Scientific Computation Project 2

Amaury Francou — CID : 01258326

March 7, 2022

Part 1

1

We analyze the `gSearch` function which implements the Dijkstra's algorithm using a binary heap data structure to manage the priority queue. The library `heapq` allows to use a python list as a binary heap, in particular performing insert and minimum removal operations, while maintaining the heap invariant, in $\log(N)$ computation time, where N is the length of said list. We consider a random graph with N nodes and L edges, where every node is reachable from the source. Dijkstra's algorithm implies three main steps. The first one consist of removing the node with the minimum provisional distance, which is made N times in $\log(N)$ time. The second one consist of computing the new provisional distances of the minimum distance node's explored neighbors and updating the binary heap accordingly. In this implementation, as operations on dictionaries - including `.pop()` - are made in constant time, this second step is made L times in $\log(N)$ time, using `heapq.heappush` insertion. The last step consist of computing the provisional distances of the minimum distance node's unexplored neighbors and inserting them in the binary heap. This last step in made N times in $\log(N)$ time, using the same `heapq` method. Thus, the overall computation time doesn't grow faster than $(L+N)\log(N)$.

We implement a wall time testing function `test_queue`, which computes specifically the wall time in which the highest-priority node is identified and removed from the queue - using `heapq.heappop` - for an increasing number of nodes N. The function also computes the wall time observed for the entire Dijkstra's algorithm with binary heap, for an increasing number of nodes. The `gSearch` function is tested on random graphs, where the number of edges L is taken 3 times greater than N, to ensure having presumable connected graphs, each node having in average 3 neighbors in this setting.

In figure 1, we observe that the computed wall times for the minimal removal operation well follows a $\log(N)$ trend. It is as expected from the use of a binary heap structured list, performed here through the use of the `heapq` python library. As we use $L = 3N$, we equally see that in figure 2, the wall times for the full Dijkstra's algorithm using a binary heap follows a $N\log(N)$ trend, which was also the expected behavior.

2

We seek to fine the widest possible path between two nodes n_1 and n_2 in a weighted undirected graph. Namely, we seek to find the route between said nodes that has the maximum possible minimum weight, in between two consecutive inner nodes, belonging to the path that goes from n_1 to n_2. To perform this task, we use a modified version of the Dijkstra algorithm with binary heap, that has a computation time of $\mathcal{O}((N+L)\log(N))$, and in which the priority used for handling nodes in the queue is not a provisional distance, but a provisional widest path.

As we are looking for the *maximum* minimum weight in the path, and as the binary heap managed by the `heapq` library performs a *minimum* removal in $\log(N)$ time, we store the inverses of the provisional widest paths. We retrieve this largest minimum weight by reinverting in the different steps of the algorithm. We start by initializing the heap in the same way as previously, except that the first provisional widest path is not set exactly to zero to avoid division errors.

Until n_2 is removed from the queue : we pop n^* - the node with the greatest provisional minimum weighted edge in his path to n_1. We transfer n^* to the finalized nodes list. We then iterate through its neighbors. Still three possibilities arise.

The first possibility is that said neighbor is a finalized node : we move to the next neighbor without performing any further operations.

The second possibility is that the considered neighbor is already in the queue : we compute the new provisional minimum weight in his path. This value is the maximum between his previous minimum weight in the already seen route and the new minimum weight in the newly discovered route that goes from n_1 to this neighbor, ultimately stopping by n^*. This new minimum weight in this newly discovered route is the minimum between n^*'s greatest minimum weight in his paths to n_1 and the weight of the edge connecting n^* and the current considered neighbor. This rule is implemented using the built-in `min` and `max` python methods. If the new provisional minimum weight of the considered neighbor is higher than the previous stored one, then we update said value in the binary heap.

The third possibility is that the neighbor has never been encountered before. In this case, we add the neighbor to the queue, computing a first provisional minimum weight in his path to n_1, that is the minimum between n^*'s greatest minimum weight and the weight of the edge connecting to n^*.

Notice that we stop the iteration process directly when n_2 has been finalized, since we are looking only for the maximum minimum weight in the routes that connects n_1 and n_2.

