Simulation of an Inventory System and Infection Simulation with the SIR Model

Simulação e Otimização
Departamento de Electrónica, Telecomunicações e Informática, Universidade de Aveiro.

April 2024

1 Inventory System Problem

In a competitive and dynamic business environment, efficient inventory management is crucial for ensuring customer satisfaction, reducing costs, and optimizing operations. A particular challenge in this area is balancing the need to maintain sufficient inventory to meet demand against minimizing the risk of excess stock or product expiration. This is especially critical in sectors like food or pharmaceuticals, where products have a limited shelf life.
Inventory simulation offers a powerful tool for navigating this complex decision-making landscape. Employing an event-based model, we can explore the impact of different reorder policies, characterized by varying reorder point.
Our simulation proceeds through a series of key events: customer demand, inventory evaluation, and order arrival. Demand events are modeled as a stochastic process, following an exponential distribution with a mean frequency, representing the uncertain and variable nature of market demand. Orders are placed based on needs detected during monthly evaluations and can be standard or express, depending on the inventory situation at the time of evaluation. The lead time for each order follows a uniform distribution, representing variations in delivery times from the supplier. An additional challenge is managing expired items [1]. Each item in inventory has a random shelf life, simulated through a uniform distribution. The simulation keeps track of expired items and removes them from inventory, a crucial process for avoiding the sale of unsuitable products and maintaining quality.
Analysis of the simulation results will allow us to draw conclusions about the feasibility and value of express ordering and develop recommendations for optimizing inventory management policies.

1.1 Base Problem

The basic problem simply asked to simulate the inventory system for nine policies shown in Figure 1

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

Figure 1: Policies

Where `s` is a parameter representing the inventory level that triggers a replenishment order. When the inventory level is less than or equal to `s`, a reorder signal is activated to purchase more goods. Conversely, `S` is a parameter indicating the target inventory level after restocking. The actual order placed will thus be `S - inventory_level`, ensuring that the inventory is adequately replenished. The results to be reported were:

- Total cost
- Ordering Cost
- Average Holding Cost
- Average Shortage Cost

The evaluation of the system has a duration set at 120 months. The development of the base code, reported in the file `base_simulation_inventory_system.py`, was developed following this approach:

- Definition of variables
- Definition of the main simulation loop 1
- Definition of the `timing()` method that returns the next event 2.
- Definition of the `demand()` event 3
- Definition of the `evaluate()` event 4

- Definition of the `arrival_of_order()` event 5
- Definition of the function that calculates statistics `update_statistical_counters()` 6

Algorithm 1 Main Simulation Loop

Initialize main variables (omitted)
$update \leftarrow$ False
while $time \leq month\_of\_evaluation$ **do**
    TIMING
    UPDATINGAREAS
    if $next\_event\_type =$ 'demand' **then**
        DEMAND
    else if $next\_event\_type =$ 'evaluate' **then**
        EVALUATE
    else if $next\_event\_type =$ 'arrival' **then**
        ARRIVALOFORDER
    end if
    if $next\_event\_type =$ 'end' **and not** $update$ **then**
        append UPDATESTATISTICALCOUNTERS **to** $results$
        $update \leftarrow$ True
        break
    end if
end while

The loop continues until the predetermined time is reached. During the loop, the current event type is checked, and based on this, different functions are invoked to handle the specific event. If the statistics have not been updated during an 'end' event, they are calculated and saved, and the loop is terminated.

Algorithm 2 Timing Function

procedure TIMING
    $min\_time\_next\_event \leftarrow \infty$
    $next\_event\_type \leftarrow$ ''
    $next\_event \leftarrow time\_next\_event[0]$
    $next\_event\_type \leftarrow next\_event[0]$
    $min\_time\_next\_event \leftarrow next\_event[1]$
    if $next\_event\_type =$ ' ' **then**
        exit
    end if
    $simulation\_time \leftarrow min\_time\_next\_event$
end procedure

The `timing()` function 2 is used to determine the next event to be simulated. It accesses the elements at position 0 because every time a new event is generated and added to the event list, the list is sorted such that the nearest event is always in the first position. This ensures that the simulation always processes events in chronological order, based on their scheduled times.

