



## **ESCOLA POLITÉCNICA**

**MAP3122 - Métodos Numéricos e Aplicações -  
Quadrimestral 2021**

---

# **Exercício Computacional 2- Métodos numéricos para resolução de EDOs**

---

**Professor Dr. Antoine Laurain**

**Engenharia da Computação - Escola Politécnica da USP**

**Felipe Bagni** **11257571**

**Gabriel Yugo Nascimento Kishida** **11257647**

**São Paulo**  
**Abril de 2021**

# Sumário

<b>1 Resumo</b>	<b>3</b>
<b>2 Introdução</b>	<b>4</b>
2.1 Método de Euler Explícito . . . . .	4
2.2 Método de Euler Implícito . . . . .	5
2.3 Método de Runge-Kutta de ordem 4 . . . . .	6
<b>3 Exercícios</b>	<b>7</b>
<b>4 Implementação dos Métodos de Solução das EDOs</b>	<b>7</b>
4.1 Método de Euler Explícito . . . . .	7
4.2 Método de Euler Implícito . . . . .	8
4.3 Método de Runge-Kutta de ordem 4 . . . . .	11
<b>5 Exercício 1 - Testes</b>	<b>12</b>
5.1 Enunciado . . . . .	12
5.2 Implementação da solução . . . . .	13
5.3 Teste 1 . . . . .	14
5.4 Teste 2 . . . . .	24
<b>6 Exercício 02 - Modelo presa-predador</b>	<b>28</b>
6.1 Enunciado . . . . .	28
6.2 Implementação da solução . . . . .	29
6.3 Resultados . . . . .	29
6.4 Interpretação e Discussão dos Resultados Obtidos . . . . .	35
<b>7 Exercício 3 - Modelo duas presas-um predador</b>	<b>35</b>
7.1 Enunciado . . . . .	35
7.2 Implementação da solução . . . . .	37
7.3 Análise do Comportamento . . . . .	37
7.4 Resultados . . . . .	38
7.5 Teste de Sensibilidade . . . . .	68
<b>8 Conclusão e Comentários Finais</b>	<b>72</b>

<b>9 Apêndice</b>	<b>73</b>
9.1 Código completo plotter.py . . . . .	73
9.2 Código completo euler_backward.py . . . . .	78
9.3 Código completo euler_forward.py . . . . .	81
9.4 Código completo rk4.py . . . . .	83
9.5 Código completo ejercicio1.py . . . . .	85
9.6 Código completo ejercicio2.py . . . . .	89
9.7 Código completo ejercicio3.py . . . . .	94

# **1 Resumo**

Este documento tem por objetivo apresentar os aspectos de desenvolvimento e relatar os resultados obtidos para o problema computacional proposto de Métodos numéricos para resolução de EDOs da disciplina de MAP3122 - Métodos Numéricos e Aplicações. A partir da divisão das atividades propostas, foram apresentadas as soluções, discussões e mídias pertinentes de acordo com o documento de enunciado apresentado no Moodle da disciplina e os materiais referentes a este experimento.

## 2 Introdução

Este exercício computacional tem como objetivo o estudo de diferentes métodos de resolução de Equações Diferenciais Ordinárias (EDOs) e a aplicação dos mesmos na resolução do problema de Modelo Presa-Predador, tais problemas serão discutidos nas próximas seções.

Todos os problemas a serem resolvidos são **problemas de valor inicial** (PVI), onde temos as seguintes informações:

$$y'(t) = f(t, y(t)) \quad (1)$$

$$y(t_0) = \alpha \quad (2)$$

Em alguns dos casos, há um Sistema de Equações Diferenciais Ordinárias, constituindo então um **Sistema de Problemas de Valores Iniciais** (SPVI). A única diferença é que, ao invés de um único valor  $y(t)$ , existe um vetor de valores  $u(t)$ , um vetor de valores iniciais  $\alpha$ . Por fim, há também uma série de funções  $f_0(t, u(t))$ ,  $f_1(t, u(t))$ , ...  $f_n(t, u(t))$  onde  $n$  é a quantidade de variáveis contidas em  $u(t)$ .

Assim, ao invés de uma equação escalar, se terá uma equação matricial do tipo:

$$u'(t) = f(t, u(t)) \quad (3)$$

$$u(t_0) = \alpha \quad (4)$$

Agora, serão analisados diferentes métodos para resolver esses **problemas de valor inicial**.

### 2.1 Método de Euler Explícito

O método de Euler explícito consiste em um método numérico de primeira ordem utilizado para resolver Equações Diferenciais Ordinárias com valor inicial dado – sendo assim, um método para resolver **problemas de valor inicial** (PVI).

Definindo que:

$$u_k = u(t_0 + hk) \quad (5)$$

Onde o valor  $h$  é o passo temporal, definido por:

$$h = \frac{t_f - t_0}{n} \quad (6)$$

Com  $t_0$  tempo inicial,  $t_f$  tempo final,  $n$  número de passos/iterações.

Já expandindo para a forma matricial, a equação principal para o Método de Euler Explícito é:

$$u_{k+1} = u_k + h f(t_k, u_k) \quad (7)$$

Para cada  $k$  entre 0 e  $n$ . Dessa forma, partindo do valor  $u_0$ , é possível obter valores para cada iteração, obtendo valores pontuais e conseguindo obter uma aproximação da função  $u(t)$ .

## 2.2 Método de Euler Implícito

Para o Euler Implícito, temos também um método iterativo. No entanto, desta vez, a equação principal é:

$$u_{k+1} = u_k + h f(t_{k+1}, u_{k+1}) \quad (8)$$

No entanto, parece um tanto contraditório obter o valor de  $u_{k+1}$  utilizando como argumento  $f(t_{k+1}, u_{k+1})$ . O valor que é passado para dentro desta função  $f$ , e utilizado para o cálculo de  $u_{k+1}$  é, na verdade, uma aproximação obtida por meio de outro método, que se chama de Método de Newton.

### Método de Newton

Para obter uma aproximação de  $u_{k+1}$ , precisa-se zerar a seguinte equação:

$$G(u_{k+1}) = u_{k+1} - h f(t_{k+1}, u_{k+1}) - u_k \quad (9)$$

Utilizando o Método de Newton para calcular raízes de funções, e escolhendo um valor inicial  $u_{k+1}^{(l)}$ , podemos calcular:

$$u_{k+1}^{(l+1)} = u_{k+1}^{(l)} - J(t_{k+1}, u_{k+1}^{(l)})^{-1} G(t_{k+1}, u_{k+1}^{(l)}) \quad (10)$$

Onde a função  $J$  é a matriz Jacobiana da função  $G$ .

Assim, com algumas iterações, conseguimos uma aproximação de  $u_{k+1}$  e conseguimos utilizar o método de Euler Implícito.

### 2.3 Método de Runge-Kutta de ordem 4

O método de Runge-Kutta também é um método iterativo, e para ele, utiliza-se a seguinte equação principal:

$$u_{k+1} = u_k + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4) \quad (11)$$

Onde os valores  $K_1$ ,  $K_2$ ,  $K_3$  e  $K_4$  são obtidos por meio de:

$$K_1 = hf(t_k, u_k) \quad (12)$$

$$K_2 = hf\left(t_k \frac{h}{2}, u_k + h \frac{K_1}{2}\right) \quad (13)$$

$$K_3 = hf\left(t_k + \frac{h}{2}, u_k + h \frac{K_2}{2}\right) \quad (14)$$

$$K_4 = hf(t_k + h, u_k + hK_3) \quad (15)$$

Vale lembrar que, se  $u(t)$  for um vetor, então os valores  $K_1$ ,  $K_2$ ,  $K_3$  e  $K_4$  também o são.

## 3 Exercícios

Os programas dos exercícios computacionais foram escritos em python 3, na versão 3.8.5, usando o pacote **numpy**. A entrada e saída foram feitas de forma a ajudar o usuário a executar o programa e facilitar a análise dos resultados. E a biblioteca **matplotlib** foi usada para as plotagens. Portanto, para rodar os exercícios, é necessário ter python3 instalado no computador e as dependências para a solução, que são o pacote **numpy** e a biblioteca **matplotlib**.

Para a realização do exercício foram criados os arquivos **exercicio1.py**, **exercicio2.py** e **exercicio3.py**, sendo cada um respectivo a um dos exercícios. Além disso, foram criados os arquivos **rk4.py**, **euler\_forward.py**, **euler\_backward.py** e **plotter.py** contendo a implementação do seu respectivo método de resolução de EDO e a última contém métodos de plotagem de gráficos comuns aos exercícios. Para se rodar o exercício programa, estando no mesmo diretório do arquivo, deve-se rodar no terminal:

```
1 $ python3 [NOME DO ARQUIVO DO PROGRAMA]
```

Você deve substituir [NOME DO ARQUIVO DO PROGRAMA] por **exercicio1.py** ou **exercicio2.py** ou **exercicio3.py**, de acordo com o exercício que se deseja rodar.

Então para rodar o programa **exercicio1.py**, por exemplo, basta fazer:

```
1 $ python3 exercicio1.py
```

Para mais informações, segue o link do repositório no GitHub com mais informações e os arquivos desenvolvidos:

- [Link para o Repositório](#)

## 4 Implementação dos Métodos de Solução das EDOs

Seguem aqui as funções que implementam os métodos supracitados em seu arquivo correspondente.

### 4.1 Método de Euler Explícito

```
1 def forwardEuler(f, yinit, I, n):
2     """
3         This function/module performs the forward Euler method steps.
4     """
```

```

5     m = len(yinit) # Number of ODEs
6
7     h = (I[1] - I[0])/n
8
9     x = I[0] # Initializes variable x
10    y = yinit.copy() # Initializes variable y
11
12    xsol = np.empty(0) # Creates an empty array for x
13    xsol = np.append(xsol, x) # Fills in the first element of xsol
14
15    ysol = []
16    ysol.append(y.copy()) # Fills in the initial conditions
17
18
19    for _ in range(n):
20
21        for j in range(m):
22            y[j] = y[j] + h*f[j](x, y) # Eq. (8.2)
23
24        x += h # Increase x-step
25        xsol = np.append(xsol, x) # Saves it in the xsol array
26        ysol.append(y.copy()) # Saves all new y's
27
28    return [xsol, ysol]

```

Listing 1: Implementação do método de Euler explícito em `euler_forward.py`

## 4.2 Método de Euler Implícito

```

1 def partial_derivative(t, u, partial_in, f, step):
2     """
3         Brief : Essa fun ao calcula a derivada parcial de uma das funcoes
4             de f (sendo
5                 f um vetor de funcoes f1, f2, ...). A derivada parcial
6             derivada na
7                 variavel de indice "partial_in" e     feita de forma
8             num rica (para um step
9                 pequeno).
10            Par metros: t - valor de tempo,
11                u - variaveis do sistema,
12                partial_in - indice (de "u") da variavel sobre a qual
13                ser    derivada f

```

```

10             f - vetor de funcoes, uma das quais ser derivada,
11             step - O tamanho do passo que define a precisao da
12             derivada numerica.
13             Retorna:    Valor da derivada parcial para os determinados valores
14             de "input".
15             """
16             step_u = u.copy()
17             step_u[partial_in] += step
18             return (f(t,step_u) - f(t,u))/step
19
20 def newton_iter(t, u, f, h, last_u):
21     """
22         Brief : Essa funcao calcula uma iteracao do m todo de aproximacao
23         de Newton para
24             obter U_k+1 - com o objetivo de se apurar o m todo
25             implementado de Euler.
26             Par metros: t - valor de tempo,
27                     u - U_k+1 (antes da iteracao) variaveis do sistema,
28                     f - vetor de funcoes, uma das quais ser derivada,
29                     h - tamanho do passo temporal percorrido a cada
30             iteracao,
31                     last_u - U_k variaveis do sistema.
32             Retorna:    Valor de U_k+1 ap s a iteracao de Newton
33             """
34             n = len(u)
35             jacobian = np.identity(n)
36             G = np.zeros(len(u))
37             new_t = t + h
38             for i in range(n):
39                 for j in range(n):
40                     jacobian[i][j] -= h*partial_derivative(new_t, u, j, f[i],
41                     0.001)
42             inv_jacobian = np.linalg.inv(jacobian)
43             for i in range(len(u)):
44                 G[i] = u[i] - h * f[i](t,u) - last_u[i]
45             return u - np.matmul(inv_jacobian, G) # == new_u
46
47 def implicit_euler_iter(u, t, h, f, newton_iter_num):
48     """
49         Brief : Essa funcao calcula uma iteracao do m todo de aproximacao
50         de Newton para
51             obter U_k+1 - com o objetivo de se apurar o m todo

```

```

impl cito de Euler.

45 Par metros: u - U_k (antes da iteracao) variaveis do sistema,
46             t - valor de tempo,
47             h - tamanho do passo temporal percorrido a cada
48 iteracao,
49             f - vetor de funcoes, uma das quais ser derivada,
50             newton_iter - numeros de iteracao de newton que serao
51 feitas.
52 Retorna:    Valor de U_k+1 para a iteracao de Euler Impl cito
53 """
54
55     newton_u = u.copy()
56     euler_u = u.copy()
57     for _ in range(newton_iter_num) :
58         newton_u = newton_iter(t, newton_u, f, h, u)
59     for i in range(len(u)):
60         euler_u[i] = u[i] + h * f[i](t, newton_u)
61     return euler_u
62
63 def implicit_euler_system(u, f, t0, tf, n, newton_iter_num):
64 """
65     Brief : Essa funcao calcula iteracoes do metodo de Euler
66     impl cito para um
67             sistema de funcoes.
68     Par metros: u - U_0 valores iniciais do sistema,
69                 f - vetor de funcoes que criam o sistema,
70                 t0 e tf - tempos iniciais e finais da iteracao,
71                 n - numero de iteracoes do metodo de Euler a serem
72 realizadas
73                 newton_iter_num - numeros de iteracao de newton que
74 serao feitas.
75     Retorna : Valores de u para cada passo temporal calculado.
76 """
77     u_values = []
78     new_u = u.copy()
79     u_values.append(new_u)
80     h = (tf-t0)/n
81     t = t0
82
83     tsol = np.empty(0) # Creates an empty array for t
84     tsol = np.append(tsol, t) # Fills in the first element of tsol
85
86     for _ in range(1, n+1):

```

```

81     new_u = implicit_euler_iter(new_u, t, h, f, newton_iter_num)
82     u_values.append(new_u)
83     t += h
84     tsol = np.append(tsol, t) # Saves it in the tsol array
85
86 return [tsol, u_values]

```

Listing 2: Implementação do método de Euler explícito em `euler_backward.py`

### 4.3 Método de Runge-Kutta de ordem 4

```

1 def rk4iter(u, t, h, f) :
2     """
3         Brief :Essa fun ao aplica o algoritmo Runge Kutta em uma unica
4             iteracao
5             Par metros: x - valor a ser iterado,
6                     t - valor de t a iterar,
7                     h - passo da fun ao ,
8                     f - funcao f para aplicar Runge Kutta.
9
10    """
11    K1 = np.zeros(len(f))
12    K2 = np.zeros(len(f))
13    K3 = np.zeros(len(f))
14    K4 = np.zeros(len(f))
15    for i in range(len(f)):
16        K1[i] = h*f[i](t,u)
17    for i in range(len(f)):
18        K2[i] = h*f[i](t + h/2,u + K1*0.5)
19    for i in range(len(f)):
20        K3[i] = h*f[i](t + h/2,u + K2*0.5)
21    for i in range(len(f)):
22        K4[i] = h*f[i](t+h,u + K3)
23    return u + (K1 + 2*K2 +2*K3 + K4)/6
24
25 def rk4system(u, f, t0, tf, n):
26     """
27         Brief: Essa fun ao aplica o algoritmo Runge Kutta resolvendo um
28             SPVI
29                     sendo o sistema de forma linear. O sistema pode ser
30             fornecido
31                     por meio da matriz A.

```

```

28     Par metros: u - valores iniciais,
29             A - matriz do sistema linear,
30             t0 - valor inicial de t,
31             tf - valor final de t,
32             n - numero de divisões entre t0 e tf.
33
34     """
35
36     newx = np.zeros(len(f))
37     x = u.copy()
38     rk4values = []
39     rk4values.append(x)
40     tsol = []
41     tsol.append(t)
42     for _ in range(1, n+1):
43         newx = rk4iter(x, t, h, f)
44         t = t + h
45         x = newx.copy()
46         tsol.append(t)
47         rk4values.append(x)
48
49     return [tsol, np.array(rk4values)]

```

Listing 3: Implementação do método de Runge-Kutta de ordem 4 em **rk4.py**

## 5 Exercício 1 - Testes

### 5.1 Enunciado

Adaptado para o relatório:

O objetivo deste exercício é de verificar se sua implementação de RK4 e Euler implícito está funcionando em caso simples onde uma solução explícita é conhecida. Nestes testes, para  $n \in \mathbb{N}$  dado, escolhemos uma discretização uniforme  $t_k = T_0 + kh$  de  $[T_0, T_f]$ , onde  $h = (T_f - T_0) / n$  e  $k \in \{1, 2, \dots, n\}$ , isto é, dividimos o intervalo  $[T_0, T_f]$  em  $n$  subintervalos do mesmo tamanho.

