Simulation Mini-Projects

Mestrado em Engenharia Informática

Universidade de Aveiro

DETI - Aveiro, Portugal

2023/2024

SO - Simulação e Otimização

Guilherme Antunes [103600], Pedro Rasinhas [103541]

Index

1 Introduction 3
2 Problem 1 - Inventory System 3
2.1 Brief Description 3
2.2 Implementation 4
2.3 Results and Questions 9
2.3.1 Question 1 - Is express ordering worth it? 10
2.3.2 Question 2 - Compute the proportion of items taken out of the inventory that are discarded due to being spoiled. 11
3 Problem 2 - Kermack-McKendrick Model 12
3.1 Brief Description 12
3.2 Implementation 13
3.3 Results and Questions 16
3.3.1 Question 1 – Trace the evolution using Forward Euler method. 16
3.3.2 Question 2 – Trace the evolution using Runge Kutta method. 16
3.3.3 Question 3 – Compare the precision of both approaches 17
4 References and resources 19

1 Introduction

This report aims to explain the methodology used to solve the two simulation problems proposed for the Simulation and Optimization course project, as well as to answer the questions raised in its statement.

2 Problem 1 - Inventory System

2.1 Brief Description

A company that sells a single product would like to decide how many items it should have in inventory for each of the next *n* months (*n* is a fixed input parameter). The time between demands is IID exponential random variables with a mean of 0.1 month. The sizes of the demands, D, are IID random variables (independent of when the demands occur), with:

D (size of demand)	With Probability
1	1/6
2	1/3
3	1/3
4	1/6

For this, the company reviews the inventory at the beginning of each month, and decides how many items to order from its supplier based on the current size of the inventory, and based on a stationary policy that the company uses (*s*, *S*):

$$Z = \begin{cases} S - I & \text{if } I < s \\ 0 & \text{if } I \geq s \end{cases}$$

where I is the inventory level at the beginning of the month. Based on the amount, the company incurs an ordering cost, calculated based on a setup cost (K = 32$) and on an incremental cost (IC = 3$), that considers the Z (amount) previously calculated. After that, a demand will occur (somewhere in the time specter), when it happens, if the demand exceeds the current inventory size, the inventory goes into stockout, leaving the current inventory level as a negative value. When an order arrives, it first eliminates that backlog (if possible) and adds the remainder to the inventory.

Besides the ordering cost, the company also incurs in two different types of costs: holding costs and shortage costs. The first one, is associated with when the inventory level is above zero (the inventory has items), and the second one is calculated when the inventory

is below zero. Besides this, there is also a differentiation between express and normal orders. The express order is placed when the inventory goes into stockout, and, although it's costlier (it has a setup cost of 48$ and an incremental cost of 4$), it arrives sooner (with a delivery lag uniformly distributed on [0.25, 0.50] month), when in comparison with the normal order mentioned above.

The inventory is also perishable, which means, having a shelf life distributed uniformly between 1.5 and 2.5 months. The company discovers if an item is determined to be spoiled, it is discarded and the next item in the inventory is examined.

We assumed that the inventory had 60 items in the beginning, and that no order was placed beforehand. We also simulated the inventory system for 120 months, and then calculated the average total cost per month, the expected proportion of time that there is a backlog, and the number of express orders placed to compare the following nine inventory policies:

s	20	20	20	20	40	40	40	60	60
S	40	60	80	100	60	80	100	80	100

2.2 Implementation

To implement the inventory system described above, and all the respective functionalities, we first need to start by defining some constant values:

```
K = 32                  # setup cost Normal order
IC = 3                  # Incremental cost per item
HC = 1                  # Holding cost per item per month
PC = 5                  # Backlog cost per item per month
N_MONTHS = 120          # 120 months
MIN_LAG = 0.5
MAX_LAG = 1

shelf_life_min = 1.5
shelf_life_max = 2.5

probs_demand = [1/6, 1/3, 1/3, 1/6]

# Inventory policies
policies = [(20, 40), (20, 60), (20, 80), (20, 100), (40, 60),
            (40, 80), (40, 100), (60, 80), (60, 100)]

init_inv_level = 60

time_next_event = [0, 0, 0, 0, 0]
next_event_type = 0

dict_events = {
    1: "Order arrival",
    2: "Demand",
    3: "End of simulation",
    4: "Inventory evaluation"
}
```

```
num_events = 4
final_results = []
time_theres_backlog = 0
n_express_orders = 0
n_spoiled_items = 0
all_items = 0
demanded_items = 0
```