Algorithm 3 Demand Function

procedure DEMAND
    $random\_number \leftarrow$ RANDINT$(0, 1)$
    $demand\_size \leftarrow 0$
    if $random\_number < \frac{1}{6}$ **then**
        $demand\_size \leftarrow 1$
    else if $random\_number < \frac{1}{2}$ **then**
        $demand\_size \leftarrow 2$
    else if $random\_number < \frac{5}{6}$ **then**
        $demand\_size \leftarrow 3$
    else
        $demand\_size \leftarrow 4$
    end if
    $inventory\_level -= demand\_size$
    POP(time_next_event)
    APPEND(list_events, new_demand)
    SORT(list_events)
end procedure

The `demand()` function 3 simulates customer demand within an inventory system. It adjusts the inventory level based on a randomly generated demand size. After deciding the size of the demand the inventory level is decremented by that quantity.

Algorithm 4 Evaluate Function

procedure EVALUATE
    if $inventory\_level < s$ **then**
        $order\_size \leftarrow S - inventory\_level$
        $last\_order \leftarrow order\_size$
        $ordering\_cost \leftarrow ordering\_cost + K + i \cdot order\_size$
        APPEND(list_events, new_arrival)
    end if
    POP(time_next_event)
    APPEND(list_events, new_evaluation)
    SORT(list_events)
end procedure

The `evaluate()` function 4 is designed to assess the current inventory level and decide whether to place an order based on predefined thresholds. The function checks if the inventory level is below a certain threshold. If it is, an order needs to be placed to replenish the inventory up to another threshold (`S`).

Algorithm 5 Arrival of Order Function

procedure ARRIVALOFORDER
    $inventory\_level += last\_order$
    POP(time_next_event)
end procedure

The `arrival_of_order()` function 5 is responsible for handling the logistics and record-keeping associated with the arrival of a replenish-

ment order in an inventory system. And finally the `update_statistics()` function 6 is used to obtained the wanted results shown in table 1 at the end of the document.

Algorithm 6 Updating Statistics

```
1: procedure UPDATESTATISTICALCOUNTERS
2:     UPDATE(totat_cost)
3:     UPDATE(average_holding_cost)
4:     UPDATE(average_ordergin_cost)
5:     UPDATE(average_shortage_cost)
6: end procedure
```

1.2 Express Orders

The first modification to implement to the basic problem is the addition of express orders. That is, when the inventory level at the beginning of the month is less than 0, an express order must be placed. Implementing this small change was quite simple; in fact, an additional check was simply added to the `evaluate()` function as shown in code 7

Algorithm 7 Evaluate Function (Express Orders)

```
1: procedure EVALUATE
2:     if inventory_level < 0 then
3:         order_size ← S − inventory_level
4:         last_order ← order_size
5:         ordering_cost ← cost_express_order
6:         APPEND(list_events, new_arrival)
7:     end if
           ▷ ... The rest of the code is in the previous implementation
8: end procedure
```

1.3 Items with shelf life

The second modification to implement concerned the lifespan of objects. Objects could ”expire” and become unusable. This functionality required the implementation of two methods. The first one, `initialize_shelf_life()`, shown in algorithm 8, initializes the lifespan of objects according to the problem specifications whenever an order arrives. The second one, `update_shelf_life()`, is called whenever there is a `demand()` 9 event and updates the inventory level by removing expired objects. Only after removing expired objects, customer demand is attempted to be satisfied.