- 1. Teste Runge-Kutta 4.** Considere a equação  $x'(t) = f(t, x(t))$  no intervalo  $[0, 2]$  com  $x(0) = (1, 1, 1, -1) \in \mathbb{R}^4$  e  $f(t, x(t)) = Ax(t)$ , onde  $A \in \mathbb{R}^{4 \times 4}$  é uma matriz dada por:

$$A = \begin{pmatrix} -2 & -1 & -1 & -2 \\ 1 & -2 & 2 & -1 \\ -1 & -2 & -2 & -1 \\ 2 & -1 & 1 & -2 \end{pmatrix}$$

Esta EDO tem a solução explícita seguinte:

$$x^*(t) = \begin{pmatrix} e^{-t}\sin(t) + e^{-3t}\cos(3t) \\ e^{-t}\cos(t) + e^{-3t}\sin(3t) \\ -e^{-t}\sin(t) + e^{-3t}\cos(3t) \\ -e^{-t}\cos(t) + e^{-3t}\sin(3t) \end{pmatrix}$$

Definimos o erro  $E_{1,n}(t) := \max_{1 \leq i \leq 4} |x_i^*(t) - x_i(t)|$ , onde  $x(t)$  é a solução calculada com RK4 usando a subdivisão de  $[0, 2]$  em  $n$  subintervalos de mesmo tamanho. Para cada  $n = 20, 40, 80, 160, 320, 640$ , calcule a solução numérica  $x(t)$  desta EDO usando RK4, calcule e plote  $E_{1,n}(t)$  no relatório. Chamando  $n_1 = 20, n_2 = 40, n_3 = 80, n_4 = 160, n_5 = 320, n_6 = 640$ , calcule:

$$R_i := \frac{\max_{t \in [0,2]} E_{1,n_i}(t)}{\max_{t \in [0,2]} E_{1,n_{i+1}}(t)} \text{ para } i = 1, 2, 3, 4, 5$$

e interprete o resultado.

2. **Teste Euler implícito.** Considere a equação  $x'(t) = F(t, x(t))$  no intervalo  $[1.1, 3.0]$  com  $x(1.1) = -8.79$  e  $f(t, x) = 2t + (x - t^2)^2$ . A solução explícita desta equação é  $x^*(t) = t^2 + \frac{1}{1-t}$ . Definimos o erro  $E_2(t) := |x^* - x|$ , onde  $x(t)$  é a solução calculada com o método numérico. Resolva esta EDO usando o método de Euler implícito para  $n = 5000$  e plote três figuras, lado ao lado, a solução explícita  $x^*(t)$ , a solução numérica  $x(t)$  e o erro  $E_2(t)$  no relatório. [...] No método de Newton, 7 passos de Newton são suficientes para atingir uma precisão razoável para calcular  $x_{k+1}$ .

## 5.2 Implementação da solução

O código completo se encontra no apêndice: ver **exercicio\_1.py**

### 5.3 Teste 1

Para os diferentes valores de  $n$ , as soluções numéricas por RK4 e exatas são:

```
Exercício 1 - Teste 1:  
0 R para a iteracao i = 1 é : 17.96037346589822  
0 valor calculado x(Tf) para n = 20 é : [ 0.12543698 -0.05700548 -0.12068056 0.0556344 ]  
0 valor exato x*(Tf) para n = 20 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
  
0 R para a iteracao i = 2 é : 17.053544234097533  
0 valor calculado x(Tf) para n = 40 é : [ 0.12543982 -0.05701161 -0.12068007 0.05562715]  
0 valor exato x*(Tf) para n = 40 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
  
0 R para a iteracao i = 3 é : 16.54275737095762  
0 valor calculado x(Tf) para n = 80 é : [ 0.12544003 -0.05701193 -0.12068001 0.05562677]  
0 valor exato x*(Tf) para n = 80 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
  
0 R para a iteracao i = 4 é : 16.275095991051092  
0 valor calculado x(Tf) para n = 160 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
0 valor exato x*(Tf) para n = 160 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
  
0 R para a iteracao i = 5 é : 16.138299002216392  
0 valor calculado x(Tf) para n = 320 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
0 valor exato x*(Tf) para n = 320 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
  
0 valor calculado x(Tf) para n = 640 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]  
0 valor exato x*(Tf) para n = 640 é : [ 0.12544005 -0.05701195 -0.12068     0.05562675]
```

Figura 1: Exercício 1 - Teste 1 - Saída

Podemos ver os gráficos das soluções exatas e pelo método e o gráfico de  $E_{1,n}(t)$  para cada valor de  $n$ :

- $n = 20$ :

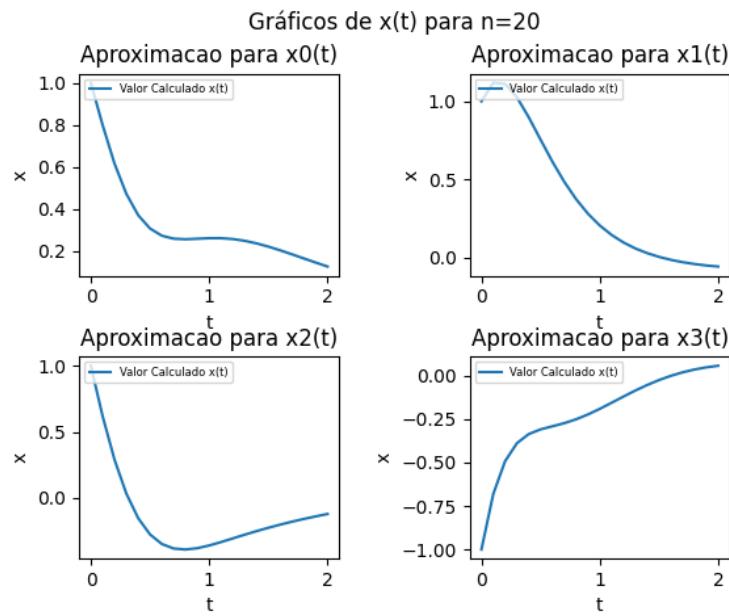


Figura 2: Exercício 1 - Teste 1 - Solução para  $n=20$

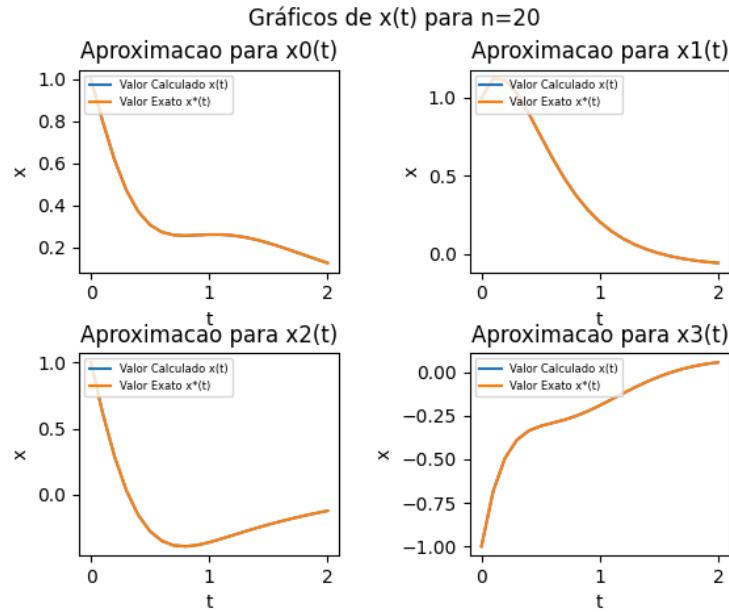


Figura 3: Exercício 1 - Teste 1 - Comparaçāo com a solução explícita para  $n=20$

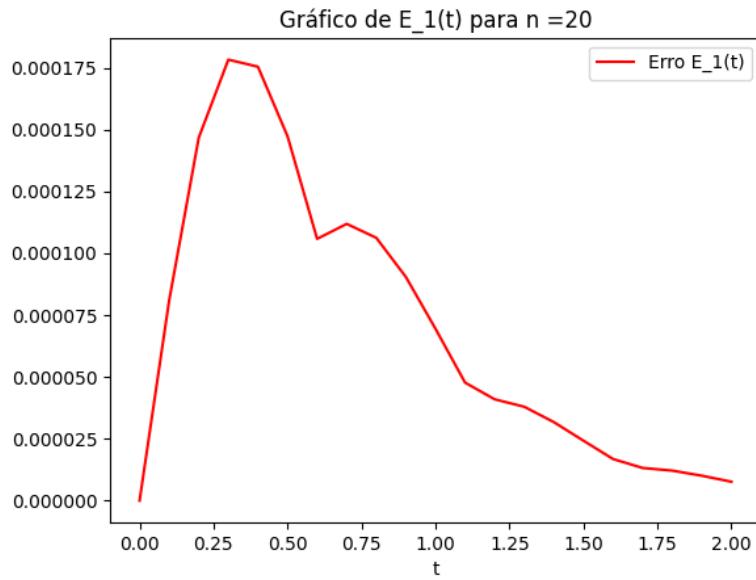


Figura 4: Exercício 1 - Teste 1 - Erro para  $n=20$

- $n = 40$ :

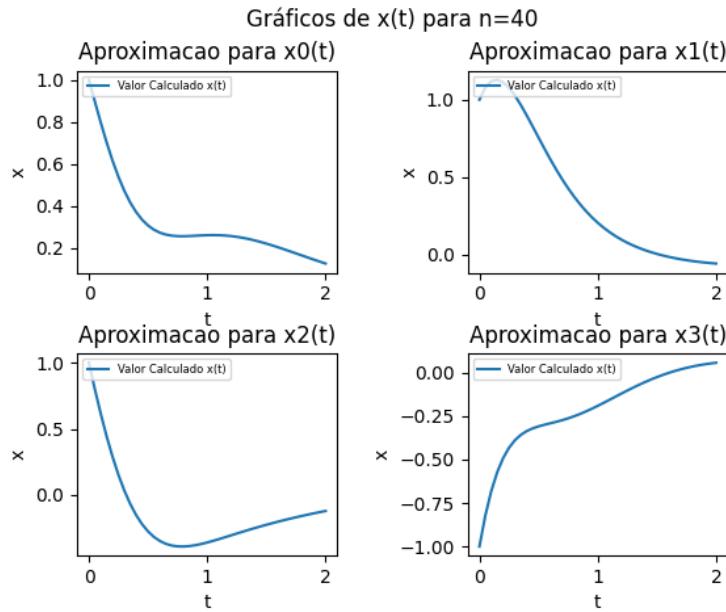


Figura 5: Exercício 1 - Teste 1 - Solução para  $n=40$

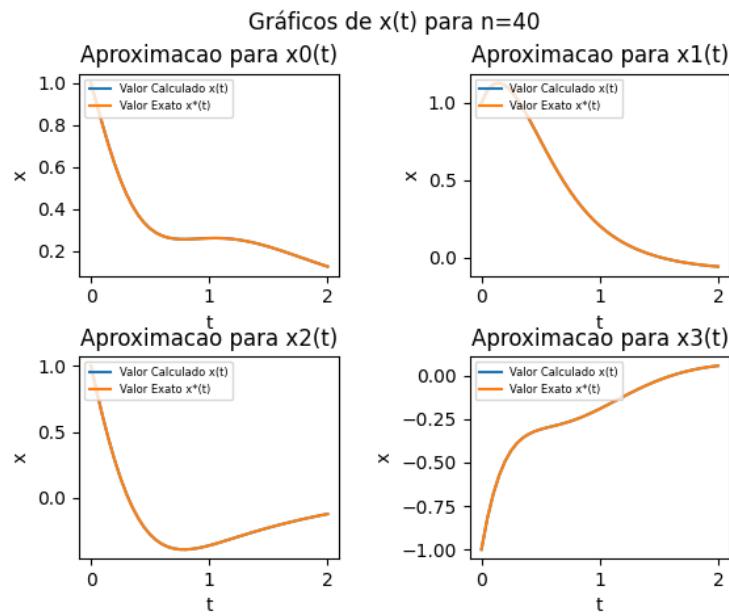


Figura 6: Exercício 1 - Teste 1 - Comparação com a solução explícita para  $n=40$

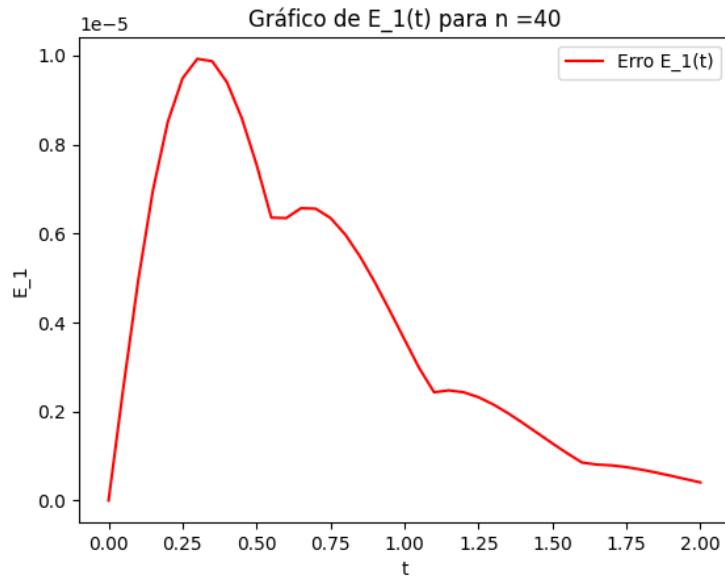


Figura 7: Exercício 1 - Teste 1 - Erro para  $n=40$

- $n = 80$ :

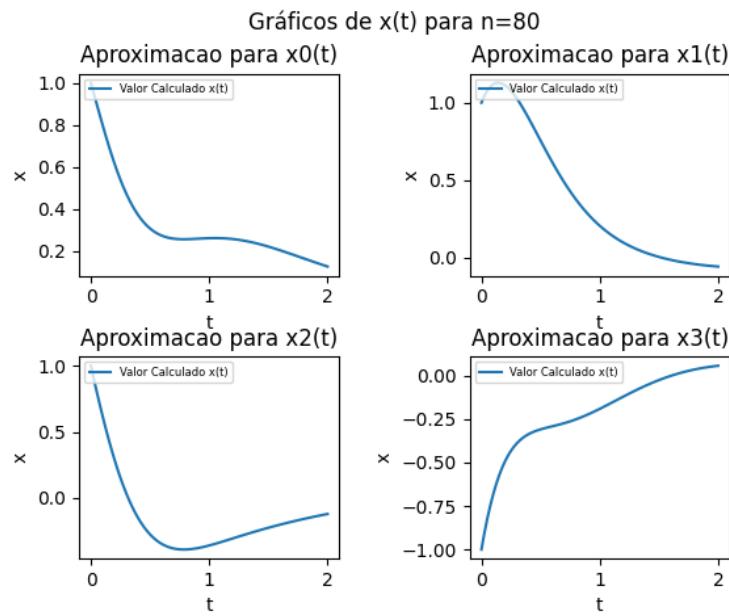


Figura 8: Exercício 1 - Teste 1 - Solução para  $n=80$

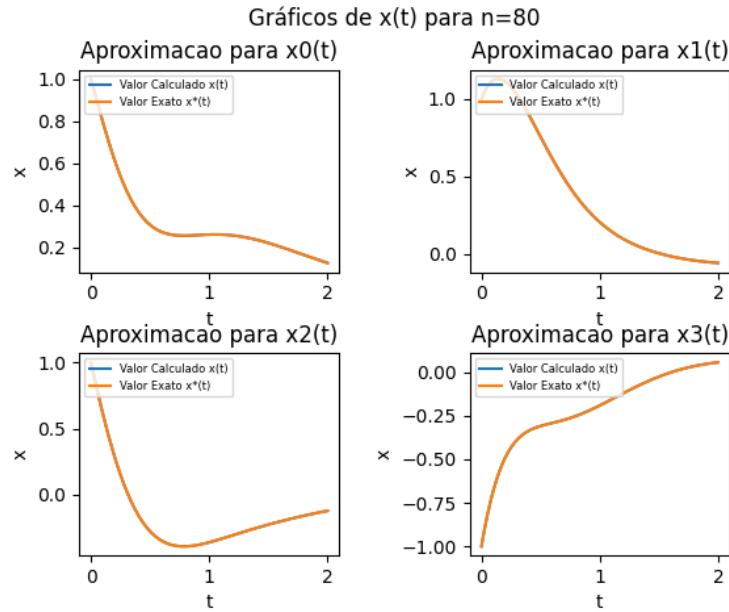


Figura 9: Exercício 1 - Teste 1 - Comparação com a solução explícita para  $n=80$

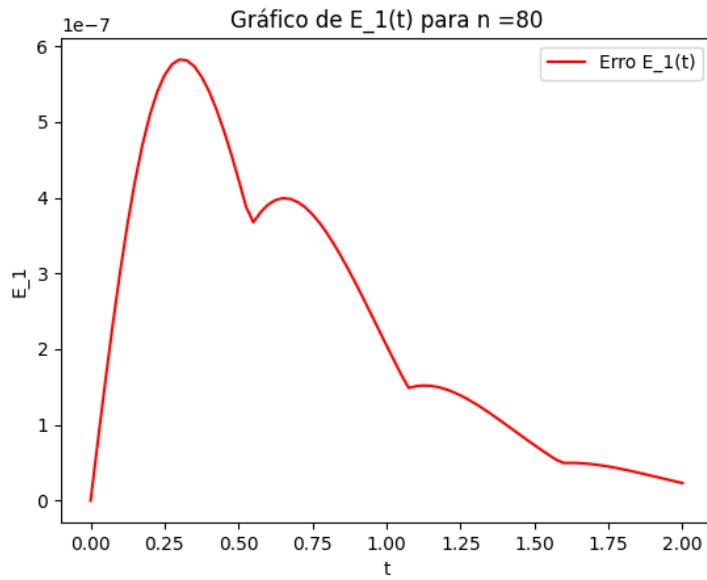


Figura 10: Exercício 1 - Teste 1 - Erro para  $n=80$

- $n = 160$ :

