—— Simulation and Optimization ——
Assigment 1

Bruno Duarte - 118326, Guido Cazzolli - 122333

Abril 2024

0.1 Prob 1

In order to create the inventory simulation, we used the code developed in lesson 2 as base. To implement the new changes, we needed to alter the way we treated the inventory, as it couldn't be just a simple integer anymore, as that wouldn't suffice to let us attribute a shelf live to the items, although it would still allow us to implement the first change, in which we just add to change how we calculate the ordering price depending on the level of our inventory. To change our inventory, we started to have it be a list with the date that a item spoils, this way. In order to do this, we created 2 functions, one to add the items and another to take them. In the function to add elements, we first check if we have items in backlog, and if so, we subtract them to the value of items to add. After that, for each item we can still add, we calculate a random date for the item to perish using the formula:

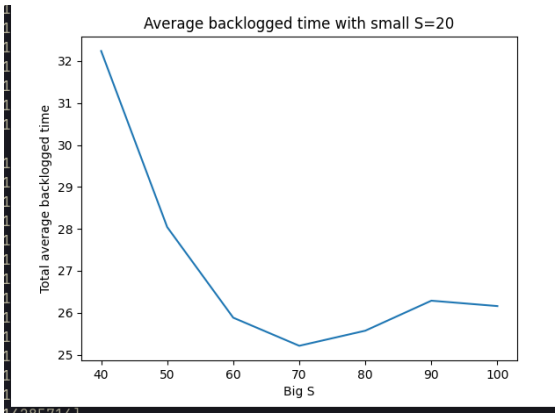
$$date\_to\_perish = sim\_time + random.uniform(1.5, 2.5)$$

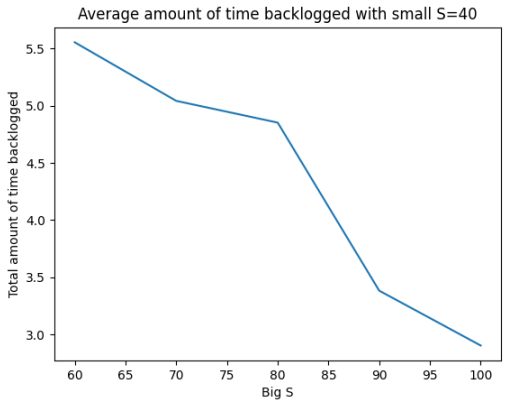
After that we update a variable that we have that represents the level of the inventory, similar to the initial implementation from class. Although this variable is only truly useful and needed to save the amount of items we have in backlog (as this variable will be negative then), we though it wouldn't hurt to just make it complement the list of the items and use this variable to also see easier the level of our inventory. When we want to take items, we make a loop that ends when we take all the items we wanted or when the inventory is empty. This can happen if we take more items than we add, or if some of the items are discovered to be perished. After this loop we update our variable with the new inventory levels, that can be negative.

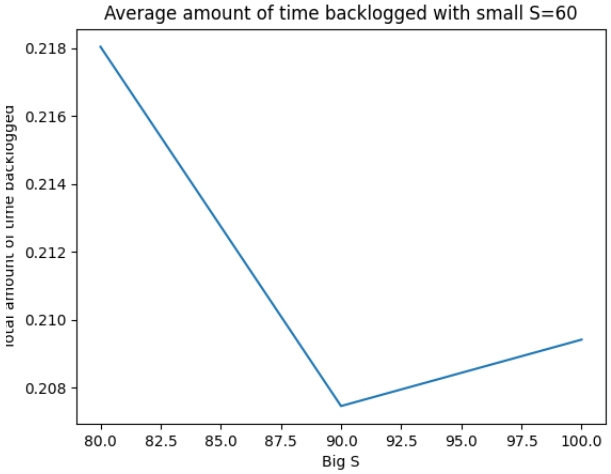
0.2 Prob 1.1

Throughout our simulations, the first data we looked at was the average items demanded by the clients per month, which was roughly 25 per month. This helps us understand some things, the first one is that for inventories with the difference between the big_s and the small_s of 20 ((20,40), (40,60), etc), we would be ordering the majority of months (in the case of (20,40), the average was 90 orders per simulation), while bigger differences between the S's would make it so we should expect a lower amount of orders, as per order we would get more than the average that we would sell. This, as expected, also affects the amount of time our inventory is backlogged, as we can see in the following graphs:

For all these graphs, we made big_s increase with a rate of 10 to have a more smooth curve. Looking at the graphs, we can see that in fact the amount of time we spend in backlog is affected by our small our difference of S's is. As we get to bigger inventory sizes, the amount of time we are backlogged decreases, as we have more product from which to take. This however, does not show the full picture, as in our simulations, despite bigger inventory sizes having less time





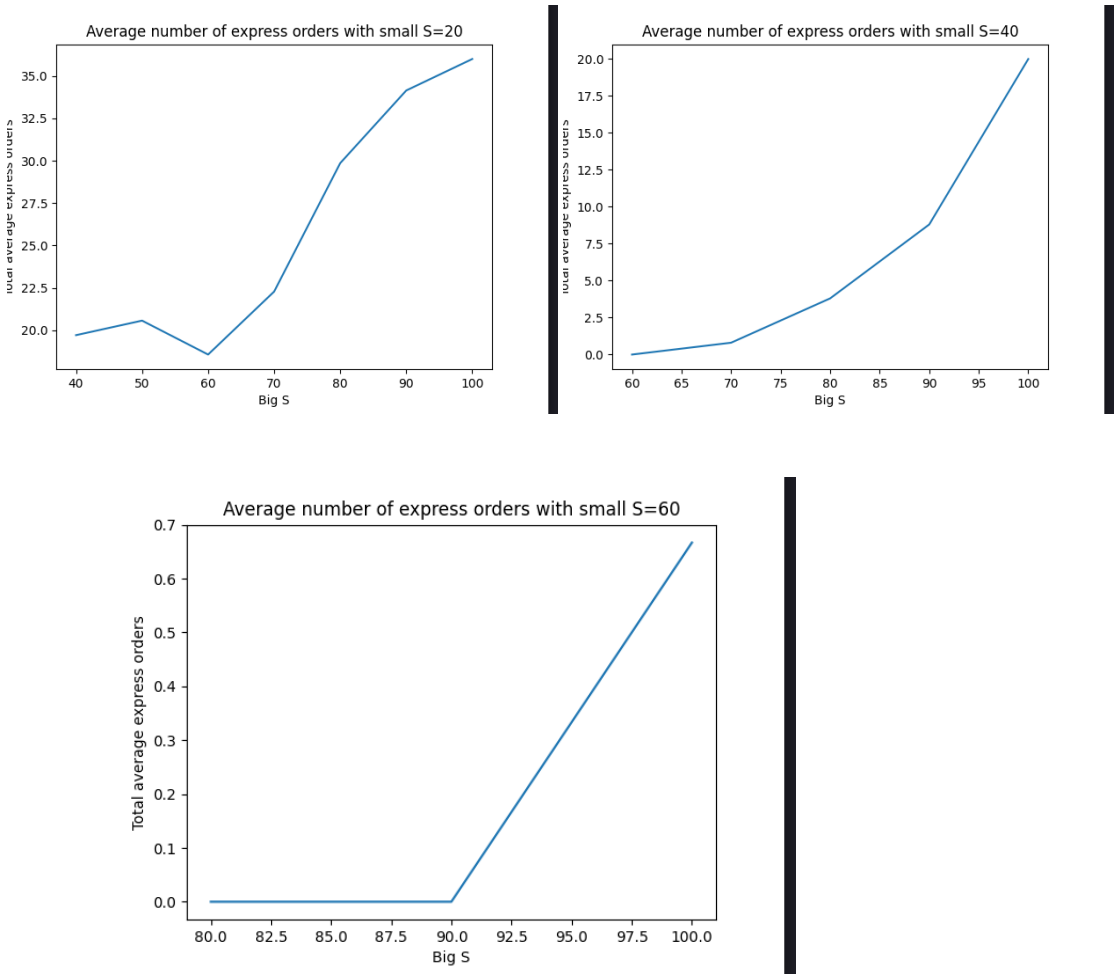


backlogged, they placed more express deliveries than smaller sizes, we show the graphs below.

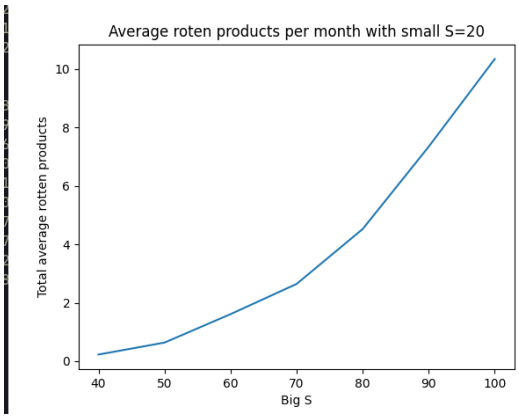
Although at first this seems contradictory, upon further and more detailed analysis of the simulation we found the reason why, it is the rotten items. We will see in the next set of graphs that the amount of rotten products increases by a lot the bigger our inventory is. Here we show the graphs.

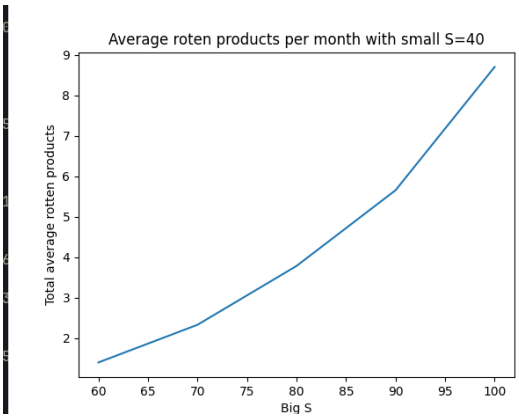
As we can see, the amount of rotten items found per month greatly increases the bigger the S's are. The way to justify this is the following:

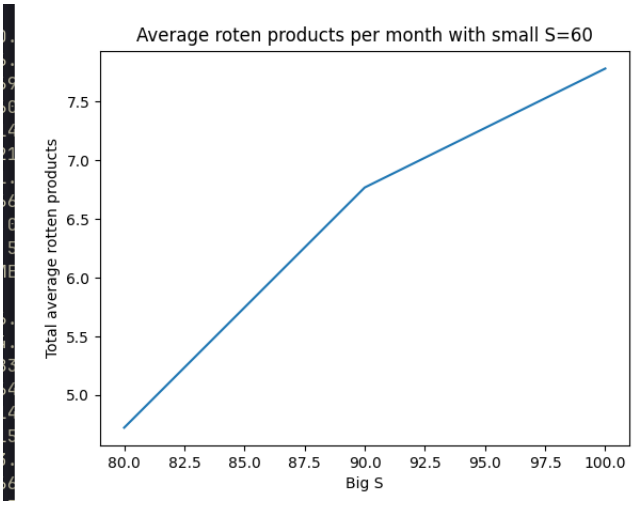
- As we concluded earlier, the average customer demand monthly is approximately 25.
- This makes it so that a difference in S's bigger than 25 (40,60,80) places a order for more product than the one consumed in a month, which will



make said product more vulnerable to perish. The bigger the difference of S's, the bigger the amount of product that risks to perish, and when it is too big, (bigger than 50 more or less), we will notice that the first month will go okay, as no product can perish, the second month will start to show some perished items, but in the third month all the items that were not sold will perish, as the max time for them was reached. This makes it so that if for example, for the next month we have 28 remaining, and the small_s is 20, we will not order anything. However in that month all the product will perish, and so we will end up having a customer demand where we eventually check and we have no more good product, and we enter in backlog. This behaviour is further enhanced the bigger the big_s is, but it also is affected by the small_s in a similar way, as we will not order more product believing that what we have is enough only to find