1.4 Other Statistics

Several updates were then made to calculate the following quantities:

Algorithm 8 Initialize Shelf Life Queue

```
1: procedure INITIALIZESHELFLIFE(order_size)
2:     for _ in range(order_size) do
3:         shelf_life ← RANDOMUNIFORM
4:         APPEND(shelf_life)
5:     end for
6: end procedure
```

Algorithm 9 Update Shelf Life

```
1: procedure UPDATESHELFLIFE
2:     while shelf_life_queue is not empty do
3:         time_of_life
4:         if time_of_life ≥ expiring_time then
5:             POP(shelf_life_queue)
6:             inventory_level − = 1
7:         end if
8:     end while
9: end procedure
```

- Expected Average total cost per month
- Proportion of backlog time $I(t) < 0$
- Number of express orders

The first value was calculated by dividing the total cost by the number of evaluation months. The number of express orders was obtained by incrementing a counter each time this type of order was placed. For calculating the backlog time, the total backlog time was computed and divided by the total evaluation time.

Algorithm 10 Backlog Time and Average Total Cost

```
1: procedure UPDATESTATISTICALCOUNTERS
2:                          ▷ ... All the previous update
3:     backlog_time ← tot_back_time/tot_month
4:     avg_mont_cost ← tot_cost/tot_month
5: end procedure
```

1.5 Proportion of spoiled items

This last modification simply required calculating the percentage of expired items compared to all items removed from inventory. Therefore, counters were simply added to keep track of expired items and total items removed. Then, in the `update_statistical_counter()` method, the calculation of the ratio between expired items and total items was added.
The first counter, used to count expired items, is updated every time an expired item is found in the `update_shelf_life()` method 12. The second counter keeps track of total items in the `demand()` method 13.

Algorithm 11 Updating Statistics (% Expired Items)

```
procedure UPDATESTATISTICALCOUNTERS
                                  ▷ ... All the previous update
    frac_expired ← expired/tot_obj
end procedure
```

Algorithm 12 Update Shelf Life

```
procedure COUNTER EXPIRED ITEMS
    while shelf_life_queue is not empty do
        time_of_life
        if time_of_life ≥ expiring_time then
                                  ▷ ... deleting expired object
            expired_items + = 1
        end if
    end while
end procedure
```

The value of `tot_obj` is determined by the size of the demand and the number of expired items (which are also removed from inventory). After all the changes in the base code the result we obtained are shown in the second table 2 also at the end of the document. Is also shown a third table 3 that does not contain the express order implemented. This table will be useful to understand if the express order are worth it.

1.6 Are Express Order Worth it ?

Based on the data provided in table 2 and table 3,it is possible to draw some conclusions regarding the use of express orders within the inventory management system.

- Total cost with express orders are significantly lower than those without express orders. For instance, for the policy (20,40), the total cost with express orders is €595.452 compared to €6803.234 without express orders. This trend is consistent across all policies, suggesting that express orders substantially reduce overall costs.
- The backlog, which indicates the percentage of time that inventory levels are below zero, is drastically lower in policies with express orders compared to those without. For example, in the policy (20,40), the backlog decreases from 95.306% to 11.660% with the introduction of express orders. This demonstrates the effectiveness of express orders in maintaining more stable inventory levels and reducing shortage periods.
- The number of express orders varies among different policies, but generally, the introduction of these orders seems to have a direct positive impact on reducing the backlog and overall costs.

Algorithm 13 Total Object Counter

```
procedure DEMAND
                                  ▷ ...generating demand size
    tot_obj = demand_size + expired
                                  ▷ .. rest of the code
end procedure
```

- The cost per month is significantly lower in policies with express orders. Moreover, the percentage of spoiled items (%Spoiled Items) remains relatively low and comparable between the two setups, indicating that express orders do not negatively impact the quality of inventory despite the increased frequency of orders.

From the analyzed data, it is evident that introducing express orders into the inventory management system offers considerable benefits. The main advantages are:

1. **Significant reduction in total costs**: Express orders appear to be an efficient investment, overall reducing the costs associated with stock shortages and high holding costs.
2. **Decrease in backlog**: Express orders help maintain an adequate inventory level, drastically reducing periods of shortage.
3. **Improved operational stability**: The ability to quickly respond to inventory shortages through express orders enhances the operational stability and responsiveness of the system.

In conclusion, based on the provided results, express orders are not only advantageous but essential for optimizing operations and reducing costs in the analyzed inventory management system. This approach seems particularly effective in scenarios with high demand variability or where inventory shortages have severe consequences.

