Simulation Mini Projects

Mestrado em Engenharia Informática

Universidade de Aveiro

DETI - Aveiro, Portugal

SO - Simulação e Otimização

Ricardo Antunes [98275], Gonçalo Silva [103668]

Index

1. Introduction

Our project aims to explore and apply simulation techniques in two distinct domains: inventory management and infectious disease modeling. Through the use of computational methods, we intend to investigate specific scenarios and address important questions in each of these contexts. We are going to approach the strategies followed for the resolution of the two simulation problems as well as the answers raised on the project statement.

This report will present the results obtained in both parts of the project, highlighting the conclusions reached and discussing the approaches taken for implementing the simulations. In the end, we hope to contribute to advancing knowledge in these areas and provide relevant insights for practical applications.

2. First Problem

2.1. Introduction

The first exercise of this project involves the simulation of an inventory system for a specific company, consisting of a single product and subject to variable demands over time. This exercise is based on a problem presented in the practical classes of the Simulation and Optimization course, aiming to make its implementation more robust and the system as close to reality as possible.

In the initial scenario, the company places orders to the supplier at the beginning of each month, taking into account the current inventory level. These orders incur associated costs and are essential to maintain adequate stock. Customer demands are met immediately if products are available in stock. Otherwise, backlogged items are processed as much as possible.

In addition to these aspects, two new variables have been introduced in the problem:

- Product shelf life: Each product in stock has a shelf life uniformly distributed between 1.5 and 2.5 months. This means that, after a certain period, the products become unsuitable for sale and are discarded.
- Express orders: If the inventory level reaches a negative state, indicating the presence of backlogged products, the company opts to place an express order to the supplier. Although more expensive, the delivery is faster.

The simulation was run for 120 months, considering various inventory policies. Monthly average costs associated with supplier orders, inventory holding, and shortages were calculated, as well as the average total cost per month. Additionally, we determined the average proportion of time the company faces product shortages and the number of express orders placed. Subsequently, we will analyze whether the use of express orders is advantageous for the company or not.

2.2. Methodology

To solve the inventory system problem, we adopted an approach similar to the ones presented in the course. Initially, we defined the necessary constants for the problem, such as the incremental cost, setup cost, and maximum simulation time. Then, we initialized the system's state variables, including the inventory level, the time of the last event, and the number of items to be ordered from the supplier, along with statistical counters such as the total cost of the company, the total cost of orders placed with the supplier, handling cost, shortage cost, the number of spoiled items, the number of express orders, the total number of orders placed with the supplier, and the time the company faces product shortages. These metrics are important for obtaining results because through careful analysis, we can develop new strategies to achieve better results for the company.

In the next step, we define the events that can occur throughout the simulation:

- Customer demand (routine customer_demand)
- Inventory evaluation (routine evaluate_inventory)
- Arrival of an order (routine supplier_arrival)
- End of simulation

and added the inventory evaluation events, customer demand, and simulation end to the event list, initiating the system simulation. The simulation begins by calculating the next event with the smallest time and updating metrics such as shortage cost, handling cost, and proportion of time that there is a backlog. Then, the routine associated with that event is executed.

1. Customer_demand

This routine is responsible for simulating customer demand. Firstly, we determine the time until the next customer demand event and generate the number of items for the current demand. Subsequently, the validity of the products in inventory is checked against the current demand. For each product, it is verified if there are valid products in inventory, i.e., those that are not spoiled. If there are, the demand unit is fulfilled, and the corresponding product is removed from inventory. If the product is spoiled, it is removed from inventory and counted as a spoiled product. Finally, the inventory level is updated according to the fulfilled demand and spoiled products.

2. Evaluate_inventory

The evaluate_inventory routine is responsible for assessing the current inventory and determining if it is necessary to place an order with the supplier. To do this, we compare the current inventory level with the minimum value of the inventory policy being used. If it is lower, the number of items to be ordered is calculated, and then it is checked whether the inventory has available products (inventory level >= 0) or if there are products on backorder (inventory level < 0). This check is important to distinguish between an urgent order or not. If the inventory level is less than 0, an express order is placed, where the order cost is calculated and the delivery time is determined, which is shorter but at a higher cost to the company compared to a normal delivery. On the other hand, if the inventory level is positive, the company places a normal order, which has a lower cost and a longer delivery time. After

evaluating the inventory, the next assessment is scheduled for the beginning of the next month.

