Scientific Computation Project 2

02027072

November 24, 2023

Part 1

1.

The problem the functions attempt to solve is to find a path within the input *networkx* graph from a specified source node to a specified target node that minimises the weight of the maximum edge weight between two nodes along the path. Let $P(s, x)$ be the set of paths from node s to node x, where a path p is the set of edges e along the path, and an edge has weight $w(e)$. Then the problem can be written as:

$$w^* = \min_{p \in P(s,x)} \left(\max_{e \in p} w(e) \right) \tag{1}$$

where w^* is the value of this minimum maximum edge weight.

Both functions return the value of w^*, and *searchGPT* also returns the optimal path as a list of nodes from source to target that the path traverses through. The strategy both functions use is an adjusted version of Dijkstra's algorithm with a priority queue, but instead of tracking the sum of the edge weights from each node to the source, it instead tracks the maximum edge weight on the path from each node to the source, which is then minimised. The priority queue is implemented with *heapq* using *heapq.push* to push elements to the list and *heapq.pop* to pop elements from the list, such that they are ordered by increasing maximum edge weight along the path to the node.

2.

a) For a graph with positive weights where the source and target node are connected, both *searchGPT* and *searchPKR2* return the same correct value that solves the problem (1).

In the case where the source and target are disconnected or unreachable in the input graph, *searchGPT* returns the value *float('inf')*, which can be interpreted as the minimum weight being infinite as there is no path. On the other hand, *searchPKR2* returns a finite value along with an empty *path* list. The value returned is the maximum solution to problem (1) for all nodes in the subgraph that are connected to the source, hence the method is inapplicable to disconnected source and target nodes.

Another feature of *searchPKR2* is that it adds a node to the input graph to act as a dummy node for dead weights in the priority queue. This is to prevent iterating through the *for* loop once a dead weight reaches the front of the priority queue, as the dummy node has no neighbours. However, a drawback of this method is that the original input graph is also modified by the search function.

b) Both functions implement an altered form of Dijkstra's algorithm. From lectures, given a graph with N nodes and L edges, using Python *heapq*, we know that Dijkstra's has a computational cost of $\mathcal{O}((N + L)\log_2(N))$ for the search part of the algorithm. On each iteration, whilst exploring the neighbours of the 'current' node, when checking the edge weights of a new neighbour or updating a neighbour that has not yet been explored, the action of adding the new edge weight to the summed edge weights of the path to the current node is replaced with taking the maximum of the edge weight and the max edge weight of the current node. The cost of taking the max of two floats is the same as adding two floats, hence the cost of the search for both algorithms is still $\mathcal{O}((N + L)\log_2(N))$.

To return the path from source to target, *searchGPT* creates a *parents* dictionary, which stores and updates the parent node of each node that is reached. Then, if the target node is found, a *path* list is created by starting with the target node and iterating the insertion of the parent node to the start of the list until the source node is inserted. Applying the Python *insert* function to a list of length n has a cost of $\mathcal{O}(n)$. Suppose the path is of length k, then the assignment of the *path* list has a computational cost

of $\mathcal{O}(k^2)$. In the worst-case scenario, this path includes all nodes of the graph, so the time complexity would be $\mathcal{O}(N^2)$.

For my code in *searchPKR2*, I introduce the *parents* dictionary in a similar way to *searchGPT*. If the target node is found, a *path* list is also then created from the *parent* dictionary. However, unlike *searchGPT*, *searchPKR2* begins with the target node and appends each parent to the end of the list until the source node is reached, then reverses the list. Suppose the path is of length k, then the looped *append* function has a cost of $\mathcal{O}(k)$, followed by a *reverse* function with a cost of $\mathcal{O}(k)$, so overall the computational cost of the path is of $\mathcal{O}(k)$. In the worst-case scenario, this path includes all nodes of the graph, so the time complexity of the *path* list creation would be $\mathcal{O}(N)$. This is significantly better than *searchGPT*.

Part 2

1.

In my new implementation of *part2q1*, I have adapted the form of the RHS of the model into *NumPy* array form. I use a *scipy* sparse matrix for the linear part of the RHS, then subtract the cubic term using element-wise broadcasting. This improves efficiency by removing the iteration over the list and utilising the fast wall time of *scipy* sparse matrix multiplication and NumPy vector methods.

To efficiently compute the solution to the IVP, I use the *scipy.integrate.solve_ivp* function with the *'BDF'* method. This method is highly robust and sophisticated as seen in lectures, and has a much faster wall-time. The default error tolerance is *atol* is 10^{-6} as required, and we maximise its efficiency for large n by setting the *vectorized* parameter to *True*.

2.