To compute a feasible route between n_1 and n_2, we keep track of each processed node's parent node. Namely, each time a provisional minimum weight is computed for a route between n_1 and a node n, we store the previously visited node in the considered path : n's parent. We obtain a list in which i-th element's value is the parent node of node i. We then use a specific algorithm to retrieve the path, starting from n_2 and going to n_1, reversing the output list. We could also use a recursive algorithm here.

Part 2

1

We analyze a given numerical method that computes the solution of the SDE $dX(t) = 2X(t)dt + X(t)dW(t)$. We first assess the accuracy of the method implemented in the `solveSDE` function, by computing the weak error $\epsilon_w(T) = |\overline{X_T} - X_0 \exp(\lambda T)|$ and the strong error $\epsilon_s(T) = \overline{|X_T - X_{exact}(T)|}$, for an increasing parameter `nt` and thus a decreasing time step $\Delta t = \frac{T}{\texttt{nt}}$. We obtain figure 3. We observe that the new method implemented has a weak order of convergence roughly equal to 1.10 and has strong order of convergence equal to 1.05. We have that this custom method has a strong order of convergence that is faster than the Euler-Maruyama method (1.05 against 0.5), and a weak order of convergence that is rather similar (1.10 against 1.13). Hence, this method is more accurate, as the average behavior is captured with a precision equivalent to what is provided by the E-M method, but with also a faster strong order of convergence. This gives us that this given method is also capable of following the expected path more accurately.

We then assess the stability of the given method. We compute the mean of several solutions X of the given SDE and plot them against time. The exact mean at time t reads : $X_0 \exp(\lambda t)$. We compute the solutions for the two different parameters `nt` $= 3$ and `nt` $= 512$. We then respectively have $\Delta t \approx 0.33$ and $\Delta t \approx 0.002$. In figure 4, we observe that the solution corresponding to `nt` $= 3$ increases too fast and drifts away from the expected theoretical mean. On the other hand, the solution corresponding to `nt` $= 512$ sticks fairly with the expected solution. In figure 5, we display the weak error of convergence, that is to say the difference between the expected mean at time t and the actually computed mean at the same given time. The error related to the solution corresponding to `nt` $= 512$ stays small and globally decreases. On the opposite, the error corresponding to the solution related to `nt` $= 3$ increases. In this case, `nt` has been chosen too small and the solution is not stable.

We finally assess the efficiency of the given method. In figure 6, we observe that the computed wall times follow a linear trend. The method used in the `solveSDE` function uses a master loop over the `nt` computations. Considering that the number M of several computations of X made is fixed, we have that the operations count does not grow faster than $\mathcal{O}(M\texttt{nt}) = \mathcal{O}(\texttt{nt})$.

2(a)

We consider solving a system of ODEs. We implement the Euler method, which reads : $x^i_{k+1} = x^i_k + h \times x^i_k \left(\alpha^i + \gamma \sum_i C_{i,j} x^k_j\right)$, with h being the step size. We set the initial conditions in an initialized array, before entering a master loop over the number of steps `Nt`. At each step, we compute the `n` operations using matrix products, with relevant `numpy` methods and slice notations. In particular, `numpy` being a C-wrapper, it allows faster wall times over those operations. Considering the number of species `n` as fixed (here `n`=20), we have that the operations count is linear in `Nt`.

2(b)

We first confirm through figure 7 that - accordingly with our analysis - the wall time observed for `model1` is linear in `Nt`.

Secondly, we observe the different trend in the abundance of species x_i. In figures 8, 9 and 10, we observe that the different species can wether grow exponentially until reaching a stationary abundance, exponentially decrease from a high initial value to zero, or exponentially increase from a low initial value before exponentially decrease to zero. In figure 11, we see that all species have an asymptotic behavior, their abundance becoming stationary at long times. This phenomenon emerges as the derivatives of the quantities x_i converge to zero for all species.

Parameter γ controls the magnitude with which the interaction between species affect the evolution of the abundance of a given one. In figure 12, γ has been set to 100000. We observe that increasing γ does not change the overall trend in the abundance, each species having the same asymptotic behavior then with $\gamma = 1$. However, for all species increasing γ decreases the transition phase between the initial value and the asymptotic value. The evolution curve becomes steeper and the final stationary behavior of the abundance is reached more quickly.