3. Supplier_arrival

The supplier_arrival routine is responsible for simulating the arrival of an order from the supplier. When an order arrives, the products are added to the inventory along with their expiration dates. These expiration dates are randomly generated following a uniform distribution between 1.5 to 2.5 months.

The simulation loop will continue until the next event is the end of simulation. At the end of each iteration of the loop, the following estimates are calculated: average total cost per month, average monthly costs associated with orders to the supplier, average monthly costs of inventory holding and shortages, the average proportion of time that there were backlogged items, and the number of express orders placed. These metrics are essential for evaluating the company's performance under different inventory policies, providing valuable insights that can assist in future optimization decisions.

2.3. Results

Table 1 presents the results obtained from the simulation with the implementation of express orders.

Policy	Average Total cost	Average Ordering cost	Average Handling cost	Average Shortage cost	Backlog time	Express orders
(20,40)	185.71	155.23	8.36	22.11	36.06	21
(20,60)	342.61	266.16	11.62	64.83	48.09	41
(20,80)	400.83	306.80	16.14	77.89	44.80	36
(20,100)	382.87	309.70	21.74	51.43	38.93	32
(40,60)	182.58	150.18	19.10	13.30	17.56	9
(40,80)	358.28	282.92	19.95	55.42	38.05	29
(40,100)	526.30	386.91	25.92	113.47	39.57	36
(60,80)	163.79	123.75	38.87	1.17	2.74	1
(60,100)	195.35	147.32	40.96	7.06	8.37	3

Table 1: Simulation with express orders.

The table 2 presents the results obtained from the simulation without the implementation of express orders.

Policy	Average Total cost	Average Ordering cost	Average Handling cost	Average Shortage cost	Backlog time
(20,40)	165.08	119.94	6.66	38.48	48.41
(20,60)	196.22	138.20	10.95	47.08	45.29
(20,80)	340.68	192.97	13.67	134.04	54.58
(20,100)	422.28	227.63	21.67	172.98	47.75
(40,60)	142.97	117.40	22.80	2.77	7.74
(40,80)	173.44	132.59	24.93	15.92	20.73
(40,100)	261.02	175.79	26.84	58.39	33.94
(60,80)	173.06	135.42	34.43	3,22	6.16
(60,100)	188.62	143.59	39.35	5.68	7.53

Table 2: Simulation without express orders.

The simulation results reveal that the implementation of express orders has a significant impact on the costs and performance of the inventory system. When conducting a comparative analysis between Tables 1 and 2, it can be observed that for inventory policies with lower minimum product numbers ((20,40) to (20,100)), the use of express orders is important. Conversely, for inventory policies with higher minimum product numbers ((40,60), etc.), the use of express orders is not justified.

For policies with a lower minimum number of products, the simulation without express orders shows a significant increase in the values associated with the proportion of time in backlog and the average monthly shortage cost compared to the other simulation. Thus, these results suggest that despite the lower average monthly ordering cost, the use of express orders in stock policies with a lower minimum number of products is crucial for minimizing backlog time and ensuring quick and efficient delivery to customers. The absence of this type of order results in longer delivery delays, which can lead to customer dissatisfaction and potentially jeopardize future orders.

On the other hand, for policies with a higher minimum number of products, the results indicate that the costs associated with express orders are not justified. In these situations, all costs (except handling cost) and backlog time are lower when express orders are not used. This suggests that for these policies, the use of regular orders is the most appropriate and cost-effective option.

3. Second Problem

3.1. Introduction

The adapted Kermack-McKendrick model, also known as the SIR Model, is used to describe the spread of an infectious disease in a population. This model is governed by the use of 3 differential equations, which depend on parameters that will be explained later.

The second problem aims to observe the evolution of the spread of an infectious disease using two variations of the Kermack-McKendrick model, namely the Euler method and the Runge-Kutta method.

3.2. Methodology

To explain further, the SIR model separates the population into 3 states:

1. **Susceptible (s(t)):** represents the fraction of the population that is susceptible to the disease and can become infected.

2. **Infectious (i(t)):** represents the fraction of the population that is infected and can transmit the disease to susceptible individuals.

