Scientific Computation Project 2

01847210

November 24, 2023

Part 1

1.

The problem searchGPT and searchPKR are solving is as follows: given two nodes in a weighted graph (a source node and a target node) to find the distance of the minimum distance path between the nodes, if a path exists. The definition of a path's distance is the maximum weight along the path and that definition is what I will be using for the remainder of Part 1.
SearchPKR additionally returns a list of nodes visited in a minimum distance path in order of nodes visited.
The strategy searchGPT uses to return the distance of the minimum distance path is a modification of Dijkstra's algorithm, where the definition of distance of a path has been redefined from the sum of weights along the path to the maximum weight along a path.
The strategy searchGPT uses:

- Initialise distances and dictionaries: The algorithm initialises a dictionary to store the distance from source to node for each node, it initialises the source node at 0 and each other node at inf. It also initialises a priority queue with the source node, the queue will contanin tuples (distance to node, node), in order of increasing distance. To help reconstruct the path, it initialises a parents dictionary.
- Loop through priority queue: heappop(priority_queue) sets current node to the node with the smallest known distance, and removes it from the queue. If the current node is the target it reconstructs the path and returns the distance, else it explores its neighbors. For each neighbor if the path via the current node is shorter than its previous distance it updates the distance and adds the neighbor to the queue.
- Path reconstruction: If the current node is the target node it reconstructs the path using the parents dictionary, returns it, and returns the minimum distance to the target node, else it returns inf.

2.

In searchPKR2 I modified searchPKR such that it returns the path with the lowest distance, an identical path to searchGPT. I included a dictionary to store the "parent" nodes, identical to the dictionary used in searchGPT, and if the target is reached in the main loop the path is constructed. The dictionary is populated in the main loop, after checking a node it updates the parent of the node to be the predecessor in the shortest path from the source node. If the target node has been found the path is iteratively constructed using deque, as this allows for O(1) left insertion, avoiding inserting into a list with order O(n). I chose this method as it is more concise than appending to a list and reversing it. Inserting to a list in searchGPT bottlenecks the time complexity in cases where the path length is very long, and contains more operations than using deque, hence searchPKR2 is more efficient at reconstructing the path.
If a path does not exist, searchPKR will return the value of dmin assigned last after all nodes in the queue have been checked. This is incorrect and problematic as it is indistingushable from the distance of the path from source to the last checked node (which is accessible from source by construction). To combat this, in searchPKR2 I have it return infinity if no path exists, like searchGPT.

- A tangible difference between searchGPT and searchPKR2 is the graph modifictaion. If the input graph does not contain a node with label equal to the length of the graph, n, then a node will be added to G. This modification is unneccessary since this node has no neighbors and will never be checked in the main loop.
- Another tangible difference in searchPKR2 is the construction of a list l. This list is created when Mdict needs to be updated by popping the current nodes list in Mdict, and is immediately overwritten to store the current node and n. Since .copy() was not used this looks as though it will create issues because overwriting l writes [dmin,n] into Mlist, however the nodes are numbered from 0 to n-1 so again this does not affect the output of the function nor does it modify the graph, this list is not used again so it is unneccessary.

- SearchGPT reconstructs the path by iteratively inserting nodes to index 0 of a list. This has time complexity $O(P^2)$ where P is the length of the path. SearchPKR2 uses deque to left append with O(1), making it more efficient at reconstructing paths. In cases where the path is very long, this bottlenecks the worst case time complexity of searchGPT to $O(N^2)$.
- searchPKR2 uses three separate dictionaries, whereas searchGPT uses only two, making searchGPT more memory efficient.