2 Infection Simulation

The spread of infectious diseases within populations can be mathematically modeled to understand and predict epidemic outbreaks. The Susceptible-Infected-Recovered (SIR) model is one of the foundational frameworks in epidemiology for modeling how an infectious disease propagates through a population. This model divides the population into three compartments:

1. Susceptible ($s(t)$)
2. Infected ($i(t)$)
3. Recovered ($r(t)$)

Each of the previous compartment representing the fraction of the total population. The dynamics of the disease are governed by a set of differential equations, showed in Figure 2, which describe the rate of movement between these compartments.

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

Figure 2: Differential Equation SIR Model

The traditional SIR model assumes a homogeneous population with no births or deaths, focusing solely on the transitions due to disease dynamics. The transition between compartments is influenced by parameters such as the infection rate (β) and recovery rate (k), which capture the disease's contagiousness and the duration of infection, respectively.

2.1 Initialization

The initialization phase of the simulation is crucial as it establishes the starting conditions for the model, ensuring that both the Forward Euler and Runge-Kutta methods operate under consistent initial parameters. The process begins with the function `get_input`, designed to handle user input.

```
def get_input(prompt, cast_type=float, min_value=None, max_value=None):
    while True:
        try:
            value = cast_type(input(prompt))
            if (min_value is not None and value < min_value) or (max_value is not None and value > max_value):
                raise ValueError
            return value
        except ValueError:
            print(f"Invalid input. Please enter a {cast_type.__name__} value", end="")
            if min_value is not None and max_value is not None:
                print(f" between {min_value} and {max_value}.")
            else:
                print(".")

def initialize_variables():
    global t_euler, t_rk, s_euler, s_rk, i_euler, i_rk, r_euler, r_rk, k,dt, beta, t_final
    t_euler= get_input("Enter initial time (t=0): ", float, 0)
    t_rk = t_euler
    beta = get_input("Enter infection rate (beta): ", float, 0)
    s_euler = get_input("Enter initial susceptible fraction (s): ", float, 0, 1)
    s_rk = s_euler
    i_euler = get_input("Enter initial infected fraction (i): ", float, 0, 1)
    i_rk = i_euler
    r_euler = get_input("Enter initial recovered fraction (r): ", float, 0, 1)
    r_rk = r_euler
    k = get_input("Enter recovery rate (k): ", float, 0)
    dt = get_input("Enter time step (dt): ", float, 0)
    t_final = get_input("Enter t_final: ",int, 0 )

    return True
```

This function prompts the user for specific values. It also checks whether the entered values fall within an acceptable range, if specified, such as ensuring that the fractions of susceptible, infected, and recovered populations are between 0 and 1, and that rates like β and k are non-negative.
Once inputs are validated and collected, they are stored in global variables such as `s_euler`, `i_euler`, `r_euler` for the Forward Euler method, and similarly named variables for the Runge-Kutta method. This dual storage ensures that each numerical method operates independently but under equivalent initial conditions.
After the validation of all the inputs the simulation begin.

```
initialize_euler()
for _ in range(t_final):  # Adjust the number of iterations as needed
    update_euler_method()
    observe_euler()

#plotting the result ...

initialize_runge_kutta()
for _ in range(t_final):
    update_runge_kutta()
    observe_runge_kutta()

#plotting the results ...
```

2.2 Euler Method

The Euler method is a straightforward numerical technique used to approximate solutions to ordinary differential equations (ODEs) through a series of steps. The Euler Method works following the equation 1

$$x(t + \Delta t) = x(t) + G(x(t)) + \Delta t \quad (1)$$

where $G(.)$ is the derivative of the function x.
The `initialize_euler` function sets up the initial state of the simulation. It initializes arrays to store the time t and the fractions of susceptible (s), infected (i), and recovered (r) individuals over time. These arrays

are populated with the starting values, ensuring that the initial conditions are recorded as the baseline of the simulation.

```
    def initialize_euler():

    #global variables ...

    result_s_euler = [s_euler]
    result_i_euler = [i_euler]
    result_r_euler = [r_euler]
    result_t_euler = [t_euler]
```