3. **Recovered (R):** represents the fraction of population that is recovered, this is no longer infectious.

As previously mentioned, this model is governed by 3 equations that help to simulate the behavior and evolution of the disease, namely:

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

1. Input Arguments

The program accepts several input arguments, which are parsed using the argparse module. These arguments allow users to customize the simulation parameters:

- **s0**: The initial proportion of the population that is susceptible to the disease.
- **i0**: The initial proportion of the population that is infected.

- **r0**: The initial proportion of the population that has recovered from the disease.
- **beta**: The infection rate, which determines how quickly the disease spreads among susceptible individuals.
- **k**: The recovery rate, which determines how quickly infected individuals recover.
- **t**: The total time for the simulation.
- **dt**: The time step for the simulation, which affects the granularity of the simulation.
- **f**: An optional file path to a file containing the simulation parameters, allowing for more complex configurations without command-line arguments.
- **method:** parameter to specify which method is gonna be used, either “euler” or “runge-kutta”. If not specified it will run both methods in sequence.

2. How to run the Program

In the ***ex2***/ directory there is gonna be a results folder with the plot of the graphs and the following python programs: **main.py**, **euler_disease_population.py** and **runge_kutte_disease_population.py.**

In **euler_disease_population.py** and **runge_kutte_disease_population.py** it will be the implementation for a simulation using each method. In both of those files there is no need to use the **method** parameter.In the **main.py** both programs are combined. This file was made to be easier to compare the results and plot the graphs of the two methods. Only in this file can the **method** parameter be used, if it's run without it, it will run both methods. Examples of ways to run the code:

```
python .\main.py --f .\parameters.txt
python .\euler_disease_population.py --f .\parameters.txt
python .\runge_kutta_disease_population.py --f .\parameters.txt
```

3. Program Structure

The program is structured around three main functions: ***initialize***, ***update***, and ***observe***, which collectively simulate the disease's progression over time.

The ***initialize*** function sets up the initial condition for the simulation to start. It takes the initial proportion of susceptible (s0), infected (i0) and recovered (r0) individuals. The ***update*** function is the core of the simulation. It calculates the changes in the susceptible, infected, and recovered populations based on the infection rate (beta), recovery rate (k), and the current time step (dt). The ***observe*** function records the current state of the susceptible, infected, and recovered populations at each time step. These

values are appended to lists, which are later used to plot the simulation results.

```
def initialize(s0, i0, r0):
    global s, i, r, s_result, i_result, r_result
    s = s0
    i = i0
    r = r0
    s_result = [s]
    i_result = [i]
    r_result = [r]
```

Figure 1: Initialize function implementation

```
def update_euler(dt, beta, k):
    global s, i, r
    s += -beta * s * i * dt
    i += (beta * s * i - k * i) * dt
    r += k * i * dt
```

Figure 2: Update_euler function implementation

```
def k_calculator(s, i, beta, k, dt):
    k_s = (-beta * s * i) * dt
    k_i = (beta * s * i - k * i) * dt
    k_r = (k * i) * dt
    return k_s, k_i, k_r


def update_runge_kutta(dt, beta, k):
    global s, i, r

    K1_s, K1_i, K1_r = k_calculator(s, i, beta, k, dt)
    K2_s, K2_i, K2_r = k_calculator(s + 0.5 * K1_s, i + 0.5 * K1_i, beta, k, dt)
    K3_s, K3_i, K3_r = k_calculator(s + 0.5 * K2_s, i + 0.5 * K2_i, beta, k, dt)
    K4_s, K4_i, K4_r = k_calculator(s + K3_s, i + K3_i, beta, k, dt)

    s += (K1_s + 2 * K2_s + 2 * K3_s + K4_s) / 6
    i += (K1_i + 2 * K2_i + 2 * K3_i + K4_i) / 6
    r += (K1_r + 2 * K2_r + 2 * K3_r + K4_r) / 6
```

Figure 3: Update_runge_kutta function implementation

```python
def observe():
    global s, i, r, s_result, i_result, r_result
    s_result.append(s)
    i_result.append(i)
    r_result.append(r)
```

Figure 4: Observe function implementation

3.3. Results

Before analyzing the results, we need to know some information about both methods and what we will concretely evaluate.
Both methods are used to solve ordinary differential equations, which are equations that involve derivatives of a function with respect to a single variable.

