universidade de aveiro deti departamento de eletrónica, telecomunicações e informática

Mini-Projeto de Simulação

Simulação e Otimização

Docentes Nuno Lau e Amaro Sousa
2022/2023

Mestrado em Engenharia Informática

Artur Romão, 98470
João Reis, 98474

Índice

1. Introdução

Este relatório tem como objetivo documentar as estratégias de resolução levadas a cabo para responder aos problemas do primeiro mini-projeto da componente de Simulação da unidade curricular Simulação e Otimização.

Serão explicadas as abordagens aos problemas, as estratégias de resolução e as respostas às questões levantadas de cada um dos problemas.

2. Problema 1

2.1. Estratégia implementada

Primeiramente, antes de explicar a abordagem implementada para a resolução do exercício 1 é necessário referir que foi criada uma classe ***Costumer***. Esta classe tem como objetivo ajudar na criação de vários e diferentes clientes, bem como armazenar toda a informação necessária de um determinado cliente.

Quando é criada uma instância desta classe, é definido o seu tempo de chegada (*arrival_time*), o seu tipo (*type*), em que evento/estado o cliente se encontra (*event_type* - quando criado, o cliente estará no evento "*arrive*"), em qual dos servidores está a trabalhar (*server_type* - quando criado, o cliente não estará a trabalhar, ou seja, *None*), e é definido também o seu identificador (*id*). Todos os métodos *getters* e *setters* necessários para a implementação foram criados.

```
class Costumer:
    identifier = 0
    type1_probability = 0.8

    def __init__(self, arrival_time) -> None:
        # initializate costumer variables
        self.id = Costumer.identifier

        # upon arrival, a customer is determined to be either a type 1 customer or a type 2 customer
        self.type         = 1 if np.random.rand() < Costumer.type1_probability else 2
        self.time         = arrival_time
        self.event_type   = "arrive"
        self.server_type  = None
        self.arrival_time = arrival_time

        self.waiting_time       = 0.0
        self.start_working_time = arrival_time      # the start working time could not be the arrival time !!!
        self.end_working_time   = 0.0

        Costumer.identifier += 1
```

Fig. 1 - Função que inicializa um objeto *Costumer*

Vamos agora analisar o ficheiro *sim1.py*. Em primeiro lugar, são inicializadas todas as variáveis necessárias ao programa. É de notar a criação de uma lista importante, chamada ***active_costumers***, que vai conter apenas os clientes que estão em atividade, isto é, que não estão em lista de espera.

Foram definidos **3 eventos** que um cliente pode ter: ***arrive***, caso o cliente acabe de chegar; ***work***, caso o cliente esteja pronto a trabalhar num servidor; e ***depart***, caso o cliente tenha acabado de fazer a sua função no servidor onde trabalhou.

Para começar é chamada a função ***timing()***. O seu objetivo é identificar qual é o próximo cliente a realizar um evento. Primeiro, vamos ver, dentro dos *active_costumers*, qual é o cliente com o evento mais próximo do tempo de simulação atual.

Caso encontre algum cliente onde o seu próximo evento é do tipo "*work*", será esse o próximo cliente, dado que esse tipo de evento indica que o cliente está pronto a trabalhar. Caso contrário, irá analisar se se reúnem todas as condições para um cliente das filas de espera começar a trabalhar. As condições necessárias para tal são: se os servidores, que são necessários devido ao seu tipo, estão disponíveis, e caso o seu tempo de realizar o evento seja menor que o até então selecionado. Basicamente, vamos dar preferência ao evento que está mais próximo do tempo de simulação atual.

É de notar que estamos a dar preferência a clientes do tipo 2 em relação aos clientes do tipo 1, no que toca a escolher qual dos clientes em lista de espera escolher.

O cliente escolhido é armazenado na variável *actual_costumer*, e o evento que o mesmo irá realizar na variável *next_event_type*. Dependendo desta variável, o *actual_costumer* irá desempenhar funções com intenções diferentes.