The `observe_euler` function is called after each update to record the current state. By appending the latest values of s, i, and r, along with the corresponding time t, to their respective arrays, the function tracks the evolution of the disease throughout the simulation.

```
def observe_euler():

    #global variables ...

    result_s_euler.append(s_euler)
    result_i_euler.append(i_euler)
    result_r_euler.append(r_euler)
    result_t_euler.append(t_euler)
```

The core of the simulation lies within the `update_euler_method` function. This function performs the actual numerical solution of the SIR model's differential equations using the Forward Euler method.

```
def update_euler_method():

    #global variables ...

    #update s
    next_s_euler = s_euler - (beta * s_euler
    * i_euler * dt)

    #update i
    next_i_euler = i_euler + (beta * s_euler
    * i_euler * dt) - (k * i_euler * dt)

    #update r
    next_r_euler = r_euler + (k * i_euler *
    dt)

    t_euler = t_euler + dt
    s_euler = next_s_euler
    i_euler = next_i_euler
    r_euler = next_r_euler
```

If we consider the following initial values:

- $s(0) = 0,99$
- $i(0) = 0,01$
- $r(0) = 0$
- $\beta = 0,5$
- $k = 0,1$
- $\Delta t = 0.1$

the Euler Method return the graph shown in Figure 3

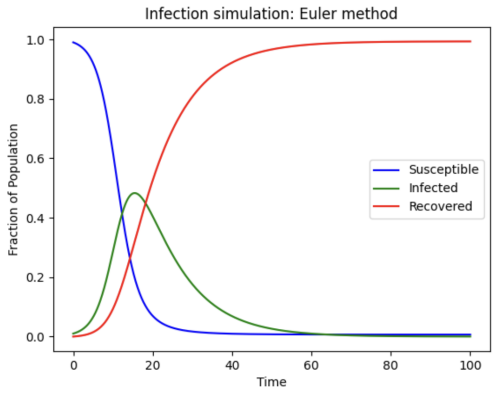

Figure 3: Euler Graph

2.3 Kunge Rutta Method

The Runge-Kutta method is a numerical technique used to approximate the solutions of ordinary differential equations (ODE) with particular effectiveness when it is not possible to obtain analytical solutions. The Runge-Kutta approach involves computing a series of incremental approximations of the solution, using evaluations of the derivative at intermediate points within each integration interval. This allows you to obtain more accurate solutions than other numerical methods, such as Euler's method, especially for non-linear or complex equations.
The first method implemented is the `initialize_runge_kutta` function. This function is responsible for initializing the global variables used in the Runge-Kutta method, as well as setting the initial conditions for variables representing the state of the epidemiological system, such as the number of susceptible, infected, and recovered individuals. Specifically, a result array is created that will contain the values of these variables at each time step during the simulator execution.

```
def initialize_runge_kutta():

    #global variables ...

    result_s_rk = [s_rk]
    result_i_rk = [i_rk]
    result_r_rk = [r_rk]
    result_t_rk = [t_rk]
```

