Scientific Computation Project 2

01854740

March 5, 2023

Part 1

1.

Option 0 first makes Hlist into a heap (sorted by the first element of the tuple) and where Hlist consists of 2-element lists, a dictionary Hdict is created which has for each list in Hlist the second element as a key and the list as the value, and Hlist and Hdict are returned. Hence, option 1 will take in Hlist as a heap. As long as Hlist is not empty, heappop is used to remove and store the root node, i.e. smallest element by weight (first element of tuple) as wpop and npop (which correspond to a weight and node number given Hlist contains 2-element lists of weights and nodes). Option 1 also returns the 2-element list. The heappop function also preserves the heap property, i.e. is restructured in place. Finally, the key value pair of npop is deleted from Hdict (the if statement is provided because the dictionary should not have a key -10000 by the way it is built while the npop can certainly take this value, so we avoid a key error). Option 2 uses heappush to push x to Hlist whilst preserving heap structure and if the node npop is already in Hdict, overwrites the second tuple element of the previous list also with npop. For Hdict however, it deletes the key corresponding to the pair already in the dictionary and adds the new key value pair instead.

In such a way, option 1 pops the root of the heap while option 2 pushes an element x to the heap, maintaining heap structure at all times, whereas Hdict gets a key value pair deleted by option 1 or added by option 2. The difference is that in the case of a node clash, Hlist adds the new pair without removing the old pair, simply overwrites its node number with -10000 so we remember it has been overwritten, while Hdict replaces it entirely.

On Hdict, we perform dictionary lookup and dictionary assignment in constant time, as well as popping from Hdict given we only use pop if an element is in the dictionary. Updating l to overwrite n to -10000 takes constant time as well. The bottleneck is heappush, which is done whether x[1] is in the dictionary or not. Heappush has worst case complexity O(log n) where n is the number of elements in the heap, i.e. O(m), so option 2 has complexity O(m) overall. In this worst case, the element inserted to the heap, has to be swapped at every level from bottom to top , there being log n levels in the heap (i.e. inserted element is the smallest element in the heap).

2. (a)

Given nodes s and x, the code first tries to find the distance between the 2 nodes as well as return the path from s to x. If x is inaccessible from s, the code returns Fdict which is a dictionary with each key being a node and the value being the distance from s to that node. The distance in this case is not additive but rather is multiplicative, i.e. if the distance from s to y is A and from y to x is B, the distance from s to x is A*B. This why we need a constraint on The graph edge-weights that they are greater than or equal to 1. Without this constraint, say for the example above that A=2, B=0.5 (which violates the constraint), then the distance from s to y to x is 1, which is less than the distance from s to y of 2, which means the path which goes along the same route and then beyond is shorter, which is impossibler intuitively.

The strategy being used is an adaptation of Dijkstra's algorithm. We initialise the provisional distance for the source as 1 (the multiplicative identity) and add it to Mdict, the dictionary of explored nodes. We also set the Plist to have empty lists for each node and the list for the source node as just the source node. We initalise an empty dictionary Fdict for the finalised distances. While Mdict still contains nodes to explore, we pop the node with the smallest distance from Mdict and put it in Fdict (though if the smallest distance is to x, we return the distance and path from s to x). For each of this node's unexplored neighbours, we update the provisional distance of its unexplored neighbour nodes (ignoring if already in the finalised dictionary). We add Mdict at the unexplored node with value of the minimum distance times its weight and update the path of the node in Plist as the current shorest path with also the node itself. If the neighbour node was already in Mdict, we update Plist and Mlist at that edge if the new distance is smaller than the previous one. Finally, if x is not found we return the whole Fdict.

2. (b)

Currently the calculation of dmin is a bottleneck, as Mdict can contain up to N nodes, and the dmin calculation is O(N), so overall, this part of the code is $O(N^2)$. Otherwise, each node is at some point added and removed from Mdict, i.e. O(N) and each edge is examined twice, O(L). I will consider the creation of Plist separately. Using a binary heap provides extraction of the node with minimal distance in O(log N) time, so the dmin calculation is O(log N) so this part of the code is O(N log N). The heap is structured such that we can remove the minimal element easily and just then need to restructure the heap to maintain the heap property in O(log N) time. Now that we are using the heap Mlist for removal, using option 1 from the previous question, we need to maintain the heap property across the function. We initialise the heap using option 0, adjust the weights of the neighbours of the removed node and hence have to restructure the heap each time, so this takes O(LlogN) time. We also insert the unexplored neighbours of the removed node in O(NlogN) time, as we must maintain heap structure. So although maintaining the heap can make individual steps more costly, the overall complexity is reduced and the bottleneck was the $O(N^2)$ dmin calculation, and we assume L is not too large compared to N.

