - Select the node with the smallest distance in Mdict and remove it from Mdict.
   - If the selected node is the target node x, return the distance and the path to x.

- Add the selected node and its distance to Fdict.
- Update the distance of the neighbors of the selected node in Mdict if it is shorter than their current distance, and update their path in Plist.
- If a neighbor is not in Mdict or Fdict, add it to Mdict with its distance and update its path in Plist.

3. If the target node x was not reached, return Fdict.

In the traditional Dijkstra's algorithm, we would update the tentative distance to a neighboring node by adding the weight of the edge connecting the current node to that neighboring node. However, in this implementation, we update the tentative distance by multiplying the current distance by the weight of the edge. Taking logarithms of the distances makes this equivalent to the traditional Dijkstra's algorithm with edge weights replaced by their logarithms. In the traditional algorithm we require the weights to be non-negative. Here since we have this slight change in updating the distances we require the weights to be greater or equal to 1 since taking the log of such a weight would result in non negative number. So the minimum distance we are working with here is equivalent to minimising the sum of the log distances between nodes along the path from s to x.

2. (b)

The implementation in part1q2 has one or more inefficiencies that negatively affect the running time when the function is applied to large sparse graphs. One of the inefficiencies is the use of a dictionary Mdict to keep track of the nodes that are still in consideration for the shortest path calculation. In each iteration, the code searches through the entire dictionary to find the node with the smallest distance. This operation takes $\mathcal{O}(n)$ time in each iteration, where n is the number of nodes in the graph. This can be very slow for large sparse graphs. Another inefficiency is the use of a list Plist to keep track of the paths to all nodes from the source node s. In each iteration, the code creates a copy of the list for each node that is added to Mdict. This operation takes $\mathcal{O}(n)$ time in each iteration. Since the list can contain up to n nodes, this can be very slow for large sparse graphs.

A more efficient implementation would be to use a priority queue (e.g. a binary heap or Fibonacci heap) instead of a dictionary Mdict to keep track of the nodes that are still in consideration for the shortest path calculation. This would allow the code to find the node with the smallest distance in $\mathcal{O}(log_2 n)$ time in each iteration. Additionally, instead of using a list Plist to keep track of the paths, the code could use a dictionary Pdict to keep track of the shortest path to each node. This would allow the code to avoid creating copies of the path list in each iteration. Overall, these changes would significantly improve the running time of the function for large sparse graphs.

In my implementation, pred is a dictionary where the keys are node indices and the values are the predecessor nodes in the shortest path from the source node. The path from the source node to a target node is constructed by starting from the target node and following the predecessor nodes until the source node is reached. The append() calls on Plist in the previous implementation are replaced with updates to the pred dictionary. This also removes the extend call which is $\mathcal{O}(n)$ to a dictionary reassignment which is $\mathcal{O}(1)$.

We utilise the part1q1 implementation to have $\mathcal{O}(log_2 n)$ minimum distance node removal, insertion and key modification.
We know from lectures that the algorithm can be summarised as so:
Start with a heap of neighbors of the source node
1. Remove n* and re-structure heap
2. Adjust weights of neighbors of n* (if needed) and re-structure heap
3. Insert unexplored neighbors of n*
Overall cost of step 1: $\mathcal{O}(nlog_2 n)$
Overall cost of step 2: $\mathcal{O}(mlog_2 n)$
Overall cost of step 3: $\mathcal{O}(nlog_2 n)$
By optimising over these two inefficiencies we reduce the time complexity of the algorithm to $\mathcal{O}((m + n)log_2 n)$, where m is the number of edges and n is the number of nodes in the graph. This is much better than the time complexity of $\mathcal{O}(n^2)$ of the previous implementation.

Part 2

2.