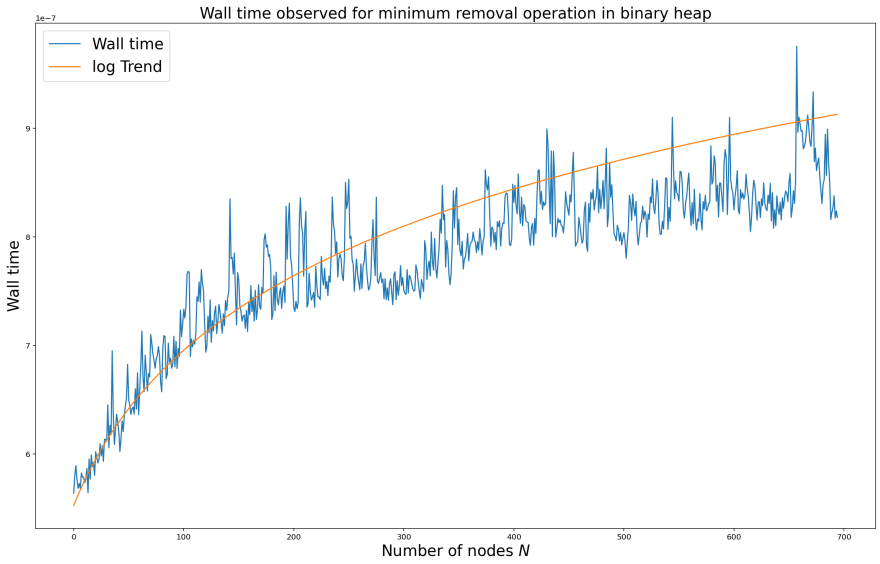

Figure 1: Part 1 - Question 1 — Wall time observed for minimum removal operation in binary heap - using `heapq.heappop` - for an increasing number of nodes N

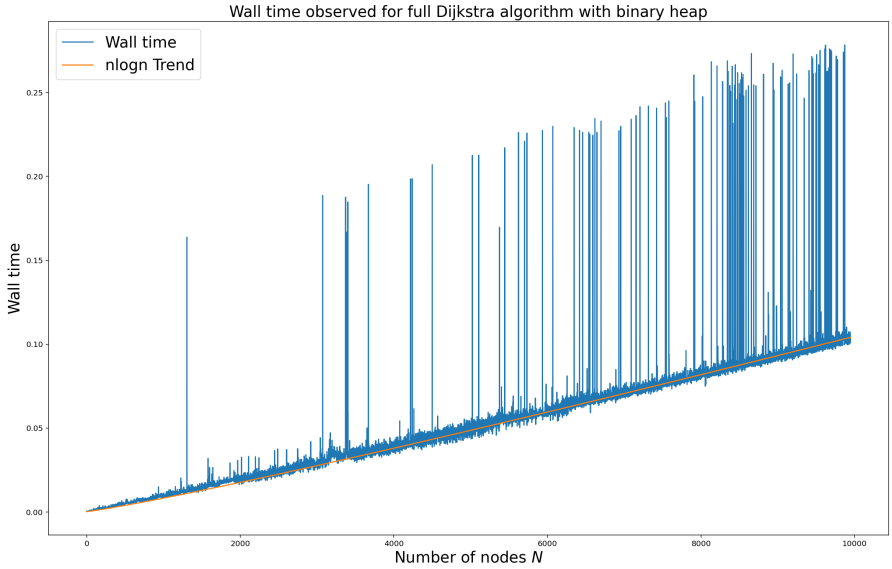

Figure 2: Part 1 - Question 1 — Wall time observed for full Dijkstra algorithm with binary heap for an increasing number of nodes N