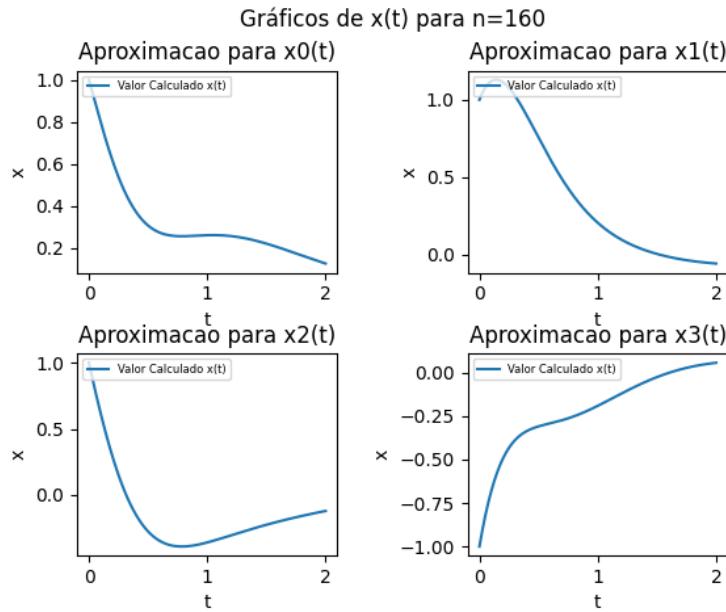


Figura 11: Exercício 1 - Teste 1 - Solução para  $n=160$

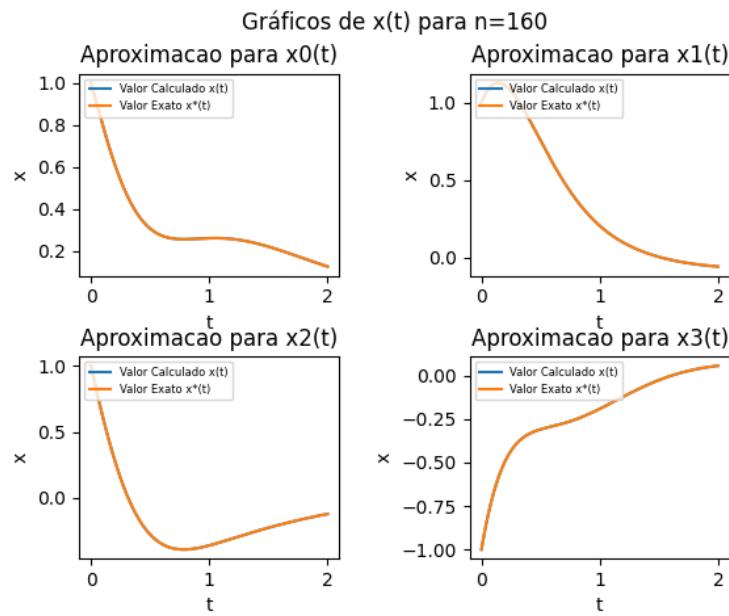


Figura 12: Exercício 1 - Teste 1 - Comparação com a solução explícita para  $n=160$

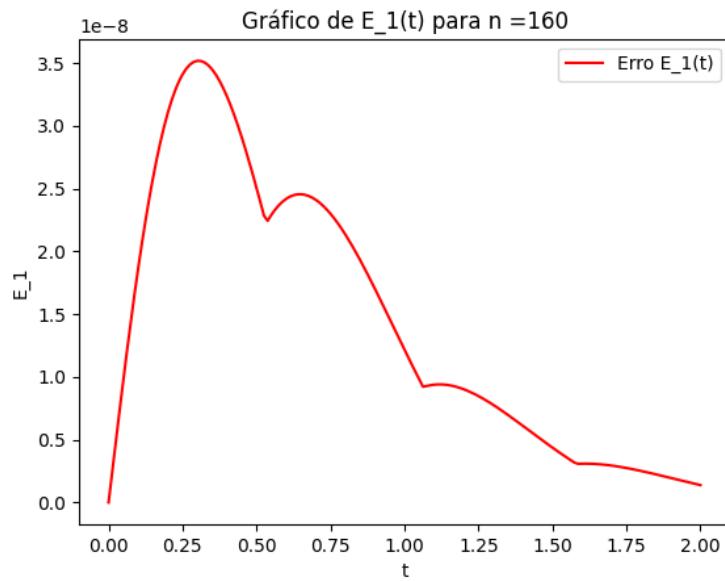


Figura 13: Exercício 1 - Teste 1 - Erro para  $n=160$

- $n = 320$ :

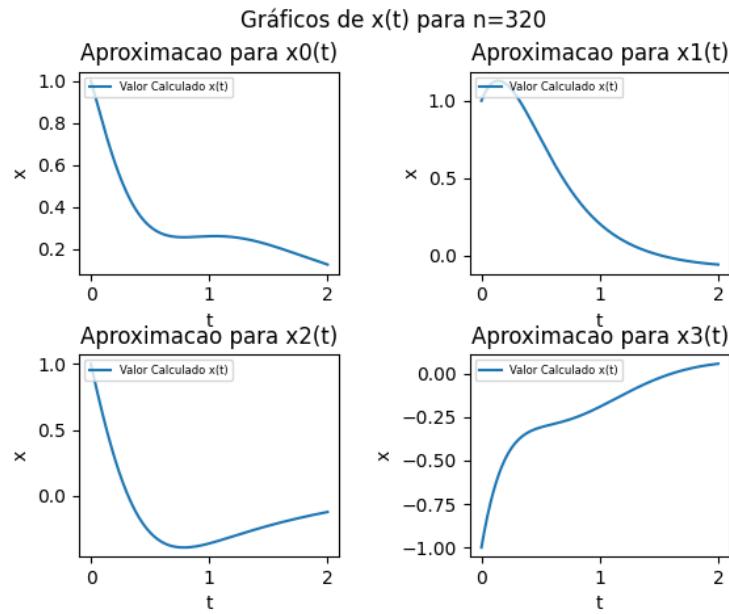


Figura 14: Exercício 1 - Teste 1 - Solução para  $n=320$

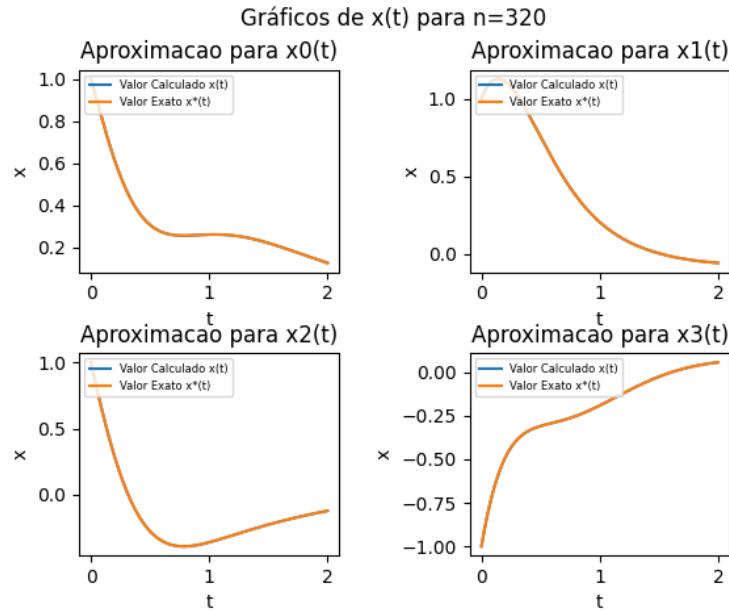


Figura 15: Exercício 1 - Teste 1 - Comparação com a solução explícita para  $n=320$

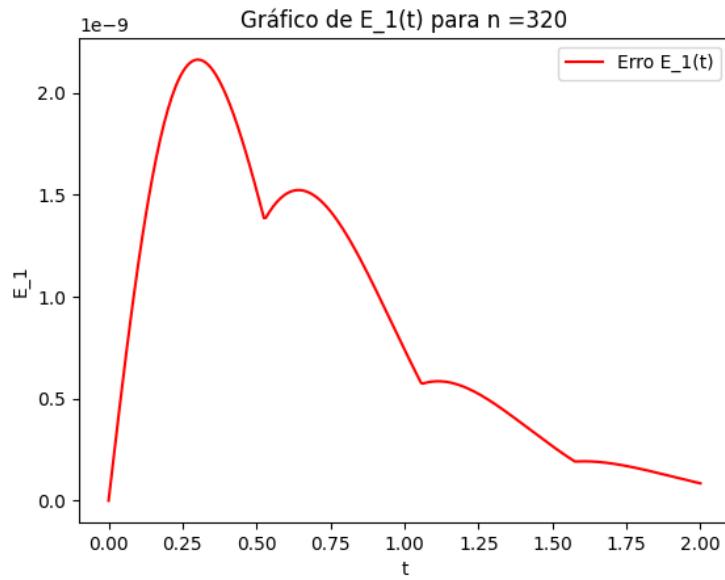


Figura 16: Exercício 1 - Teste 1 - Erro para  $n=320$

- $n = 640$ :

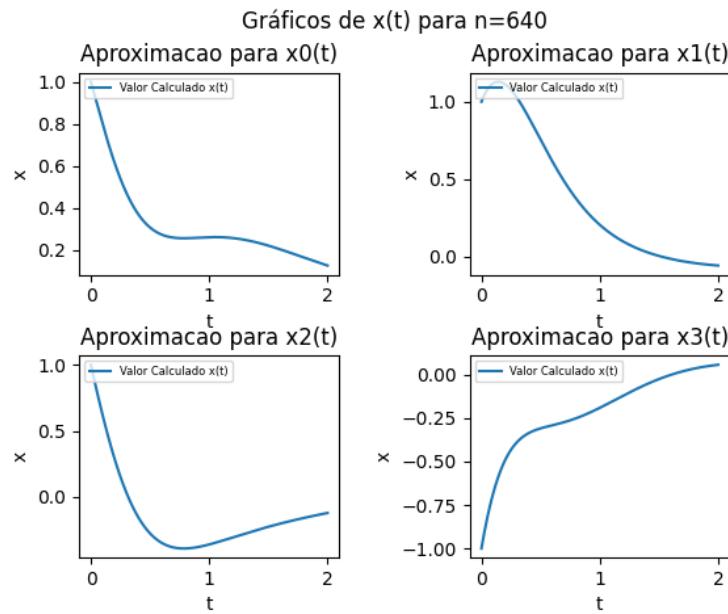


Figura 17: Exercício 1 - Teste 1 - Solução para  $n=640$

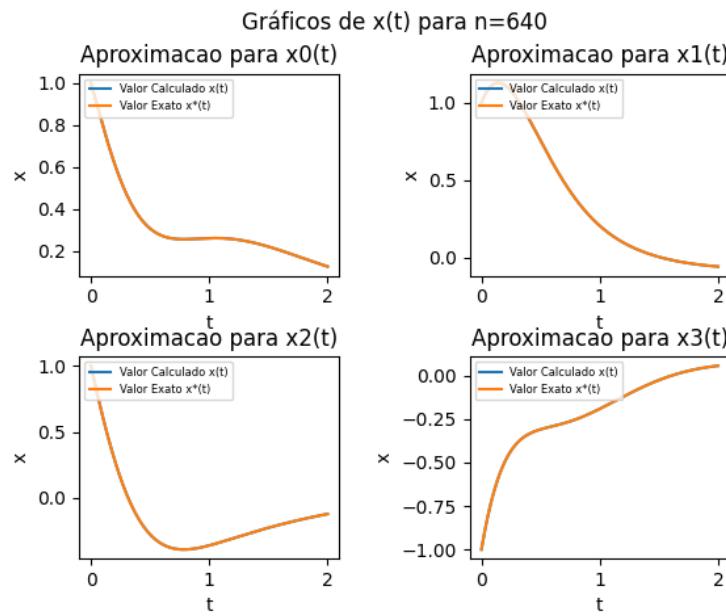


Figura 18: Exercício 1 - Teste 1 - Comparação com a solução explícita para  $n=640$

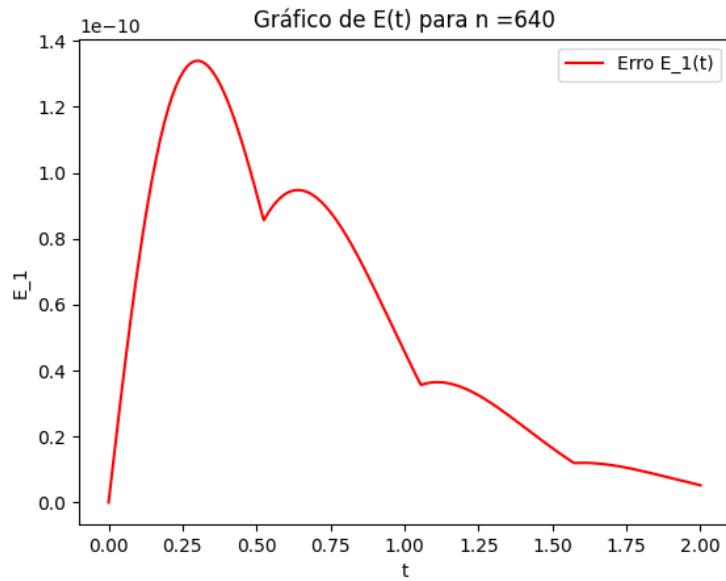


Figura 19: Exercício 1 - Teste 1 - Erro para  $n=640$

Segue aqui a tabela dos valores calculados de  $R_i$  para  $i = 1, 2, 3, 4, 5$ :

$i$	$n_i$	$R_i$
1	20	17.96037346589822
2	40	17.053544234097533
3	80	16.54275737095762
4	160	16.275095991051092
5	320	16.138299002216392

Tabela 1: Soluções de  $x(T_f)$  para os diferentes  $n$

### Interpretação dos resultados

Destes dados recolhidos, os tópicos mais importantes a serem notados são:

- A análise do sistema de equações diferenciais por Runge-Kutta foi efetiva, já que os resultados obtidos se assemelham à solução explícita;
- Conforme cresce "n", menor fica o erro em relação à resposta explícita;
- Conforme cresce "n", menor fica o R – isto é. que a taxa de precisão cresce cada vez menos conforme "n" cresce.
- Para as equações fornecidas, o erro do método RK4 diminui conforme se aproxima do  $t_f$ , tendo um grande pico de erro próximo ao  $t_0$ , e diminuindo conforme se aproxima do valor final. Isso pode ser decorrente do formato das funções dadas, e não inerente ao método RK4.

### 5.4 Teste 2

Sabendo que a solução exata em  $T_f$  é dada por:  $x*(T_f) = 8.5$

A solução em  $T_f$  para o método de Euler implícito é dado por:  $x(T_f) = 8.49769092$

Assim temos um erro relativo de 0.027165647%, assim, podemos considerar que a implementação foi bem sucedida.