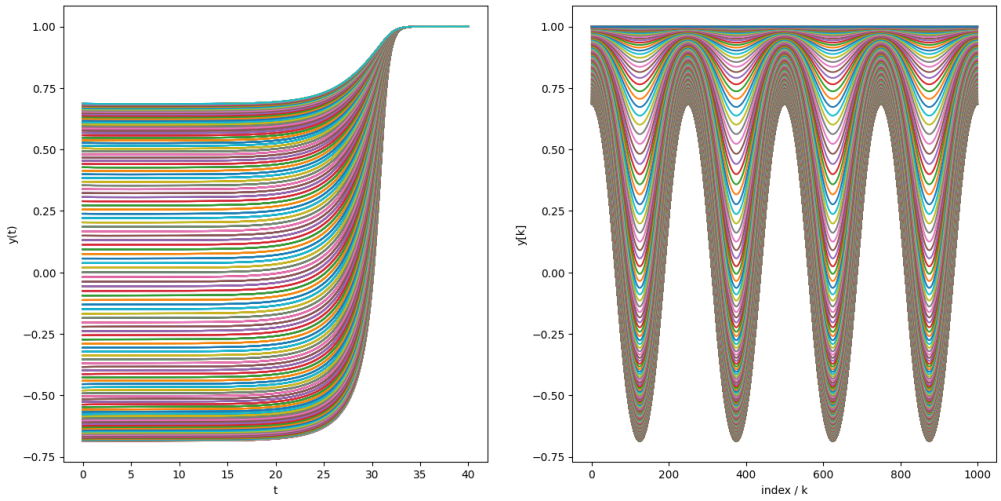

Figure 1: Solution to initial condition $\mathbf{y_{0A}}$

From Figure 1, we observe that from initial condition $\mathbf{y_{0A}}$, the values converge towards 1 at around $t = 25$ and diverge away from its nearby equilibrium.

From Figure 2, we observe that with the initial condition $\mathbf{y_{0B}}$, the values remain stable and close to the initial condition throughout.

We assume that each initial condition $\mathbf{y_{0A}}$ and $\mathbf{y_{0B}}$ is "close" to an equilibrium point, so by applying *scipy.optimize.root* onto the ODEs with initial guesses being the initial conditions, we find the equilibrium states $\overline{\mathbf{y_A}}$ and $\overline{\mathbf{y_B}}$.

By introducing the power series expansion for y_i, where $\bar{y}_i$ is an equilibrium state and $0 < \epsilon \ll 1$:

$$y_i = \bar{y}_i + \epsilon\tilde{y}_i + \mathcal{O}(\epsilon^2), i = 0, 1, \cdots, n-1$$

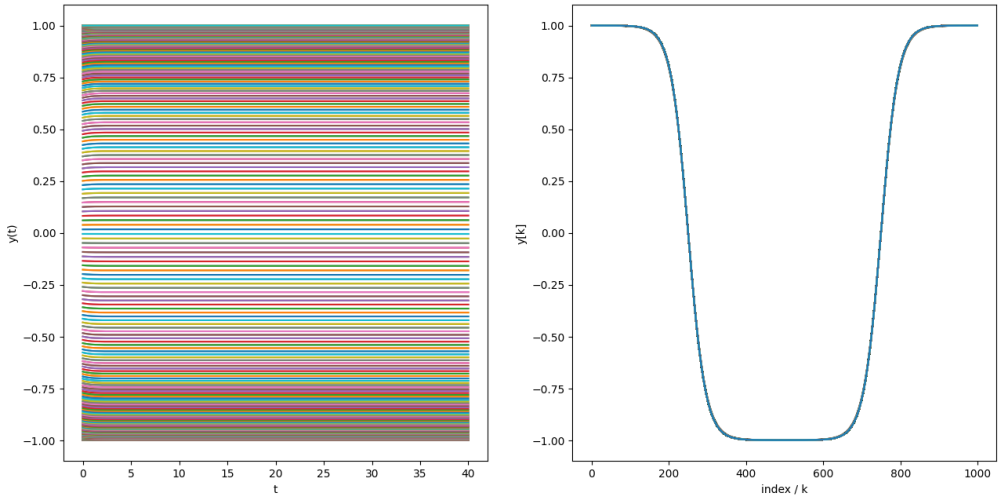

Figure 2: Solution to initial condition $\mathbf{y_{0B}}$

we get the following linearised equations for the perturbations:

$$\frac{d\tilde{y}_0}{dt} = (\alpha - 3\bar{y}_0^2)\tilde{y}_0 + \beta(\tilde{y}_{n-1} + \tilde{y}_1)$$
$$\frac{d\tilde{y}_i}{dt} = (\alpha - 3\bar{y}_i^2)\tilde{y}_i + \beta(\tilde{y}_{i-1} + \tilde{y}_{i+1}), i = 1, \cdots, n-1$$
$$\frac{d\tilde{y}_{n-1}}{dt} = (\alpha - 3\bar{y}_{n-1}^2)\tilde{y}_{n-1} + \beta(\tilde{y}_0 + \tilde{y}_{n-2})$$