A função ***arrive()*** vai verificar se há servidores disponíveis para o *actual_costumer* trabalhar. Em caso positivo, irá definir o próximo evento do *actual_costumer* como "work", dado que o cliente está pronto a trabalhar nos servidores. Em caso negativo, irá alterar também o próximo evento desse cliente para "*work*", porém vai adicioná-lo à lista de espera correspondente ao seu tipo.

Em suma, esta função avalia se o *actual_costumer* pode começar a trabalhar diretamente nos servidores ou se irá para a fila de espera.

```python
def arrive():
    global num_serverA_available, num_serverB_available, queue_type1_costumers, queue_type2_costumers, actual_costumer, activ
    log_msg = False

    # check if costumer needs to go to waiting queue: if no servers available
    if actual_costumer.get_type() == 1:
        if not (num_serverA_available > 0 or num_serverB_available > 0):    # no servers available, so waitigng queue
            log_msg = True
            queue_type1_costumers.put( actual_costumer )
            active_costumers.remove( actual_costumer )
            number_in_queue_type1 += 1  # for statistics

    elif actual_costumer.get_type() == 2:
        if not (num_serverA_available > 0 and num_serverB_available > 0):   # no servers available, so waitigng queue
            log_msg = True
            queue_type2_costumers.put( actual_costumer )
            active_costumers.remove( actual_costumer )
            number_in_queue_type2 += 1  # for statistics

    actual_costumer.set_event_type("work")  # costumer ready to work, no mather if we stays on waiting queue or not

    active_costumers.append( Costumer( sim_time + np.random.exponential(1.0) ) )    # add new costumer arrival event
    print('[Costumer {} (id = {})] arrival event at {:.2f} | waiting_queue_1 = {} | waiting_queue_2 = {}'.format(actual_costu
    if log_msg: print(f"[Info] No servers available for that type of costumer!")
```

Fig. 2 - Implementação da função *arrive()*

A função ***work()*** irá fazer com que os clientes trabalhem nos servidores. Basicamente, o que ela irá fazer é decrementar as variáveis *num_serverA_available* e/ou *num_serverB_available*, que indicam quantos servidores estão disponíveis, e definir o tempo de saída do servidor (*depart*) do *actual_costumer*.

Como se pode verificar na imagem abaixo, caso o tipo do cliente seja 1, estamos a dar preferência ao servidor A. Apenas quando os dois servidores A estão indisponíveis é que será selecionado o servidor B.

```python
def work():
    global num_serverA_available, num_serverB_available, actual_costumer

    if actual_costumer.get_type() == 1:
        # a type 1 customer can be served by any server but will choose a type A server if one is available
        if num_serverA_available > 0:
            num_serverA_available -= 1
            actual_costumer.set_server_type('A')
            actual_costumer.set_event_time( sim_time + np.random.exponential(0.8) )
            actual_costumer.set_event_type('depart')

        elif num_serverB_available > 0:
            num_serverB_available -= 1
            actual_costumer.set_server_type('B')
            actual_costumer.set_event_time( sim_time + np.random.exponential(0.8) )
            actual_costumer.set_event_type('depart')

    elif actual_costumer.get_type() == 2:
        # a type 2 customer requires service from both a type A server and a type B server simultaneously
        num_serverA_available -= 1
        num_serverB_available -= 1
        actual_costumer.set_server_type('AB')
        actual_costumer.set_event_time( sim_time + np.random.uniform(0.5, 0.7) )
        actual_costumer.set_event_type('depart')

    print('[Costumer {} (id = {})] work event at {:.2f} | servers A = {} | servers B = {}'.format(actual_costumer.ge
```

Fig. 3 - Implementação da função *work()*

Por fim, a função ***depart()*** irá incrementar as variáveis que controlam a disponibilidade dos servidores, já que o cliente acabou de fazer o seu trabalho.

