Scientific Computation Project 2

02099078

November 24, 2023

Part 1

1.

Both functions are attempting to find the shortest path from a source node to a target node (both given as inputs) within a graph. By exploring nodes with the shortest distance from the source node and repeatedly updating distances, the shortest path can be reconstructed, allowing us to find the shortest distance to the target node.

The function searchGPT is using a strategy similar to Dijkstra's algorithm. Initially, the distances from the source node to all nodes other than the source node itself are set to infinity. A priority queue is then created, which is used for deciding which order we explore the nodes in. A dictionary is also used to keep track of the parent node for each node, so that when the target node has been found, the shortest path can be remade.

While the priority queue is non-empty, we pop the node with shortest distance from the source node. If this is the target node, then the shortest path is remade and returned as a list of nodes from the source node to the target node, and the distance is also returned. If it is not the target node, we check the neighbouring nodes based on the distance to the current node, and the distances and parent nodes are updated. The distances are updated by picking the maximum value between the distance to the neighbour from the current node and the weight of the edge to the neighbour. If the target node is not found, infinity is returned.

2.

The searchPKR2 function has been modified to efficiently construct a list that is the same as the list returned by searchGPT. It now keeps track of parent nodes using the dictionary created and once the target node x is found, the shortest path is remade backwards from the target node to the source node and returned as a list.

We first look at the differences in correctness between the two functions. The significant difference is in the way the distances are calculated. searchGPT uses the maximum value between the distance to the neighbour from the current node and the weight of the edge to the neighbour. In Dijkstra's algorithm, the distances are updated by summing the distance from the source node to the current node and the weight of the edge to the neighbour. Due to this difference, the method used by searchGPT may incorrectly update distances, meaning the calculated shortest path might not in fact be the shortest path, since the shortest path could be reconstructed incorrectly. On the other hand, searchPKR2 uses the maximum value between the minimum distance from the source node to the current node and the weight of the edge to the neighbour. This is similar to Dijkstra's algorithm, and is more likely in comparison to searchGPT to update distances accurately and hence return the correct shortest path, since the shortest path is more likely to be reconstructed correctly. However, it is important to note that if the source node and the target node are disconnected, searchPKR2 will fail whereas searchGPT will not since infinity is returned if the target node is not found.

Next, we look at the differences in computational cost between the two functions. To do this, we analyse the time complexity of both. The computational cost is the same for most parts of both functions. The main difference is in the initial stage of the code. In searchGPT, the initial distances are set to infinity for all nodes. While doing this for each node has time complexity O(1) since a dictionary is used, if we have n nodes, initialising the distances for all nodes will be O(n). On the other hand, in searchPKR2, initialising dmin is an O(1) operation. Creating Fdict, Mdict, and Mlist all have constant time complexity, and calculating n, as well as adding a node to G are also O(1) operations. This results in searchPKR2 being a slightly more efficient function with regards to computational cost as there is no O(n) operation in the initialisation stage of the code. Overall, both functions will still have similar computational costs due to the distance calculations and updating of the priority queue having the same time complexity for both functions.

Part 2

1.

Various changes were made to compute solutions to this problem more efficiently. The RHS function computes the derivatives based on the system of equations and stores them in the dydt array. To make this more efficient, the task has been carried out without using a for loop, unlike in the original function part2q1.

In addition, solve_ivp has been implemented to solve the system of linear ODEs. It takes as parameters the RHS function, the initial condition, the time interval, and the number of time steps. I chose to use the 'LSODA' method after some experimentation, as this produced the quickest wall time. The relative tolerance was also set to $1e-6$ to provide the desired level of accuracy. The times and their corresponding solutions are assigned to arrays and then returned. In comparison, part2q1 uses the explicit Euler method to compute the numerical solutions to the system, which is far less efficient.

2.

In order to analyse the evolution of the system from these initial conditions, I decided to create four functions, each plotting a graph, as this would help me to deduce the stability of the system.