The `update_runge_kutta` method implements the algorithm of the Runge-Kutta method to calculate the evolution of the variables of the epidemiological system over time. This method calculates the derivatives of the variables with respect to time using the differential equations that describe the specific epidemiological model under consideration. Subsequently, the increase coefficients $k1$, $k2$, $k3$ and $k4$ are calculated, and the values of the susceptible, infected and recovered variables are updated based on these coefficients, maintaining a constant time step.

```
def update_runge_kutta():

    #global variables ...

    function_derivate_s = -beta * s_rk * i_rk
    function_derivate_i = beta * s_rk * i_rk
    - k * i_rk
    function_derivate_r = k * i_rk

    k1_s = dt * function_derivate_s
    k1_i = dt * function_derivate_i
    k1_r = dt * function_derivate_r

    k2_s = dt * (-beta * (s_rk + k1_s/2) * (
    i_rk + k1_i/2))
    k2_i = dt * (beta * (s_rk + k1_s/2) * (
    i_rk + k1_i/2) - k * (i_rk + k1_i/2))
    k2_r = dt * (k * (i_rk + k1_i/2))

    k3_s = dt * (-beta * (s_rk + k2_s/2) * (
    i_rk + k2_i/2))
    k3_i = dt * (beta * (s_rk + k2_s/2) * (
    i_rk + k2_i/2) - k * (i_rk + k2_i/2))
    k3_r = dt * (k * (i_rk + k2_i/2))

    k4_s = dt * (-beta * (s_rk + k3_s) * (
    i_rk + k3_i))
    k4_i = dt * (beta * (s_rk + k3_s) * (i_rk
     + k3_i) - k * (i_rk + k3_i))
    k4_r = dt * (k * (i_rk + k3_i))

    next_s_rk = s_rk + (k1_s + 2*k2_s + 2*
    k3_s + k4_s) / 6
    next_i_rk = i_rk + (k1_i + 2*k2_i + 2*
    k3_i + k4_i) / 6
    next_r_rk = r_rk + (k1_r + 2*k2_r + 2*
    k3_r + k4_r) / 6

    t_rk = t_rk + dt
    s_rk = next_s_rk
    i_rk = next_i_rk
    r_rk = next_r_rk
```

The `observe_runge_kutta` method aims to record the values of the epidemiological system variables in each time step. These values are stored in the resulting arrays previously initialized within the `initialize_runge_kutta()` function. This allows you to analyze and visualize the trend of the variables over time and to evaluate the effectiveness of the epidemic containment strategies implemented in the simulator.

```
def observe_runge_kutta():

    #global variables ...

    result_s_rk.append(s_rk)
    result_i_rk.append(i_rk)
    result_r_rk.append(r_rk)
    result_t_rk.append(t_rk)
```

If we consider the following initial values, that are the same used with the Euler Method:

- $s(0) = 0,99$
- $i(0) = 0,01$
- $r(0) = 0$
- $\beta = 0,5$
- $k = 0,1$
- $\Delta t = 0.1$

the Runge Kutta Method return the graph shown in Figure 4.

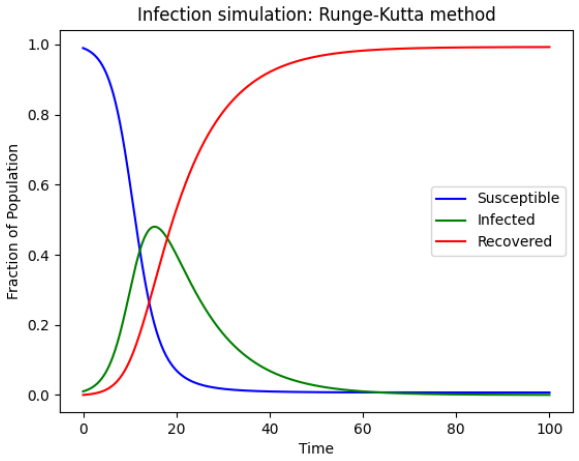

Figure 4: Runge Kutta Graph

3 Comparison

If we compare the two results obtained, we notice that they are identical, as highlighted in the Figure 5.

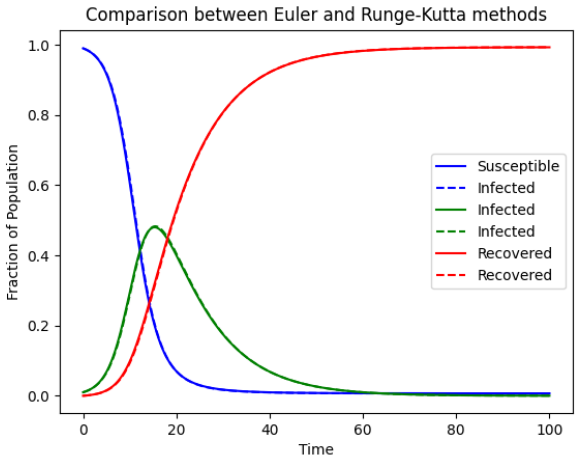

Figure 5: Method Comparation

Changing the number of iterations and increasing the time step, for example, bringing it to 100 iterations and setting the step dt to 0.5, will result in differences in the graphs generated by the Euler and Runge-Kutta methods, as shown in Figure 6
Greater is Δt grater will be the difference. In all the graphs the Runge-Kutta method is represented by the