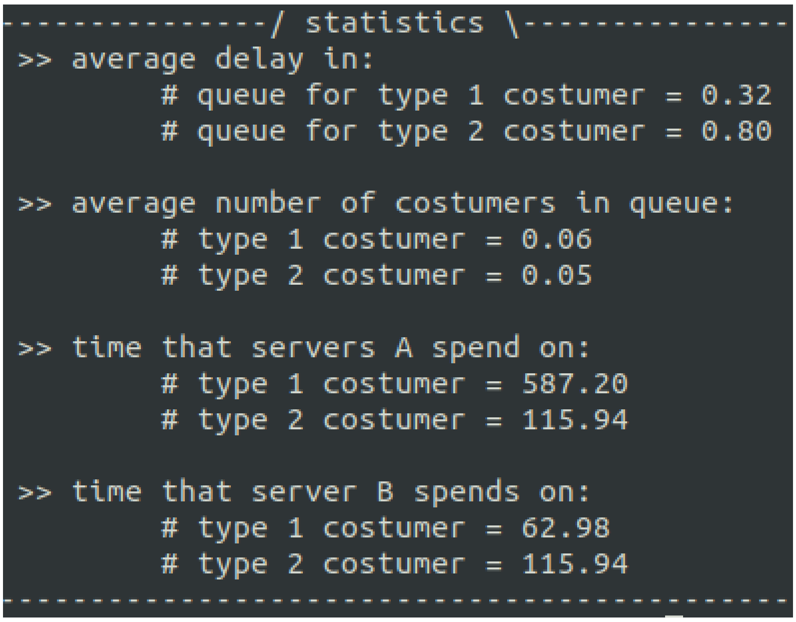
```python
def depart():
    global actual_costumer, active_costumers, num_serverA_available, num_serverB_available
    global time_serverA_spends_on_type1, time_serverA_spends_on_type2, time_serverB_spends_on_type1, time_se

    active_costumers.remove( actual_costumer )

    if actual_costumer.get_server_type() == 'A':
        num_serverA_available += 1
        time_serverA_spends_on_type1 += actual_costumer.get_working_time()  # for statistics

    elif actual_costumer.get_server_type() == 'B':
        num_serverB_available += 1
        time_serverB_spends_on_type1 += actual_costumer.get_working_time()  # for statistics

    elif actual_costumer.get_server_type() == 'AB':
        num_serverA_available += 1
        num_serverB_available += 1
        time_serverA_spends_on_type2 += actual_costumer.get_working_time()  # for statistics
        time_serverB_spends_on_type2 += actual_costumer.get_working_time()  # for statistics

    print('[Costumer {} (id = {})] departure event at {:.2f} | servers A = {} | servers B = {}'.format(actua
```

Fig. 4 - Implementação da função *depart()*

Em relação ao cálculo das estatísticas, foram criadas várias variáveis que são atualizadas ao longo do programa. Todas as iterações nestas variáveis estão identificadas no código com o comentário “*for statistics*”.

2.2. Respostas às questões

Em relação à questão 1.1, utilizando uma seed igual ao número mecanográfico de um dos alunos, 98474, os resultados são os indicados na imagem abaixo.



```
--------------/ statistics \--------------
>> average delay in:
        # queue for type 1 costumer = 0.32
        # queue for type 2 costumer = 0.80

>> average number of costumers in queue:
        # type 1 costumer = 0.06
        # type 2 costumer = 0.05

>> time that servers A spend on:
        # type 1 costumer = 587.20
        # type 2 costumer = 115.94

>> time that server B spends on:
        # type 1 costumer = 62.98
        # type 2 costumer = 115.94
--------------------------------------------
```

Fig. 5 - Resultados à questão 1.1

Em relação à questão 1.2, é pedido para comparar o *average delay* nas filas de espera. Na questão 1.2.1, com mais um servidor do tipo A, os resultados são os seguintes:

```
---------------/ statistics \---------------
>> average delay in:
        # queue for type 1 costumer = 0.18
        # queue for type 2 costumer = 0.44

>> average number of costumers in queue:
        # type 1 costumer = 0.01
        # type 2 costumer = 0.03

>> time that servers A spend on:
        # type 1 costumer = 647.34
        # type 2 costumer = 114.15

>> time that server B spends on:
        # type 1 costumer = 12.39
        # type 2 costumer = 114.15
--------------------------------------------
```

Fig. 6 - Resultados à questão 1.2.1

Com três servidores do tipo A, os resultados melhoram, uma vez que o *average delay* diminui significativamente em ambas as filas de espera.