A second inefficiency is the way Plist is created, which is a list of lists. Instead I will use a dictionary to store instead of the whole path from s to x for each node, just store the previous node in the path, where previously we stored the whole path. In the case where x in not found, the dictionary will have all the nodes as keys but has been constructed in O(1) time for each adjustment or insertion, so O(N+L) time (as we insert each unexplored node and also might adjust weights of each of the neighbours of each node), rather than O((N+L)dmax) time, where dmax is the maximum distance from s to any node, as each insertion or adjustment to the list of lists means appending at most dmax elements in O(dmax) time. In the case where x is found, neither the whole list nor dictionary is constructed. The new implementation will have to reconstruct the path from the dictionary and reverse the list in O(dmax) time for one specific node (in addition to the cost of creating the dictionary thus far),as the number of iterations of the while loop is bounded by the distance of x from s. Meanwhile the old implementation has been constructing paths for all nodes up to the point of finding x.

Part 2

2.

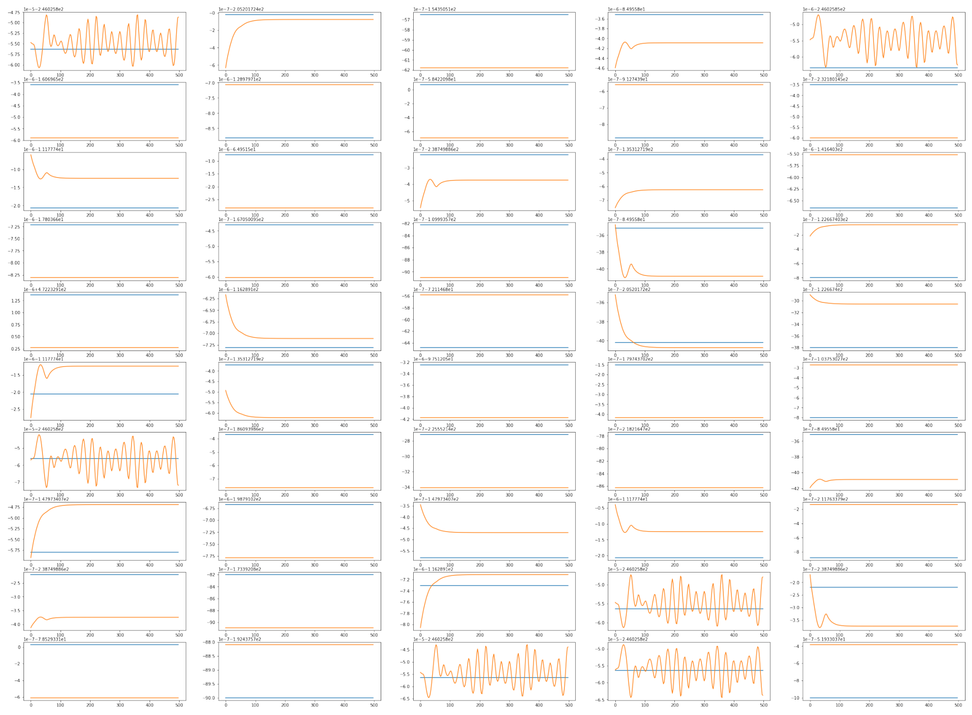
My approach to finding the equilibrium satisfying the conditions consisted of generating a random initial state of length 50, and then using the optimize root function from scipy which quickly finds the 0s of a function applied to the function defining the right hand side of the differential equation, hence corresponding to a solution of $dx_i/dt = 0$ for each i. The root function takes a starting point and a function and uses various solver methods to get from x0 to the x which makes the function value as close to 0 as possible. I checked the first condition by checking that the magnitude the absolute value of each element in the provided solution (using np.all) is greater than 0 and less or equal to 1000. Then in the second if statement, I check using np.unique which reduces an array to its unique elements and finding the length of this new array that the number of unique elements is at least 25. The while loop allows to generate new solutions from random initial conditions until one is found that satisfies the conditions. I checked for the accuracy of the solution by noting that each dx_i/dt should be within computing error of 0, hence the L1-norm must as be within 10e-15 of 0, as it will be the sum of the magnitudes of each RHS of i (the distances from 0). I check this using an assertion statement.