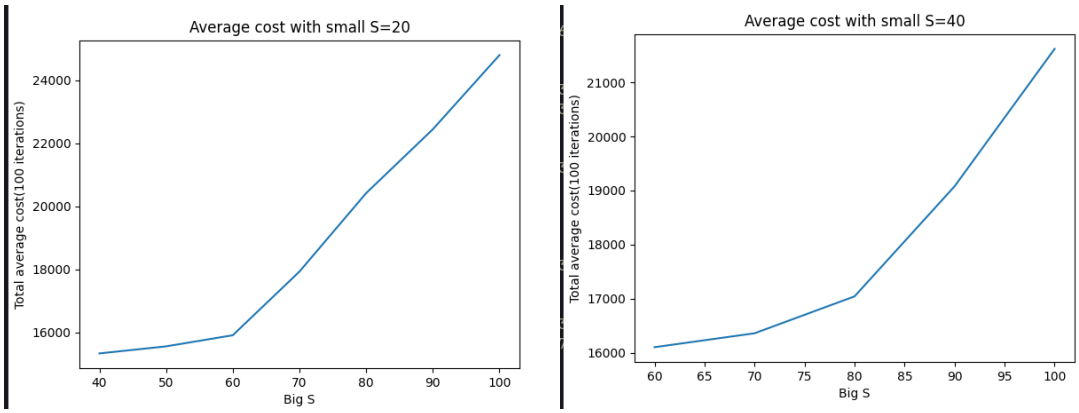




out that all the product was rotten.

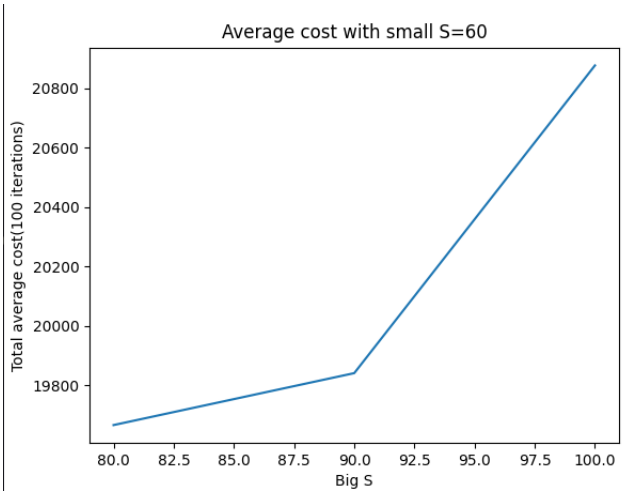
- This is however mitigated by smaller inventory sizes. As they order more times, their product will be a lot more fresh, and so it will be more reliable and the orders will be more precise, as they don't have the usual "all product rotten" associated with bigger inventories.
- This smaller sizes are however more vulnerable to shortage in theory, but because their inventory is so much more reliable, they don't enter into backlog so much, although they also place some express orders.