Besides that, for each policy, we need to reset to start/reset the following variables:

```
for s, S in policies:
    # simulation clock
    sim_time = 0

    # state variables
    INV_LEVEL = init_inv_level
    time_last_event = 0

    # statistical counters
    total_ordering_cost = 0
    area_holding = 0
    area_shortage = 0
    time_theres_backlog = 0

    # time of next events
    time_next_event = [0, 0, 0, 0, 0]
    time_next_event[1] = 1e30           # order arrival
    time_next_event[2] = sim_time + random.expovariate(1/0.1) # demand
    time_next_event[3] = N_MONTHS       # end of simulation
    time_next_event[4] = 0.0            # inventory evaluation (1st event - at the start of each month)

    inventory = np.zeros(INV_LEVEL)    # init_inv_level items and their shelf life

    n_spoiled_items = 0
    all_items = 60
    demanded_items = 0
    n_express_orders = 0


    # generate the shelf life of the items in the inventory
    for i in range(INV_LEVEL):
        shelf_life = random.uniform(shelf_life_min, shelf_life_max)
        inventory[i] = shelf_life
```

Here, we are starting the simulation clock (that will run for 120 months), the statistical counters, the *time_next_event* array, that has the next time for each of the events, and the inventory.

From this it's important to note two things: First, that we are setting up the "Inventory evaluation" as the first event (0.0); And second that we are starting the *inventory* variable as an array of zeros, and then generating the shelf life for each of the items in it.

Then we have the main loop:

```
while True:

    # determine the next event
    timing()
    # update time-average statistical variables
    update_time_avg_stats()

    if next_event_type == 1:
        order_arrival()
    elif next_event_type == 2:
        demand()
    elif next_event_type == 4:
        evaluate()
    elif next_event_type == 3:
        report()
        break
```

In the main loop, that will run for 120 months, we first need to determine which will be the next event, using the *timing* function described as follows:

```
def timing():
    global next_event_type, num_events, time_next_event, sim_time, time_last_event
    next_event_type = 0
    min_time_next_event = 1e29

    for i in range(1, num_events + 1):
        if time_next_event[i] < min_time_next_event:
            min_time_next_event = time_next_event[i]
            next_event_type = i # 1 - Order arrival, 2 - Demand, 3 - End of simulation,
                                # 4 - Inventory evaluation


    sim_time = min_time_next_event
```

So, in the beginning we set up the next event (the first event) as being the inventory evaluation one, as we setted up its simulation time to be 0.0. Then we make the simulation clock “jump” to the time of the next event. As the first event is the “evaluate inventory” one, we run the next function (*evaluate*).

```
def evaluate():

    global INV_LEVEL, amount, total_ordering_cost, inventory, n_express_orders

    """ EXPRESS ORDER """
    if INV_LEVEL < 0:
        # print("Express order!")
        amount = S - INV_LEVEL
        total_ordering_cost += 48 + 4 * amount
        time_next_event[1] = sim_time + random.uniform(0.20, 0.5)
        n_express_orders += 1

    elif INV_LEVEL < s:    # h
        amount = S - INV_LEVEL
        total_ordering_cost += 32 + 3 * amount      # K = 32, IC = 3
        time_next_event[1] = sim_time + random.uniform(0.5, 1)


    time_next_event[4] = sim_time + 1.0

    # update shelf life of the items in the inventory
    for i in range(len(inventory)):
        inventory[i] -= 1.0
```

In this function, running at the start of each month, we evaluate the current state of the inventory. If the inventory is stockout (*INV_LEVEL < 0*), we need to make an express order, increasing the total ordering cost as described there. If the inventory is not out of stock, but it's below *s*, we make a normal order, also increasing the total ordering cost but in a different way. After this, for both cases, we generate the next “Order Arrival” event, following the uniformly distributed delivery lags for each of the cases.
After that, we set up the next “Inventory Evaluation” event for the next month.

At last, we update the shelf lives of the items in the inventory, decreasing it by 1 (month).

After the evaluation being made, an order should be demanded (at some point), and for that we use the *demand* function.

```
def demand():
    global INV_LEVEL, inventory, sim_time, time_next_event, n_spoiled_items, demanded_items

    # Generate the size of this demand
    val = random.random()   # random number between 0 and 1
    lx = 4
    if val <= 1/6:
        lx = 1
    elif val <= 1/2:
        lx = 2
    elif val <= 5/6:
        lx = 3

    # Check for spoilage -> items that are expired
    items_spoiled = 0

    for i, item in enumerate(inventory):
        if item < 0:
            # print("\n\nItem expired!")
            items_spoiled += 1
            # remove the item from the inventory
            inventory = inventory[i:]


    # update the inventory level
    INV_LEVEL -= items_spoiled
    n_spoiled_items += items_spoiled
    # Adjust the inventory level based on the number of items actually demanded
    INV_LEVEL -= lx
    demanded_items += lx
    # print(f"Demanded {lx} items, New inventory level: {INV_LEVEL}")
    time_next_event[2] = sim_time + random.expovariate(1/0.1)
```

In this function, we first calculate the size of the order to be demanded (*lx*), following those cumulative probabilities, then we check for items that might be spoiled, which means, items that have negative values (because we are decreasing it in the *evaluate* function).
After that we update the inventory by removing the spoiled items and the demanded items.
And at last, we set up the next demand event.