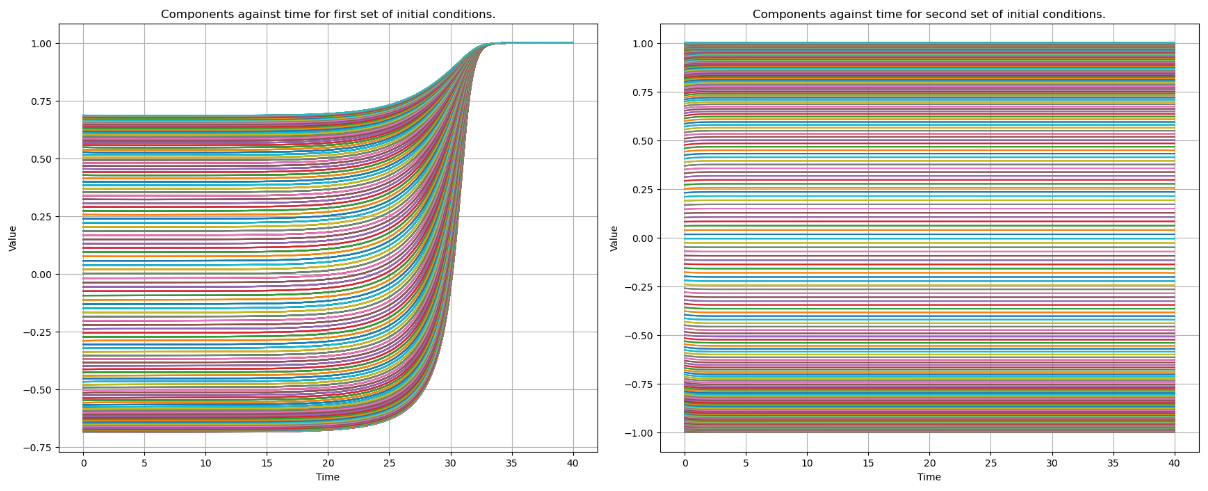
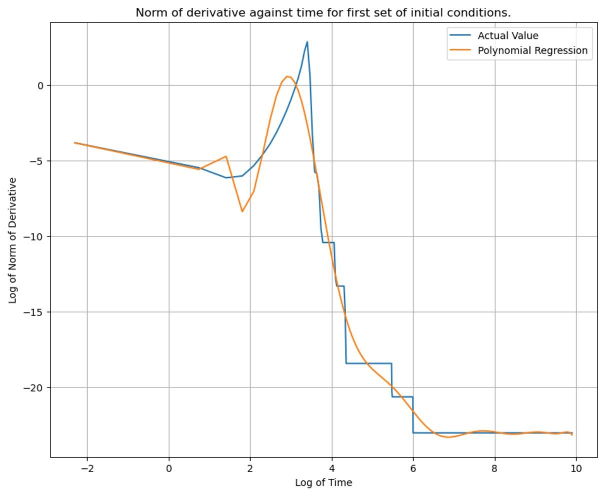


Figure 1: Components against time for both sets of initial conditions

The first two plots investigate the evolution of each component against time for both sets of initial conditions. The graphs show that the system has an oscillatory behaviour for both sets of initial conditions. This could suggest there is stability if the oscillations converge to a periodic solution or are bounded around an equilibrium point. To further investigate this, I decided to create plots showing how the norm of the derivatives vary with time.



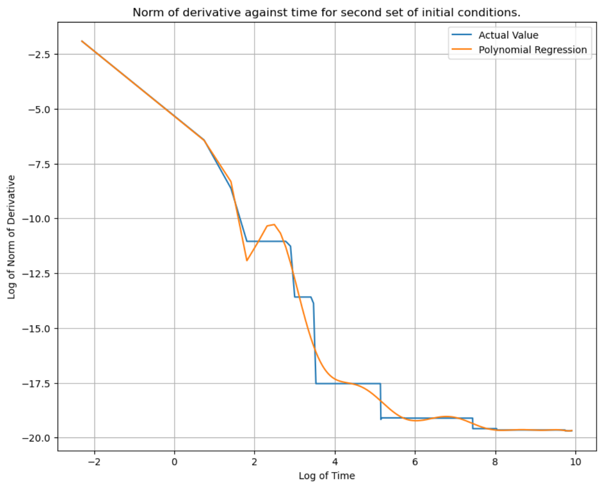


Figure 2: Norms of derivatives against time for both sets of initial conditions

Both plots show a decrease in the log of the norm of the derivatives over time. For both sets of initial conditions this value tends to negative infinity as time tends to infinity, therefore the norm of the derivatives tend to zero. This implies convergence towards some equilibrium state for both cases. The fact that the derivatives of the system approach zero over time indicates stability at these equilibrium points. Due to this stability, small perturbations from these equilibrium states would likely lead the system back towards these equilibria rather than cause the system to diverge. Hence, we can conclude that the equilibria for both sets of initial conditions are stable.

This also agrees with our simulation results, since we find when testing the function that there is convergence to an equilibrium as time tends to infinity.

3.

The function part2q3 solves a system of three stochastic differential equations. It simulates the time evolution of a system of three variables, y_0, y_1, y_2, taking into account noise (randomness). The RHS function is used to compute the derivatives at each time step. The randomness is then implemented by adding the μdW_i term using the Euler-Maruyama method. dW_0, dW_1, and dW_2 are randomly generated to represent Brownian motion. The parameter μ scales the effect of these terms, allowing the level of noise to be altered. These two components are combined together and the final computed value for that time step is stored. Finally, the values of y_0, y_1, y_2, at each time step are returned, showing the time evolution of the system.

To illustrate the effect of the parameter μ, I added code to the p2q3Analyze function to generate two plots.