Both functions return a shortest path and the 'distance' of this shortest path for identical inputs and return infinity when no path exists, making them equally correct. Correctness of searchPKR2 follows from correctness of Dijkstra's algorithm under the distance metric of maximum weight (a well defined metric) hence both algorithms are correct.
Step by step computational cost for searchGPT:
1) Initialising the dictionary for each node: O(N). Initialising the priority queue and parents dictionary:O(1)
2) In the main loop we will check each nodes neighboring edges, each edge is inserted into the queue by the order $O(log_2(N))$ heappush, which gives : $O(Elog_2(N))$. In each loop we also heappop a node from the queue, so N heappops of order $O(log_2(N))$ gives: $O(Nlog_2(N))$.
3) Reconstructing a path of length p will have p insertions into index 0 of a list: $O(p^2)$. In the rare case the path is very long this will make the overall complexity $O(N^2)$, however for the average case I will take p to be small.
4) Adding the complexities we see the overall computational cost is $O((E + N)log_2(N))$ where E is the number of edges and N is the number of nodes. However it is worth noting that for very long paths the order is dominated by $O(N^2)$, and for very densely connected graphs E is $O(N^2)$ and the order is $O(N^2log_2(N))$.
For searchPKR2:
1)Initialisation: Initialising empty dictionaries and lists is O(1), heappush to an empty list is O(1) as there is no rearranging necessary. Appending the source node to Mdict is O(1). Overall this step is O(1), which is different to searchGPT's initialisation step of O(N).
2)Main loop: Similarly to searchGPT the main loop iterates over the nodes in the priority queue, and every node may be added to the priority list so there are O(N) iterations, each node is heappopped from the queue with order $O(log_2(N))$, so the loop includes $O(Nlog_2(N))$ operations.
In the loop we also visit every edge at most twice, and for each edge we heappush to Mlist, making this step $O(Elog_2(N))$. Overall the loop therefore has $O((E + N)log_2(N)$ operations.
3)Reconstructing the path: Unlike searchGPT I have used deque to left append to the path with O(1) operations, this is done for every node in the path hence is O(p). In the worst case it is O(N) so this does not affect the overall complexity.
4)Adding the complexities as before yields $O((E + N)log_2(N))$, the same as searchGPT. However it is worth noting when the path is very long the path reconstruction becomes O(N), a stark comparison to searchGPT's construction of $O(N^2)$, and the overall complexity will remain at $O((E+N)log_2(N))$. Also, similar to searchGPT, for extremely well connected graphs the overall complexity becomes $O(N^2log_2(N))$ as E is of order $O(N^2)$ which dominates.

Part 2

1.

The function part2q1 is inefficient in multiple ways: It uses loops to initialise a system of equations, and to compute the solution. Techniques like vectorisation and slicing are especially helpful as it avoids looping over the dimension of the problem, which may be large.
The first change I made is switching np.zeros_like() to np.empty_like() and np.zeros() to np.empty(). This is more efficient since we are overwriting this array immediately, so it is unneccessary to write zeros into it, which will waste time.
The second change is for the RHS evaluation function, I removed the loop and made use of ndarray slicing, which is more effective than looping.
The final change is removing the final loop which computes the numerical solution, as the scipy function solve_ivp() is optimised for this specific problem. I chose the BFD solving method for the best efficiency-accuracy trade-off.

2.

We are given two sets of initial conditions, A and B, which we know are close to equilibrium points. To analyse the solution from the two sets of initial conditions we can solve the system of ODEs for each dataset separately and plot the solution for each dimension individually. I chose tf=40 as at this time point the two sets of solutions are completely different.