After a demand is done, an order should arrive (at some point), and when it does, we are going to work inside the *order_arrival* function:

```
def order_arrival():
    global INV_LEVEL, amount, inventory, time_next_event, sim_time, all_items

    # increment the inventory level by the amount of the order
    INV_LEVEL += amount
    # print(f"Order arrived, Got {amount} items, New inventory level: {INV_LEVEL}")
    all_items += amount
    # Append new shelf lives to the inventory list
    for _ in range(amount):
        shelf_life = sim_time + random.uniform(shelf_life_min, shelf_life_max)
        inventory = np.append(inventory, shelf_life)

    # eliminate order-arrival event from consideration
    time_next_event[1] = 1e30
```

When an order arrives, first we update the inventory level, and then we generate the shelf lives for those new items that just arrived, appended at the end of the inventory array (that works as a FIFO).

Between the events and the *timing* function, we update the statistical counters:

```
def update_time_avg_stats():

    global time_last_event, area_holding, area_shortage, INV_LEVEL


    time_since_last_event = sim_time - time_last_event
    time_last_event = sim_time

    if INV_LEVEL < 0:
        area_shortage -= INV_LEVEL * time_since_last_event
    elif INV_LEVEL > 0:
        area_holding += INV_LEVEL * time_since_last_event
```

Here we update the shortage area variable when the inventory is out of stock, and the holding area when not.

In the end, we use the *report* function to append all the results:

```
def report():
    avg_ordering_cost = total_ordering_cost / N_MONTHS
    avg_holding_cost = HC * area_holding / N_MONTHS
    avg_shortage_cost = PC * area_shortage / N_MONTHS
    # print(f"({s}, {S}) {avg_ordering_cost + avg_holding_cost + avg_shortage_cost} {avg_ordering_cost} {avg_holding_cost} {avg_s
    final_results.append((s, S, avg_ordering_cost + avg_holding_cost + avg_shortage_cost, avg_ordering_cost, avg_holding_cost,
                          avg_shortage_cost, n_express_orders, n_spoiled_items, demanded_items, all_items))
```

2.3 Results and Questions

In terms of results, as we are using *random* generators each run will most likely provide a different result in the end, but one example is:

```
Final results:
s    S    Avg Total          Avg Order          Avg Hold           Avg Shortage       Express            (Spoiled,DemandedIt)
20   40   145.2158           117.2333           9.9636             18.0188            28                 (196 ,3023)
Time with backlog:  40.700188046699125
20   60   131.7987           100.2667           20.4535            11.0785            18                 (171 ,2832)
Time with backlog:  22.636175352247154
20   80   144.7460           104.0167           27.0053            13.7241            12                 (226 ,3112)
Time with backlog:  20.72554488594868
20   100  151.6225           102.6333           38.8676            10.1216            13                 (230 ,2938)
Time with backlog:  13.807726348061948
40   60   136.9443           110.7667           22.5086            3.6690             0                  (223 ,3162)
Time with backlog:  10.030137705486485
40   80   135.3143           97.8917            32.4096            5.0130             2                  (199 ,3059)
Time with backlog:  7.623157955542814
40   100  145.9480           96.4917            43.4435            6.0128             3                  (259 ,2987)
Time with backlog:  7.369385893741734
60   80   149.3460           106.2667           42.1675            0.9118             0                  (175 ,3111)
Time with backlog:  3.042571469378108
60   100  152.4023           98.6917            49.9929            3.7177             0                  (258 ,3054)
Time with backlog:  3.627444407125923
```

As evident from the data, each policy exhibits distinct characteristics in terms of managing these costs.

For ordering costs:

The cost component tends to decrease as the inventory policy allows for larger orders less frequently. For example, when comparing policies with the same reorder point (s) but varying order-up-to levels (S), such as (20, 40) and (20, 100), the ordering cost decreases from $117.23 to $102.63, respectively. This reduction should be primarily attributed to the fact that there is a smaller number of orders placed, which might mean a smaller overall ordering cost, for example: ordering 50 items in one order would be: 32 + 3 * 50 = 182$; but ordering 50 items in 5 orders (with 10 items per order) would be: 5 * (32 + 3 * 10) = 310$

For holding costs and shortage costs:

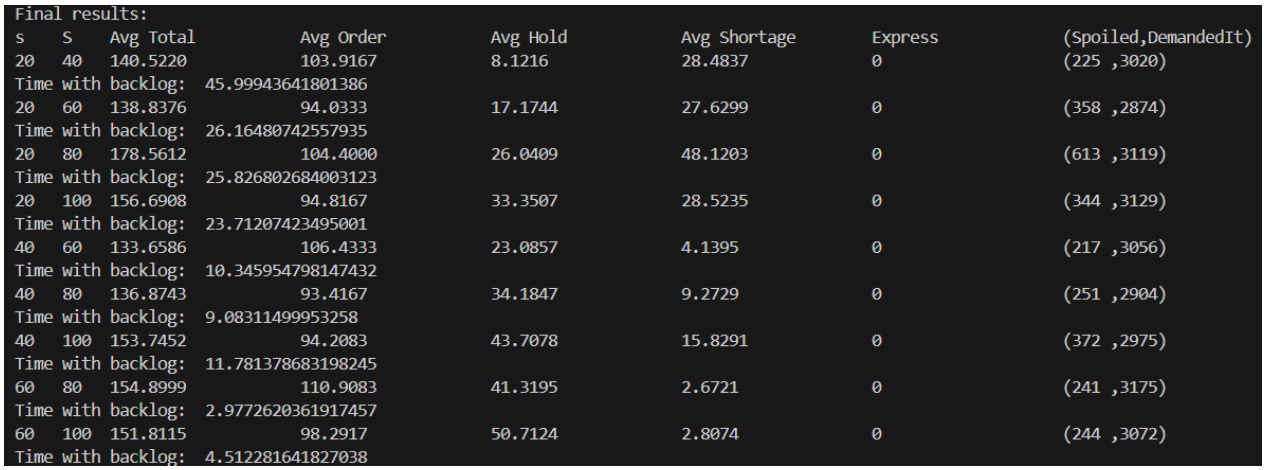
Holding costs tend to increase with higher order-up-to levels (S), as policies maintaining higher inventory levels (e.g., (20, 100) and (60, 100)) incur greater holding costs due to the larger quantity of inventory held in stock, leading to increased storage and therefore holding costs. On the other hand, shortage costs tend to decrease as the inventory system becomes more resilient (as we go up in the policy levels). Policies with higher reorder points (S) and safety stock levels (s) translate into a lower probability of stockouts, which are the primary driver of shortage costs, which is also shown on the decreasing value of “Time with backlog”. Obviously, and just to reinforce the idea, as it’s harder to go into stockouts, the average holding costs increases.

2.3.1 Question 1 - Is express ordering worth it?

To answer this question, we set up a **seed** so that we could make a run considering express orders, and another run (**for the same seed**) with just normal orders.
And the results are as follows:

```
Final results:
s    S    Avg Total          Avg Order          Avg Hold          Avg Shortage       Express    (Spoiled,DemandedIt)
20   40   145.2158           117.2333           9.9636            18.0188            28         (196 ,3023)
Time with backlog:  40.700188046699125
20   60   131.7987           100.2667           20.4535           11.0785            18         (171 ,2832)
Time with backlog:  22.636175352247154
20   80   144.7460           104.0167           27.0053           13.7241            12         (226 ,3112)
Time with backlog:  20.72554488594868
20   100  151.6225           102.6333           38.8676           10.1216            13         (230 ,2938)
Time with backlog:  13.807726348061948
40   60   136.9443           110.7667           22.5086           3.6690             0          (223 ,3162)
Time with backlog:  10.030137705486485
40   80   135.3143           97.8917            32.4096           5.0130             2          (199 ,3059)
Time with backlog:  7.623157955542814
40   100  145.9480           96.4917            43.4435           6.0128             3          (259 ,2987)
Time with backlog:  7.369385893741734
60   80   149.3460           106.2667           42.1675           0.9118             0          (175 ,3111)
Time with backlog:  3.042571469378108
60   100  152.4023           98.6917            49.9929           3.7177             0          (258 ,3054)
Time with backlog:  3.627444407125923
```

Figure 1 - Results with express orders



```
Final results:
s    S    Avg Total          Avg Order          Avg Hold          Avg Shortage       Express    (Spoiled,DemandedIt)
20   40   140.5220           103.9167           8.1216            28.4837            0          (225 ,3020)
Time with backlog:  45.99943641801386
20   60   138.8376           94.0333            17.1744           27.6299            0          (358 ,2874)
Time with backlog:  26.16480742557935
20   80   178.5612           104.4000           26.0409           48.1203            0          (613 ,3119)
Time with backlog:  25.826802684003123
20   100  156.6908           94.8167            33.3507           28.5235            0          (344 ,3129)
Time with backlog:  23.71207423495001
40   60   133.6586           106.4333           23.0857           4.1395             0          (217 ,3056)
Time with backlog:  10.345954798147432
40   80   136.8743           93.4167            34.1847           9.2729             0          (251 ,2904)
Time with backlog:  9.08311499953258
40   100  153.7452           94.2083            43.7078           15.8291            0          (372 ,2975)
Time with backlog:  11.781378683198245
60   80   154.8999           110.9083           41.3195           2.6721             0          (241 ,3175)
Time with backlog:  2.9772620361917457
60   100  151.8115           98.2917            50.7124           2.8074             0          (244 ,3072)
Time with backlog:  4.512281641827038
```