Gráfico da solução pelo método de Euler implícito, solução explícita, os gráficos sobrepostos:

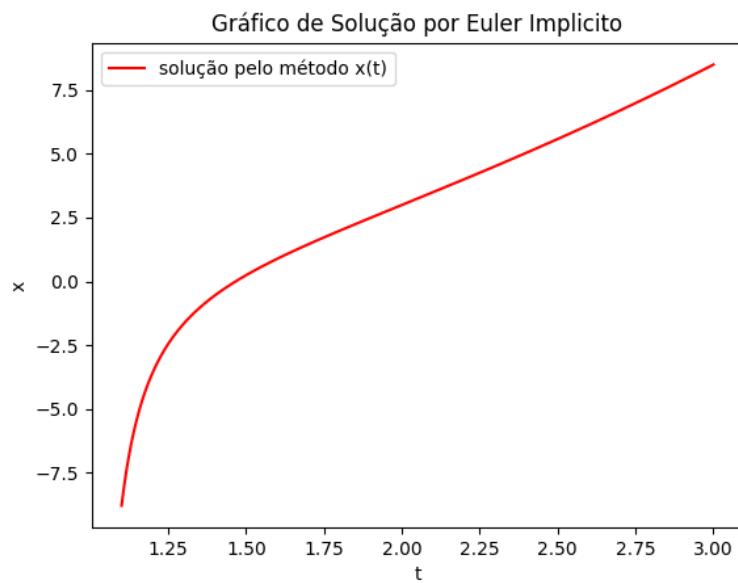


Figura 20: Exercício 1 - Teste 2 - Solução pelo Método de Euler Implícito



Figura 21: Exercício 1 - Teste 2 - Solução explícita

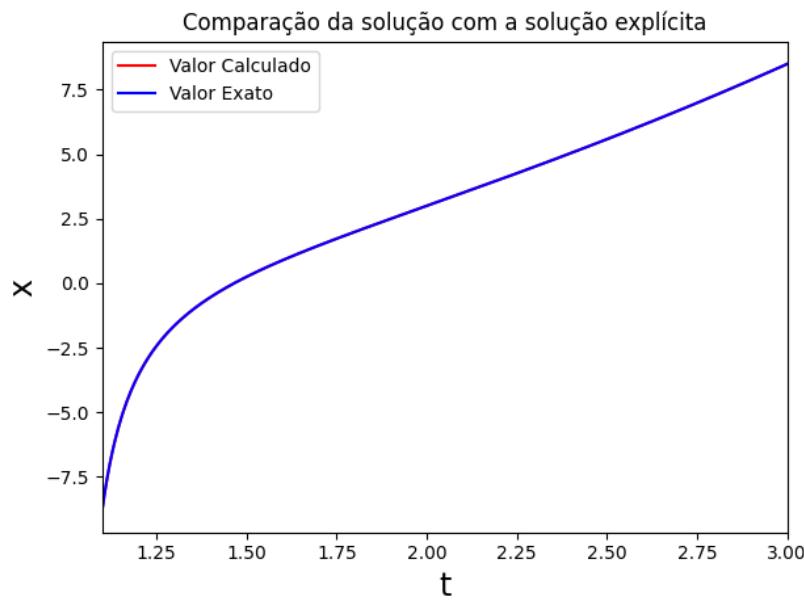


Figura 22: Exercício 1 - Teste 12 - Comparação das solução pelo método com a solução explícita

Gráfico de  $E_2(t)$ :

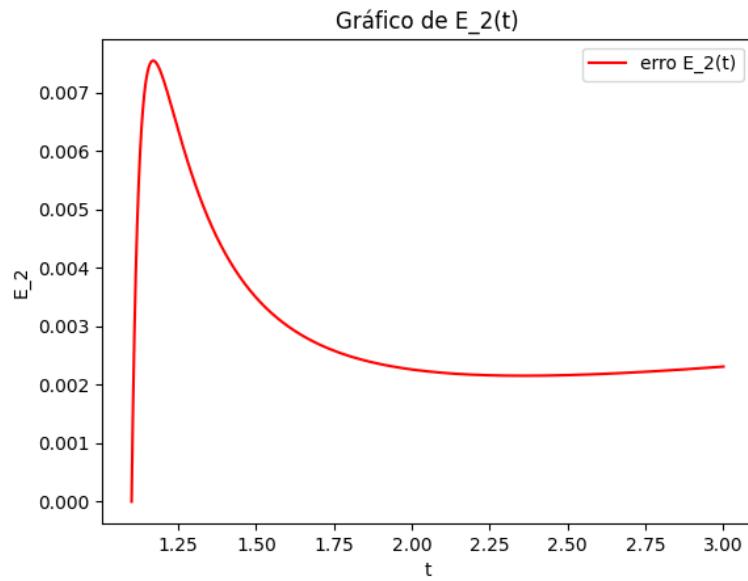


Figura 23: Exercício 1 - Teste 2 - Erro

## Interpretação dos resultados

Percebe-se que a ordem de grandeza do erro foi de  $10^{-3}$ , ou seja, ficando abaixo de 1% em todos os momentos. Por conta do chute inicial, para as primeiras iterações o erro cresce, com o avanço do número de iterações (representada pelo próprio eixo do tempo), o erro decai e se estabiliza em um valor próximo de 0.2%, o que pode ser considerado bem satisfatório. É notável que o erro é maior na seção onde a derivada da função se altera rapidamente. Quando a derivada se estabiliza, o erro decai, e se estabelece.

Além disso, pelo gráfico apresentado na imagem (22), podemos visualizar que o modelo foi implementado de forma correta e igualmente satisfatória.

## 6 Exercício 02 - Modelo presa-predador

### 6.1 Enunciado

Adaptado para o relatório:

Considere um ambiente em que convivam duas espécies, uma que se alimenta dos recursos naturais (digamos, uma população de coelhos) e uma que se abastece da primeira (por exemplo, uma população de raposas).

Podemos descrever a população (número de animais vivos) de cada por meio do sistema de equações diferenciais conhecido como modelo de Lotka-Ventura (sendo  $x(t)$  a população de coelhos em função do tempo e  $y(t)$  a população de raposas):

$$x'(t) = \lambda x(t) - \alpha x(t)y(t) \quad (16)$$

$$y'(t) = \beta x(t)y(t) - \gamma y(t) \quad (17)$$

Neste exercício, trabalhará-se com  $\lambda = \frac{2}{3}$ ,  $\alpha = \frac{4}{3}$ ,  $\beta = 1$ ,  $\gamma = 1$ ,  $x(0) = 1.5$ ,  $y(0) = 1.5$  e  $[T_0, T_f] = [0, 10.0]$ .

Neste caso, se houver inicialmente membros das duas espécies no ambiente, o tamanho das respectivas populações irá variar periodicamente no tempo, ou seja, o retrato de fase (curva paramétrica  $(x(t), y(t))$  para  $[T_0, T_f]$ , será composta por órbitas periódicas.

Para  $n \in \mathbb{N}$  dado, escolhemos uma discretização uniforme  $t_k = T_0 + kh$  de  $[T_0, T_f]$ , onde  $h = (T_f - T_0)/n$  e  $k \in \{1, 2, \dots, n\}$ , isto é, dividimos o intervalo  $[T_0, T_f]$  em  $n$  subintervalos do mesmo tamanho.

1. Escreva um código para resolver (16)-(17) usando o método de Euler explícito. Você deve plotar um gráfico do retrato de fase. Faça também um outro gráfico mostrando simultaneamente o tamanho de cada população ao longo do tempo (plota as raposas em vermelho e os coelhos em azul). Use um valor  $n \geq 5000$ .
2. Escreva um código para resolver (16)-(17) usando o método de Euler implícito. Você deve plotar um gráfico do retrato de fase. Faça também um outro gráfico mostrando simultaneamente o tamanho de cada população ao longo do tempo (plota as raposas em vermelho e os coelhos em azul). Use um valor  $n \geq 500$ .
3. Sejam  $x_{im}(t)$ ,  $y_{im}(t)$  e  $x_{ex}(t)$ ,  $y_{ex}(t)$  as soluções usando o método de Euler implícito e explícito, respectivamente. Vamos definir  $E_x(t) := x_{im}(t) - x_{ex}(t)$ ,  $E_y(t) := y_{im}(t) - y_{ex}(t)$ . Para cada valor  $n = 250, 500, 1000, 2000, 4000$ , plote um gráfico

com a função  $E_x(t)$  em azul e a função  $E_y(t)$  em vermelho. Comente brevemente o resultado.

4. Escreva um código para resolver (16)-(17) usando o método de Runge-Kutta 4. Você deve plotar um gráfico do retrato de fase. Faça também um outro gráfico mostrando simultaneamente o tamanho de cada população ao longo do tempo (plota as raposas em vermelho e os coelhos em azul). Use um valor  $n \geq 500$ .

## 6.2 Implementação da solução

O código completo se encontra no apêndice: ver **exercicio\_2.py**

## 6.3 Resultados

### Exercício 2.1 - Euler Explícito

Gráfico do retrato de fase:

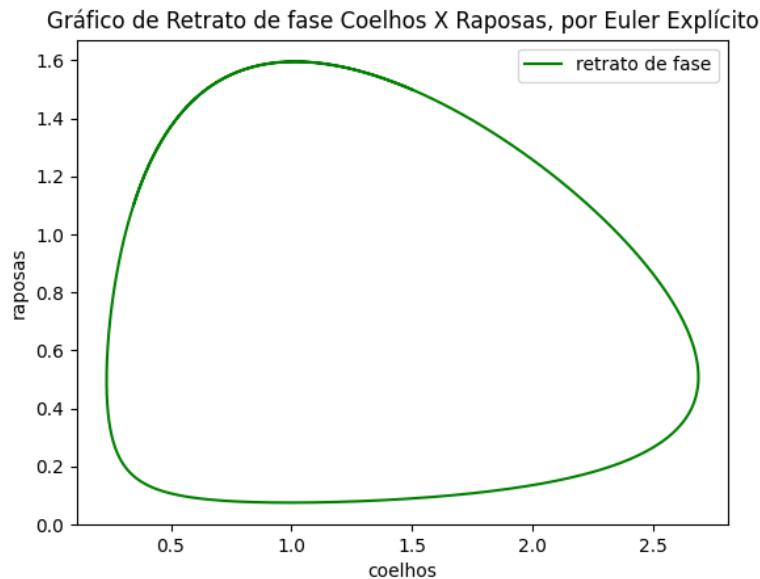


Figura 24: Exercício 2.1 - Retrato de Fase

Gráfico do tamanho das populações ao longo do tempo:

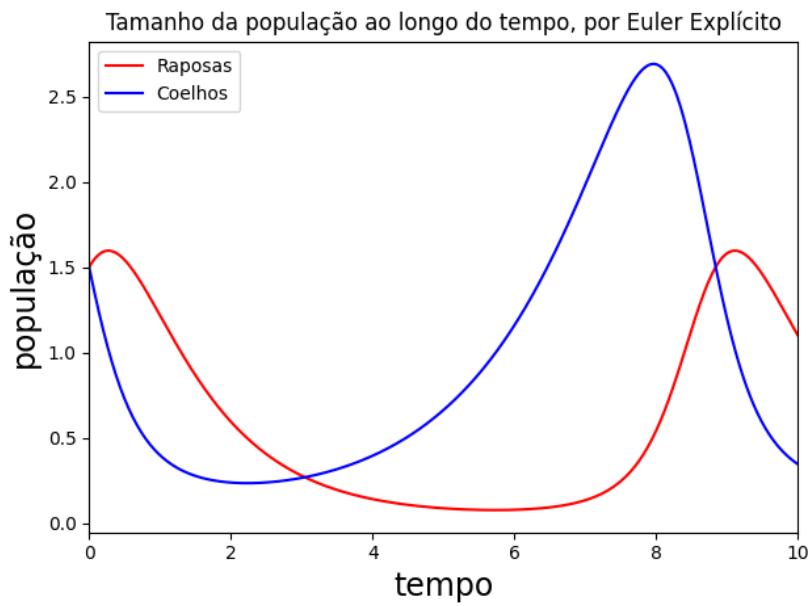


Figura 25: Exercício 2.1 - Tamanho da População ao longo do tempo

### Exercício 2.2 - Euler Implícito

Gráfico do retrato de fase:

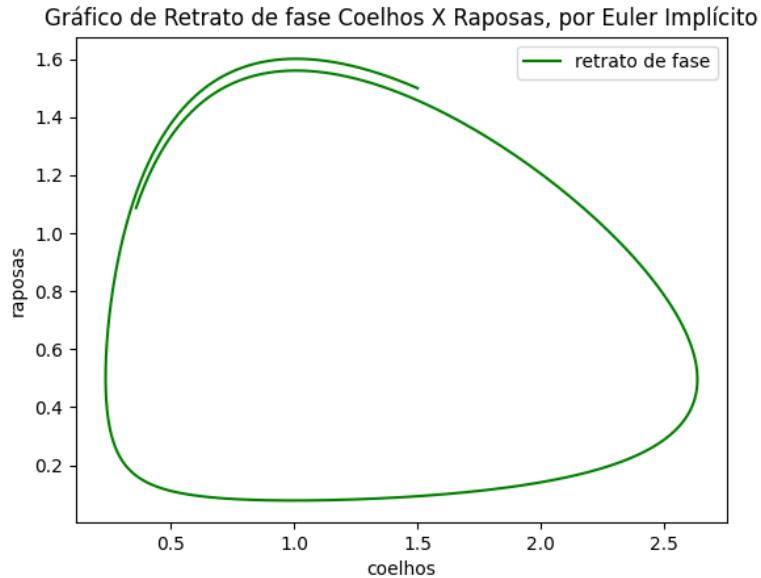


Figura 26: Exercício 2.2 - Retrato de Fase

Gráfico do tamanho das populações ao longo do tempo:

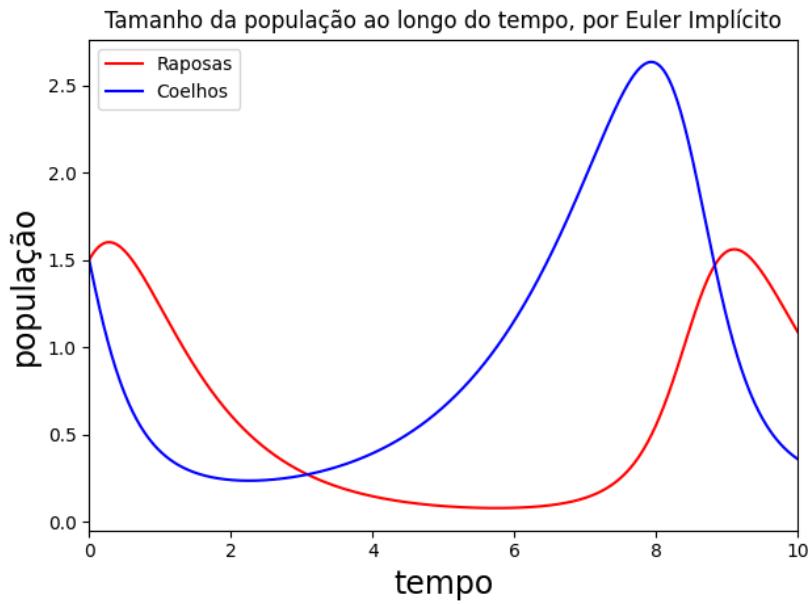


Figura 27: Exercício 2.2 - Tamanho da População ao longo do tempo

### Exercício 2.3 - Análise Euler Explícito e Implícito

Podemos ver os gráficos de  $E_x(t)$  em azul e  $E_y(t)$  em vermelho para cada valor de  $n$ :

- $n = 250$ :

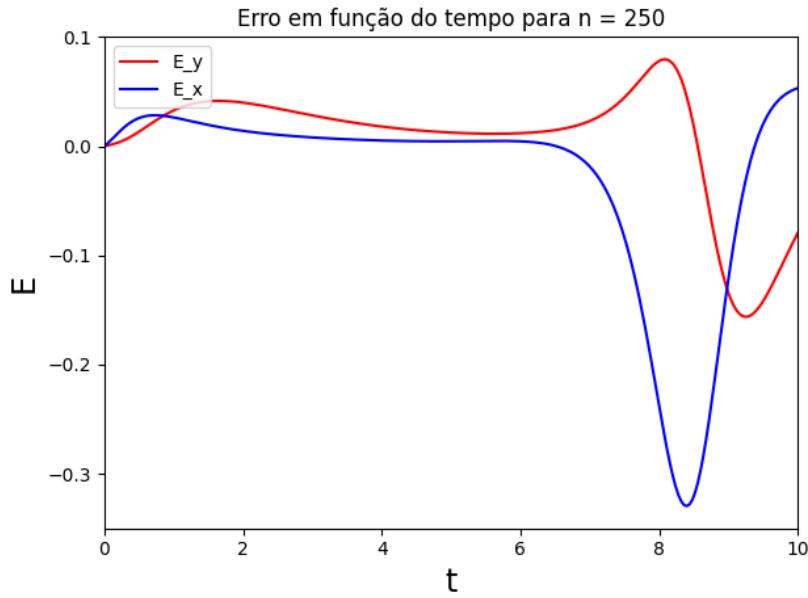


Figura 28: Exercício 2.3 - Gráfico dos Erros pelo tempo para  $n = 250$

- $n = 500$ :

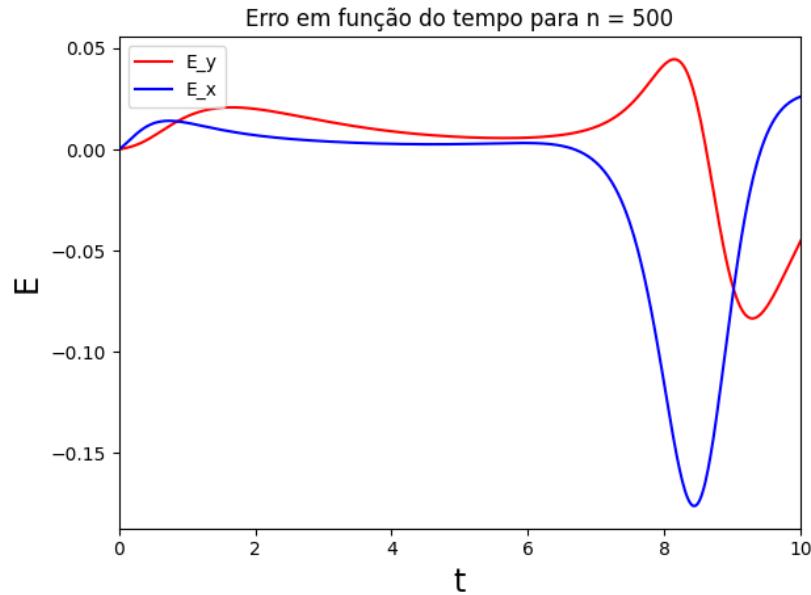


Figura 29: Exercício 2.3 - Gráfico dos Erros pelo tempo para  $n = 500$

- $n = 1000$ :