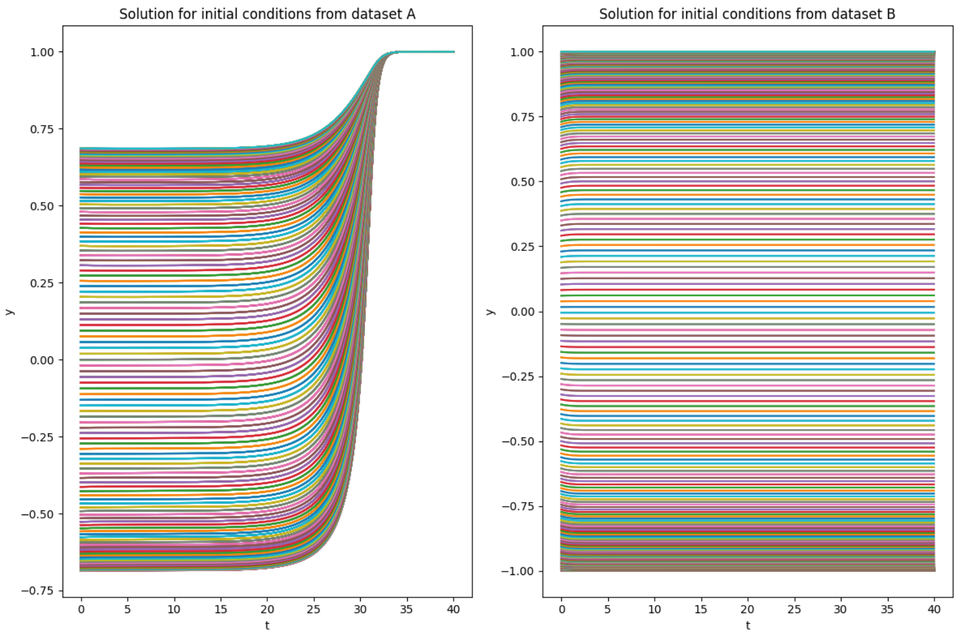

Figure 1: Solution paths for every y_i

Despite the initial conditions looking similar they are actually very different, they exhibit totally different behaviors after t=25. Paths from A become rapidly attracted to 1 and paths from B stay constant.
Looking back at our system of ODE's:

$$\frac{dy_i}{dt} = \alpha y_i - y_i^3 + \beta(y_{i-1} + y_{i+1})$$

(Here the edge cases are included by wrapping around mod n) We notice that each solution is dependent on neighboring dimension solutions.
To explain the differences in paths we should study the equilibrium points nearby them

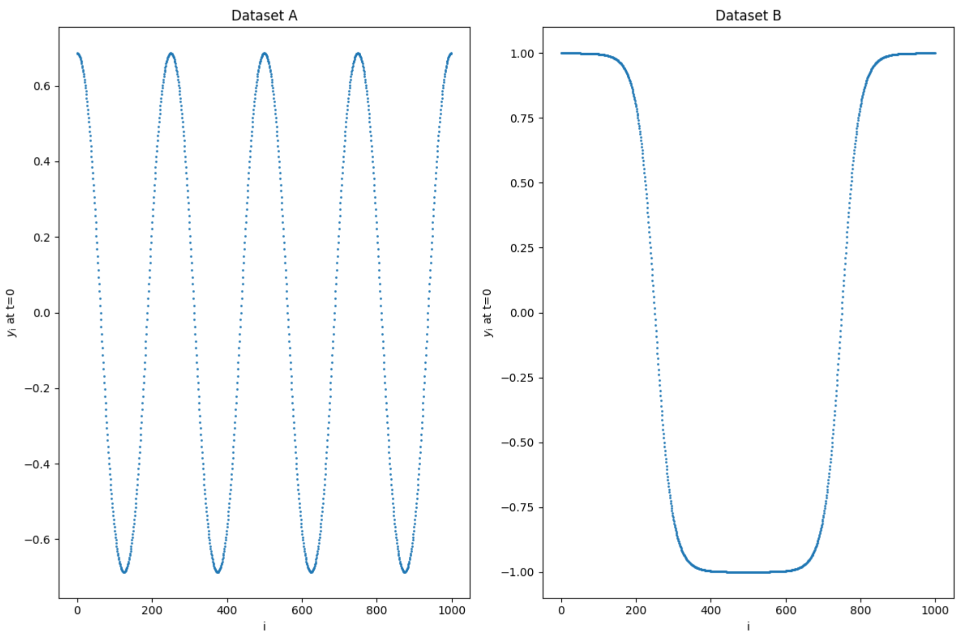

Figure 2: Initial conditions

Since we know these initial conditions are close to equilibrium solutions we can find the closest equilibrium solutions using scipy.optimize.root(). We can plot the equilibrium solutions to see how close A and B are:

