Simulation and Optimization: Simulation Mini-Projects

João Clemente
DETI
University of Aveiro
Aveiro, Portugal
jclemente@ua.pt

***Abstract*—This report presents a comprehensive analysis of two simulation-based projects. The first part consists of simulating an inventory system to evaluate the effectiveness of different inventory policies. Goals include estimating the average total cost per month, the proportion of time with backlog, and the frequency of express orders. The second part of the assignment focuses on the implementation of the SIR model using both the Forward Euler and Runge-Kutta 4th order methods to simulate the spread of an infectious disease within a population. Key performance metrics and comparative analysis are provided for both tasks, highlighting the cost-effectiveness and accuracy of the simulated models.**

***Index Terms*—Simulation, Inventory System, Kermack-McKendrick model, Forward Euler Method, Runge Kutta Method**

Introduction

This report addresses two different simulation problems, in order to understand its relevance and applicability to real-world scenarios.

The first part simulates an inventory system for a company selling a single product, whose objective is to evaluate the performance of nine different inventory policies over a 120-month period. Simulation aims to estimate the average total cost per month, the proportion of time with backlog, and the number of express orders, to determine the cost-effectiveness of these policies.

On the second part of the assignment, we focus on an epidemiological model, specifically the Kermack-McKendrick SIR model, whose goal is to simulate the spread of an infectious disease within a population using two numerical methods: the Forward Euler method and the Runge-Kutta 4th order method. The simulation considers initial values and key parameters to evaluate the accuracy and computational efficiency of both methods.

Both tasks use python for implementation, demonstrating practical applications of simulation in different contexts. This report details the methodology, implementation, and analysis of results, providing a comprehensive understanding of the effectiveness of different strategies in inventory management and disease modeling.

I. Inventory System

This section presents the implementation of an inventory system simulation from a company, designed to evaluate the effectiveness of different inventory policies.

Problem Description

The company sells a single product and aims to evaluate nine different inventory (s, S) policies through a 120-month simulation. The primary goals of the simulation are to estimate:

- The expected average total cost per month.
- The expected proportion of time that there is a backlog $(I(t) < 0)$.
- The expected number of express orders placed.
- Whether express ordering is cost-effective.

Inventory Policies

The simulation tests various policies, where each policy is defined by two parameters: s (reorder point) and S (maximum inventory level). These parameters dictate when to place a new order to complete the stock. In the below table are represented the tested policies:

Reorder Point (s)	20	20	20	20	40	40	40	60	60
Maximum Level (S)	40	60	80	100	60	80	100	80	100

TABLE I
COMPARISON OF INVENTORY POLICIES

Ordering Mechanism

The company utilizes a stationary (s, S) ordering policy, where the reorder quantity Z is determined as follows:

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

This mechanism ensures that inventory levels are optimized according to the predefined thresholds.

Demand Satisfaction

- Demands are satisfied immediately if inventory levels (I) meet or exceed the demand.
- If inventory levels are below demand, the excess demand is backlogged and satisfied by future deliveries.

Cost Components

Each order incurs a fixed setup cost ($K = 32$) and an incremental cost ($i = 3$) per item ordered, leading to the total order cost formula:

$$\text{Cost} = K + i \cdot Z$$

This formula calculates the direct costs associated with replenishing the inventory, crucial for evaluating the cost-effectiveness of different policies.

In addition to the setup and incremental costs associated with ordering inventory, the company faces other costs during the operation of its inventory system, such as:

Handling Costs: incurred for inventory items that are not sold throughout the month. Specifically, the costs are:

- $h = 1$ per item per month for items held in a positive inventory.

Shortage Costs: When the inventory level is insufficient to meet the demand, shortage costs are incurred. These costs reflect the negative impact of not being able to immediately fulfill orders, including:

- $\pi = 5$ per item per month that the demand remains unmet.

Simulation Process and Execution

This subsection details the monthly simulation process, illustrating how inventory levels are checked, orders are placed, and goods are received. A step-by-step breakdown ensures clarity in understanding the operational mechanics of the simulation.

