Simulation Mini-Projects

Paulo Pereira[98430],Rui Fernandes[92952]

Simulação e Otimização, Universidade de Aveiro, 3810-193 Aveiro, Portugal

1 Introduction

This document covers the approach for the first mini-project assignment in the course *Simulação e Otimização*.

The assignment was comprised by two exercises, the first one consisting in simulating a service facility, that had two types of customers and servers, in this exercise it was asked of us to calculate metrics and evaluate the performance of the system when we added one more server to each type independently.

The objective of the second one was to simulate approximate results for the Lotka-Volterra model using two methodologies, a first order one, and a fourth order one, the Euler and Runge-Kutta methods respectively.

2 First exercise

To solve this exercise, we started by looking over all the code developed in the classes and upon doing this we found a library called sympy, previously used to calculate primefactors in an exercise that aimed to find if a Linear Congruential Generator had a full period given certain paramaters. This finding took us to take a look into this library and it's features. After reading the documentation, we realized that this library was capable of simulating queues and servers. So we decided to explore this form of implementation over the way we had previously done in class.

2.1 Simpy and it's uses

Simpy is able to simulate both servers and queues using it's own classes. In simpy, the Resource class is able to simulate a server that only serves one client at a time, when given a simpy environment and a capacity, which in this case is the number of servers of that type in the system. To simulate a customer queue simpy has a Store class that acts like a FIFO queue with infinite storage, if no argument for capacity is given. These classes met exacly what the exercise was aiming for and, with them, we decided to take on the challenge of implementing them.

To make use of these features, each class has different methods we utilized, in the case of the Resources we mostly used the **request** method. For the Stores since they are being used like queues we used the **put** and **get** methods, and we also used the Environment class to create the processes(each process was treated like a customer by the system) and provide all the timings using the **now** method.

2.2 Implementation

To implement the system, we developed some functions, `customer_arrival` , `type1_customer` , `type2_customer` and `queue_manager`.

- `customer_arrival` - this function takes as arguments the servers of each type and, after randomly determining the type of customer, creates a process that will start the service process for the type of customer generated
- `type1_customer` - this function is responsible for serving type 1 customers, assigning them to a queue or a server
- `type2_customer` - this function is responsible for serving type 2 customers, assigning them to a queue or a server
- `queue_manager` - this function is responsible for setting the priority to type 2 customers when they are leaving a queue, as well as, general queue management.

Customer Generation Like previously mentioned, the customer generation falls under the responsibility of the `customer_arrival` function, this function will randomly (according to the probabilities mentioned in the assignment) generate a customer and create a process, that in essence is that customer's life-cycle.

Queue Management As asked in the assignment we needed to implement a queue for each type of customer and, for that, we decided to develop a function that would be responsible for both queues and all the operations of a queue (except the queue insertion). This function starts by checking if any queue has customers (first the type 2 customer queue) and if there are customers in any queue, it try to find a way to serve them, if a way is found it starts a new process with that customer and removes it from the queue.

Customer of type 1 life-cycle When a new customer of type 1 is generated the system will try to find a server of type A for it and if not possible a server of type B, in case a server is not found it is then added to a the type 1 customers' queue. If a server is found, it will be requested using a **with** statement, upon receiving the confirmation from the **.request()** method from the **simpy.Resource** class, the servers list will be updated using the call *.requested[i]* to the according server type list, the *.requested* list is a list of booleans where True means the server is in use and False means it's free, which is how we can easily keep track of which servers are available. Despite we not really needing this list for the server requests, since simpy is responsible for handling that part, we kept this list since it is easier to debug and to provide the status of the system in real time. After recording the server usage and waiting times we then use a **yield** statement with a timeout equal to the calculated required time to serve that customer.

Customer of type 2 life-cycle In essence type 2 customers and type 1 customers are treated with the same logic by the system, but there are a few key differences. Firstly, type 2 customers need two servers (one of each type), so the system tries to find one of each, if any of the searches fails the customer is then put into the type 2 customers queue. If both searches are successful, and once again using a **with** statement the servers are requested. After being requested the statistics are updated and the timeout signal is sent. After that both servers are released.

2.3 Results

In this section, we will present the results to the first mini-project.