Figure 2 - Results without express orders

From both images presented above (Figure 1 and Figure 2), we can see that for most of the policies the average total cost (that is the sum of the ordering cost, the holding cost and the shortage cost) is **lower** when using express orders, than when not using. We can also see that the shortage cost is bigger when not express ordering, which means that the express orders did exactly what they were supposed to do. We can also verify that the number of spoiled items is lower when using express ordering.

To validate all this, we summed up all the values of the “Avg Total” column for both experiments and got:

Without express ordering ($)	**With** express ordering ($)
~1293.3378	~1345.6009

So, we can conclude that express ordering **is** in fact **worth it**, obviously there are some policies that do not corroborate that result, for example and following the notation (s, S): (20, 40), (40, 80), (60, 80); but in general, it is worth it.

2.3.2 Question 2 - Compute the proportion of items taken out of the inventory that are discarded due to being spoiled.

To compute the proportion of items taken out of the inventory that are discarded due to being spoiled, we need to consider 2 variables:

1. number of items spoiled.
2. number of items demanded (sum of the number of all demanded items)

The number of the items taken out of the inventory is:

$$\boldsymbol{items\ taken\ out\ =\ number\ of\ spoiled\ items\ +\ number\ of\ items\ demanded}$$

Then, to calculate the proportion we do:

$$\boldsymbol{proportion\ =\ \frac{number\ of\ spoiled\ items}{items\ taken\ out}}$$

Policy	Proportion	Proportion%
(20, 40)	0.0609	6.09%
(20, 60)	0.0569	5.69%
(20, 80)	0.0677	6.77%
(20, 100)	0.0726	7.26%
(40, 60)	0.0659	6.59%
(40, 80)	0.0611	6.11%
(40, 100)	0.0798	7.98%
(60, 80)	0.0533	5.33%
(60, 100)	0.0779	7.79%

Note that these results used 103541 as seed.

3 Problem 2 - Kermack-McKendrick Model

3.1 Brief Description

The Kermack-McKendrick model, also known as the SIR Model, is a hypothesis that attempts to predict the impact and evolution of an infectious disease on a given population. For this purpose, the population is divided into 3 groups: the susceptible population (s), who can be infected, the infected population (i), who are currently infected and can transmit the disease to susceptible individuals, and the recovered population, who have been infected, acquired immunity, and can no longer contract the disease (r).

The differential equations that govern the model are:

- $\frac{ds(t)}{dt} = -\beta \times s(t) \times i(t)$
- $\frac{di(t)}{dt} = \beta \times s(t) \times i(t) - k \times i(t)$
- $\frac{dr(t)}{dt} = k \times i(t)$

The constants present in the formulas β and k correspond, namely, to the infection rate and the recovery rate associated with the disease evolution process.

With the aim of simulating the evolution of a disease, two simulation techniques, the Forward Euler and the Runge-Kutta, were implemented within the structure of components: initialize, observe, and update. To initiate each simulation, it will be necessary to define the values of each population category at time 0, the values of the constants previously mentioned, the time interval between simulation iterations, the maximum simulation time and the simulation method.

While the Forward Euler method is given by:

$$dx/dt = G(x):$$

$$x(t + \Delta t) = x(t) + G(x(t))\Delta t$$

the fourth order Runge-Kutta (RK4) method is given by:

$$\frac{dx}{dt} = F(x,t) \qquad \mathrm{x}(t+\Delta t) = x(t) + \frac{K_1 + 2.K_2 + 2.K_3 + K_4}{6}$$

Where:

$$\mathrm{K_1} = \Delta \mathrm{t}.\,\mathrm{F}(\mathrm{x},\mathrm{t})$$

$$\mathrm{K_2} = \Delta \mathrm{t}.\,\mathrm{F}\left(\mathrm{x} + \frac{\mathrm{K_1}}{2}, \mathrm{t} + \frac{\Delta \mathrm{t}}{2}\right)$$

$$\mathrm{K_3} = \Delta \mathrm{t}.\,\mathrm{F}\left(\mathrm{x} + \frac{\mathrm{K_2}}{2}, \mathrm{t} + \frac{\Delta \mathrm{t}}{2}\right)$$

$$\mathrm{K_4} = \Delta \mathrm{t}.\,\mathrm{F}(\mathrm{x} + \mathrm{K_3}, \mathrm{t} + \Delta \mathrm{t})$$