The simulation runs over 120 months and operates as follows:

1) At each time point, check if orders have arrived and update the inventory level accordingly.
2) If the effective inventory (inventory plus scheduled orders) falls below the reorder point s, place an order to increase inventory up to S. Express orders are placed if the inventory is negative, with faster delivery times.
3) Demands are generated with a size of D, independent and identically distributed (IID) random variable with the following distribution:

$$D = \begin{cases} 1 & \text{with probability } \frac{1}{6} \\ 2 & \text{with probability } \frac{1}{3} \\ 3 & \text{with probability } \frac{1}{3} \\ 4 & \text{with probability } \frac{1}{6} \end{cases}$$

at random intervals, also based on an IID exponential distribution with a mean of 0.1 month.

```
def exponential_demand_time(mean):
    return random.expovariate(1/mean)

demand_interval = sim_time +
    ↪ exponential_demand_time(0.1)
```

If inventory is insufficient, the shortage is recorded as a backlog.
4) Calculate holding costs for inventory and shortage costs for backlogged items at the end of each month and update total costs (holding costs + shortage costs + order costs(both normal and express)), as visible below:

```
def place_order():
    if effective_inventory < 0: #
        ↪ Express order conditions
        total_cost += 48 + 4 * needed
            ...
    elif effective_inventory < s: #
        ↪ Normal order conditions
        total_cost += setup_cost +
            ↪ incremental_cost * needed
            ...
...

def simulate(s, S):
...
    while sim_time < months:
        if int(sim_time * 10) % 10 ==
            ↪ 0: # Check if it's the end
            ↪  of a month

            current_holding_cost = max
                ↪ (0, inventory_level) *
                ↪ holding_cost
            total_holding_cost +=
                ↪ current_holding_cost

            current_shortage_cost =
                ↪ backorders *
                ↪ shortage_cost
            total_shortage_cost +=
                ↪ current_shortage_cost

            total_cost +=
                ↪ current_holding_cost +
                ↪ current_shortage_cost
```

Performance Metrics Calculation

Key performance indicators are calculated to evaluate the effectiveness of the inventory management system:

- **Average Total Cost per Month:** Total costs divided by the number of months in the simulation.

```
   average_cost_per_month = total_cost
       ↪  / months
```

- **Proportion of Time with Backlog:** Ratio of time points with backlogged demand to total time points, indicating supply reliability.

```
def simulate(s, S):
   months = 120 # Duration of
      ↪ simulation in months
   while sim_time < months:
      time_points.append(sim_time)
      if inventory_level < 0:
         backlog_time_points += 1
         ...
      sim_time += 1/10 # Increment
         ↪ time by approximately a
         ↪ 1/10 of month
      ...
   proportion_of_time_with_backlog = (
      ↪ backlog_time_points / len(
      ↪ time_points))*100
```

- **Number of Express Orders:** Total express orders placed throughout the simulation, reflecting the urgency and additional costs incurred.

```
def place_order():
   effective_inventory =
      ↪ inventory_level +
      ↪ scheduled_order_amount
   if sim_time >= next_order_month:
      if effective_inventory < 0 #
         ↪ Express order conditions
         needed = max(0, S +
            ↪ effective_inventory)
         if needed > 0:
            arrival_time = sim_time +
               ↪ random.uniform(0.25,
               ↪  0.5)
            express_orders += 1
            ...
   return express_orders
```

Analysis of Results

1) Average Total Cost per Month: In the figure 1, each line represents how the cost per month for the each policy performed for a total of ten simulations.

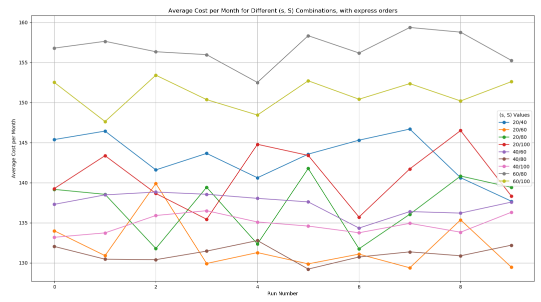

Fig. 1. Cost per Month for Different (s, S) Combinations, over 10 simulations

It is possible to compare that combinations with higher (s, S), such as $(60/80)$ and $(60/100)$ result in a higher average total cost per month, as also represented in the figure 2. This can be explained because ordering new inventory for these policies is more expensive, since mantaining the inventory between the desired thresholds requires a bigger load of items.