These are the statistics we gathered from a normal scenario using two type A servers and one type B server

```
----- Results -----


Simulation with 3 servers and 1019 customers
Served 800 Type 1 customers
Served 219 Type 2 customers
Queue Type 1 size at end of simulation: 0
Queue Type 2 size at end of simulation: 0
Average time spent in the system per Type 1 customer: 0.829791296569312
Average time spent in the system per Type 2 customer: 0.5966216418530347
Average delay in queue for any customer: 0.08413702198106095
Average delay in queue for Type 1 customers: 0.036027312176790824
Average delay in queue for Type 2 customers: 0.25988025414277827
Expected time average number in queue for type 1 customers: 0.064
Expected time average number in queue for type 2 customers: 0.185
Proportion of time server A1 was in use by type 1 customers: 37.02
Proportion of time server A1 was in use by type 2 customers: 9.90
Proportion of time server A2 was in use by type 1 customers: 23.08
Proportion of time server A2 was in use by type 2 customers: 3.17
Proportion of time server B1 was in use by type 1 customers: 6.28
Proportion of time server B1 was in use by type 2 customers: 13.07
Maximum of the average delay in queue for both types of customers: 0.25988025414277827
```

Fig. 1. Results for Normal scenario (2 type A, 1 type B server)

For the second part of the mini-project, we were asked to add one of each server independently and find which helped reduce the maximum delay in queue for the customers.

As we can see in the last line of the the results adding one more server of the type A was better for the system and substantially reduced the maximum average delay in queue for the customers.

```
----- Results -----


Simulation with 4 servers and 1014 customers
Served 792 Type 1 customers
Served 222 Type 2 customers
Queue Type 1 size at end of simulation: 0
Queue Type 2 size at end of simulation: 0
Average time spent in the system per Type 1 customer: 0.8248258222718009
Average time spent in the system per Type 2 customer: 0.5948846840074187
Average delay in queue for any customer: 0.019068323806531562
Average delay in queue for Type 1 customers: 0.0057718888333388235
Average delay in queue for Type 2 customers: 0.06650425398116512
Expected time average number in queue for type 1 customers: 0.003
Expected time average number in queue for type 2 customers: 0.018
Proportion of time server A1 was in use by type 1 customers: 35.63
Proportion of time server A1 was in use by type 2 customers: 9.36
Proportion of time server A2 was in use by type 1 customers: 20.59
Proportion of time server A2 was in use by type 2 customers: 2.84
Proportion of time server B1 was in use by type 1 customers: 1.09
Proportion of time server B1 was in use by type 2 customers: 13.21
Proportion of time server A3 was in use by type 1 customers: 8.02
Proportion of time server A3 was in use by type 2 customers: 1.01
Maximum of the average delay in queue for both types of customers: 0.06650425398116512
```

Fig. 2. Results for scenario with 3 type A and 1 type B servers

```
----- Results -----


Simulation with 4 servers and 1012 customers
Served 795 Type 1 customers
Served 217 Type 2 customers
Queue Type 1 size at end of simulation: 0
Queue Type 2 size at end of simulation: 0
Average time spent in the system per Type 1 customer: 0.8261351815726923
Average time spent in the system per Type 2 customer: 0.5971135509856436
Average delay in queue for any customer: 0.07856483252339119
Average delay in queue for Type 1 customers: 0.03389862070455592
Average delay in queue for Type 2 customers: 0.2422037191407831
Expected time average number in queue for type 1 customers: 0.059
Expected time average number in queue for type 2 customers: 0.131
Proportion of time server A1 was in use by type 1 customers: 37.00
Proportion of time server A1 was in use by type 2 customers: 9.79
Proportion of time server A2 was in use by type 1 customers: 22.33
Proportion of time server A2 was in use by type 2 customers: 3.17
Proportion of time server B1 was in use by type 1 customers: 3.35
Proportion of time server B1 was in use by type 2 customers: 6.56
Proportion of time server B2 was in use by type 1 customers: 2.99
Proportion of time server B2 was in use by type 2 customers: 6.40
Maximum of the average delay in queue for both types of customers: 0.2422037191407831
```

Fig. 3. Results for scenario with 2 type A and 2 type B servers

3 Second Exercise

3.1 Objective

The aim of this exercise was centered around the Lotka-Volterra model, a pair of first-order differential equations, used to describe the evolution of the population of two species, one a predator and one a prey in a biological system in which only they interact.