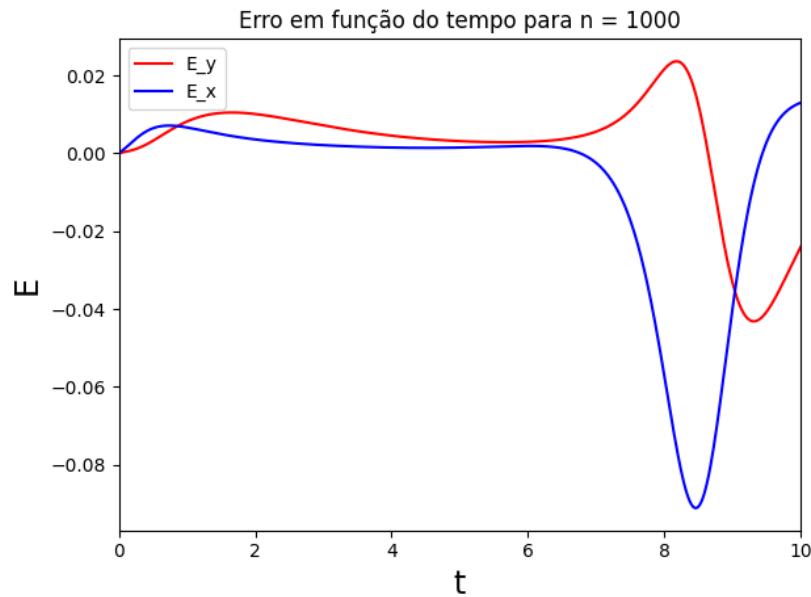


Figura 30: Exercício 2.3 - Gráfico dos Erros pelo tempo para  $n = 1000$

- $n = 2000$ :

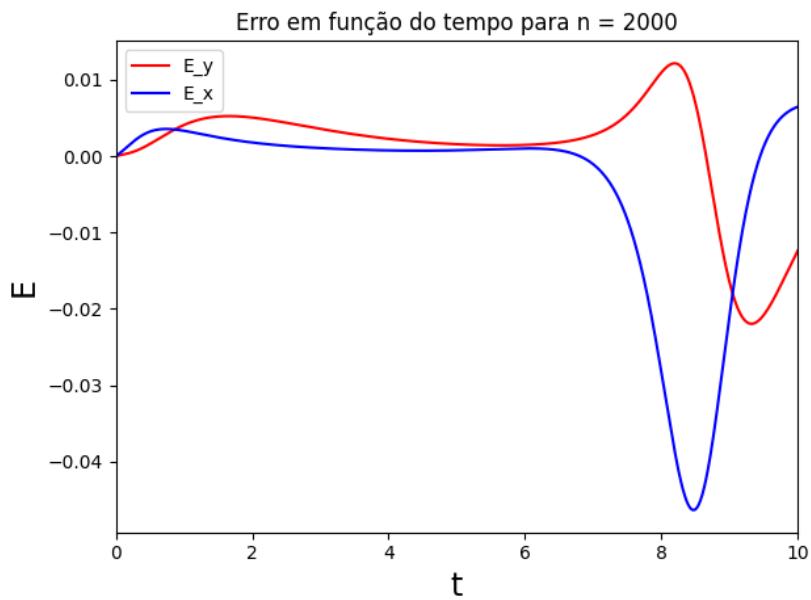


Figura 31: Exercício 2.3 - Gráfico dos Erros pelo tempo para  $n = 2000$

- $n = 4000$ :

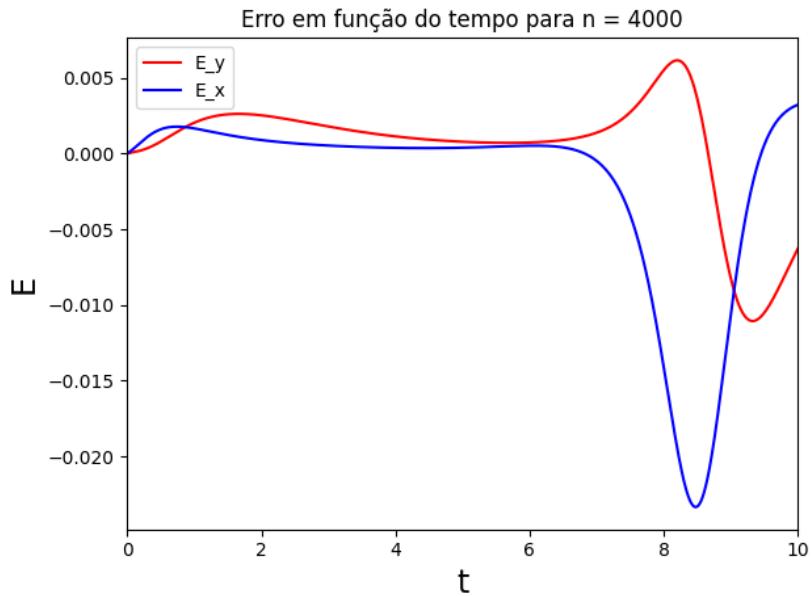


Figura 32: Exercício 2.3 - Gráfico dos Erros pelo tempo para  $n = 4000$

#### Exercício 2.4 - Runge-Kutta Ordem 4

Gráfico do retrato de fase:

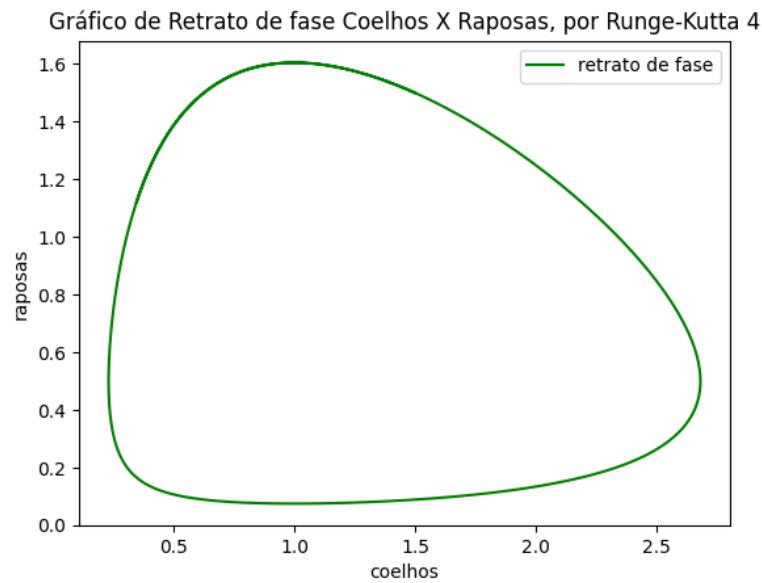


Figura 33: Exercício 2.4 - Retrato de Fase

Gráfico do tamanho das populações ao longo do tempo:

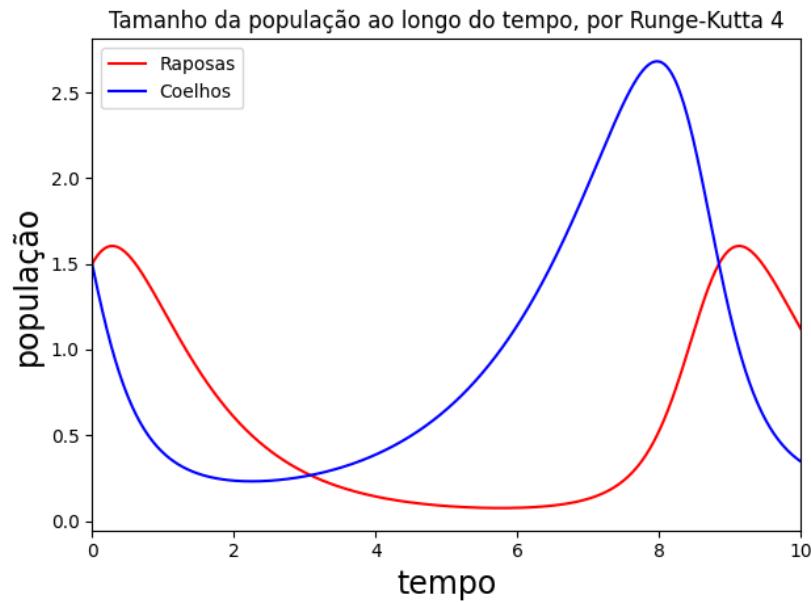


Figura 34: Exercício 2.4 - Tamanho da População ao longo do tempo

## 6.4 Interpretação e Discussão dos Resultados Obtidos

Em uma primeira análise, vale notar que todos os métodos (Euler Explícito, Euler Implícito e Runge-Kutta de ordem 4) chegaram em resultados bem próximos - basta visualizar os gráficos dos tamanhos das populações em função do tempo.

Uma diferença mais notável se dá no retrato de fase: onde o Euler Explícito e o RK4 obtém um retrato de fase que se assemelha a uma curva paramétrica fechada, o Euler implícito apresenta uma curva de formato semelhante, mas que não se "fecha", isto é, que não se conecta de ponta a ponta.

Essa diferença é visualizável no exercício 2.3, onde estuda-se a diferença entre os métodos implícito e explícito de Euler. Neste exercício, visualiza-se que os erros se maximizam próximos aos pontos em que existem uma grande alteração das populações, o que leva a crer que o erro está relacionado com a taxa de variação da derivada das populações. Isso é compreensível, pois o método de Euler é sensível a mudanças bruscas de derivadas, já que se baseia em sua grande parte na derivada do sistema.

No entanto, conforme o número de iterações  $n$  cresce, menor fica o erro, já que a distância de tempo entre os pontos analisados são encurtadas, e mudanças bruscas são estudadas com mais cautela. No entanto, o pico do erro se mantém neste mesmo ponto, onde a taxa de variação da taxa de variação atinge um pico.

## 7 Exercício 3 - Modelo duas presas-um predador

### 7.1 Enunciado

Adaptado para o relatório:

Vamos considerar agora que mais um espécie conviva com os coelhos e as raposas, digamos uma população de lebres que também se alimente apenas dos recursos naturais. Ou seja, coelhos e lebres estarão competindo pelos mesmos recursos. Por outro lado, as raposas poderão se alimentar tanto de coelhos como de lebres. A equação do modelo pode ser posta na forma:

$$x'(t) = x(t)(B_1 - A_{1,1}x(t) - A_{1,2}y(t) - A_{1,3}z(t)) \quad (18)$$

$$y'(t) = y(t)(B_2 - A_{2,1}x(t) - A_{2,2}y(t) - A_{2,3}z(t)) \quad (19)$$

$$z'(t) = z(t)(B_3 - A_{3,1}x(t) - A_{3,2}y(t) - A_{3,3}z(t)) \quad (20)$$

onde  $x(t)$  representa os coelhos,  $y(t)$  as lebres e  $z(t)$  as raposas. Nesta forma os coeficientes  $B_i$  e  $A_{i,j}$  são positivos para  $i = 1$  e  $i = 2$ , enquanto para  $i = 3$  são negativos. Desta forma, o modelo acima engloba todos os modelos já apresentados.

Neste exercício, vamos considerar os seguintes parâmetros para o modelo com duas presas e um predador:

$$B = \begin{pmatrix} 1.0 \\ 1.0 \\ -1.0 \end{pmatrix}, \quad A = \begin{pmatrix} 0.001 & 0.001 & 0.015 \\ 0.0015 & 0.001 & 0.001 \\ -\alpha & -0.0005 & 0.0 \end{pmatrix}$$

com  $0.001 \leq \alpha \leq 0.0055$ .

Com esta escolha de parâmetros temos que os coelhos irão levar vantagens sobre as lebres quando há poucas raposas. No entanto, as raposas se nutrem melhor de coelhos do que de lebres. O objetivo desse exercício computacional é estudar esse modelo de competição fazendo variações nos parâmetros do modelo. Para tanto, você irá usar os métodos de Euler explícito e Runge-Kutta 4. Para  $n \in \mathbb{N}$  dado, escolhemos uma discretização uniforme  $t_k = T_0 + kh$  de  $[T_0, T_f]$ , onde  $h = (T_f - T_0) / n$  e  $k \in \{1, 2, \dots, n\}$ , isto é, dividimos o intervalo  $[T_0, T_f]$  em  $n$  subintervalos do mesmo tamanho. Em todos os casos teremos  $T_0 = 0$  e usaremos  $n \geq 5000$ .

- Analise o comportamento do retrato de fase do modelo com relação ao parâmetro  $\alpha$ , escolhendo os valores 0.001, 0.002, 0.0033, 0.0036, 0.005 e 0.0055. Inicie com 500 coelhos, 500 lebres e 10 raposas. Nos dois primeiros casos, use  $T_f = 100$ , nos dois seguintes  $T_f = 500$  e nos dois últimos  $T_f = 2000$ . Determine os casos em que há órbitas periódicas, equilíbrios estáveis ou instáveis.
- Para cada caso você deve fazer gráficos do retrato de fase, ou seja, um gráfico 3D cujos eixos são o número de coelhos, lebres e raposas e cada ponto do gráfico se refere a um instante de tempo da integração. Para melhor visualização, plote também gráficos dos retratos de fase em 2D (um gráfico para coelhos x lebres, um gráfico para coelhos x raposas, um gráfico para lebres x raposas). Faça também um gráfico mostrando simultaneamente o tamanho das três populações ao longo do tempo (plote as raposas em vermelho, os coelhos em azul e as lebres em preto, e coloque legenda em todos os gráficos).
- Teste a sensibilidade em relação aos valores iniciais quando  $\alpha = 0.005$ , executando o modelo até  $T_f = 400$  iniciando com 37 coelhos, 75 lebres e 137 raposas. Imprima o

número de raposas, lebres e coelhos no instante final. Repita o mesmo experimento, trocando apenas o número inicial de lebres de 75 para 74. Compare os valores finais e a evolução das populações nos dois casos.

## 7.2 Implementação da solução

O código completo se encontra no apêndice: ver **exercicio\_3.py**

## 7.3 Análise do Comportamento

Caso	$\alpha$	$T_f$	Comportamento
1	0.001	100	Sem periodicidade, equilíbrio estável
2	0.002	100	Há uma periodicidade limitada, lembra oscilação amortecida
3	0.0033	500	Há periodicidade, equilíbrio instável
4	0.0036	500	Há periodicidade irregular, equilíbrio instável
5	0.005	2000	Periodicidade com alterações a cada ciclo
6	0.0015	2000	Semelhança de periodicidade com irregularidades

Tabela 2: Soluções de  $x(T_f)$  para os diferentes  $n$

## 7.4 Resultados

### Caso 1

Retrato de fase para  $T_f = 100.0$  e  $\alpha = 0.001$  com Runge-Kutta 4

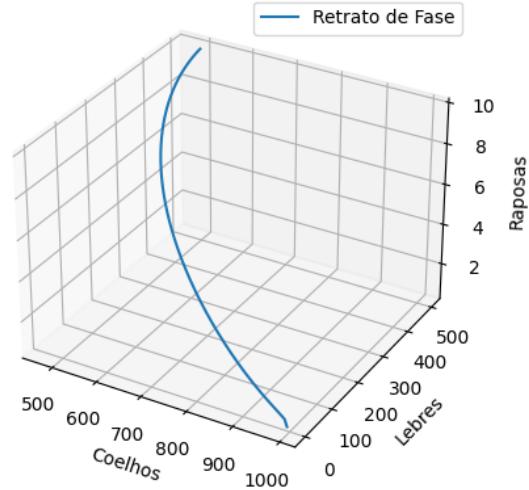


Figura 35: Exercício 3 - Caso 1 - Runge-Kutta 4 - Gráfico de Retrato de Fase 3D

Retrato de fase para  $T_f = 100.0$  e  $\alpha = 0.001$  com Euler Explícito

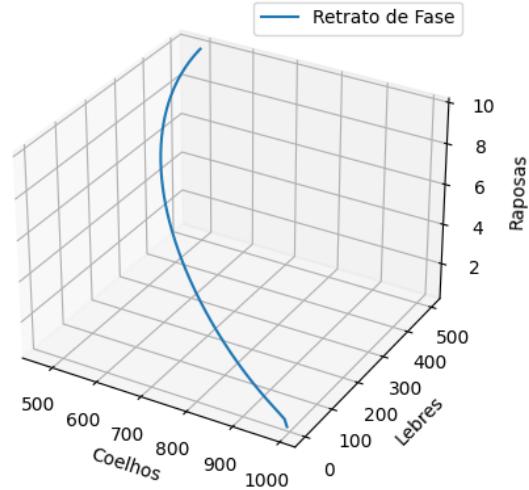


Figura 36: Exercício 3 - Caso 1 - Euler Explícito - Gráfico de Retrato de Fase 3D

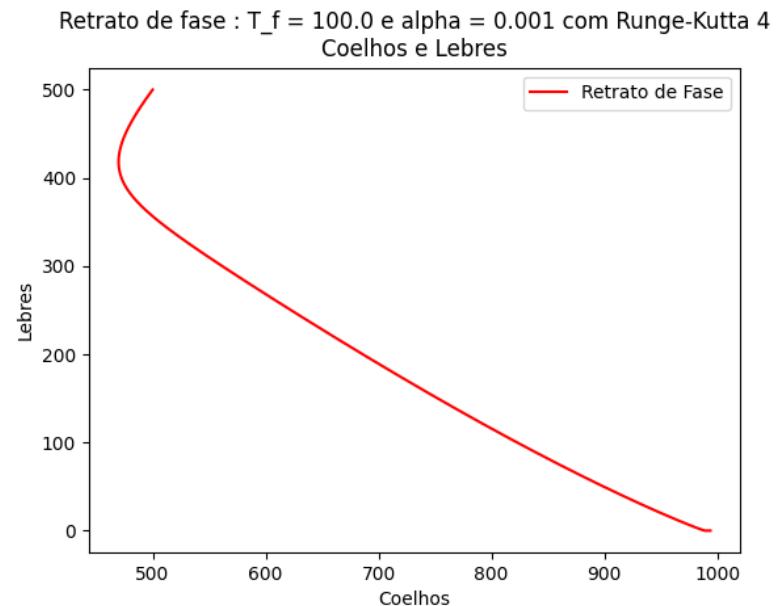


Figura 37: Exercício 3 - Caso 1 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