Also, policies with a higher S value, such as $(20/100)$, might show increased costs due to higher holding costs.

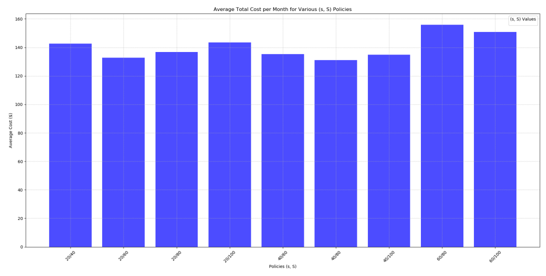

Fig. 2. The Average Cost per Month for each different (s, S)

2) Express Orders: Since express ordering happens only when the inventory level falls below 0, and ordering from the supplier can only happen in the beginning of the month, if the inventory level is slightly above s and this s is a low value, it leaves a greather chance of running out of inventory during the month and needing to place an express order in the following one.

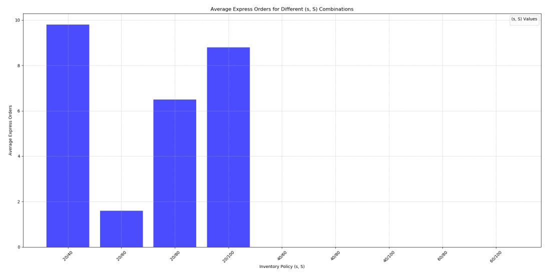

Fig. 3. Average Express Orders for Different (s, S) Combinations

Therefore, as plotted in the figure 3, the three policies that stay above the remaining are $(20/40)$, $(20/80)$, $(20/100)$.

This also justifies why the average cost of the $(20/40)$ and $(20/100)$ policies is greather in the 2 figure, since it requires express orders more often.

3) Proportion of Time with Backlog: The bar chart of figure 4, showing the proportion of time with backlog indicates the policies more effective at avoiding stockouts. Lower values in this metric are desirable as they indicate higher service levels. This represents all of the policies that have a s higher than 20.

4) Normal versus Express Orders: Cmparing express ordering with normal ordering results for the different policies, either for the proportion of time with backlog (figure 5), or the average cost per month (figure 6), is depicted below.

Express ordering is not worth it, since it has higher proportion of time with backlog and normal ordering has generally lower and more stable costs.

The fluctuations are less pronounced, indicating that normal ordering avoids the peaks of express ordering but might risk

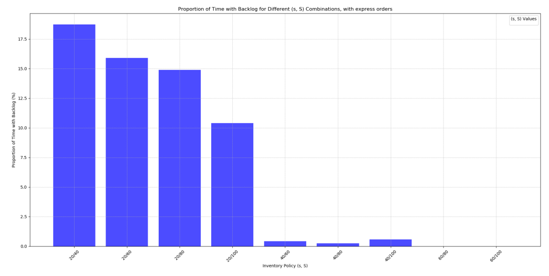
Fig. 4. Proportion of Time with Backlog for Different (s, S) Combinations, with express orders

more frequent low inventory situations without the buffer of quick replenishment.