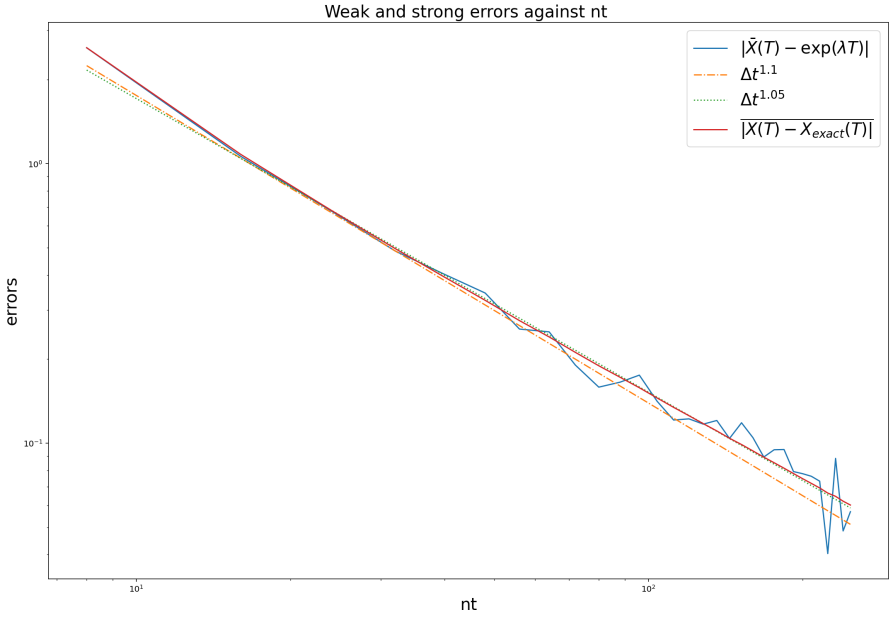

Figure 3: Part 2 - Question 1 — Weak and strong errors of convergence against `nt` in `loglog` scale

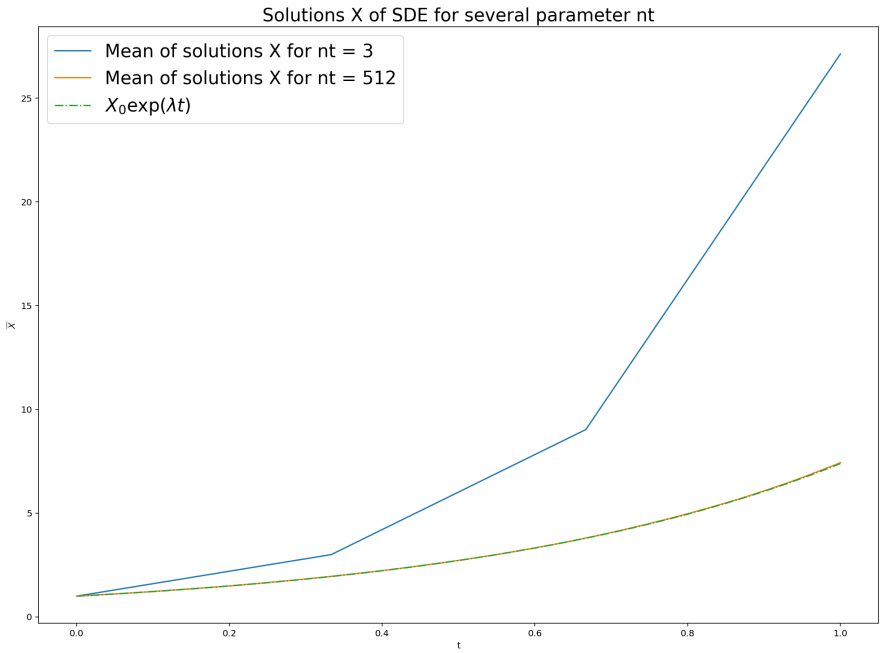

Figure 4: Part 2 - Question 1 — Mean of solutions X of SDE for several parameter `nt`