A naive approach to find an equilibrium with the given conditions is as follows:

1. set the seed to 1
2. compute the solution to the model using part2q1 code
3. check the equilibrium point that the solution converges to meets the given conditions
4. if meets conditions then plot to verify that equilibrium converges to 25 or more different opinions
5. if plot reveals this is not true then increase seed by one and go back to 2
6. if we do not meet conditions then increase seed by one and go back to 2
7. if plot confirms conditions met then return the equilibrium

The reason behind the plot is because it can be difficult to decide what a unique opinion is. Because of all the decimal points the solution to the model with any seed will have 50 unique opinions but in reality they may all be infinitesimally close to each other. So we define uniqueness of opinion to 1 decimal place and then use the plot to distinguish if two opinions look like they are about to converge to the same one or not. With this approach I obtained the following plot until convergence to an equilibrium point (seed=285084.) My approach is to try a more regularly spaced initial condition x_0. This is so that we do

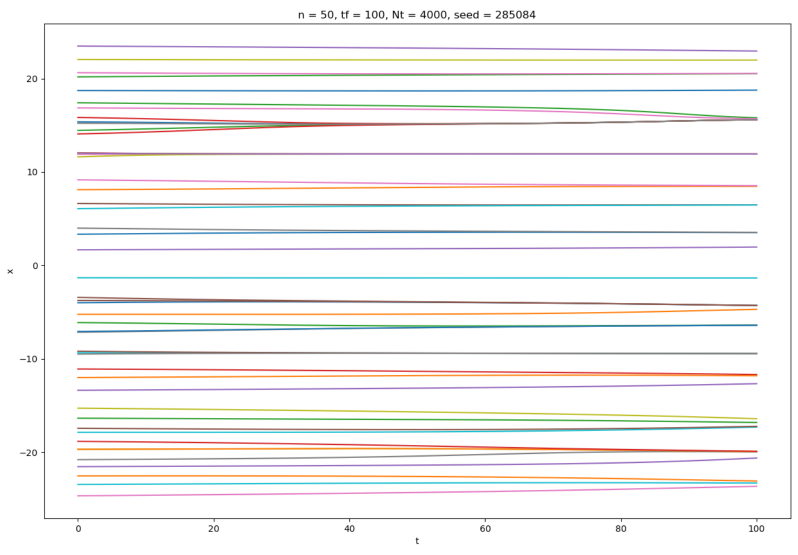

Figure 1: Equilibrium solution satisfying constraints using naive approach

not have multiple very close opinions initially (since these would end up all converging to one opinion.) We use numpy linspace between -1000 and 1000 for the same number of opinions as our initial condition. Now we get the following opinion dynamics until convergence. Now we are able to observe far more clearly that the equilibrium satisfies the conditions. In fact we discover a non trivial equilibrium point with our initial guess as the RHS of the dynamics equation is $\underline{0}$ for the initial guess so the opinions do not change from the initial opinions.

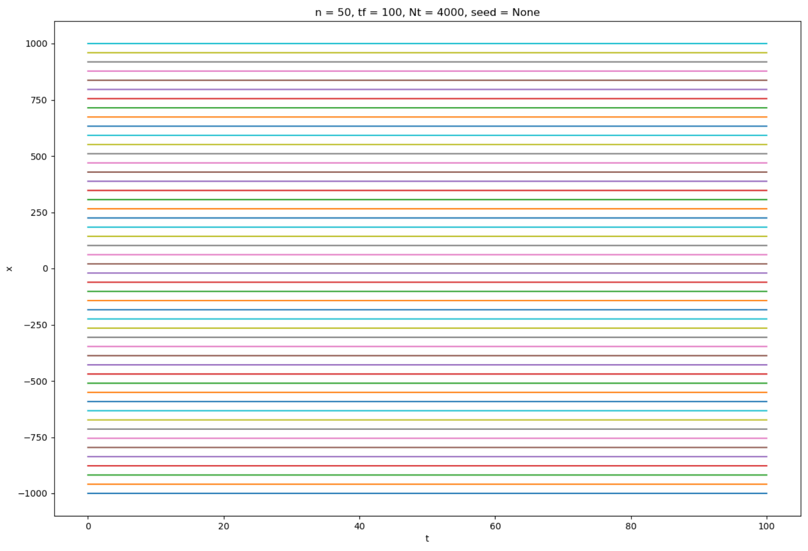

Figure 2: Equilibrium solution satisfying constraints using spaced out initial condition approach

3.