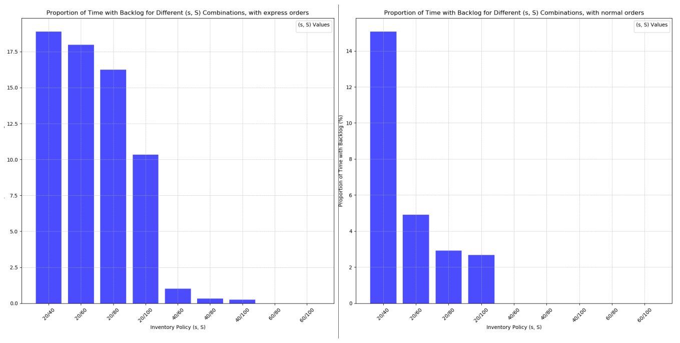
Fig. 5. Proportion of Time with Backlog for Different (s, S) Combinations, for express (left) and normal (right) ordering

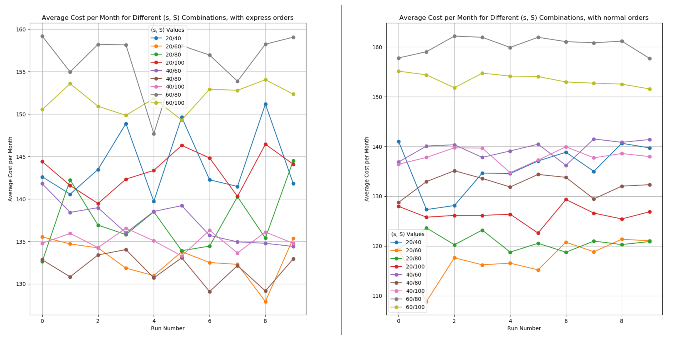
Fig. 6. Average Cost per Month for Different (s, S) Combinations, for express (left) and normal (right) ordering

Concluding remarks for 1 task

The extensive 10 simulations covering nine different (s, S) inventory policies over a 120-month period has provided valuable insights:

The policies with a higher reorder point (s) and a substantial maximum inventory level (S) generally demonstrated cost-effectiveness by balancing the trade-offs between holding costs and the costs associated with frequent reordering, as well as the importance of setting higher s values to mitigate the need for costly express restocking, also obtaining lower backlog percentages.

Normal Ordering, when paired with appropriate (s, S) configurations, provides a more cost-effective and predictable management strategy, when compared with express ordering.

II. SIR Model

Regarding the second task of the assignment, an adapted Kermack-McKendrick model, also called SIR Model, was implemented.

Model description

The SIR model allows to evaluate the evolution of an infectious disease in a population, by keeping track of different metrics, such as:

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

The differential equations that govern this SIR model are depicted below:

$$\frac{ds(t)}{dt} = -\beta s(t)i(t), \tag{1}$$

$$\frac{di(t)}{dt} = \beta s(t)i(t) - ki(t), \tag{2}$$

$$\frac{dr(t)}{dt} = ki(t). \tag{3}$$

Simulation Objectives: It was requested that two different simulation methods were implemented in order to model the evolution of SIR: the Forward Euler method and the Runge Kutta 4 order method.

Both methods should be given the initial values:

- $s(0)$: initial proportion of susceptible individuals in the population at the start of the simulation (time t=0)
- $i(0)$: initial proportion of infected individuals in the population at the start of the simulation (time t=0)
- $r(0)$: initial proportion of recovered individuals in the population at the start of the simulation (time t=0)

and crucial variables such as:

- β: infection/transmission rate
- k: recovery rate
- Δt: time step
- t_{final}: total simulation time

Both the initial values and parameters can be specified in the command line or in a file.

Implementation Strategy

1) Forward Euler method: is a first-order numerical procedure to solve ordinary differential equations. It approximates the derivatives by using a simple forward difference approach.

The following developed python code shows how it was implemented for the SIR model, according to the equations above:

```
for t in range(1, num_steps):
    ds_dt = -beta * s[t-1] * i[t-1]
    di_dt = beta * s[t-1] * i[t-1] - k * i[
        ↪ t-1]
    dr_dt = k * i[t-1]

    s[t] = s[t-1] + ds_dt * dt
    i[t] = i[t-1] + di_dt * dt
    r[t] = r[t-1] + dr_dt * dt
```

Since the Forward Euler method is a first-order method, the error per step is proportional to the time step (Δt), therefore, the total error is also proportional to Δt*number of steps and the bigger the number of time steps, the less accurate the method will be.