With all this, we can see that the smaller inventory (20,40), is the most cost effective, because even tho it spends more of it's time backlogged, the orders from the previous month arrive before the actual month ends, and so they can end most months with a very small, but positive, available product. This size



also benefits the most out of express orders, as it is the one more susceptible to big variances in costumer demand (if a lot of customers demand in one month), but due to it's small size the order arrives and can almost always clear the backlog and have enough items, or very close, to satisfy the current month. One other advantage of this size is that by being more time backlogged, it does not risk so much to perish it's items, as a lot of them are immediately delivered the moment they arrive, with the rest fulfilling the rest of the orders from the month. In other bigger sizes, because we don't order so many times and order a lot more product, a bigger percentage of our product will rotten, decreasing immensely the inventory reliability to fulfill orders, as we can't be sure if the inventory levels are close or not to the amount of usable product until we check it. A way to increase the effectiveness of bigger inventory sizes would be to make the date of the oldest product or products influence the amount of product we order, and maybe place a bigger order if we can count on a lot of product being a bigger part next month. Other option would be to simply check more regularly the state of the product and throw away rotten product, but that is also not very efficient and both methods fail in comparison to having a smaller inventory size. With all this, it can be useful to sometimes use express orders when in a pinch and with a lot of shortage, but as the normal orders arrive at the max in the next month, it is almost always not fully needed with the right inventory size, as the product ordered by the normal order should suffice to not enter in backlog at the end of the month. To end, we will show the graphs for the average total cost. We mention that for all the graphs showed the values were obtained by running 100 iterations of the simulation for each pair of S's.

One way to change the results could be to increase the maximum amount of time that a order can take to being delivered (bigger than one month), in which the reliability of the smaller size would also be affected and some bigger sizes could gain the edge by maintaining more stock at the end of the month.



0.3 Problem 2: Kermack-McKendrick model

The differential equations of the Kermack-McKendrick model that govern the process are utilized for describing the evolution of an infectious disease in a population, by simulating the 3 functions of the system:

- s(t): fraction of the population that is susceptible
- i(t): fraction of population that is infected
- r(t): fraction of population that is recovered (no longer infectious)

The differential equations that govern this SIR model are solved by simulating with python two different methods.

0.3.1 Problem 2.1: Solution with Euler-Forward method

In order to solve a system of 3 differential equations the initial conditions of the 3 functions s(0)=99, r(0)=0 and i(0)=1, and the parameters of the process β=0.5, k=0.1 and dt=0.01 are needed; the 6 parameters are given through command line (sys.argv[]). The problem is solved using the code developed in lesson 4 as base and mainly divided into 3 subfunctions: initialize(), update() and observe(). In the initialization function the time is set to 0 and the initial values of the 3 function are set in the respective vectors sr, rr, ir and tr. In the update function the following values of the functions are evaluated by applying the numerical integration of the differential equations using the Forward-Euler method:

```
s1=s-B*s*i*dt
i1=i+(B*s*i-k*i)*dt
r1=r+k*i*dt
s, i, r = s1, i1, r1
```

And so the time variable:

```
t= t + dt
```

Finally in the function observe the new values of the 3 functions are appended to the previous ones. In the main program the simulation is made by a for cycle which updates the time variable a discrete number of times tf=200.

In the end of the simulation the graphs of the 3 simulated function are shown in Figure 1 respect to the time variable and the computation time is plotted.

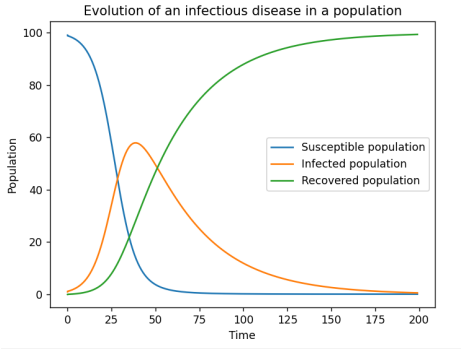


Figure 5: Euler-Forward Method

0.3.2 Problem 2.2: Solution with Runge-Kutta method

The program is solved by using the same base code but by performing a different numerical method.
In order to simulate the 3 function the population in necessary to evaluate the 4 constants k1, k2, k3, k4 for every function by using the same parameters of the problem 2.1, and in the end calculate the numerical integration by using the constants:

```
s1=s+(k1s+2*k2s+2*k3s+k4s)/6
i1=i+(k1i+2*k2i+2*k3i+k4i)/6
r1=r+(k1r+2*k2r+2*k3r+k4r)/6
```

As shown in Figure 2 it's possible to see the result, very similar to the previous one.