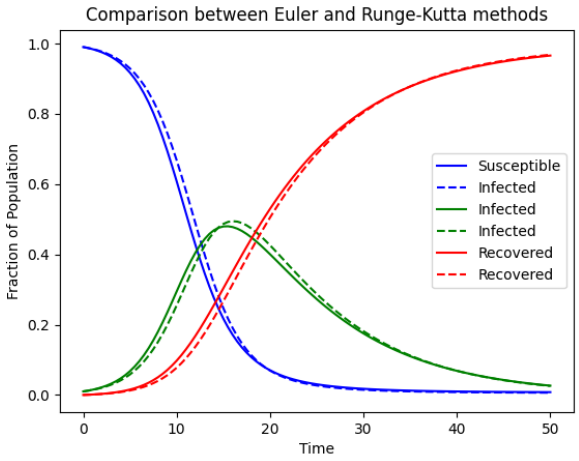

Figure 6: Method Comparation

continuous line, the dotted one represents the Euler Method. For a more detailed analysis, graphs representing individual population fractions were also included. Figure 7 shows the graph of the susceptible, in Figure 8 the graph relating to the infected, while in Figure 9 the graph relating to the recovered is presented.

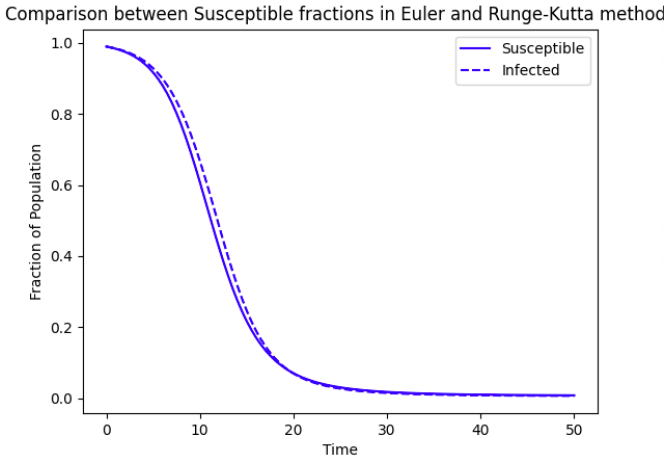

Figure 7: Comparison Susceptible

This happens because Euler's method is more sensitive to the size of the time step. By increasing the time step, the Euler method approximation becomes less accurate because it does not take into account intermediate changes in the derivative. In contrast, the Runge-Kutta method is more robust and handles larger time steps better due to its iterative approach and calculation of increment coefficients. Increasing the number of iterations further improves the accuracy of the solution with the Runge-Kutta method, while Euler continues to show significant discrepancies.