Na questão 1.2.2, com mais um servidor do tipo B, os resultados são os seguintes:

```
---------------/ statistics \---------------
>> average delay in:
        # queue for type 1 costumer = 0.32
        # queue for type 2 costumer = 0.57

>> average number of costumers in queue:
        # type 1 costumer = 0.01
        # type 2 costumer = 0.04

>> time that servers A spend on:
        # type 1 costumer = 576.12
        # type 2 costumer = 115.83

>> time that server B spends on:
        # type 1 costumer = 85.91
        # type 2 costumer = 115.83
--------------------------------------------
```

Fig. 7 - Resultados à questão 1.2.2

Com dois servidores do tipo B, o *average delay* para a fila de espera dos clientes do tipo 1 mantém-se igual, enquanto que para a dos clientes do tipo 2 diminui. Contudo, esta diminuição não é tão significativa quanto a da questão 1.2.1.

Portanto, respondendo à pergunta 1.2, é melhor ter mais um servidor do tipo A do que do tipo B, dado que é melhor no que toca a reduzir ao máximo o *average delay* das duas filas de espera.

3. Problema 2

3.1. Estratégia implementada

Para o segundo problema, foi-nos pedido para escrever dois programas que simulassem a evolução das populações de presas e predadores, com base numa adaptação do modelo *Lotka-Volterra*. Tendo *x(t),* que representa o número de presas e *y(t),* que representa o número de predadores, temos as seguintes equações diferenciais, que regem o modelo:

$$\frac{dx(t)}{dt} = \alpha . x(t) - \beta . x(t) . y(t)$$
$$\frac{dy(t)}{dt} = \delta . x(t) . y(t) - \gamma . y(t)$$

Fig. 8 - Equações diferenciais que compõem o modelo

Assim, na primeira alínea do exercício, é-nos pedido para fazer a simulação da evolução das populações usando o método *Forward Euler*.

Para tal, seguiu-se a mesma estratégia levada a cabo na resolução do exercício 5.1 da aula 3, de Simulação com Tempos Discretos e Contínuos. Esta abordagem consiste no uso de três componentes: *Initialize*, *Observe* e *Update*. Cada uma destas componentes corresponde a um método do programa. A primeira, é necessária para inicializar as variáveis de estado do sistema e fazer as atribuições dos valores iniciais e dos parâmetros com os valores recebidos via *Command Line* ou via ficheiro de texto. A segunda, *Observe*, é necessária para monitorizar o estado do sistema e adicionar os novos resultados calculados no método *Update* à lista de resultados, previamente inicializada. Por fim, a componente *Update* irá implementar o modelo propriamente dito e atualizar as variáveis de estado em cada *time step*.

Como foi supramencionado, o método a implementar na primeira alínea é o *Forward Euler*, o método mais simples de integração numérica de equações diferenciais. Para o implementar, é apenas necessário incrementar as variáveis de estado com o resultado das fórmulas da Figura 5, multiplicando o seu valor por *Δt*.

```
def update():
    global x, y, a, b, d, g, dt
    x_new = x + (a * x - b * x * y) * dt
    y_new = y + (d * x * y - g * y) * dt
    x, y  = x_new, y_new
```

Fig. 9 - Método *update* que traduz o método *Forward Euler* (sendo a, alpha; b, beta; d, delta; g, gamma e dt, *Δt*)

A interpretação deste método é bastante direta. As populações atuais e os parâmetros são utilizados para calcular *x_new* e *y_new*, os novos valores das populações, com base nas equações diferenciais, sendo as variáveis de estado *x* e *y* posteriormente atualizadas.