0.3.3 Problem 2.3: Comparation

It's possible to save the data of the 2 programs by saving the resulting vectors and the time variable in 2 different text files

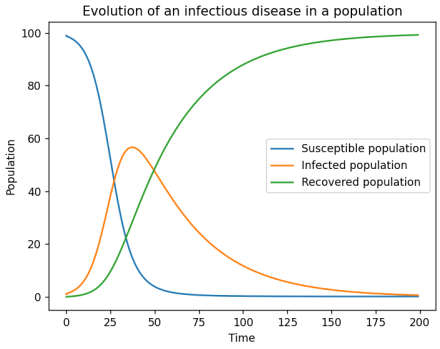


Figure 6: Runge-Kutta Method

(data_rk.txt and data_euler.txt).

```
data = np.column_stack((tr, sr, ir, rr))
np.savetxt('data_rk.txt', data, delimiter='\t', header='Time\tPopulation', fmt='%.4f')
```

The data are then charged on another file and compared between each other by plotting the graphs of the 2 methods and by performing and plotting the absolute error between them (Figure 3).

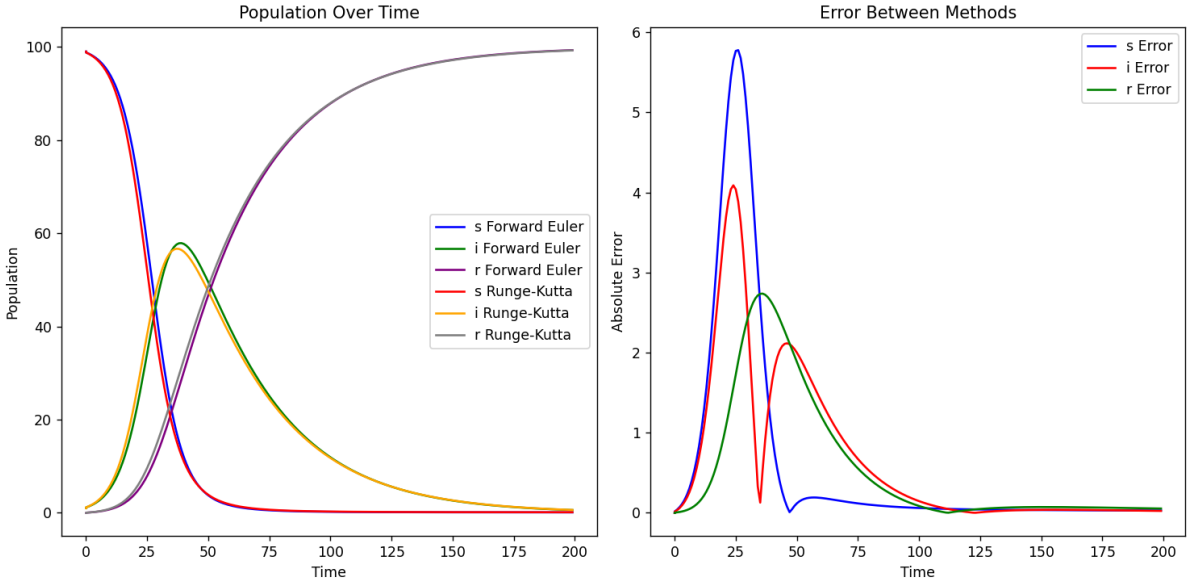


Figure 7: Comparison

By looking at the comparison graph we can make a relative evaluation of the precision between the two approaches: the results are consistent and the two methods are similar; it's possible to notice the only few deviation points during the rise of the i(t) function and during the dropping of the s(t).