3.

In order to investigate the perturbations to the equilibrium, I write x_i as a function of the perturbation to the equilibrium, x_i^* and the equilibrium, x_i^0, as $x_i = x_i^0 + \epsilon x_i^*$. Then by using the Taylor expansion of f around the equilibrium up to 1st order, summing across j and dividing by n, and dividing both sides by ϵ, we get $dxi^*/dt = -\frac{1}{n}\sum_{j=1}^{n}(x_i^* - x_j^*)\exp^{-(x_i^0 - x_j^0)^2}$. We can rearrange this into a matrix M such that $Mx^* = \frac{d(x^*)}{dt}$ is equivalent. Each non-diagonal element ends up being $\exp^{-(x_i^0 - x_j^0)^2}$ and the diagonal elements minus the sum of this expression across j.

I then compute the eigenvalues and eigenvectors of M as the perturbation can be written as a linear combination of the eigenvectors scaled by the exponential of the eigenvalues times the time, which means that if the decomposition of an x_i^* is has components with positive eigenvectors, the perturbation should grow over time (not return to this equilibrium and perhaps settle at another equilibrium, perturbation analysis ignores the other equilibria), if all are negative then the perburbation should decay to 0 and x_i should return to its original equilibrium. If some eigenvalues of the decomposition have a complex part then the oscillatory dynamics are to be expected, though the real parts dictate if the solution grows or decays. I check the eigenvalues for my particular equilibrium and I find that there are both positive and negative real parts and that some eigenvalues have an imaginary part. Hence, we expect some of the x_i to be stable, some to go to another equilibrium and some to oscillate.

I plot for each i the path of x_i after perturbation with the equilibrium for reference, and we can see that some of the x_i do indeed go to another equilibrium, some oscillate and some decay to 0.



4.(b)

Plotting the average solutions across many simulations for each x_i for μ values of 0, 0.2, 0.5 and 5 in a grid of 2x2 subplots, we can see that in each case, the 50m solutions will agglomerate into groups. $\mu = 0$ is equivalent to the deterministic solution as all randomness is removed. Of course, increasing the μ value progressively introduces more noise into the system so the solutions oscillate more for larger μ. We also see more importantly that the agglomeration into groups takes longer to happen for 0.2 than 0 and even longer for 0.5. For example, looking at the x_i near 0, for $\mu = 0$, by around time 10, the solutions have already formed a single line, i.e. converged to a single value. For $\mu = 0.2$, the solutions have not yet converged by time 50. This is because the non-Gaussian term takes the direction of opinion of i as the average of the difference of itself and its neighbours, hence this term will tend to pull the solutions towards the average of its neighbours. Increasing a completely random term changes this dynamic and make it happen slower, and not at all when μ is large enough as the random term dominates making the dynamics random.

From plotting the variance of the solutions for each x_i for μ values of 0.2 and 5, we see also that the variance which is theoretically (for the Gaussian term) the square root of the number of timesteps scaled by μ of the different x_i match the theoretical value more closely the more μ is increased. This is because as the noise is increased the noise term eventually dominates and all the variance is explainable as Gaussian, which is not the case for smaller μ.