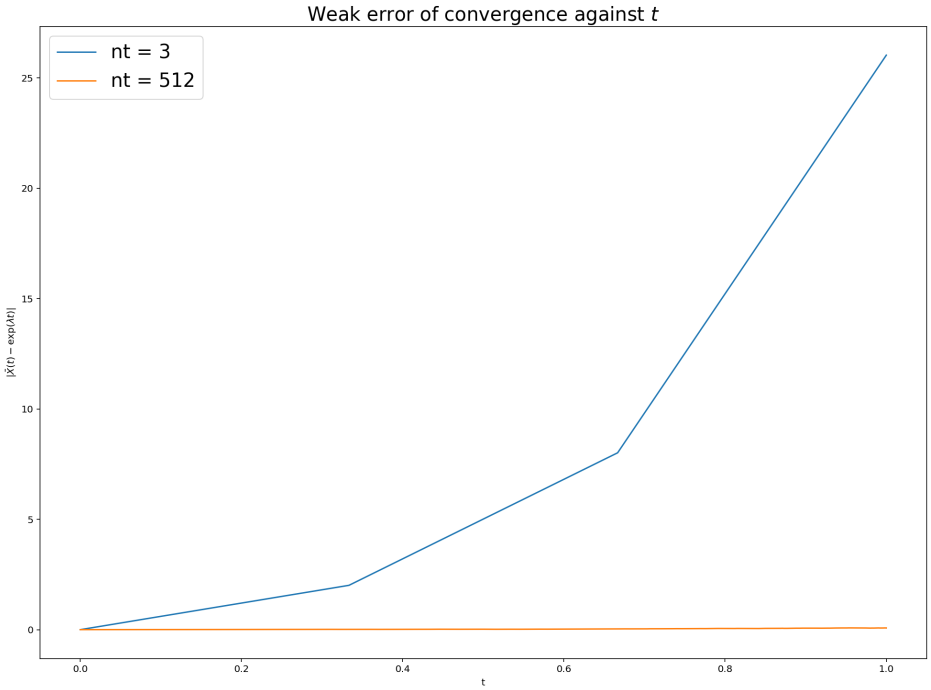

Figure 5: Part 2 - Question 1 — Weak error of convergence at time t, in normal scale

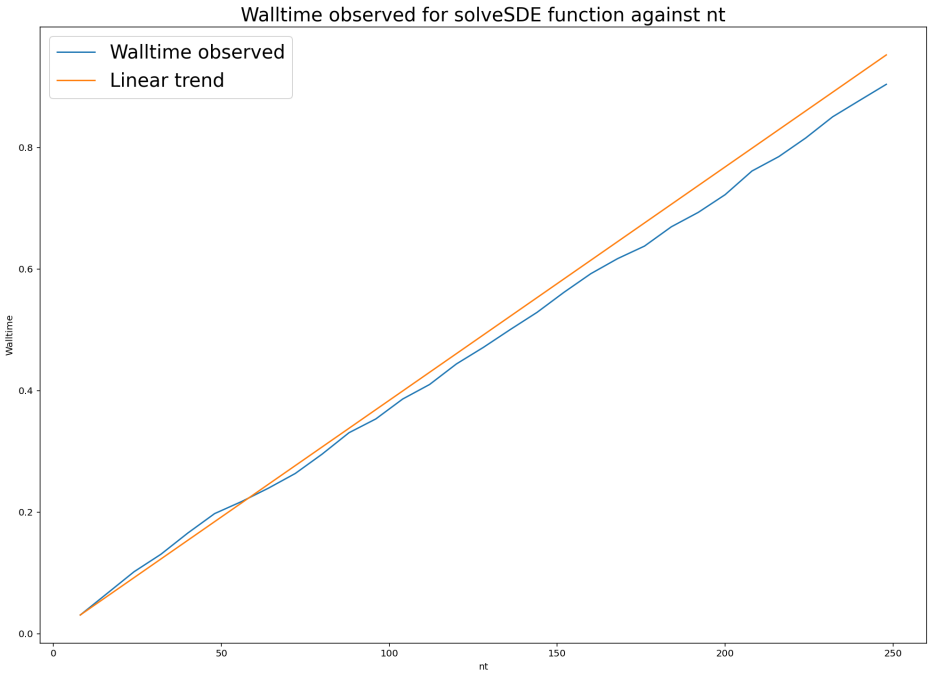

Figure 6: Part 2 - Question 1 — Wall times observed for the `solveSDE` function against `nt`