3.2 Implementation

As mentioned, to simulate the evolution of the disease, it is necessary to define a set of initial values, constants, and choose the simulation method.

```
5   euler_kutta = 1      # 1 for Euler, 0 for Runge-Kutta
6
7   S0 = 0.8             # fraction of the population that is susceptibe
8   I0 = 0.1             # fraction of the population that is infected
9   R0 = 0.1             # fraction of the population that is recovered
10  BETA = 0.5           # infection rate
11  K = 0.1              # recovery rate
12  DELTA_T = 0.1        # time step
13  T_FINAL = 100        # final time
```

From line 5 to 13, it will be necessary to define the simulation method, with 1 for the Forward Euler and 0 for the Runge-Kutta, the values of s, i, and r for time 0, represented by S0, I0, and R0, the value of β should be defined in the constant BETA, and the value of k in the constant K, the time interval between iterations defined by DELTA_T, and the total simulation time defined by T_FINAL, assuming that all simulations start at time 0.

From lines 15 to 22, the functions that implement the calculation of each of the differential equations for a given time t have been defined.

```
15  def ds_dt(s, i):
16      return -BETA * s * i
17
18  def di_dt(s, i):
19      return BETA * s * i - K * i
20
21  def dr_dt(i):
22      return K * i
```

In the initialize function, we assign each of the initial values to the respective result list and initialize a list of instants at which each iteration of the simulation occurs. In the observe function, we monitor the system's state by adding each current state to the respective result list initialized in the previously described function, as well as storing the instant at which these states occurred.

```
24  def initialize():
25      global s0, i0, r0, results_s, results_i, results_r, t_list
26      s0 = S0
27      i0 = I0
28      r0 = R0
29
30      results_s = [s0]
31      results_i = [i0]
32      results_r = [r0]
33      t_list = [0]
34
35  def observe():
36      global s0, i0, r0, results_s, results_i, results_r, t_list
37      results_s.append(s0)
38      results_i.append(i0)
39      results_r.append(r0)
40      t_list.append(t_list[-1] + DELTA_T)
```

Since we are considering two possible simulation methods, the update function will have two possible implementations depending on the flag chosen at the beginning of the program. For either case, depending on the chosen method, the calculation of the values for the next iteration is implemented, just as described in the formulas stated earlier. The only difference regarding the formula presented is in the Runge-Kutta method, where for simplification purposes commonly used, the value of t in $F(x,t)$ is ignored, and all Ks are calculated for the same t.

```
42  def update():
43      global s0, i0, r0, euler_kutta
44
45      if euler_kutta:          # Euler Forward
46          s0 += ds_dt(s0, i0) * DELTA_T
47          i0 += di_dt(s0, i0) * DELTA_T
48          r0 += dr_dt(i0) * DELTA_T
49      else:                    # Runge-Kutta
50          k1_s = ds_dt(s0, i0) * DELTA_T
51          k1_i = di_dt(s0, i0) * DELTA_T
52          k1_r = dr_dt(i0) * DELTA_T
53
54          k2_s = ds_dt(s0 + k1_s / 2, i0 + k1_i / 2) * DELTA_T
55          k2_i = di_dt(s0 + k1_s / 2, i0 + k1_i / 2) * DELTA_T
56          k2_r = dr_dt(i0 + k1_i / 2) * DELTA_T
57
58          k3_s = ds_dt(s0 + k2_s / 2, i0 + k2_i / 2) * DELTA_T
59          k3_i = di_dt(s0 + k2_s / 2, i0 + k2_i / 2) * DELTA_T
60          k3_r = dr_dt(i0 + k2_i / 2) * DELTA_T
61
62          k4_s = ds_dt(s0 + k3_s, i0 + k3_i) * DELTA_T
63          k4_i = di_dt(s0 + k3_s, i0 + k3_i) * DELTA_T
64          k4_r = dr_dt(i0 + k3_i) * DELTA_T
65
66          s0 += (k1_s + 2 * k2_s + 2 * k3_s + k4_s) / 6
67          i0 += (k1_i + 2 * k2_i + 2 * k3_i + k4_i) / 6
68          r0 += (k1_r + 2 * k2_r + 2 * k3_r + k4_r) / 6
```

To execute the simulation from start to finish, we initialize the system once and then execute the update and observe operations together, in this order, once at each instant until the number of instants equals the integer division of T_FINAL (final instant/total simulation time) by DELTA_T (step value).