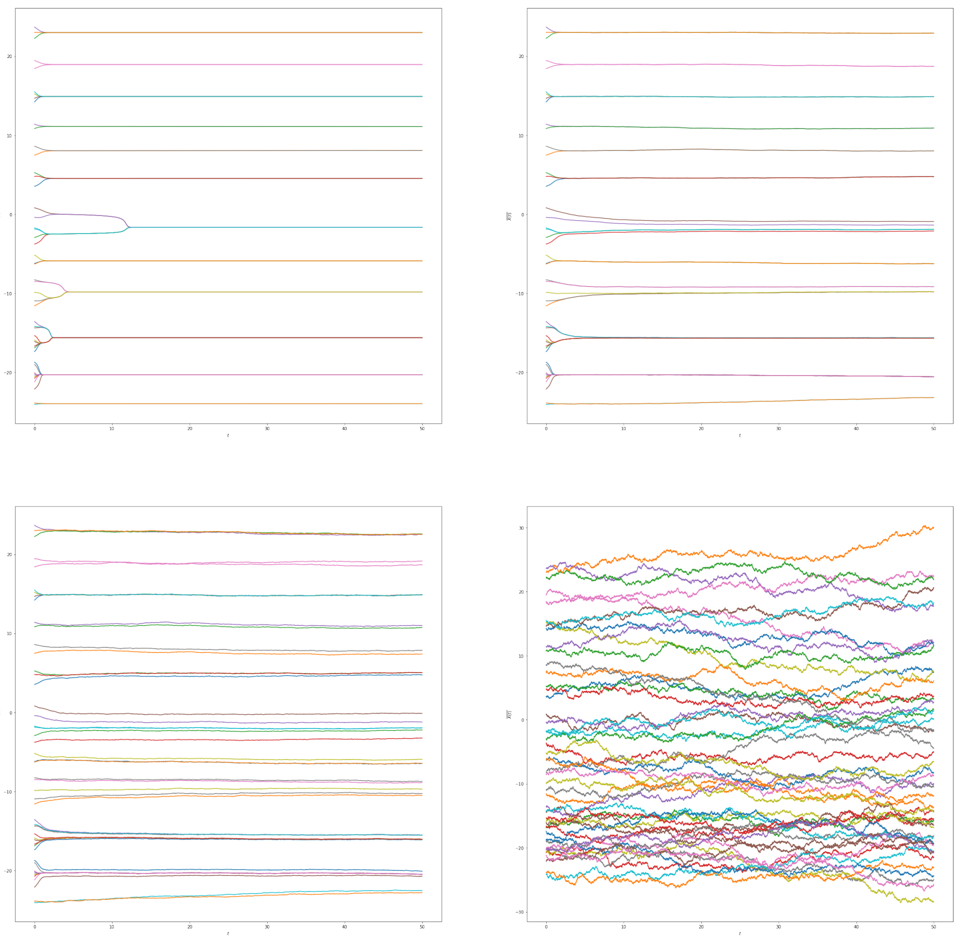


Figure 1: Xave for μ=0,0.2,0.5,5

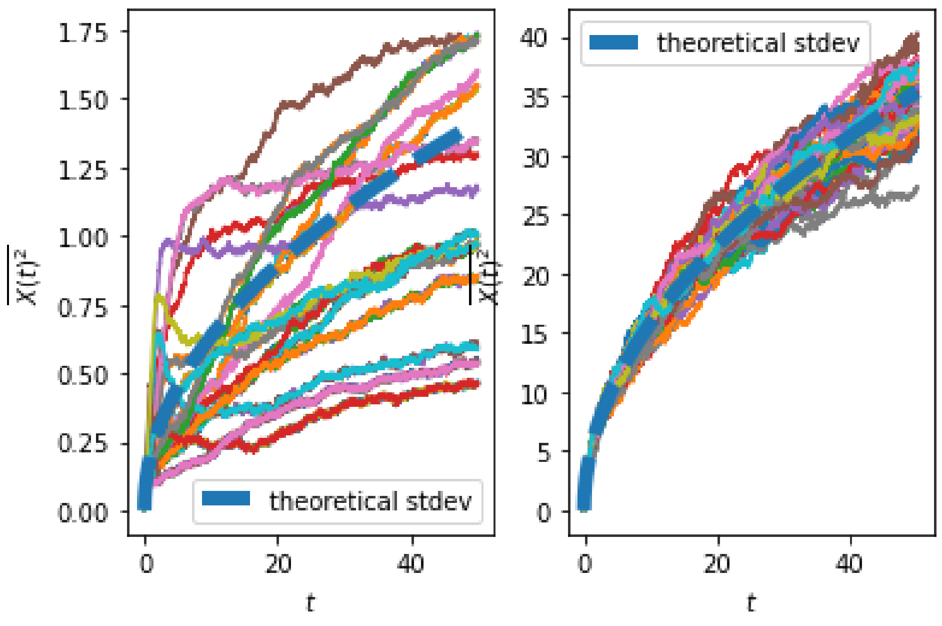


Figure 2: Xstdev for μ=0.2,5