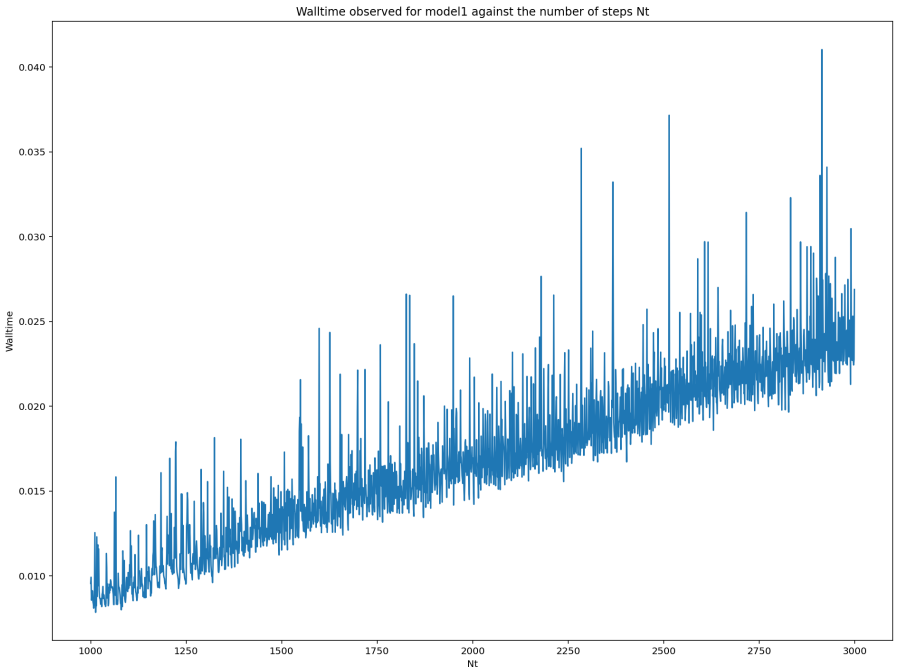

Figure 7: Part 2 - Question 2 — Wall times observed for the `model1` function against `Nt`

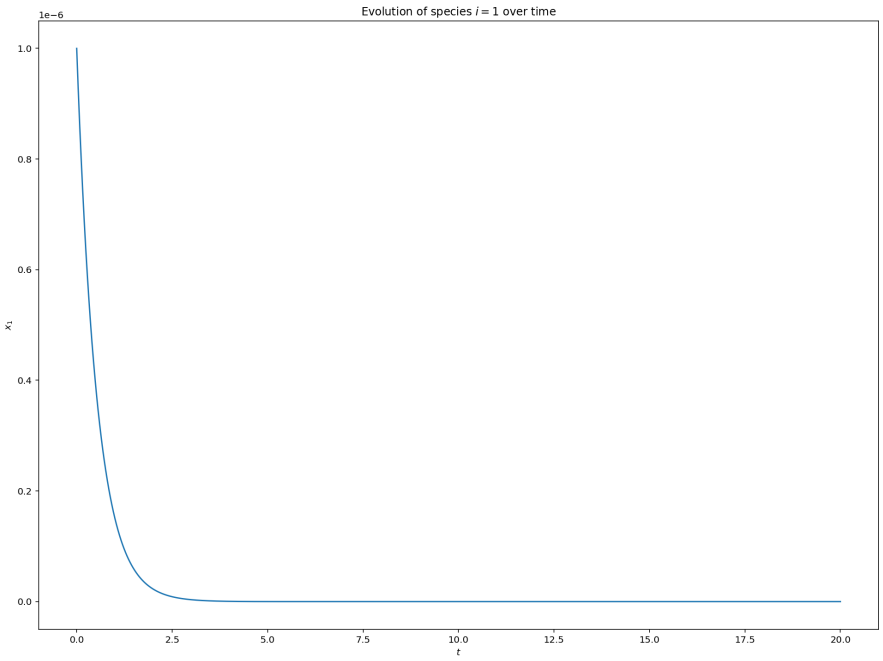

Figure 8: Part 2 - Question 2 — Evolution of species $i = 1$ over time, with $\gamma = 1.0$