For some context, the equations are as follow:

$$\frac{dx}{dt} = \alpha x - \beta xy$$
$$\frac{dy}{dt} = \delta xy - \gamma y$$

where:

- x represents the population of the prey species
- y represents the population of the predator species
- α represents the natural growth rate of the prey
- β represents the effect of the presence of predators on the prey growth rate
- δ represents the effect of the presence of prey on the predator's growth rate
- γ represents the natural death rate of the predator.

We had to write programs that simulated the evolution of this model, getting approximate results by resorting to two methods meant for solving ordinary differential equations, a first order one, the Euler method, and a fourth order one, the Runge-Kutta method (RK4).

We also had to write simple code to permit the user to input custom parameters through the command line.

3.2 Common Implementation

In terms of implementation the code is quite simple, two python scripts were written, one for each method, a portion of the code is common to both and so will be explained first.

Firstly, the parameters of the Lotka-Volterra model, as well as a time step, total time, and initial values for the prey and predator counts, were all defined as constants and given default values. The default values represent a common, stable, example of the model and will be the ones used for all graphs in this document.

```
# Default Values
ALPHA = 0.1
BETA = 0.02
GAMMA = 0.4
DELTA = 0.02
DELTA_T = 0.1
T_FINAL = 300
X0 = 10
Y0 = 10
```

Fig. 4. Constant definition

Secondly the simple code meant to allow the user to input parameters trough the command line that substitute the default values.

```
# Remove 1st argument from the
# list of command line arguments
argumentList = sys.argv[1:]

# Options
options = "a:b:g:d:t:f:x:y:"

# Long options
long_options = ["Alpha=", "Beta=", "Gamma=", "Delta=", "D_Time=", "T_Final=", "X0=", "Y0="]

try:
    # Parsing argument
    arguments, values = getopt.getopt(argumentList, options, long_options)

    # checking each argument
    for currentArgument, currentValue in arguments:

        if currentArgument in ("-a", "--Alpha"):
            ALPHA = float(currentValue)

        elif currentArgument in ("-b", "--Beta"):
            BETA = float(currentValue)

        elif currentArgument in ("-g", "--Gamma"):
            GAMMA = float(currentValue)

        elif currentArgument in ("-d", "--Delta"):
            DELTA = float(currentValue)

        elif currentArgument in ("-t", "--D_Time"):
            DELTA_T = float(currentValue)

        elif currentArgument in ("-f", "--T_Final"):
            T_FINAL = int(currentValue)

        elif currentArgument in ("-x", "--X0"):
            X0 = float(currentValue)

        elif currentArgument in ("-y", "--Y0"):
            Y0 = float(currentValue)

except getopt.error as err:
    # output error, and return with an error code
    print (str(err))
```

Fig. 5. Command line argument processing

Finally we define the time vector with the final time and time-step, followed by the initialize, observe, update routine.

```
# Determine time vector
num_steps = int(T_FINAL / DELTA_T)
time_vector = [i * DELTA_T for i in range(0,num_steps)]
initialize()

for t in time_vector:
    observe()
    update()
```

Fig. 6. Base code

3.3 Euler Method Implementation

The Euler method is a numerical approximation technique used to solve ordinary differential equations. It involves dividing the interval into small steps and estimating the function's values at each step by using the derivative at the current point. Although simple and computationally efficient, the Euler method is a first-order method and can introduce errors, especially in complex systems.

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

where:

- $x(t)$: The approximate value of the function at the current step t.
- $y(t + 1)$: The approximate value of the function at the next step $t + 1$.
- Δt: The step size, determining the interval between successive steps.
- $G(x(t))$: The derivative of the function evaluated at the current point $(x(t))$.