Para a segunda alínea, na qual é pedido para fazer a simulação da evolução das populações usando o método *Runge-Kutta*, a abordagem seguida foi a mesma, utilizaram-se as mesmas três componentes, *Initialize*, *Observe* e *Update*. O método de *Runge-Kutta* também é utilizado para resolver equações diferenciais ordinárias. É, no entanto, mais complexo do que o método *Forward Euler*, mas, em contrapartida, fornece uma maior precisão nos resultados estimados. O método que implementámos em específico é o *RK4*, cuja fórmula se encontra na próxima figura:

$$\boxed{y(x+h) = y(x) + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4)}$$

where

$$F_1 = hf(x, y)$$

$$F_2 = hf\left(x + \frac{h}{2}, y + \frac{F_1}{2}\right)$$

$$F_3 = hf\left(x + \frac{h}{2}, y + \frac{F_2}{2}\right)$$

$$F_4 = hf(x + h, y + F_3)$$

Fig. 10 - Método *Runge-Kutta*, de 4ª ordem

Os métodos *initialize* e *observe* são iguais para ambos os problemas, sendo apenas o método *update* que os distingue. De maneira a facilitar a leitura e interpretação deste método, foram criados dois novos métodos auxiliares, *f_x* e *f_y,* que retornam o resultado das duas equações diferenciais:

```
def f_x(x, y):
    global a, b
    return a * x - b * x * y

def f_y(x, y):
    global d, g
    return d * x * y - g * y
```

Fig. 11 - Métodos *f_x* e *f_y* que retornam o resultado das equações diferenciais

Assim sendo, vamos analisar o método *update* para o segundo programa:

```python
def update():
    global x, y, a, b, d, g, dt

    k1_x = dt * f_x(x, y)
    k1_y = dt * f_y(x, y)

    k2_x = dt * f_x(x + dt / 2, y + k1_y / 2)
    k2_y = dt * f_y(x + dt / 2, y + k1_y / 2)

    k3_x = dt * f_x(x + dt / 2, y + k2_y / 2)
    k3_y = dt * f_y(x + dt / 2, y + k2_y / 2)

    k4_x = dt * f_x(x + dt, y + k3_y)
    k4_y = dt * f_y(x + dt, 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
```

Fig. 12 - Método *update* que traduz o método *Runge-Kutta* (sendo a, alpha; b, beta; d, delta; g, gamma e dt, Δt)

Como podemos observar, este método segue a lógica exibida na Figura 9. Começamos com o primeiro *step*, definido por *k1_x* e *k1_y*, que na realidade corresponde ao método *Forward Euler*. De seguida, calculamos o segundo *step*, definido por *k2_x* e *k2_y,* tal como descrito na fórmula, o *x* passa a ser *x + dt / 2*, sendo *dt* o nosso *h*, e o *y* passa a ser *k1_y / 2*. O terceiro *step* é igual ao segundo, com a substituição de *k1_y* por *k2_y*. No último *step*, tal como ilustrado na Figura 9, não dividimos *dt* e *k3_y* por 2. Finalmente, podemos calcular os novos valores das populações, seguindo a fórmula.

Assim, resumimos os passos essenciais para a resolução dos dois programas.

É importante notar que, no enunciado do mini-projeto, é referido que os valores iniciais e os parâmetros podem ser especificados via *Command Line* ou num ficheiro de texto. De maneira a satisfazer este requisito, foi utilizada a biblioteca de *Python*, *argparse*, que permite fazer *parse* dos argumentos passados na *Command Line* e atribuir-lhe valores *default*, caso não sejam passados. Foram criados os seguintes argumentos:

```python
# Initialize the parser and parse the arguments from CLI
parser = argparse.ArgumentParser()

parser.add_argument("--x0", type=int, default=10, help="Initial population of prey")
parser.add_argument("--y0", type=int, default=10, help="Initial population of predators")
parser.add_argument("--a", type=float, default=0.1, help="Parameter alpha")
parser.add_argument("--b", type=float, default=0.02, help="Parameter beta")
parser.add_argument("--d", type=float, default=0.02, help="Parameter delta")
parser.add_argument("--g", type=float, default=0.4, help="Parameter gamma")
parser.add_argument("--dt", type=float, default=0.1, help="Parameter delta_t")
parser.add_argument("--tf", type=int, default=5000, help="Parameter tfinal")
parser.add_argument("--f", type=str, default=None, help="Path to the input file")
```