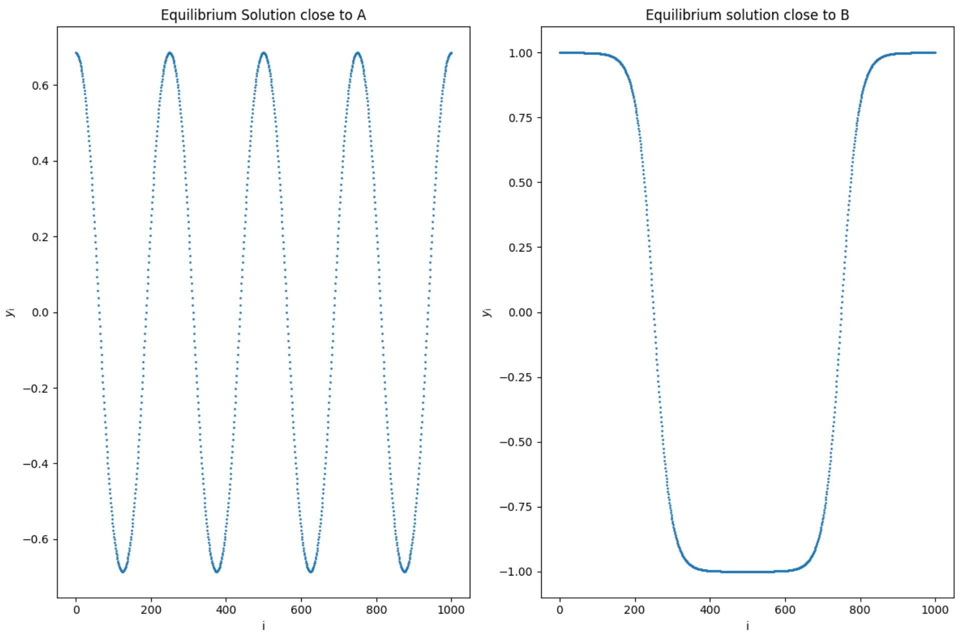

Figure 3: Equilibrium solutions visualised

Clearly the initial conditions are very close to the equilibrium solutions.
Initial condition B reaches equilibrium very quickly, as we can see the gradient for each dimension becomes zero almost immediately, however initial condition A exhibits equilibrium like behaviour for a long time before each dimension shoots to 1. A reasonable guess is that equilibrium point A is unstable, equilibrium point B is stable and the 1 vector is stable. We can check this by analysing the eigenvalues of the jacobian matrix for the system at these points.
My function part2q2 calculates the jacobian at each of these points and prints the positive eigenvalues for each one, note after finding the eigenvalues we see that all of them are real valued. I did not use a sparse

matrix as sparse.linalg.eig cannot return every eigenvalue, and we need all eigenvalues for analysing the stability of a point. From this we see that the 1 vector has all negative eigenvalues, equilibium point A has multiple positive eigenvalues, and equilibrium B has all negative and one near zero eigenvalue. We see that the y0B path converges to the equilibrium near it so it is safe to assume this eigenvalue should be zero, or that y0B is completely contained in the attractive subspace.
Initial condition A therefore will only be attracted to equilibrium point A if it is contained in the subspace spanned by eigenvectors corresponding to a negative eigenvalue only, to check this I have projected initial condition A onto the subspace of eigenvectors with positive eigenvalues, and part2q2 will return False if this projection is non zero. Since it returns false we know that initial condition A is not attracted to equilibrium A in the long term. This explains why we see each path shoot to 1 as the vector tends away from equilibrium A and towards the 1 vector. The stability of equilibrium point B explains why initial conditions B tend towards equilibrium B.

3.