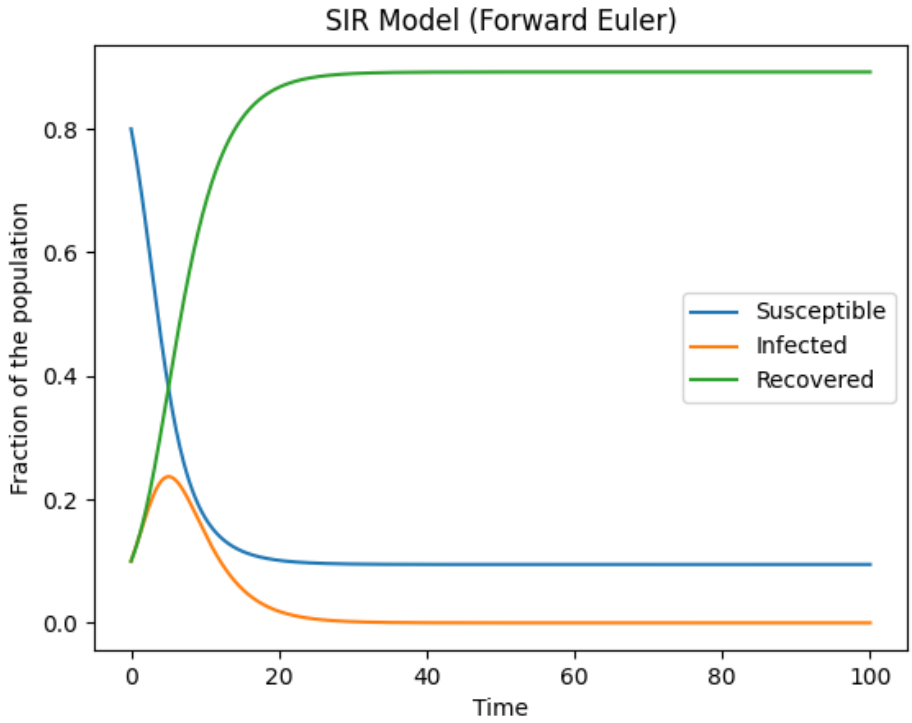
```
70  def main():
71
72      initialize()
73      for _ in range(0, int(T_FINAL/DELTA_T)):
74          update()
75          observe()
```

3.3 Results and Questions

3.3.1 Question 1 – Trace the evolution using Forward Euler method.

To carry out the task requested in question 2.1, the code already presented in the observe function was implemented, more precisely what is found between lines 46 and 48. For the input values presented below, the graph depicting the evolution of the simulation for the population sets was obtained.

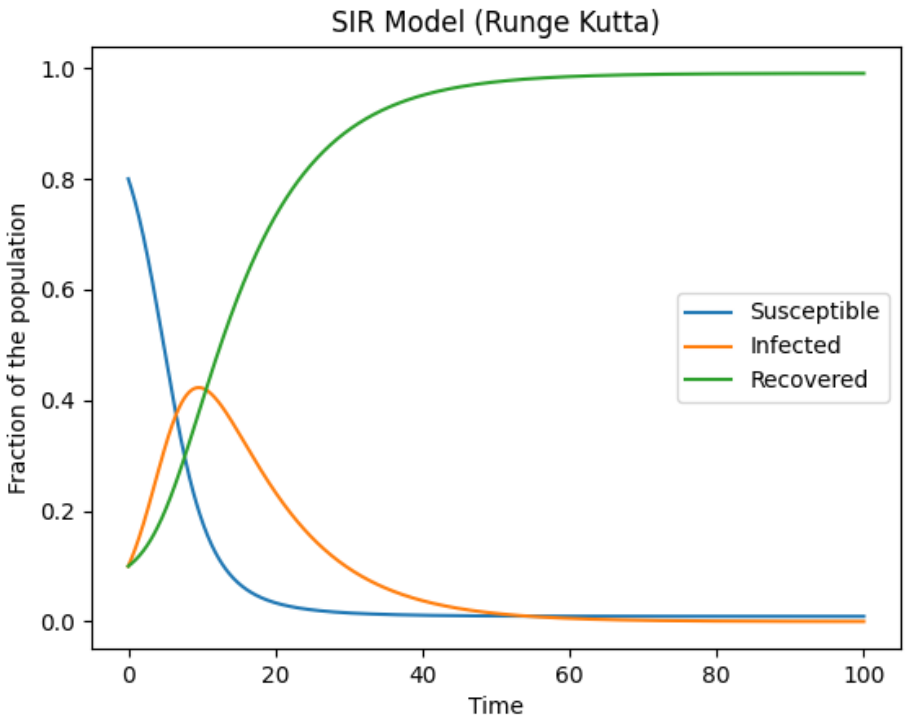
```
S0 = 0.8
I0 = 0.1
R0 = 0.1
BETA = 0.8
K = 0.3
DELTA_T = 0.1
T_FINAL = 100
```



3.3.2 Question 2 – Trace the evolution using Runge Kutta method.

To carry out the task requested in question 2.2, the code already presented in the observe function was implemented, more precisely what is found between lines 50 and 68. For the input values presented below, the graph depicting the evolution of the simulation for the population sets was obtained.