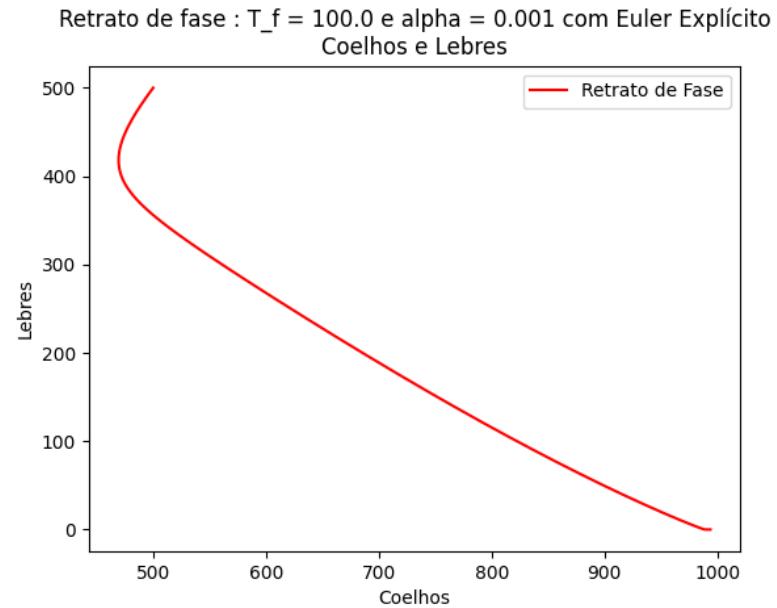


Figura 38: Exercício 3 - Caso 1 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.001$  com Runge-Kutta 4  
Coelhos e Raposas

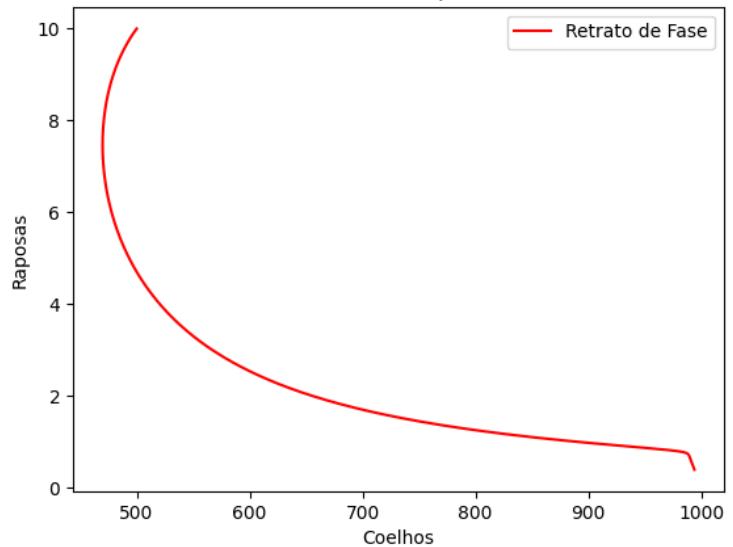


Figura 39: Exercício 3 - Caso 1 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.001$  com Euler Explícito  
Coelhos e Raposas

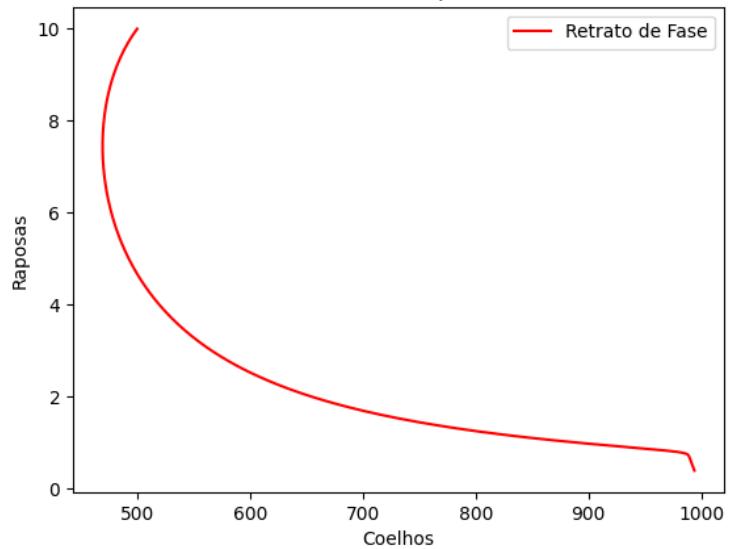


Figura 40: Exercício 3 - Caso 1 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.001$  com Runge-Kutta 4  
Lebres e Raposas

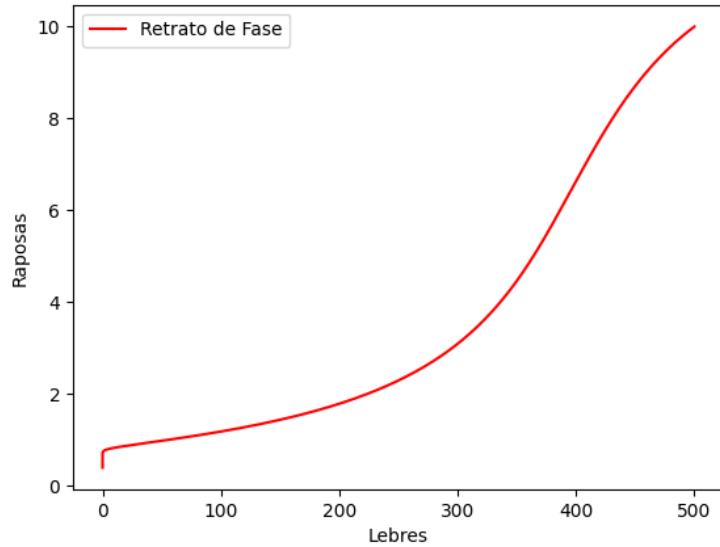


Figura 41: Exercício 3 - Caso 1 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.001$  com Euler Explícito  
Lebres e Raposas

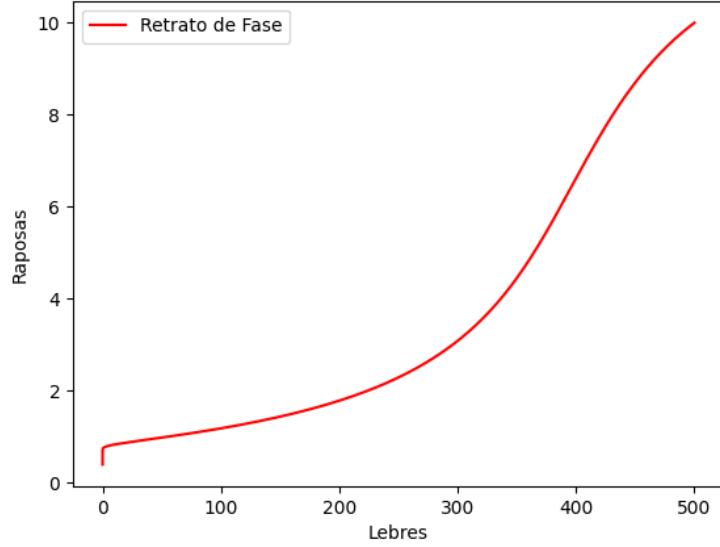


Figura 42: Exercício 3 - Caso 1 - Euler Explícito - Gráfico de Retrato de Fase 2D

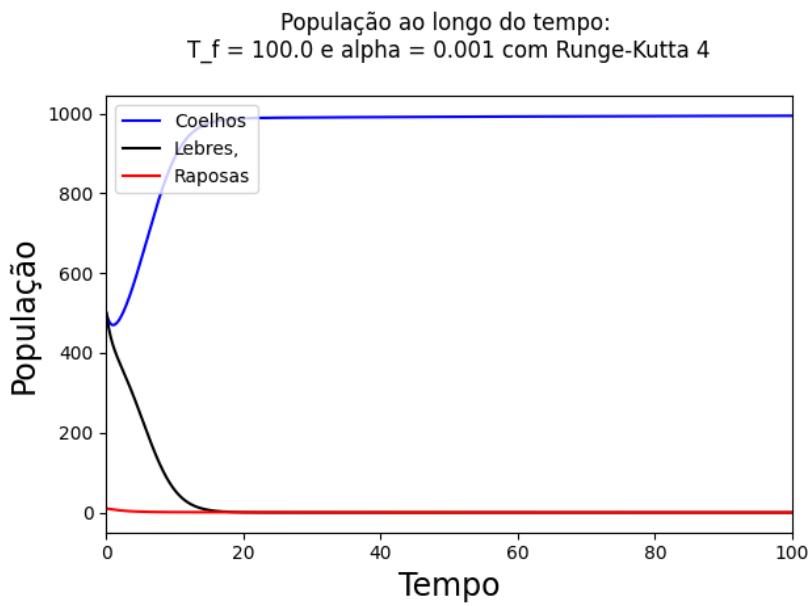


Figura 43: Exercício 3 - Caso 1 - Runge-Kutta 4 - Gráfico de tamanho da população pelo tempo

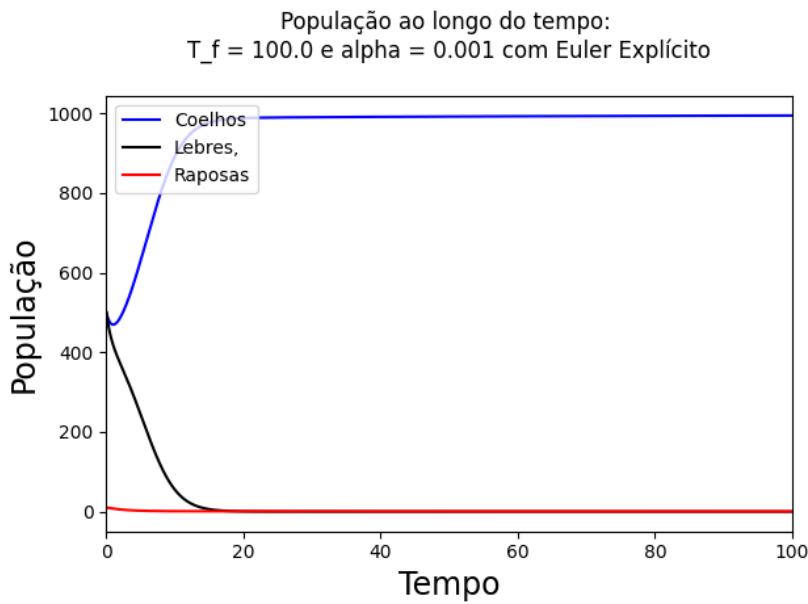


Figura 44: Exercício 3 - Caso 1 - Euler Explícito - Gráfico de tamanho da população pelo tempo

## Caso 2

Retrato de fase para  $T_f = 100.0$  e  $\alpha = 0.002$  com Runge-Kutta 4

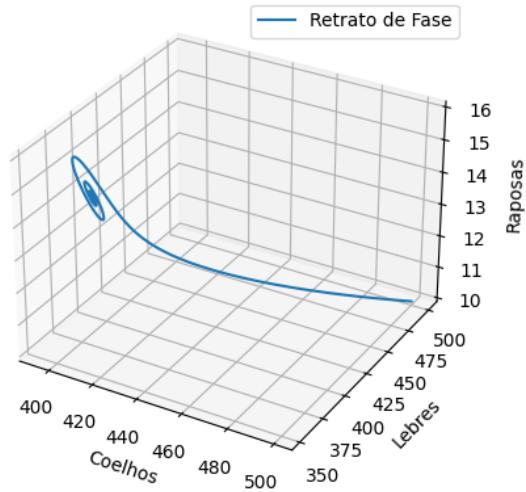


Figura 45: Exercício 3 - Caso 2 - Runge-Kutta 4 - Gráfico de Retrato de Fase 3D

Retrato de fase para  $T_f = 100.0$  e  $\alpha = 0.002$  com Euler Explícito

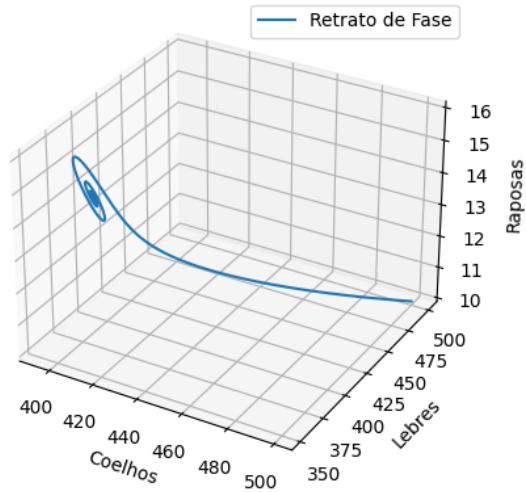


Figura 46: Exercício 3 - Caso 2 - Euler Explícito - Gráfico de Retrato de Fase 3D

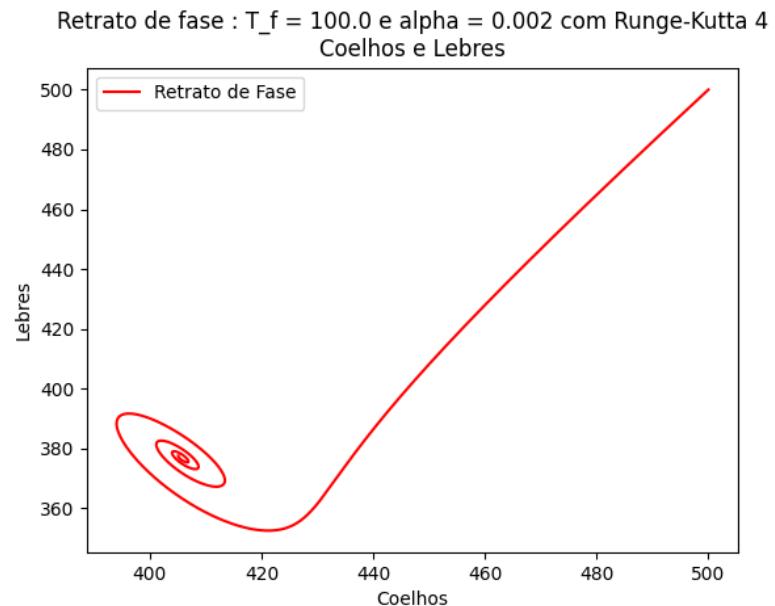


Figura 47: Exercício 3 - Caso 2 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

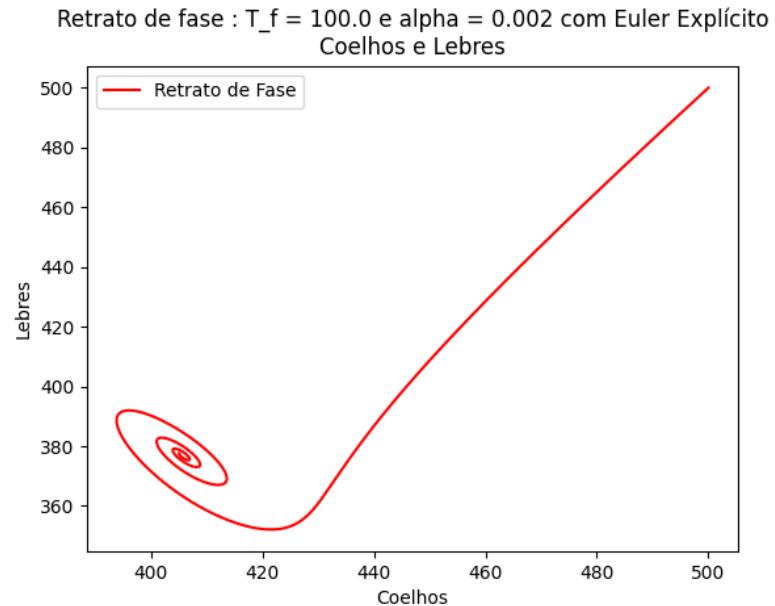


Figura 48: Exercício 3 - Caso 2 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.002$  com Runge-Kutta 4  
Coelhos e Raposas