Fig. 13 - Inicialização do *Parser* e criação dos argumentos

É possível, ao correr o programa na *Command Line*, passar as flags “-h” ou “--help” de maneira a imprimir o menu de ajuda, que explica cada um dos argumentos e fornece um exemplo genérico de como correr o programa.

Após receber os argumentos, estes são transformados num dicionário e o programa verifica se o argumento “--f” foi passado. Se sim, o ficheiro passado como argumento é aberto e os valores iniciais e parâmetros são devidamente importados. Se o ficheiro não existir, o programa imprime o erro e termina. Caso o argumento “--f” não seja passado, serão utilizados os argumentos passados na *CLI* ou os argumentos *default*.

É importante referir que os parâmetros definidos no ficheiro têm prioridade sobre os parâmetros passados na *CLI*, pelo que se todos os argumentos forem passados (incluindo o do ficheiro), os da *CLI* serão ignorados e serão utilizados os do ficheiro.

A estrutura do ficheiro deve obedecer ao seguinte modelo (com os parâmetros na mesma ordem):

```
x0 = 10
y0 = 10
a = 0.1
b = 0.02
d = 0.02
g = 0.4
dt = 0.1
tf = 5000
```

Fig. 14 - Estrutura do ficheiro de importação de parâmetros

Por fim, são criados os gráficos para auxiliar na análise da solução concebida.

3.2. Resultados obtidos

De maneira a analisar os resultados obtidos em cada programa, foram criados dois gráficos em cada programa. O primeiro analisa a evolução das populações de presas e predadores ao longo do tempo, o segundo analisa a evolução da população de predadores (y) em relação à evolução da população de presas (x).

Para estes casos específicos, foram usados os mesmos valores iniciais e parâmetros para os dois programas: *x0 = 10, y0 = 10, a = 0.1, b = 0.02, d = 0.02, g = 0.4, dt = 0.1* e *tf = 5000.* Estes parâmetros foram escolhidos após a análise de diversos *papers* sobre o tema (disponíveis nas **Referências**) e a realização de vários testes com diferentes parâmetros.

Para o método *Forward Euler* obtivemos os seguintes gráficos:

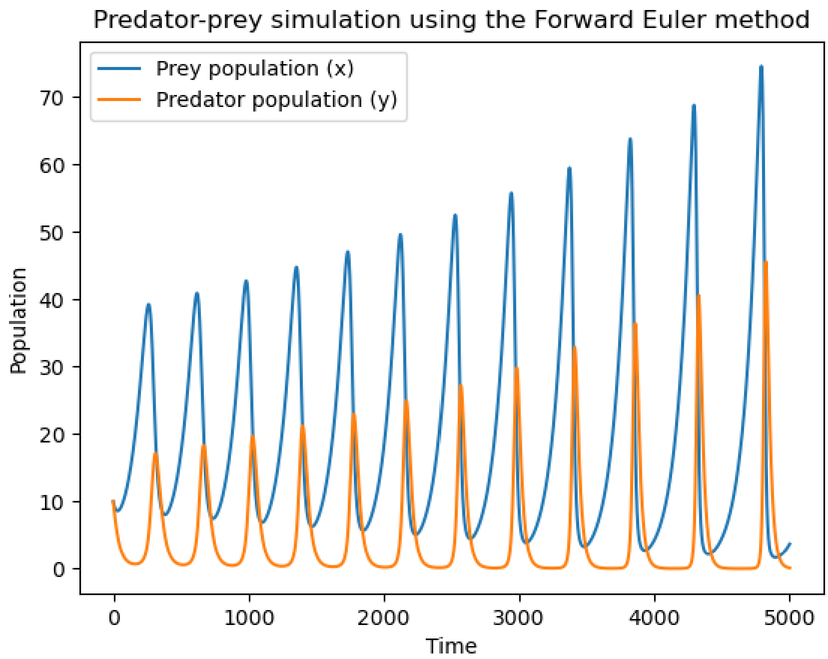


Fig. 15 - Gráfico da evolução das populações ao longo do tempo para o método *Forward Euler*