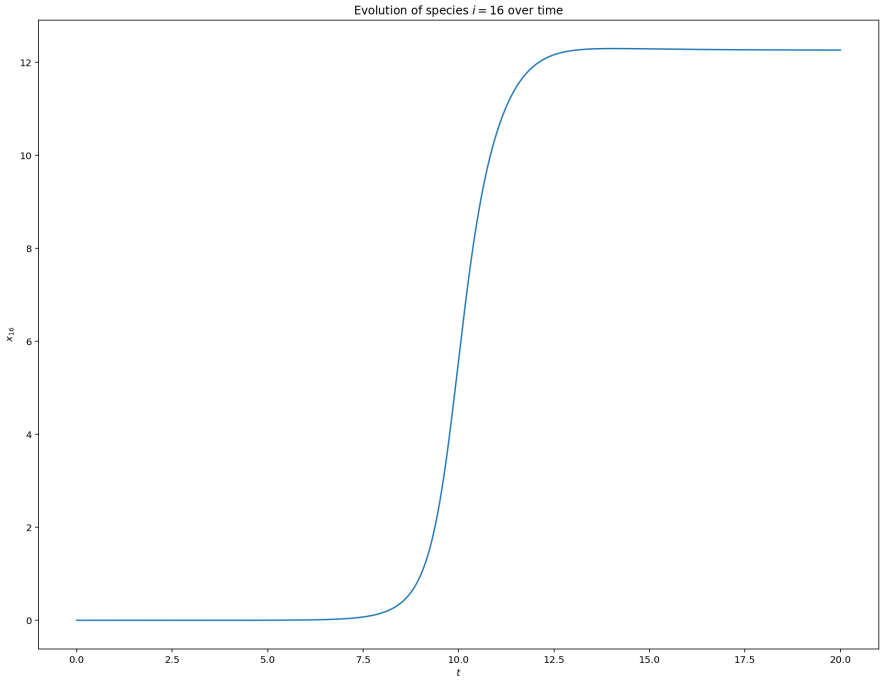

Figure 9: Part 2 - Question 2 — Evolution of species $i = 16$ over time, with $\gamma = 1.0$

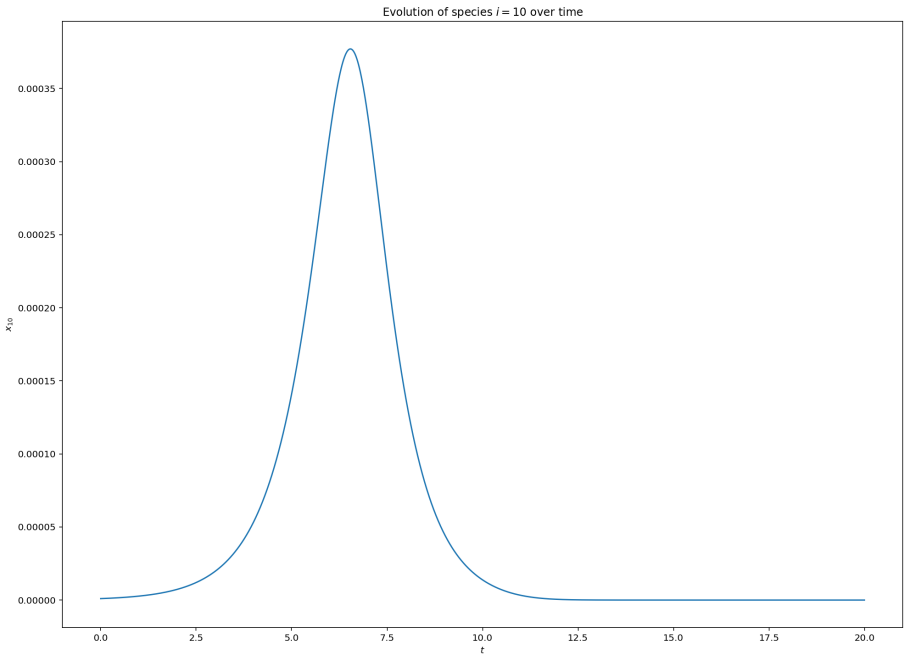

Figure 10: Part 2 - Question 2 — Evolution of species $i = 10$ over time, with $\gamma = 1.0$