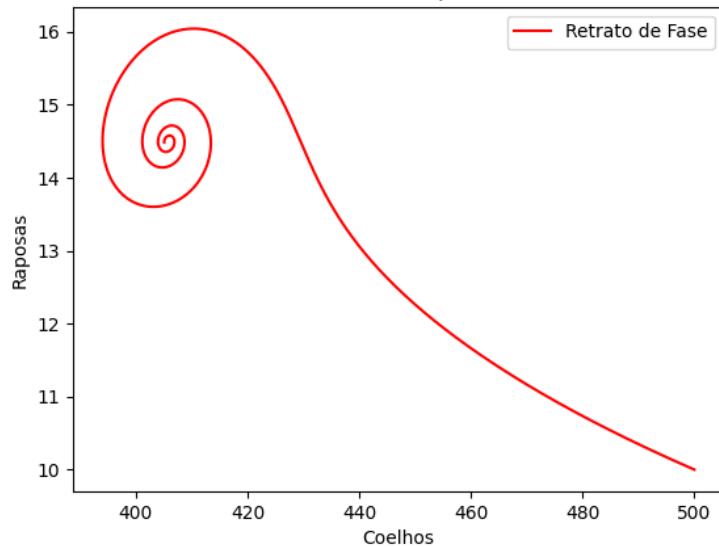


Figura 49: Exercício 3 - Caso 2 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.002$  com Euler Explícito  
Coelhos e Raposas

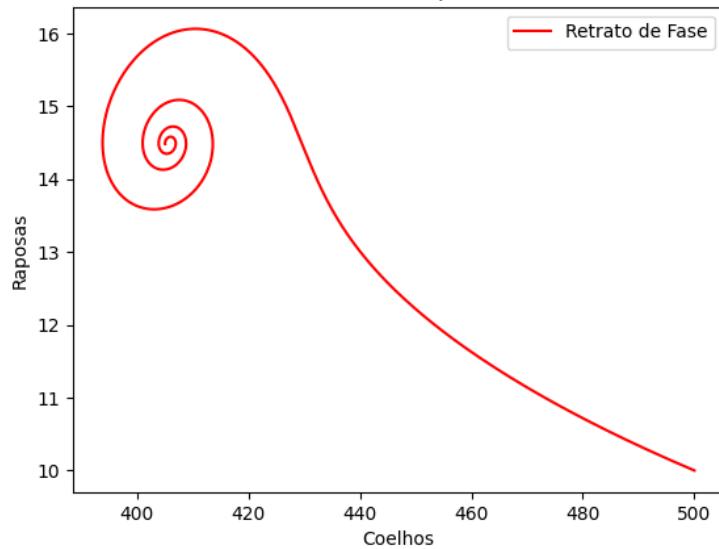


Figura 50: Exercício 3 - Caso 2 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.002$  com Runge-Kutta 4  
Lebres e Raposas

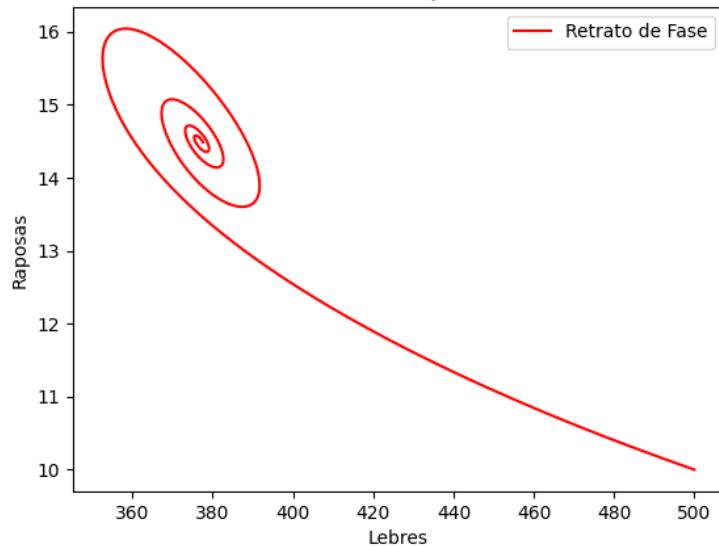


Figura 51: Exercício 3 - Caso 2 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 100.0$  e  $\alpha = 0.002$  com Euler Explícito  
Lebres e Raposas

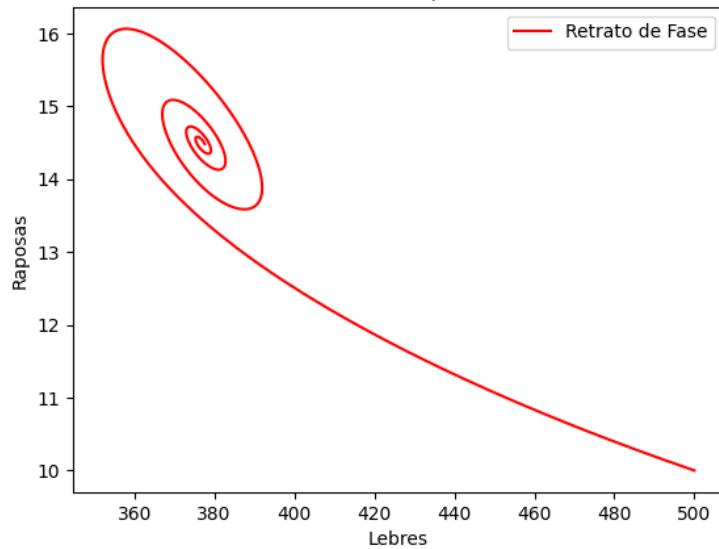


Figura 52: Exercício 3 - Caso 2 - Euler Explícito - Gráfico de Retrato de Fase 2D

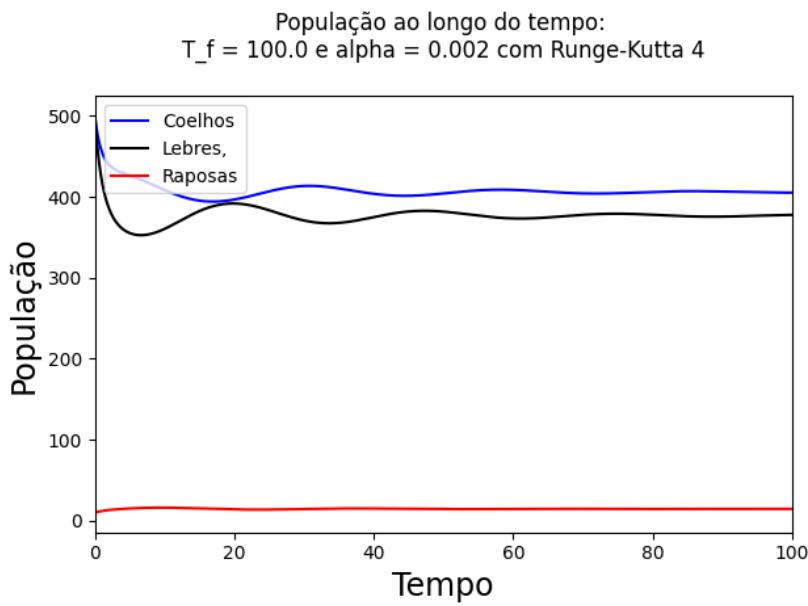


Figura 53: Exercício 3 - Caso 2 - Runge-Kutta 4 - Gráfico de tamanho da população pelo tempo

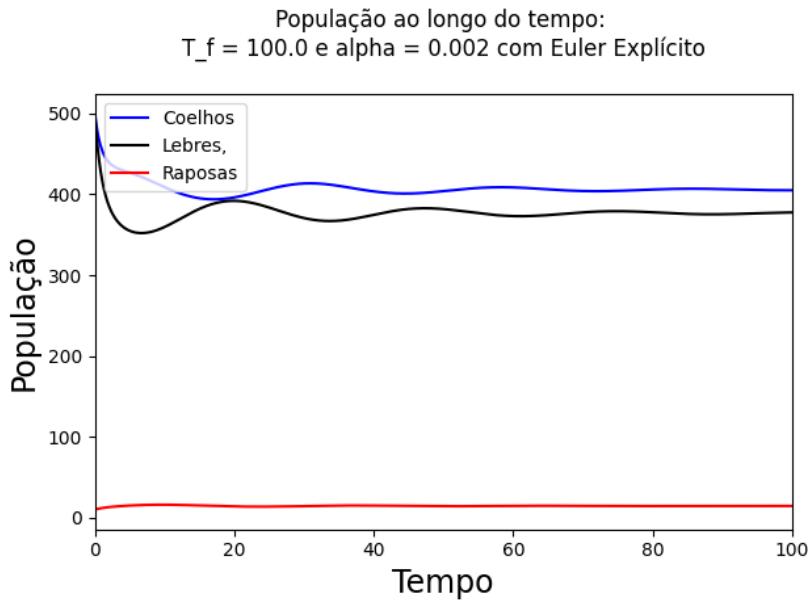


Figura 54: Exercício 3 - Caso 2 - Euler Explícito - Gráfico de tamanho da população pelo tempo

### Caso 3

Retrato de fase para  $T_f = 500.0$  e  $\alpha = 0.0033$  com Runge-Kutta 4

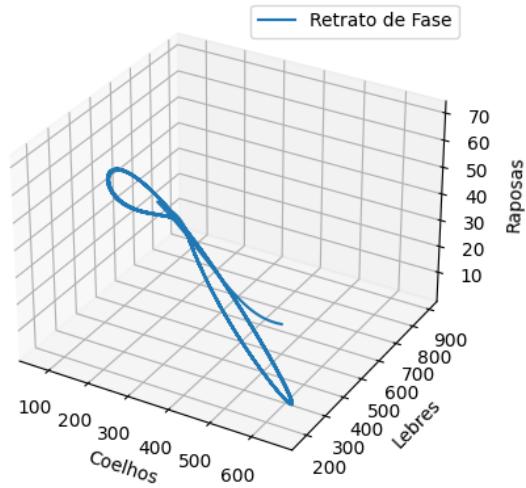


Figura 55: Exercício 3 - Caso 3 - Runge-Kutta 4 - Gráfico de Retrato de Fase 3D

Retrato de fase para  $T_f = 500.0$  e  $\alpha = 0.0033$  com Euler Explícito

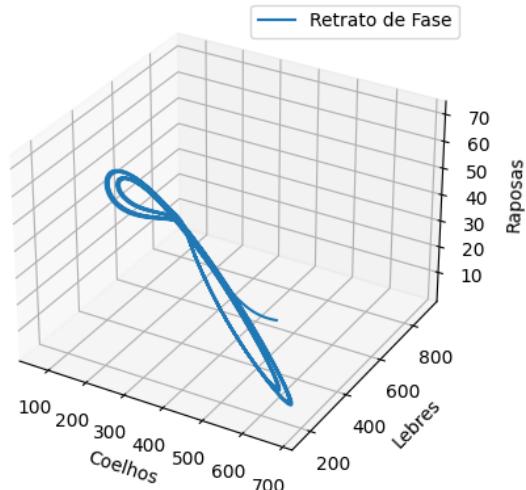


Figura 56: Exercício 3 - Caso 3 - Euler Explícito - Gráfico de Retrato de Fase 3D

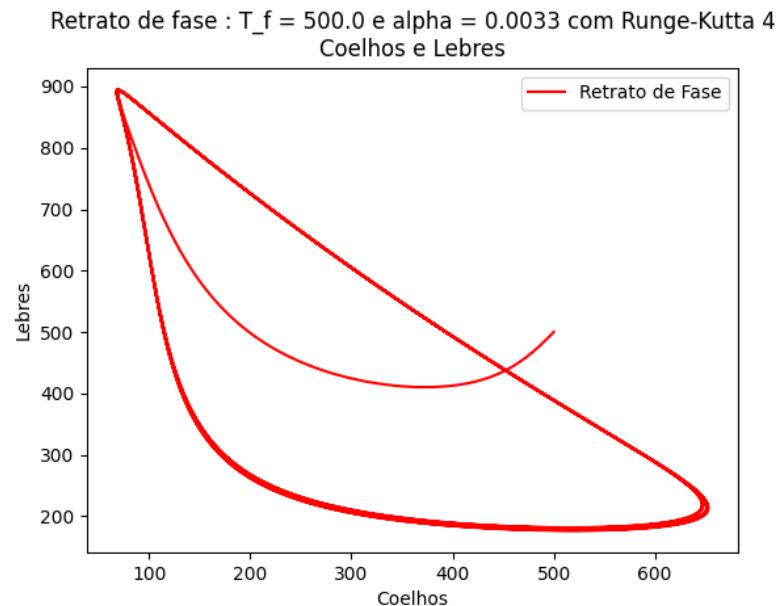


Figura 57: Exercício 3 - Caso 3 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

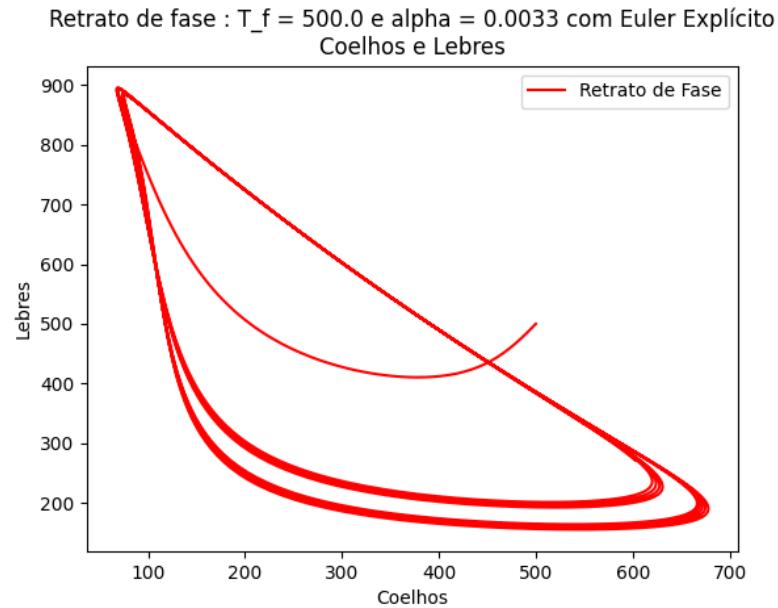


Figura 58: Exercício 3 - Caso 3 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 500.0$  e  $\alpha = 0.0033$  com Runge-Kutta 4  
Coelhos e Raposas

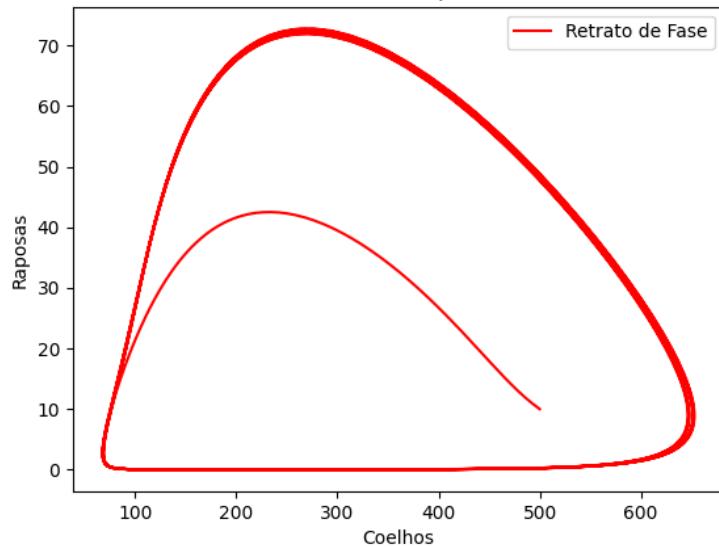


Figura 59: Exercício 3 - Caso 3 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 500.0$  e  $\alpha = 0.0033$  com Euler Explícito  
Coelhos e Raposas

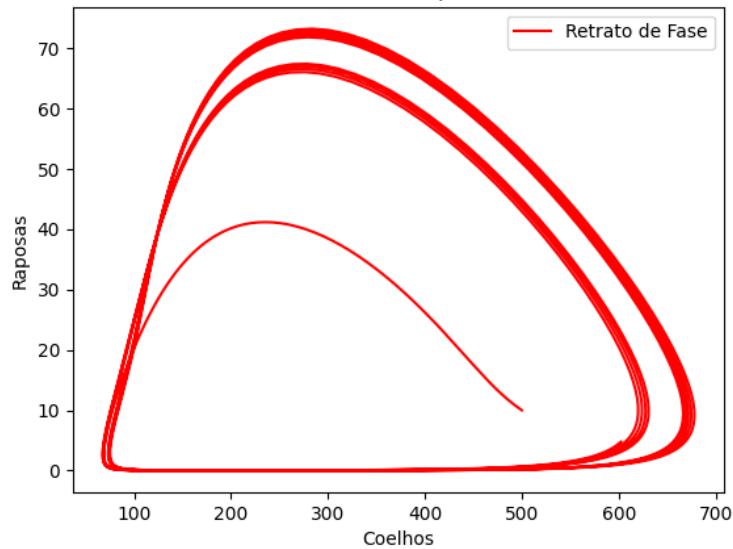


Figura 60: Exercício 3 - Caso 3 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 500.0$  e  $\alpha = 0.0033$  com Runge-Kutta 4  
Lebres e Raposas