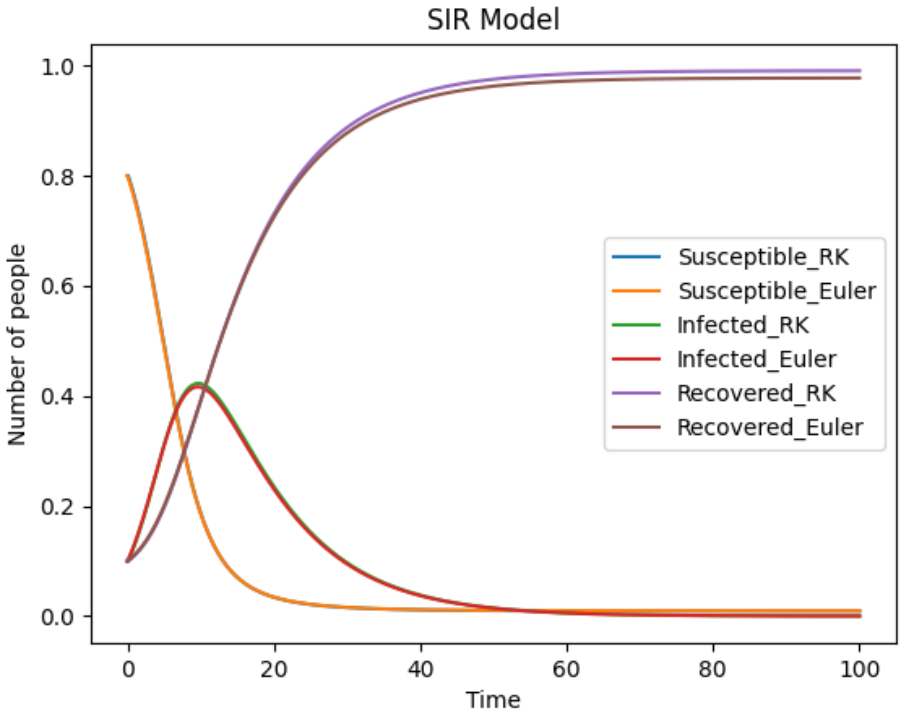
```
S0 = 0.8
I0 = 0.1
R0 = 0.1
BETA = 0.5
K = 0.1
DELTA_T = 0.1
T_FINAL = 100
```



3.3.3 Question 3 – Compare the precision of both approaches.

For the input values presented below, the graph depicting the evolution of the simulation for the population sets was obtained for both methods.

```
S0 = 0.8
I0 = 0.1
R0 = 0.1
BETA = 0.5
K = 0.1
DELTA_T = 0.1
T_FINAL = 100
```



Unfortunately, the Kermack-McKendrick model does not have an analytical solution from which we can derive results to compare with those obtained through simulation and thus calculate the accuracy of the simulated results. However, we can affirm that in general, the Runge Kutta method is more accurate, stable, and less prone to error propagation than the Forward Euler method. This is due to the simplification that the Forward Euler method makes in the calculation of the derivative, which contrasts with the way Runge Kutta considers intermediate steps that allow smoothing estimates of the derivative and reducing local error. The cancellation of these errors will allow the result of the Runge Kutta method to also have a lower global error, thus providing greater accuracy to the result obtained by it than that obtained by the Forward Euler method.

We could also consider as real values those simulated with a much smaller Δt. However, we would still be using simulation methods subject to error propagation which, no matter how minimized, would not guarantee complete fidelity of the data to the point of being able to estimate the accuracy of other methods based on the results of this approach.

4 References and resources

[1] A. M. Law, Simulation modeling and analysis (Vol. 5), New York: McGraw Hill LLC, 2014.

[2] N. Lau e A. Sousa, “Simulation Mini-Projects,” [Online]. Available: https://elearning.ua.pt/pluginfile.php/1824265/mod_resource/content/4/simopt2324_SimProject.pdf. [Acedido em May 2024].

[3] N. Lau e A. Sousa, “12/03/2024 - Queuing Systems; Discrete-Time and Continuous-Time Simulation,” [Online]. Available: https://elearning.ua.pt/pluginfile.php/1577257/mod_resource/content/4/simopt_2324_0312_DiscreteContinuous.pdf. [Acedido em May 2024].

[4] D. I. Lanlege, R. Kehinde, D. . A. Sobanke, A. Abdulganiyu e U. M. Garba, “Comparison of Euler and Range-Kutta methods in solving ordinary differential equations of order two and four,” [Online]. Available: http://ljs.academicdirect.org/A32/010_037.htm. [Acedido em May 2024].