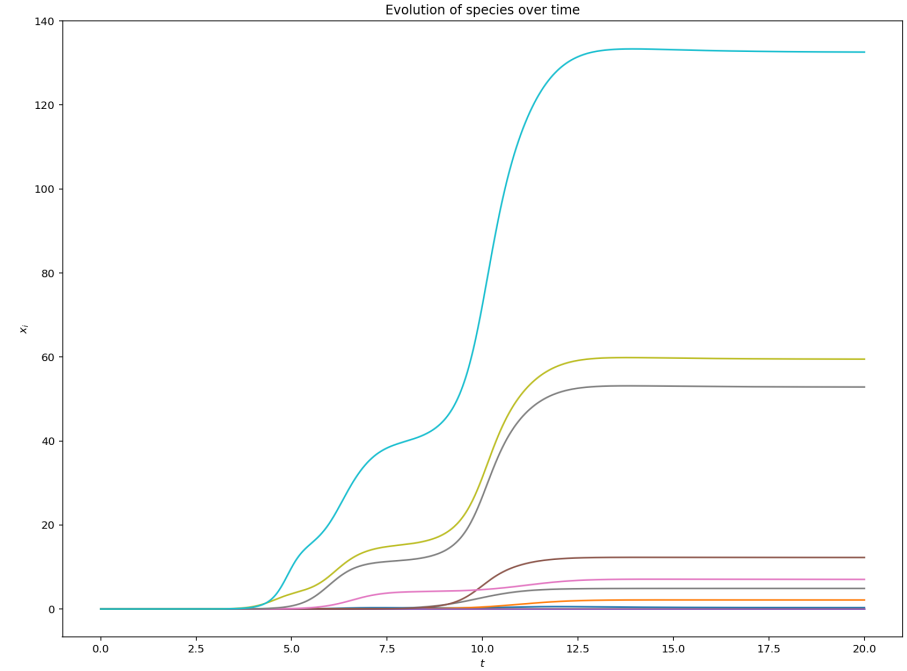

Figure 11: Part 2 - Question 2 — Evolution of all species over time, with $\gamma = 1.0$

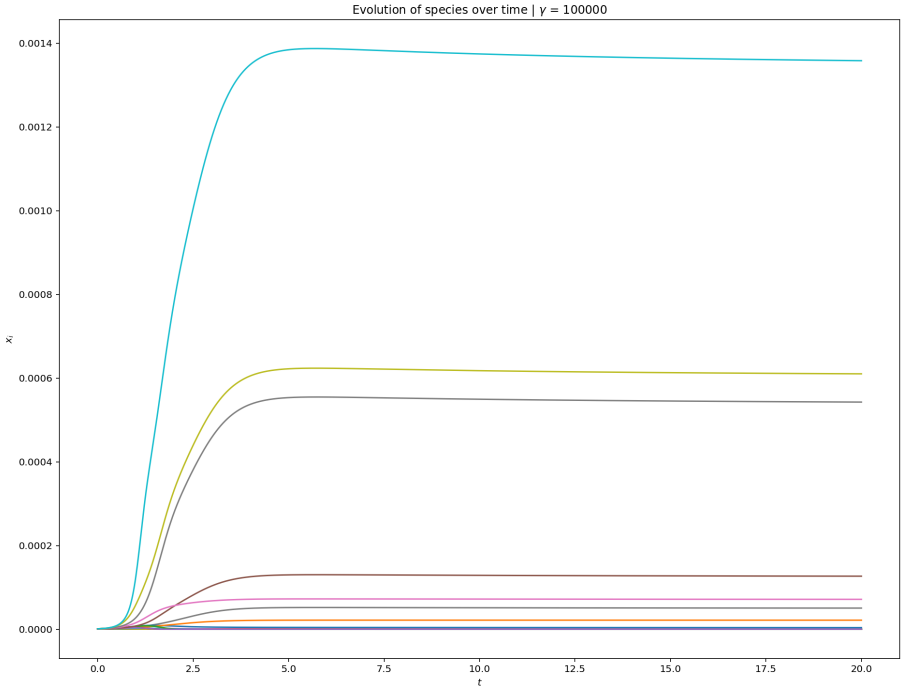

Figure 12: Part 2 - Question 2 — Evolution of all species over time, with $\gamma = 100000$