On the other hand, the method is computationally less demanding, requiring fewer calculations per time step, thus, being useful for quicker approximations or when computational resources are limited.

2) Runge-Kutta 4th Order method: is the most widely known member of the Runge–Kutta family, which are implicit and explicit iterative methods used in temporal discretization for aproximating solutions of simultaneous nonlinear equations.

It is a more computationally demanding process which improves accuracy by taking weighted average derivatives at different points within each time step.

Similarly to what was implemented in the previous method, the code for the Runge-Kutta 4th Order method can be seen below:

```
for t in range(1, num_steps):

    k1s = -beta * s[t-1] * i[t-1]
    k1i = beta * s[t-1] * i[t-1] - k * i[t
        ↪ -1]
    k1r = k * i[t-1]

    k2s = -beta * (s[t-1] + 0.5 * dt * k1s)
        ↪  * (i[t-1] + 0.5 * dt * k1i)
    k2i = beta * (s[t-1] + 0.5 * dt * k1s)
        ↪ * (i[t-1] + 0.5 * dt * k1i) - k *
        ↪  (i[t-1] + 0.5 * dt * k1i)
    k2r = k * (i[t-1] + 0.5 * dt * k1i)

    k3s = -beta * (s[t-1] + 0.5 * dt * k2s)
        ↪  * (i[t-1] + 0.5 * dt * k2i)
    k3i = beta * (s[t-1] + 0.5 * dt * k2s)
        ↪ * (i[t-1] + 0.5 * dt * k2i) - k *
        ↪  (i[t-1] + 0.5 * dt * k2i)
    k3r = k * (i[t-1] + 0.5 * dt * k2i)

    k4s = -beta * (s[t-1] + dt * k3s) * (i[
        ↪ t-1] + dt * k3i)
    k4i = beta * (s[t-1] + dt * k3s) * (i[t
        ↪ -1] + dt * k3i) - k * (i[t-1] +
        ↪ dt * k3i)
    k4r = k * (i[t-1] + dt * k3i)

    s[t] = s[t-1] + (1/6) * dt * (k1s + 2*
        ↪ k2s + 2*k3s + k4s)
    i[t] = i[t-1] + (1/6) * dt * (k1i + 2*
        ↪ k2i + 2*k3i + k4i)
    r[t] = r[t-1] + (1/6) * dt * (k1r + 2*
        ↪ k2r + 2*k3r + k4r)
```

Since it is a fourth-order method, the error per step is significantly lower, obtaining a much better accuracy for the same time step size, compared to the Forward Euler method.

However, this efficiency comes at the cost of a more demanding computational effort.

Results and Analysis

In this part, some analysis was done by tuning different parameters, in order to compare the precision of both approaches.

In the figure 7, the results obtained from the simulation were plotted. The initial conditions and parameters defined for the simulation were:

```
s0, i0, r0 = 0.99, 0.01, 0.0
beta, k = 0.5, 0.1
dt, tfinal = 0.1, 100
```

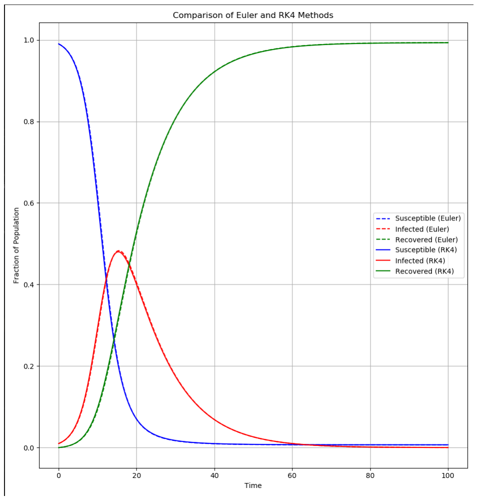

Fig. 7. Comparison of Euler and Runge-Kutta 4 Methods for the same parameters

Since it is difficult to recognize the differences between both approaches, it was considered that the RK4 method is more accurate, due to the larger complexity of the calculation, thus, plotting the relative error of the Euler method.