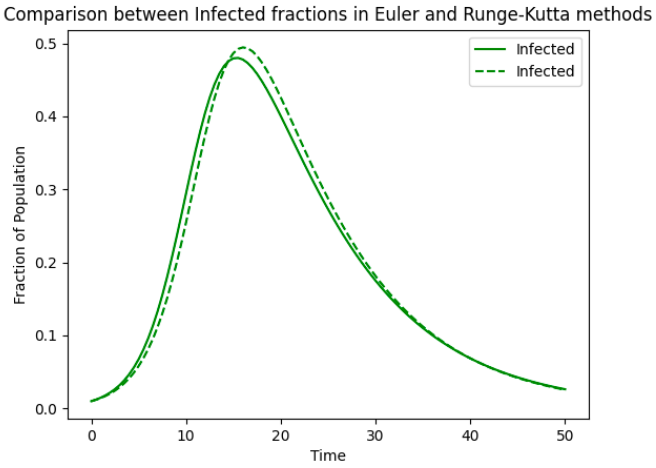

Figure 8: Comparison Infected

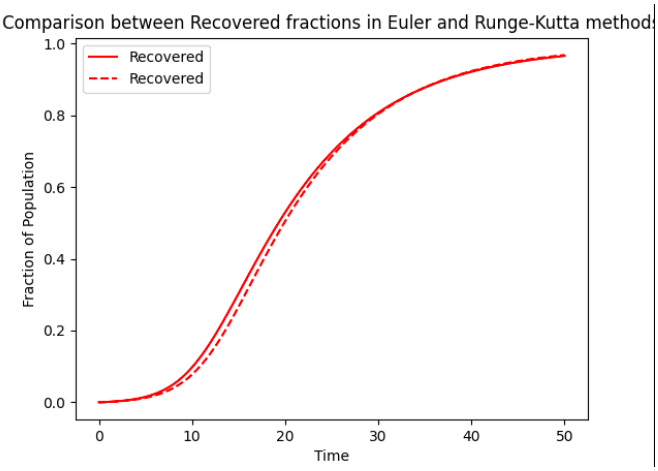

Figure 9: Comparison Recovered

4 Appendix

References

[1] A. M. Law, *Simulation Modeling Analysis*, 5th ed. McGraw-Hill.

Table 1: Inventory Simulation Results (Base Simulation)

Policy	Total	Order_Cost	Hold_Cost	Short_Cost
(20,40)	126.54	93.98	9.20	23.36
(20,60)	120.84	91.98	18.09	10.76
(20,80)	120.25	83.47	27.60	9.17
(20, 100)	128.637	84.825	36.333	7.479
(40,60)	126.63	99.59	25.66	1.38
(40,80)	123.20	87.68	34.30	1.21
(40,100)	132.91	85.98	46.18	0.74
(60,80)	145.85	99.20	46.60	0.04
(60,100)	140.19	84.95	55.23	0.01

Table 2: Inventory Simulation Results

Policy	Total	Order_Cost	Hold_Cost	Short_Cost	Cost per Month	%I(t) < 0	Express Orders	%Spoiled Items
(20,40)	595.452	72.333	464.377	58.741	4.962	11.660	13	0.201
(20,60)	313.636	107.392	134.964	71.280	2.614	14.359	16	0.162
(20,80)	785.628	75.250	640.859	69.519	6.547	6.483	8	0.151
(20,100)	892.980	99.700	710.529	82.750	7.441	6.246	7	0.194
(40,60)	286.816	101.450	145.069	40.297	2.390	16.485	16	0.176
(40,80)	741.260	101.992	542.905	96.363	6.177	6.735	8	0.215
(40,100)	261.522	119.058	92.560	49.904	2.179	12.330	12	0.213
(60,80)	285.617	172.250	56.548	56.819	2.380	14.101	14	0.175
(60,100)	275.860	139.108	88.137	48.615	2.299	11.407	12	0.187

Table 3: Inventory Simulation Results (No Express Order)

Policy	Total	Order_Cost	Hold_Cost	Short_Cost	Cost per Month	%I(t) < 0	Express Orders	%Spoiled Items
(20,40)	6803.234	4.217	0.912	6798.105	56.694	95.306	0	0.091
(20,60)	7305.355	1.642	1.737	7301.977	60.878	95.151	0	0.085
(20,80)	7638.081	4.033	1.311	7632.738	63.651	95.500	0	0.086
(20,100)	7137.991	5.008	2.529	7130.453	59.483	93.789	0	0.090
(40,60)	7247.668	4.025	1.198	7242.445	60.397	95.329	0	0.090
(40,80)	6842.598	4.400	2.217	6835.981	57.022	94.669	0	0.089
(40,100)	7209.186	10.983	5.043	7193.160	60.077	89.063	0	0.097
(60,80)	6810.013	19.300	5.191	6785.522	56.750	85.837	0	0.103
(60,100)	7133.740	3.358	2.590	7127.792	59.448	95.013	0	0.086