which can be written in matrix form as

$$\frac{d\tilde{\mathbf{y}}}{dt} = \mathbf{M}_{\bar{\mathbf{y}}}\tilde{\mathbf{y}}$$

where

$$\mathbf{M}_{\bar{\mathbf{y}}} = \begin{pmatrix} \alpha & \beta & & & \beta \\ \beta & \alpha & \beta & & \\ & \ddots & \ddots & \ddots & \\ & & \beta & \alpha & \beta \\ \beta & & & \beta & \alpha \end{pmatrix} - \begin{pmatrix} 3\bar{y}_0^2 & & & & \\ & 3\bar{y}_1^2 & & & \\ & & \ddots & & \\ & & & 3\bar{y}_{n-2}^2 & \\ & & & & 3\bar{y}_{n-1}^2 \end{pmatrix}$$

We can then compute the n eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of $\mathbf{M}_{\bar{\mathbf{y}}}$ to find a general solution of the form

$$\tilde{\mathbf{y}} = c_1\mathbf{v}_1 \exp \lambda_1 t + \cdots + c_n\mathbf{v}_n \exp \lambda_n t$$

We find $c_i, i = 1, \cdots, n$ that satisfy the initial conditions $\mathbf{y}_0$ by solving the linear equations:

$$\mathbf{V}\mathbf{c} = \mathbf{y}_0$$

where $\mathbf{V}$ is a $n \times n$ matrix with columns as the eigenvectors, and $\mathbf{c}$ is a column vector of c_i.
When applying the perturbation analysis to the equilibrium points $\overline{\mathbf{y_A}}$ and $\overline{\mathbf{y_B}}$ with initial values $\mathbf{y_{0A}}$ and $\mathbf{y_{0B}}$, we get the following results:

	$\overline{\mathbf{y_A}}$	$\overline{\mathbf{y_B}}$
$\lambda_{\max}$	0.3582534659916265	2.1646883070509115e-08
$\lambda_{\min}$	-4052.676267072026	-4054.7764873427172

Table 1: Notable eigenvalues of $\mathbf{M}_{\bar{\mathbf{y}}}$

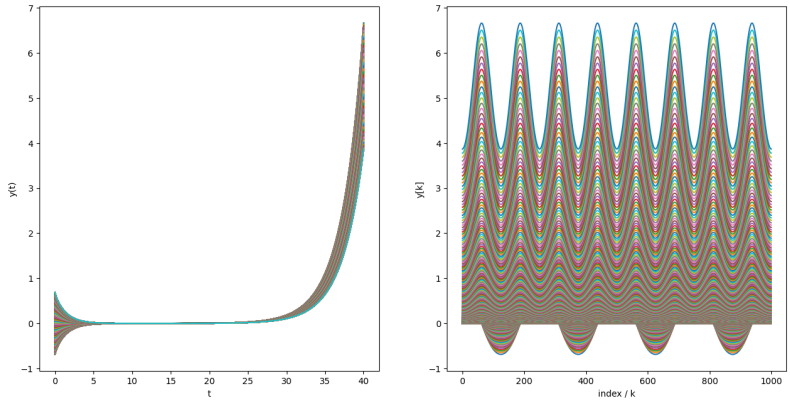

Figure 3: Perturbations of equilibrium $\overline{\mathbf{y_A}}$

From Figure 3, we observe that perturbations from the equilibrium point $\overline{\mathbf{y_A}}$ appear to initially converge quickly before diverging at around $t = 25$, which shows it is unstable. This is supported by Table 1, where $\lambda_{\max}$ is positive and $\lambda_{\min}$ is negative, indicating that it is a saddle point.