The code for part2q3 is for finding a numerical solution to a three dimensional SDE. It uses an Euler-Maruyama method which is an adapted Euler method for Stochastic differetial equations.
Inside part2q3 we have the function RHS which returns the right hand side of the same system as in part2q1, except with $\beta = 0.04/\pi^2$. It then computes dW, which represents the random flucuations in the system by a normally distributed random variable scaled by the square root of the time step Dt. This corresponds to the Brownian steps the solution will take for each time step.
Finally the update equations are defined, updating the deterministic part of the solution using RHS and also adding the stochastic part of the solution which is scaled by the parameter mu. The loop is efficient as dW is computed as a matrix. Intuitively a larger mu will result in a noisier solution, which may impact how closely the solution to the SDE reflects the solution to the ODE (the no noise solution).
The noiseless system
The system of equations is the same as in part2, with the exception that β is far smaller. Nonetheless we still have stable attractive equilibrium at the 1 vector and the -1 vector. So we would expect a slightly noisy solution to the SDEs to follow the solution of the ODEs roughly.
Effect of mu
To see the effect of mu on the solution we should plot each dimension of a no noise solution, mu = 0, and test larger mu to see the effect. Each solution is represented by a colour, where each dimension is a different shade for clarity:

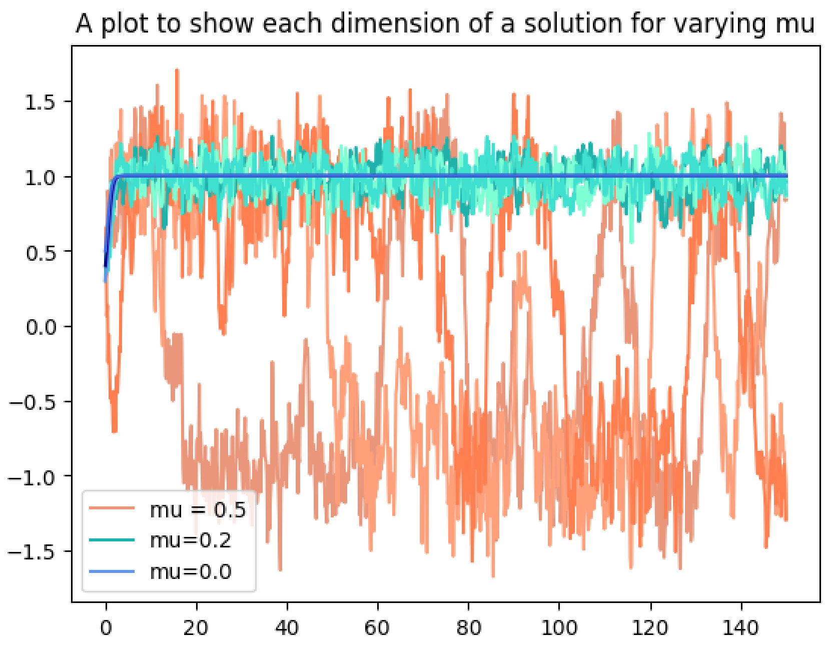

Figure 4: three different solutions, three lines per solution

For initial conditions y0 = (0.3,0.4,0.5) we see the no noise solution tends towards the 1 vector

equilibrium point and stays there asymptotically. When we make mu 'well behaved' and small, mu = 0.2, the probability the path will jump away from the 1 vector and land in the basin of attraction of another equilibrium point is very small, hence this also tends towards the 1 vector, with the noise level remaining constant forever.
Small mu is 'well behaved' because if you denoise this solution you can recover the noiseless solution of the system of ODEs.
For larger mu the probability the solution jumps away from the 1 vector is much larger, so we see each dimension jumping away from 1 and being more attracted to -1, and jumping back to 1 very frequently. The result of this is if we denoise this solution we recover nothing of value, as there is no way to recover the noiseless solution which tends to 1. Denoising this will have a very high probability of giving a meaningless path that oscillates between 1 and -1 for each dimension in question.
Another approach would be taking the 'average' path over many simulations. It is clear to see that the noiseless and well behaved small mu will return similar average paths and the average path for large mu will become 0 in each dimension due to its frequent and random jumping between -1 and 1.