Equilibria can be unstable, and their stability is crucial in determining the dynamics of a system. When dealing with autonomous ordinary differential equations, equilibrium points can be classified as stable or unstable based on the behavior of trajectories around them. Suppose that we have a set of autonomous ordinary differential equations, written in vector form:

$$\dot{x} = f(x). \tag{1}$$

Let x^* be an equilibrium point. By definition, $f(x^*) = 0$. We can take a multivariate Taylor expansion of the differential equation and evaluate the Jacobian at the equilibrium point. This allows us to write a linear differential equation that can be solved using the eigenvalues of the Jacobian.

$$\begin{aligned} \dot{x} &= f(x^*) + \left.\frac{\partial f}{\partial x}\right|_{x=x^*} (x - x^*) + \dots & (2) \\ &= \left.\frac{\partial f}{\partial x}\right|_{x=x^*} (x - x^*) + \dots & (3) \end{aligned}$$

The partial derivative in Equation (3) is to be interpreted as the Jacobian matrix. If the components of the state vector x are $(x_1, x_2, ..., x_n)$ and the components of the rate vector f are $(f_1, f_2, ..., f_n)$, then the Jacobian is

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}. \tag{4}$$

Now define $\delta x = x - x^*$. Taking a derivative of this definition, we get $\dot{\delta x} = \dot{x}$. If δx is small, then only the first term in Equation (3) is significant since the higher terms involve powers of our small displacement from equilibrium which rapidly approach zero. If we want to know how trajectories behave near the equilibrium point, e.g. whether they move toward or away from the equilibrium point, it is enough to just study this term. Then we have

$$\dot{\delta x} = J^* \delta x, \tag{5}$$

where J^* is the Jacobian evaluated at the equilibrium point. The matrix J^* is a constant, so this is simply a linear differential equation. From the theory of linear differential equations, the solution can be written as a superposition of terms of the form $e^{\lambda_j t}$ where λ_j is the set of eigenvalues of the Jacobian.

The eigenvalues of the Jacobian are, in general, complex numbers. Let $\lambda_j = \mu_j + i\nu_j$, where μ_j and ν_j are, respectively, the real and imaginary parts of the eigenvalue. Each of the exponential terms in the expansion can therefore be written

$$e^{\lambda_j t} = e^{\mu_j t} e^{i\nu_j t}$$

The complex exponential in turn can be written

$$e^{iv_j t} = cos(v_j t) + isin(v_j t)$$

The complex part of the eigenvalue therefore only contributes an oscillatory component to the solution. It's the real part that matters: If $\mu_j > 0$ for any j, $e^{\mu_j t}$ grows with time, which means that trajectories will tend to move away from the equilibrium point. This leads us to the conclusion that an equilibrium point x^* of the differential equation 1 is stable if all the eigenvalues of J^* , the Jacobian evaluated at x^* , have negative real parts. The equilibrium point is unstable if at least one of the eigenvalues has a positive real part. The construction of the Jacobian for our specific differential equation is as follows:

$$J_{ij} = \frac{\partial}{\partial x_j}(\frac{dx_i}{dt}) = \frac{\partial}{\partial x_j}(\frac{1}{n}\sum_{k=1}^{n} -(x_i - x_k)e^{-(x_i-x_k)^2})$$

if $i \neq j$:

$$J_{ij} = \frac{1}{n}(1 - 2(x_i - x_j)^2)e^{-(x_i-x_j)^2}$$

if $i = j$:

$$J_{ij} = \frac{1}{n}\sum_{k\neq i}(-1 + 2(x_i - x_k)^2)e^{-(x_i-x_k)^2}$$

In the code for part2q3 we implement the jacobian function and pass the first equilibrium we found in part2q2 into the jacobian function. We then find the eigenvalues using numpy inbuilt functions and return them as the output in part2q3. From these we can make two conclusions. Fistly, all the eigenvalues have no complex part and so we expect no oscillatory behaviour from small perturbations to the equilibrium. Secondly, we observe that we have some eigenvalues with positive real part and some with negative real part meaning that the equilibrium is unstable and more specifically a saddle point. We repeat this for the second equilibrium we found in part2q2. Here we find that all eigenvalues are 0. Hence all real parts are 0 and so there is no growth or decay and all complex parts are 0 so there are no oscillations. Hence this equilibrium point is stable and more specifically a centre point.