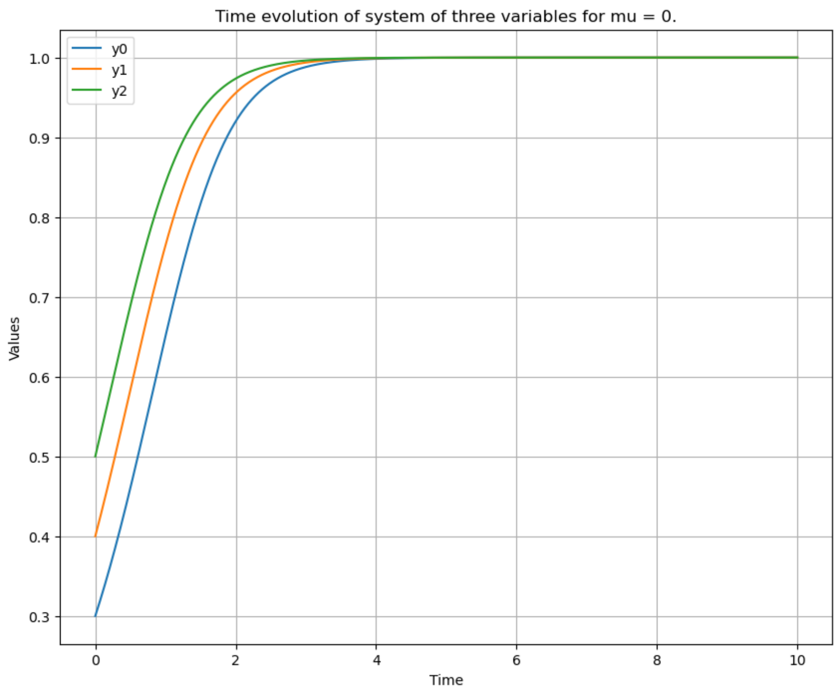


Figure 3: $\mu = 0$

The first plot shows the time evolution of the system when $\mu = 0$. This is the case where no noise is added using the Euler-Maruyama method and so there are no random variations in the values of the three variables. Here, the variables are determined solely by the equations in the RHS function. The graph shows that the system is asymptotically stable in this case as the solutions converge to a stable equilibrium as time tends to infinity. It also shows the system is mean-square stable as the system's solution remains bounded as time tends to infinity.

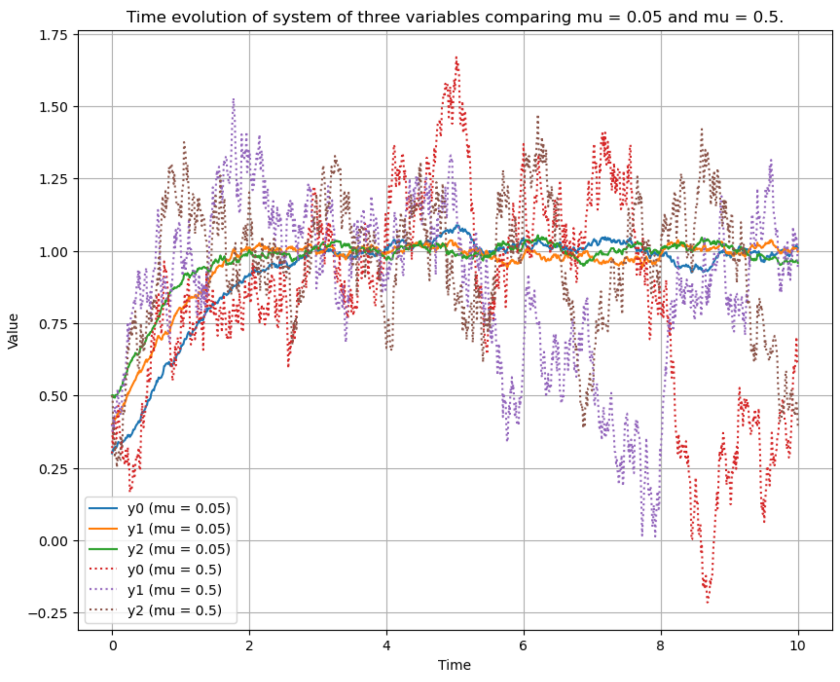


Figure 4: Comparing $\mu = 0.05$ and $\mu = 0.5$

The second plot compares the time evolution of the system for two different values of μ. It is clear that a larger value of μ leads to greater variations in the value of each variable at any given time due to the increased level of randomness. For $\mu = 0.5$, it appears that the random shifts have overshadowed the computed solutions from the RHS function, making the system appear random and indicating instability. For $\mu = 0.05$, we see the contrary. The level of randomness is very low and therefore is overshadowed by the computed solutions from the RHS function. This implies that even though there is no asymptotic stability due to the noise, there is mean-square stability as the system's solution remains bounded as time tends to infinity.