In terms of code implementation it was straightforward, as we can see in the following image:

```
def initialize():
    global x, y, x_result, y_result

    # x0 and y0
    x = X0
    y = Y0

    x_result = []
    y_result = []

def observe():
    global x, y, x_result, y_result
    x_result.append(x)
    y_result.append(y)

def update():
    global x, y
    new_x = x + (ALPHA*x - BETA*x*y) * DELTA_T
    new_y = y + (DELTA*x*y - GAMMA*y) * DELTA_T
    x = new_x
    y = new_y
```

Fig. 7. Euler method

3.4 Runge-Kutta Method Implementation

The Runge-Kutta method is a numerical technique used to approximate the solutions of ordinary differential equations. It is an iterative approach that calculates the values of a function at different points by considering weighted averages of function evaluations at various intermediate stages within each step. The method typically uses a higher-order approximation than the Euler method, resulting in improved accuracy. By evaluating the derivatives at multiple points within each step, the Runge-Kutta method provides more accurate estimates of the function's values. It is widely used in scientific and engineering applications to solve ODEs with higher precision and handle complex systems more effectively.

$$
\begin{aligned}
k_1 &= hf(x_n, y_n) \\
k_2 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right) \\
k_3 &= hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right) \\
k_4 &= hf(x_n + h, y_n + k_3) \\
y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned}
$$

where:

- k_1: The derivative of the function evaluated at the current point (x_n, y_n).
- k_2: The derivative of the function evaluated at the midpoint of the interval using an intermediate value of y.
- k_3: The derivative of the function evaluated at the midpoint of the interval using another intermediate value of y.
- k_4: The derivative of the function evaluated at the endpoint of the interval using the final intermediate value of y.
- h: The step size, determining the interval between successive steps.
- x_n: The value of the independent variable at the current step n.
- y_n: The approximate value of the function at the current step n.
- y_{n+1}: The approximate value of the function at the next step $n + 1$.

The code implementation of the Runge-Kutta method is similar to the Euler method:

```
def initialize():
    global x, y, x_result, y_result

    # x0 and y0
    x = X0
    y = Y0

    x_result = []
    y_result = []

def observe():
    global x, y, x_result, y_result
    x_result.append(x)
    y_result.append(y)

def update():
    global x, y

    k1_x, k1_y = f(x,y)

    k2_x, k2_y = f(x + DELTA_T/2, y + k1_y/2)

    k3_x, k3_y = f(x + DELTA_T/2, y + k2_y/2)

    k4_x, k4_y = f(x + DELTA_T, y + k3_y)

    x = x + (k1_x + 2*k2_x + 2*k3_x + k4_x) /6
    y = y + (k1_y + 2*k2_y + 2*k3_y + k4_y) /6

def f(x,y):
    return (DELTA_T * (ALPHA * x - BETA * x * y), DELTA_T * (DELTA * x * y - GAMMA * y) )
```

Fig. 8. Runge-Kutta method

3.5 Results

In this section we will show the graphs that describe our results, for each method there are four graphs, two describing the prey and predator populations over time, one for a time step of 0.1 and another for a time step of 0.01, and two describing the predator vs prey values, for the same time steps.

We can observe that the smaller time step gives more accurate and stable results for both methods, and that the Runge-Kutta method is more accurate to start with.