In the figure 8 it is possible to see that the relative error for the recovered population has a maximum of around 5% for time=9, and then, stagnates very close to 0 until the end of the simulation.

On the other hand, both the susceptible and infected have an early prime of 2.8% and 3.5%, respectively, lowering afterwards, and then again, both ending the simulation very close to the 4%.

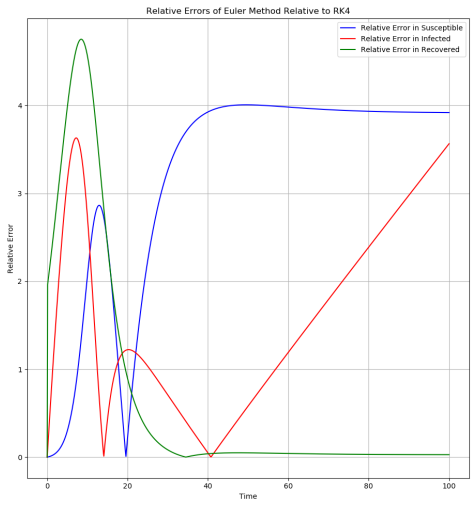

Fig. 8. Relative Error of Euler Method relative to Runge-Kutta 4

In the figure 9, can be seen how different values of β (transmission rate) influence the infected population. The below values were tested,

```
beta = [0.1, 0.3, 0.5]
```

and, despite all of them converging very close to 0, for β=0.5, both methods infect almost half of the population. Besides being the larget β that achieves a higher fraction of infected population, it also ends the simulation with the lowest value.

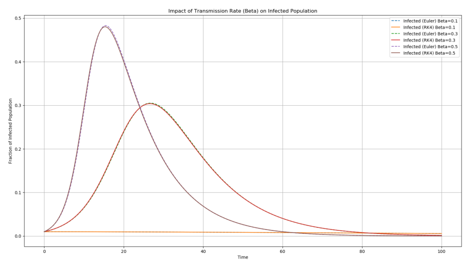

Fig. 9. Impact of Transmission Rate (beta) on Infected Population

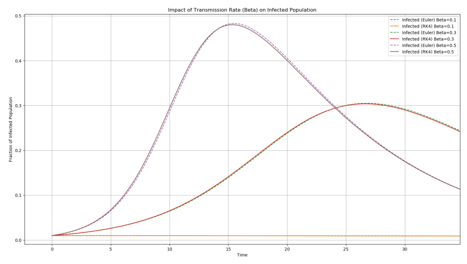

Fig. 10. Impact of Transmission Rate (beta) on Infected Population, for Euler and Runge-Kutta 4 Methods

In the figure 10 it is possible to see that both methods are very close, but it is important to note that the Euler method always gets to a higher value later than the Runge-Kutta method. Afterwards, both methods end the simulation superimposed (figure 9).

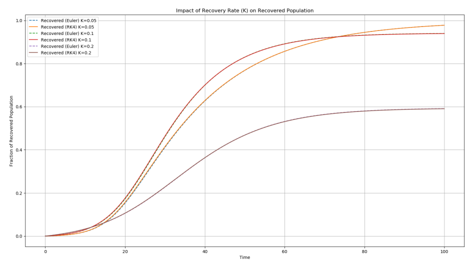

Fig. 11. Impact of Recovery Rate (K) on Recovered Population

For different values of k (recovery rate), the same behaviour is repeated, with both methods being very close to each other. It is possible to see, however, that the lower k curve achieves a higher value at the end of the simulation, as the figure 11 shows.

III. CONCLUSION

In conclusion, regarding the first task of the assignment, the best choice is adopting inventory policies that favor higher reorder points and broader inventory levels to reduce total operational costs and diminish the frequency of express orders, enhancing overall service reliability.

Although, the Runge-Kutta 4th Order model was considered the most precise one, with the trade-off of being more computationally demanding and resource consuming, both models' implementation was successful, and even with different parameter tuning, it was very difficult to detect differences between them.

Overall, this assignment allowed us to gain a better understanding of the importance of simulation by applying the concepts learned in class to real-world problems.