Ao analisar este gráfico, rapidamente concluímos que o número de predadores aumenta com o aumento do número de presas, embora com menos intensidade. É também percetível que, com o aumento do número de predadores, o número de presas diminui e, consequentemente, o número de predadores também irá baixar. A tendência repete-se ao longo do tempo. Alcançou-se uma população máxima de presas de cerca de 75 e uma população máxima de predadores de cerca de 45.

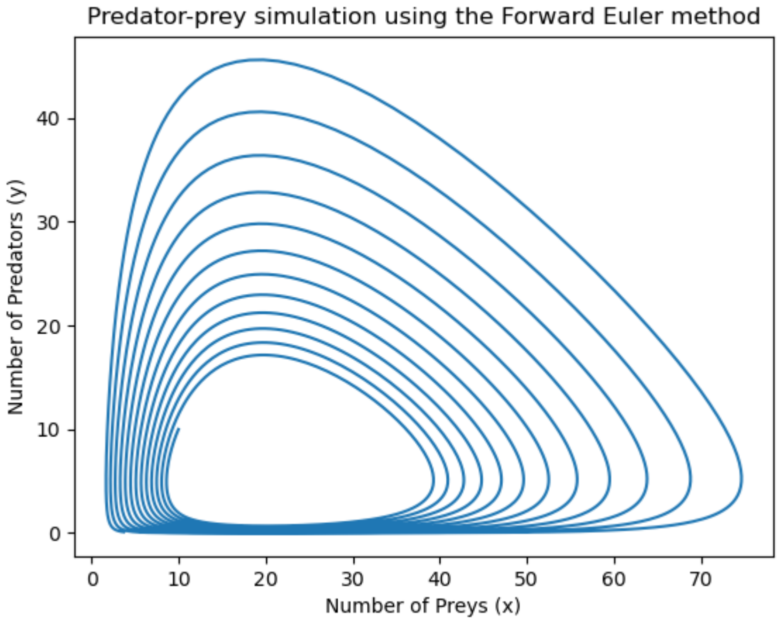


Fig. 16 - Gráfico do número de predadores em ordem ao número de presas para o método *Forward Euler*

A análise deste gráfico é um pouco mais complexa. Conseguimos inferir que a relação entre presas e predadores é algo cíclica, dividida por “fases” que correspondem a cada uma das “ovais” que conseguimos observar. Quando a população de presas é alta (eixo x), os predadores têm comida abundantemente e a população deles (eixo y) começa a aumentar. À medida que a população de predadores aumenta, eles começam a consumir excessivamente as presas, o que faz com que a população de presas diminua. De seguida, com a escassez de comida, a população de predadores começa a diminuir. Com essa diminuição, as presas reproduzem-se com maior facilidade, aumentando a sua população e voltando ao início do ciclo.

Para o método *Runge-Kutta* obtivemos os seguintes gráficos:

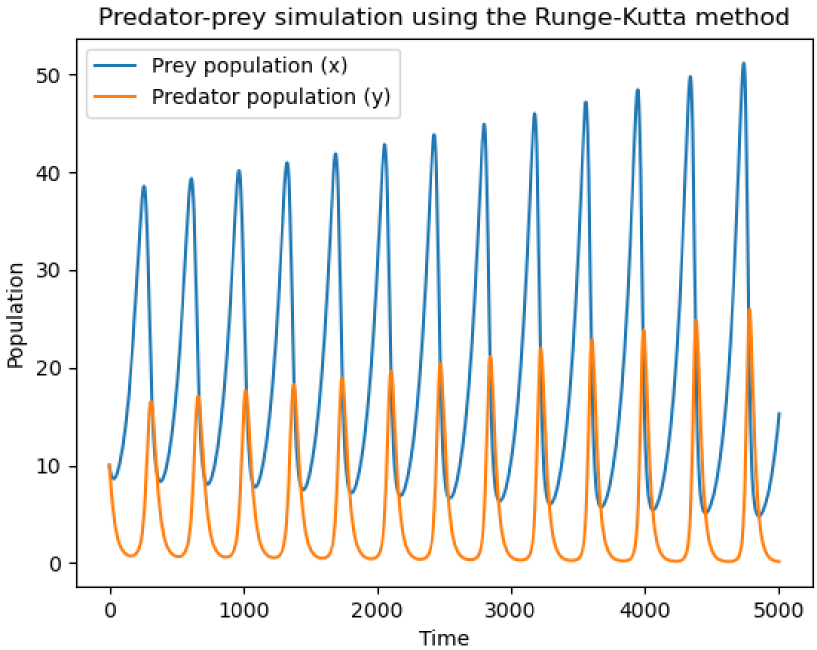


Fig. 17 - Gráfico da evolução das populações ao longo do tempo para o método *Runge-Kutta*

A análise deste gráfico é semelhante à do gráfico da Figura 14, com a exceção de que o número máximo de presas obtido foi de cerca de 50 e o número máximo de predadores obtido foi cerca de 25.

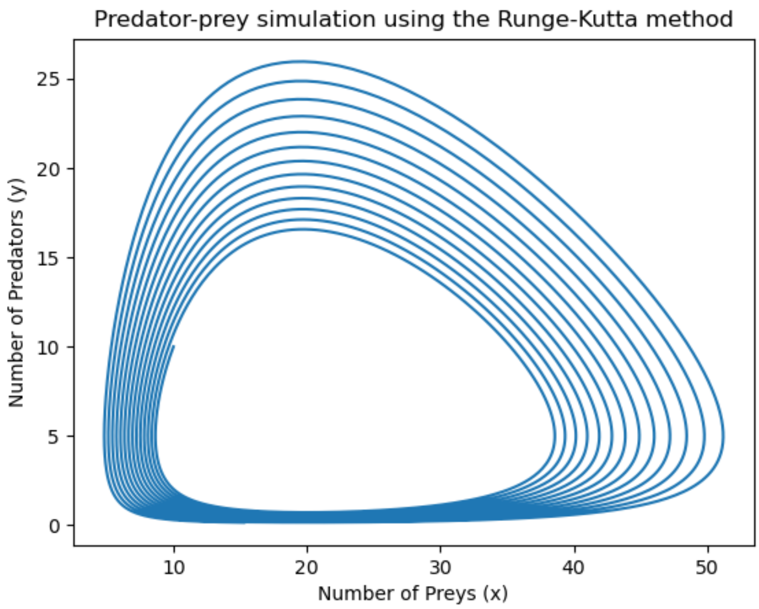


Fig. 18 - Gráfico do número de predadores em ordem ao número de presas para o método *Runge-Kutta*

Ao analisar este gráfico, denotamos alguma semelhança entre a forma do mesmo e a forma do gráfico da Figura 15. No entanto, é notória a diferença entre o tamanho da “abertura” de ambos os gráficos. De acordo com o que estudámos, as estimativas obtidas no gráfico da Figura 17 são mais confiáveis, uma vez que as aproximações do método *Runge-Kutta* são mais precisas do que as do método *Forward Euler*.

4. Referências

[1] - determination of the parameters in lotka-volterra equations from population measurements—algorithms

[2] - Lotka-Volterra Predator-Prey Model

[3] - Lotka-Volterra Systems

[4] - Parameters Estimation of a Lotka-Volterra Model in an Application for Market Graphics Processing Units