3.6 Euler Method Results

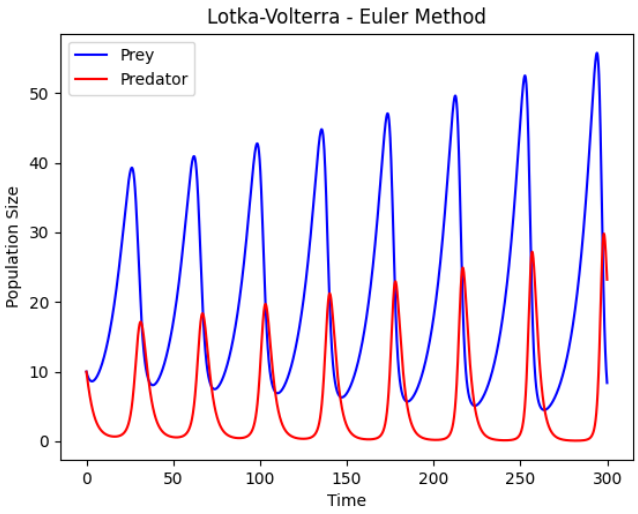

Fig. 9. Euler method for Δ t = 0.1

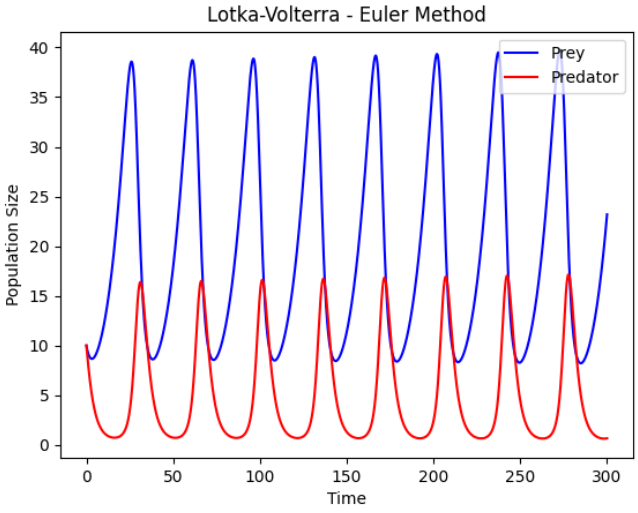

Fig. 10. Euler method for Δ t = 0.01

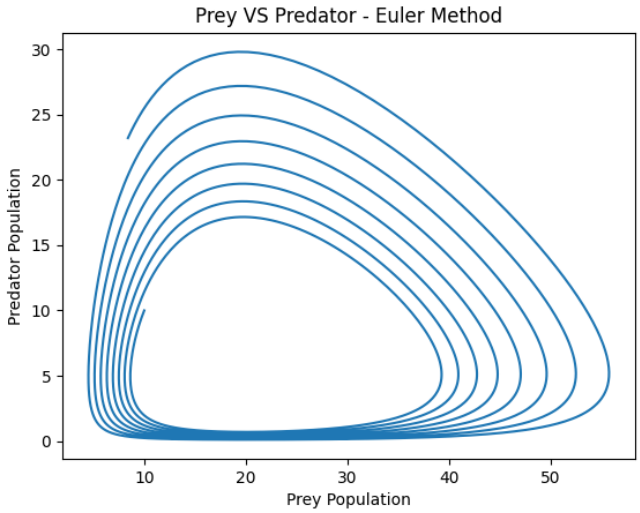

Fig. 11. Euler method Predator vs Prey for Δ t = 0.1

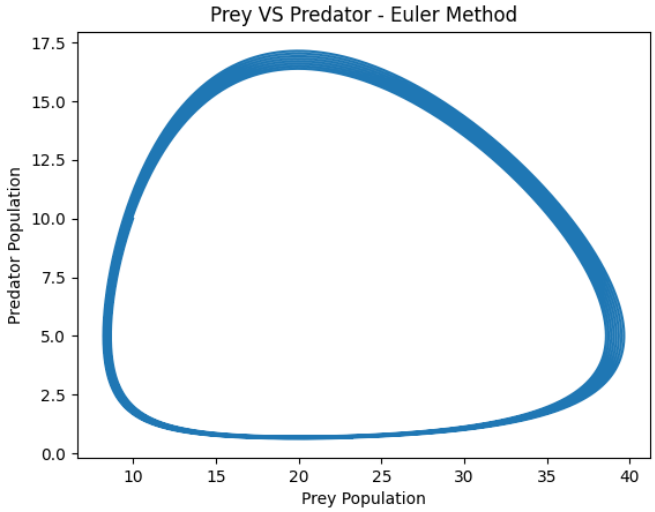

Fig. 12. Euler method Predator vs Prey for Δ t = 0.01

3.7 Runge-Kutta Results