There are multiple differences between the methods, across multiple factors like, accuracy, complexity and the way they approximate solutions, but i want to cover more the accuracy and the respective error associated.

The euler method is the simplest method, it approximates the solution by taking small steps in the direction of the derivative at the current point. Being that the euler method is a first order method, it means that the error is proportional to the square of the step size.

The Runge-Kutta method, specifically the fourth-order Runge-Kutta method (RK4), is more accurate than the Euler method. It is a fourth-order method, meaning its error is proportional to the fourth power of the step size. This results in a much smaller error for the same step size compared to the Euler method.

Since we don't have real/correct values it's not possible to evaluate the precision of the methods. So one approach we can make is to consider the Runge-Kutta values as the "true values" and evaluate the error of the euler method.

All the graphs presented in the report are generated by using the following parameters: s0 = 0.80, i0 = 0.10, r0 = 0.10, beta = 0.5, k = 0.1, dt = 0.1, tfinal = 100

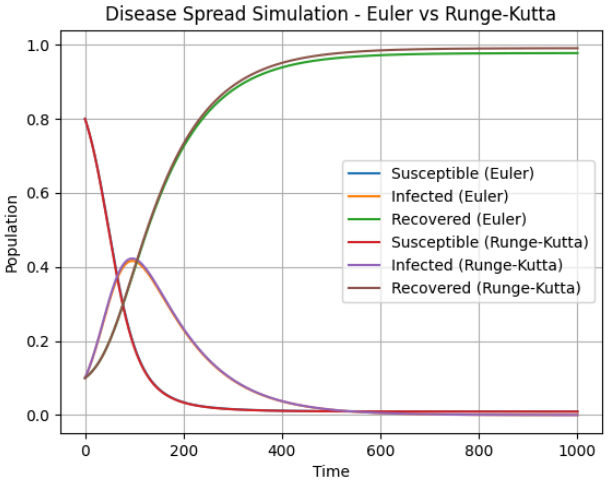

Figure 5: Disease Spread Simulation - Euler vs Runge-Kutta

By looking at the figure above, we can see that the values of the euler method and the runge-kutta method aren't the same during all of the simulation time, this difference being caused by the better accuracy of the runge-kutta method.

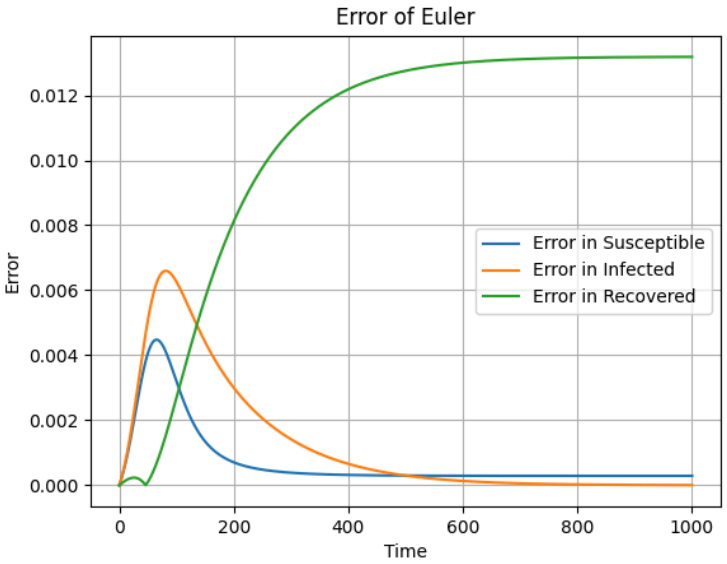

Figure 6: Error of Euler

When looking at the error graph, we can conclude that it aligns with the simulation graph. For instance, in the Recovered state, the largest discrepancy in values between the Euler method and the Runge-Kutta method is noticeable, and this can now be seen in the error graph where the error is seen increasing over time.

4. Conclusion

The analysis of real-world scenarios, as is the case with the problems presented in this report, through the implementation of simulation programs based on the theoretical and practical concepts learned in the simulation and optimization unit of the curriculum, proves to be highly beneficial for understanding how certain aspects of the world works and how to think more critically of the obtained results.