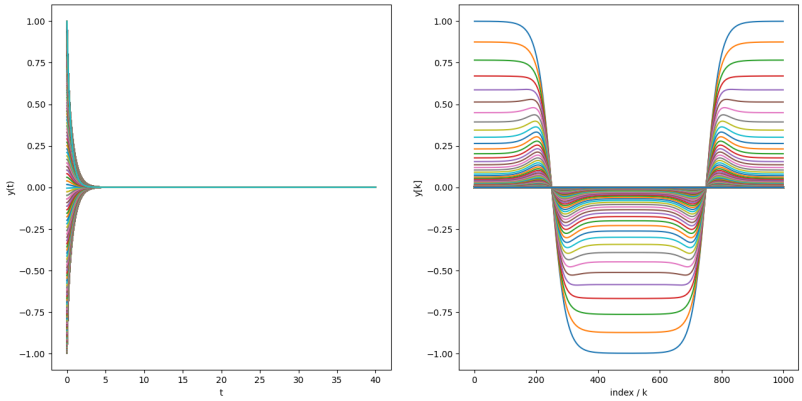

Figure 4: Perturbations of equilibrium $\overline{\mathbf{y_B}}$

From Figure 4, we observe that perturbations from the equilibrium point $\overline{\mathbf{y_B}}$ appear to converge very quickly, which indicates it is stable for at least $t \leq 40$. Table 1 suggests that $\lambda_{\max}$ is positive but of order 10^{-8} and $\lambda_{\min}$ is negative, so either the equilibrium point is saddle and will diverge for big enough t, or because the error tolerance of *part2q1new* is 10^{-6}, the true value of $\lambda_{\max}$ is 0 and the point is stable.

3.

The function *part2q3* attempts to solve the system of stochastic differential equations

$$dY(t) = f(Y(t))dt + g(Y(T))dW(t)$$

where $f(Y(t))$ is the set of equations in *part2q1* and $g(Y(t)) = \mu$. The "Brownian step" of our system is denoted by $dW(t)$. Our initial value for the stochastic differential equations is

$$\mathbf{y_0} = \begin{pmatrix} 0.3 \\ 0.4 \\ 0.5 \end{pmatrix}$$

The function solves this by applying the Euler-Maruyama method for 1000 timesteps until $t = 10$, so $\Delta t = 0.01$. We consider cases for $\mu = 0, \pm 0.2, \pm 0.4, \pm 0.6, \pm 0.8, \pm 1$. We take the average of 1000 simulations of the solution for each value of μ.

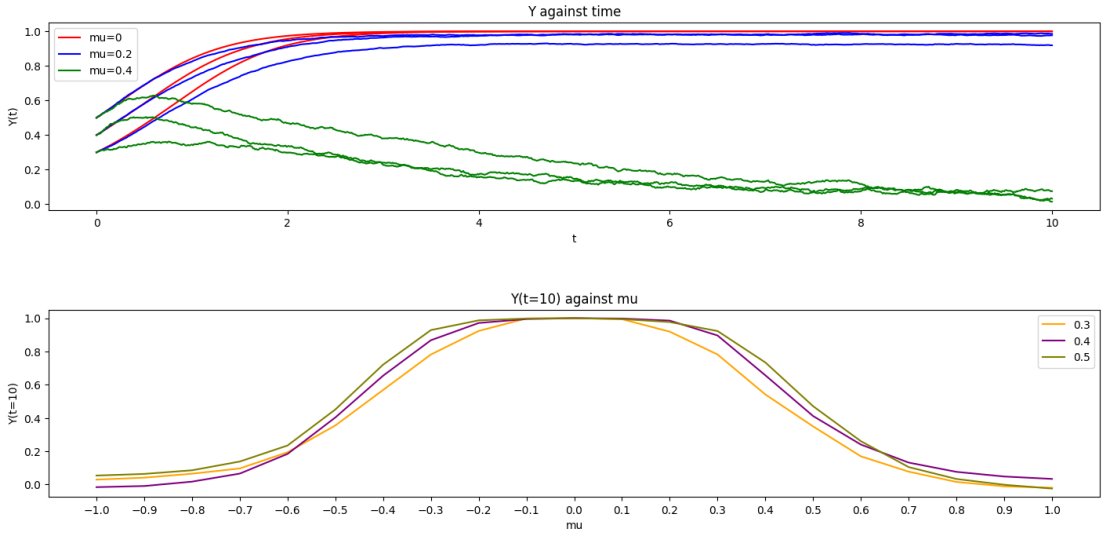

Figure 5: Plots of *part2q3analyze*

In Figure 5, in our upper plot, we see the average solution from our starting conditions for $\mu = 0, 0.2, 0.8$. In our lower plot, we see the average value of Y at $t = 10$ for a range of values of μ.

From the upper plot of Figure 5, we observe that when $\mu = 0$, without the Brownian motion component, our solution quickly converges to 1 for all initial starting points. When $\mu = 0.2$, the solution appears to converge to values below 1, and when $\mu = 0.8$, the solution appears to converge to 0. It is clear that for larger values of μ, there is more weight on the stochastic component of our system, hence there is more fluctuation in our solutions.

From the lower plot of Figure 5, we observe that μ is symmetric for negative and positive values, which is expected as the Normal distribution is symmetric. We also observe that the greater the absolute value of μ, the closer to 0 the solution is at $t = 10$. This is due to the fact that increasing μ means the stochastic component of the SDE dominates the deterministic part more, hence the solutions fluctuate more around the average of the 0-mean Normal distribution.