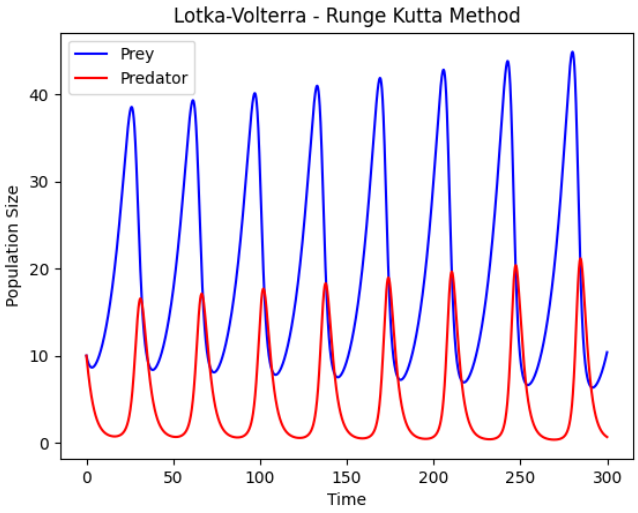

Fig. 13. Runge-Kutta method for Δ t = 0.1

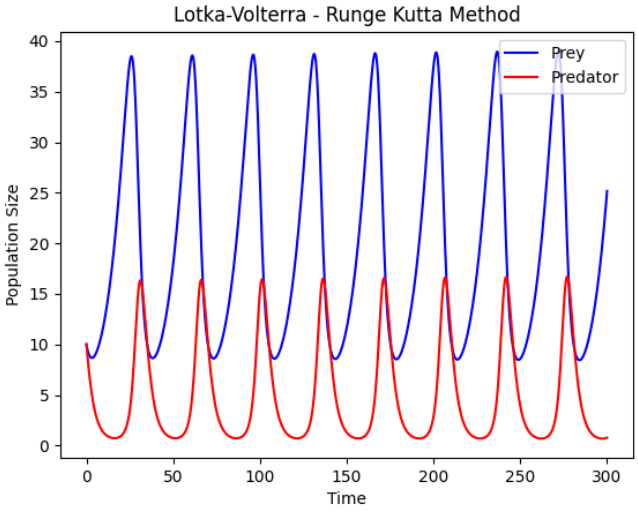

Fig. 14. Runge-Kutta method for Δ t = 0.01

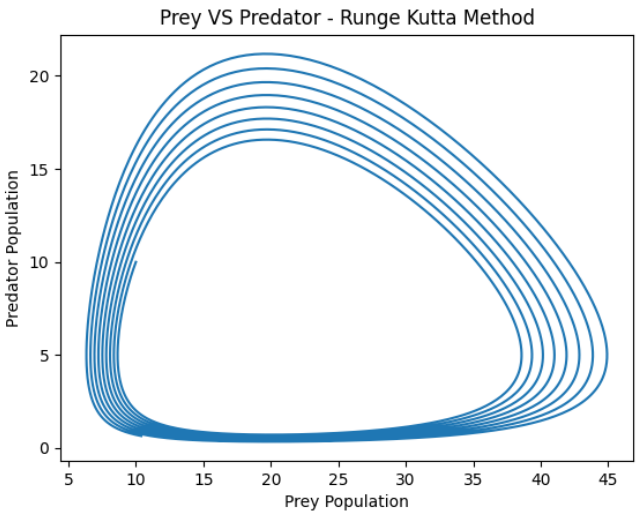

Fig. 15. Runge-Kutta method Predator vs Prey for Δ t = 0.1

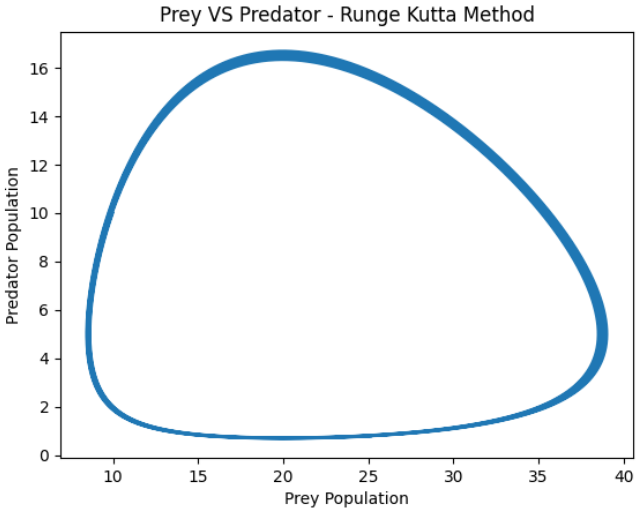

Fig. 16. Runge-Kutta method Predator vs Prey for Δ t = 0.01

4 Conclusion