4.(b)

There are specific differences in the dynamics of the two models and the magnitude of the differences is dependent on μ. We plot the average opinion for 10 individuals, until $tf = 50$, for both models as well as the standard deviation. The average for the stochastic model is over 100 simulations and the deterministic model is itself the average since there is no randomness between simulations. For $\mu = 0.2$ we see that the the stochastic model average opinions stay relatively similar to their initial position and seem to be largely unaffected by other opinions, having only very slight fluctuations from initial opinion whereas the deterministic opinions average are observed to quite strongly be influenced by those of similar opinion and so opinions of individuals end up merging into one where we have multiple individuals with similar initial opinion all of the same opinion in the end. When we increase μ to 2 we observe that the stochastic model average opinions are no longer as unwavering as before. They seem fluctuate a lot more but more importantly they do not seem to oscillate around the same initial opinion rather tending to convolve to other opinions similar to theirs. However, they do not completely stay of the same opinion as once they agree with another individual but again fluctuate and can even end up altering their opinion completely. This would seem like a far more realistic model since in real life peoples opinions will not stay constant once they agree with other people on something. Rather, humans are far more complex beings that are able to think critically for themselves hence being able to tend away from opinions of others similar to themselves. As we see from the standard deviation plots, the μ parameter affects the volatility of the model. As μ increase we have increased variability in opinions and individuals become subject to radical change of opinion as time increases. As we would expect the standard deviation follows a square root rule. So as time goes on we see that the variability of an individuals opinion increases with respect to this rule. The rule is also scaled by μ hence when increasing μ we have higher scaled volatility of individuals opinions. The deterministic model opinions have no volatility as expected.

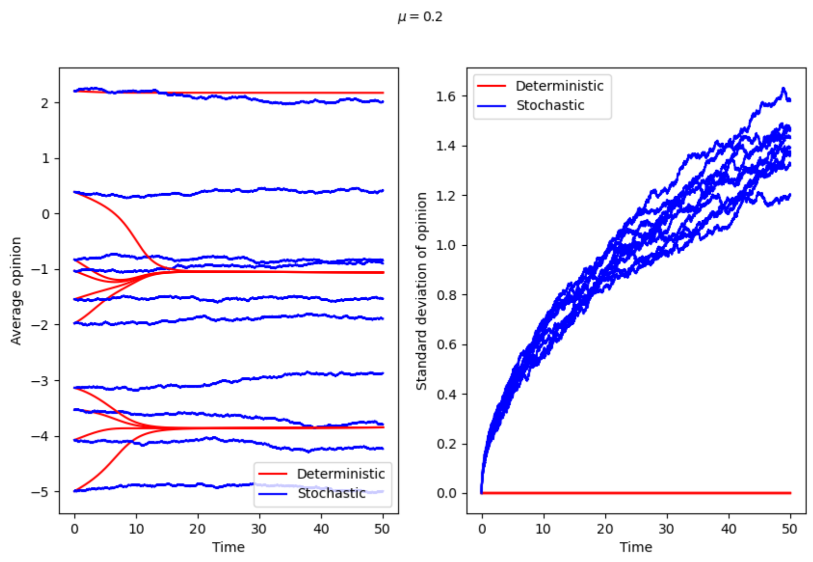

Figure 3: Average and standard deviations of opinion over time with $\mu = 0.2$

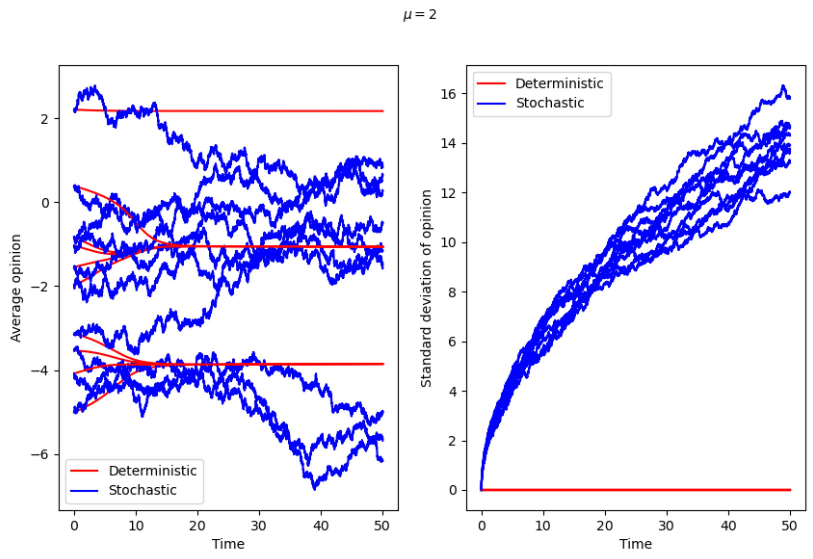

Figure 4: Average and standard deviations of opinion over time with $\mu = 2$