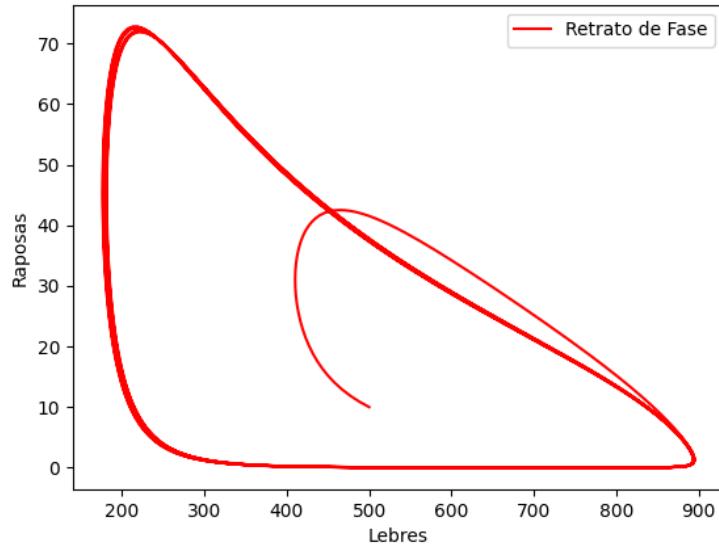


Figura 61: Exercício 3 - Caso 3 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 500.0$  e  $\alpha = 0.0033$  com Euler Explícito  
Lebres e Raposas

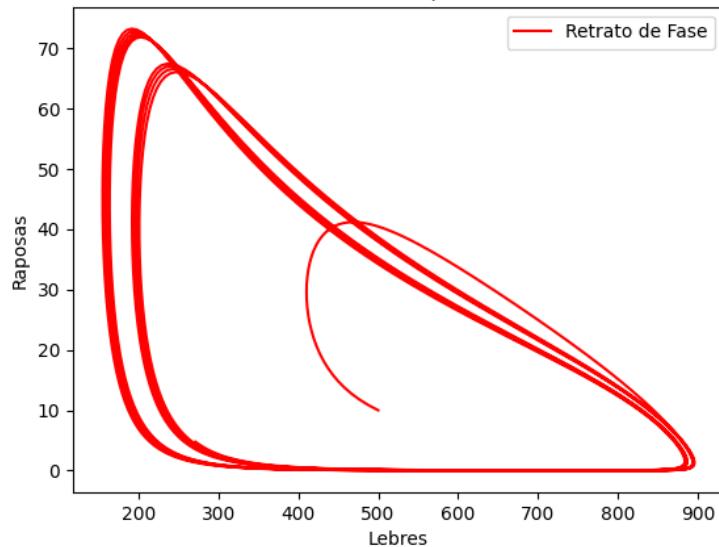


Figura 62: Exercício 3 - Caso 3 - Euler Explícito - Gráfico de Retrato de Fase 2D

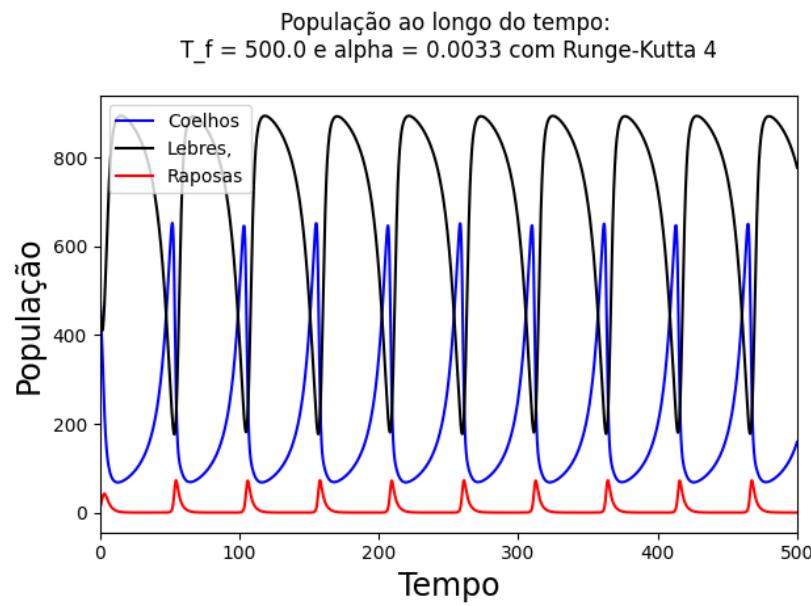


Figura 63: Exercício 3 - Caso 3 - Runge-Kutta 4 - Gráfico de tamanho da população pelo tempo

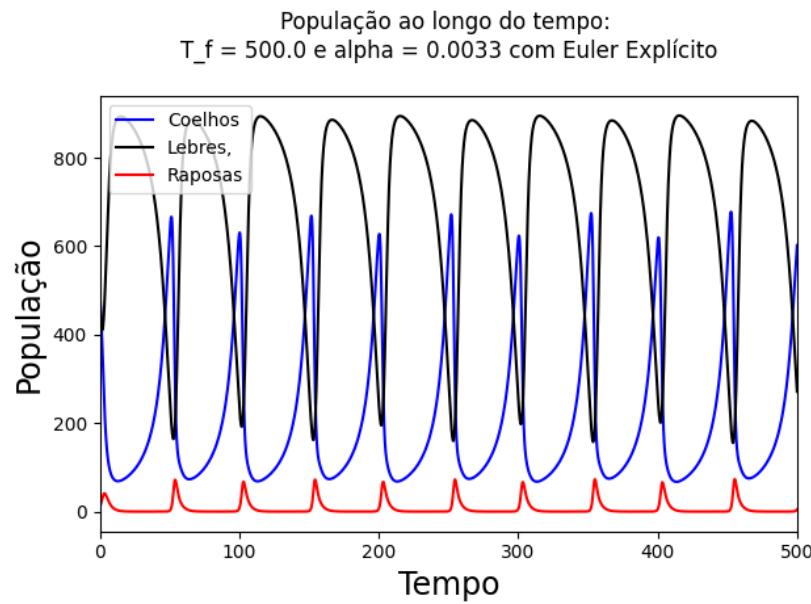


Figura 64: Exercício 3 - Caso 3 - Euler Explícito - Gráfico de tamanho da população pelo tempo

## Caso 4

Retrato de fase para  $T_f = 500.0$  e  $\alpha = 0.0036$  com Runge-Kutta 4

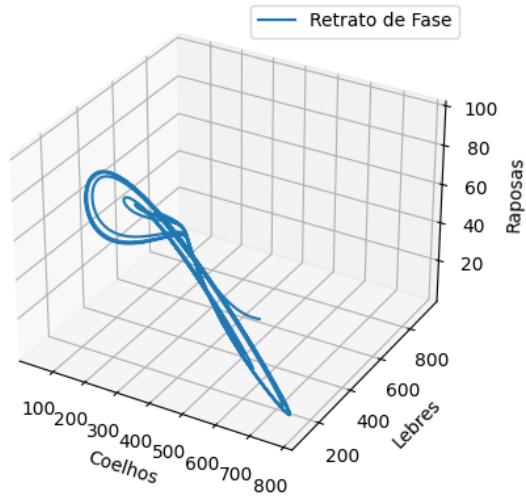


Figura 65: Exercício 3 - Caso 4 - Runge-Kutta 4 - Gráfico de Retrato de Fase 3D

Retrato de fase para  $T_f = 500.0$  e  $\alpha = 0.0036$  com Euler Explícito

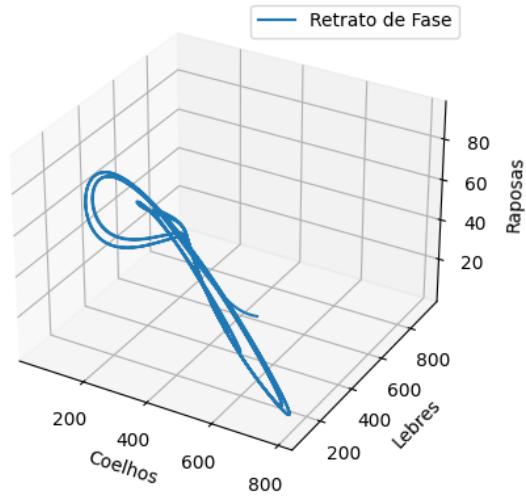


Figura 66: Exercício 3 - Caso 4 - Euler Explícito - Gráfico de Retrato de Fase 3D

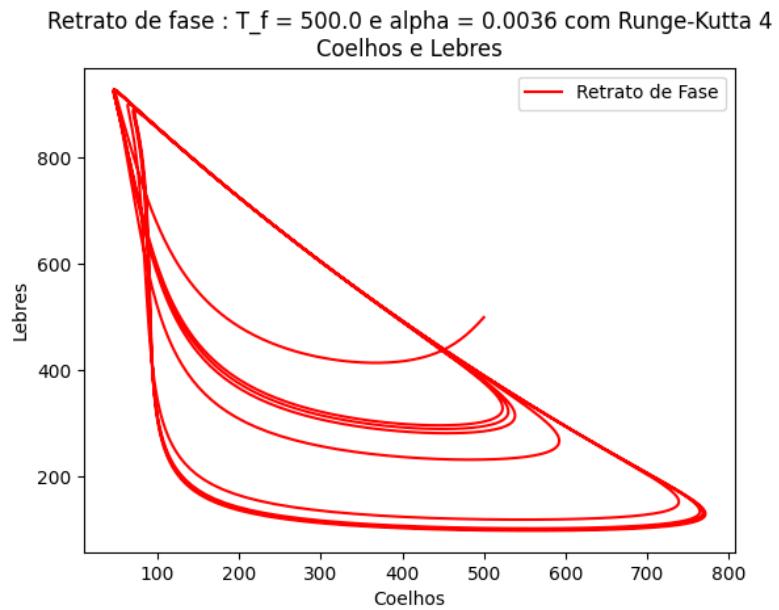


Figura 67: Exercício 3 - Caso 4 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

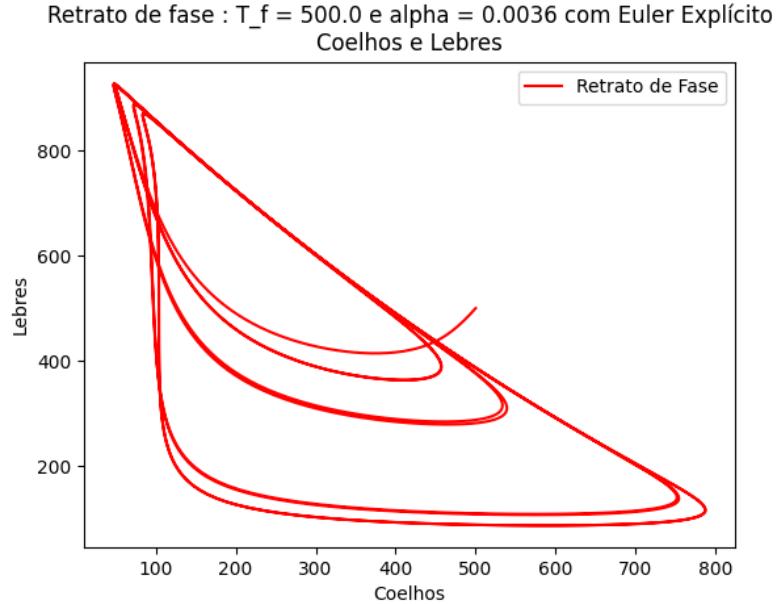


Figura 68: Exercício 3 - Caso 4 - Euler Explícito - Gráfico de Retrato de Fase 2D

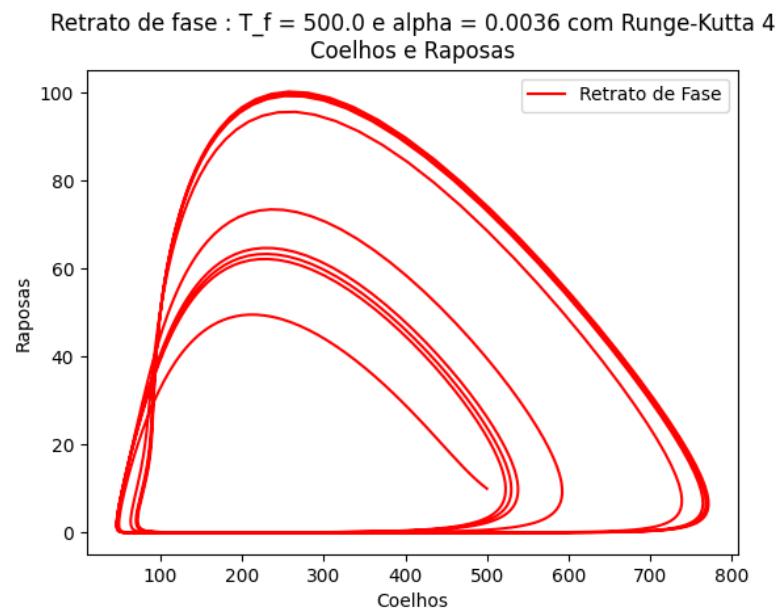


Figura 69: Exercício 3 - Caso 4 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

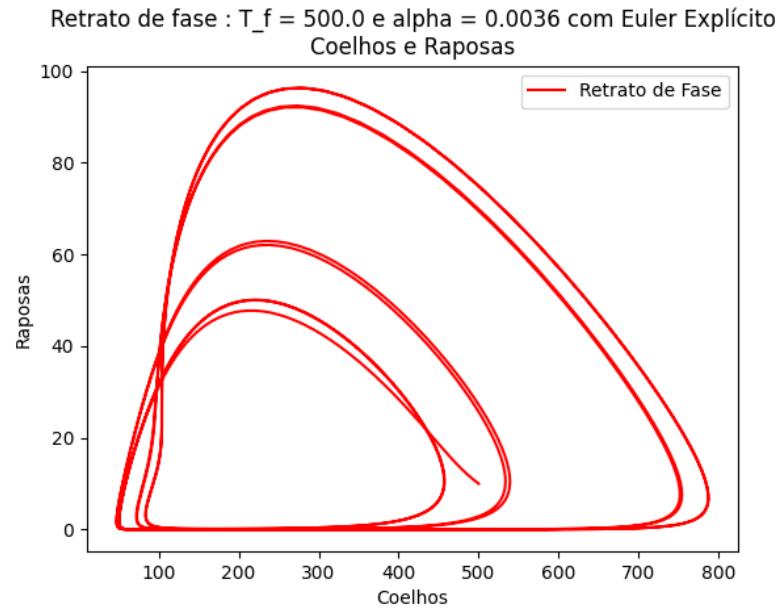


Figura 70: Exercício 3 - Caso 4 - Euler Explícito - Gráfico de Retrato de Fase 2D

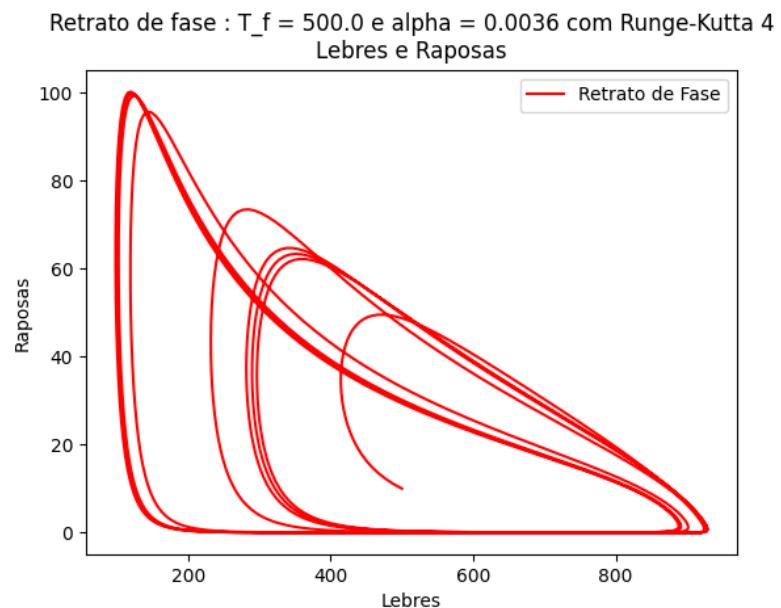


Figura 71: Exercício 3 - Caso 4 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

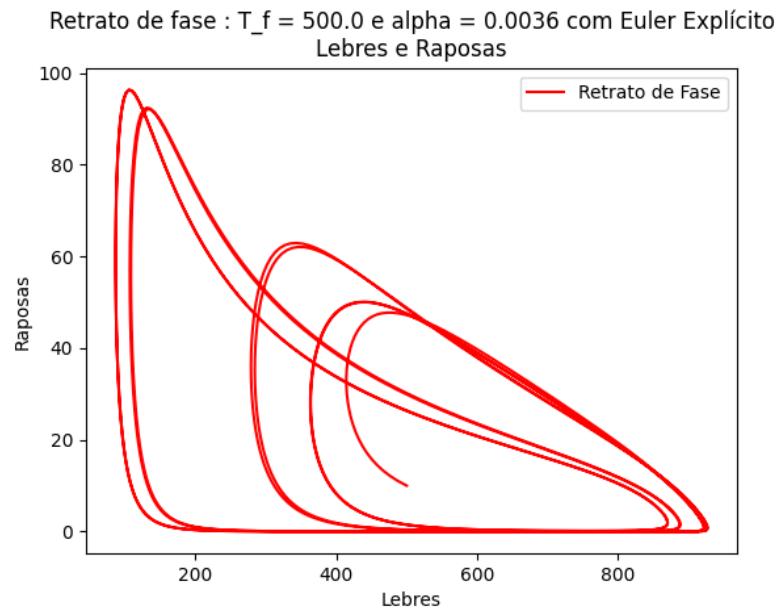


Figura 72: Exercício 3 - Caso 4 - Euler Explícito - Gráfico de Retrato de Fase 2D

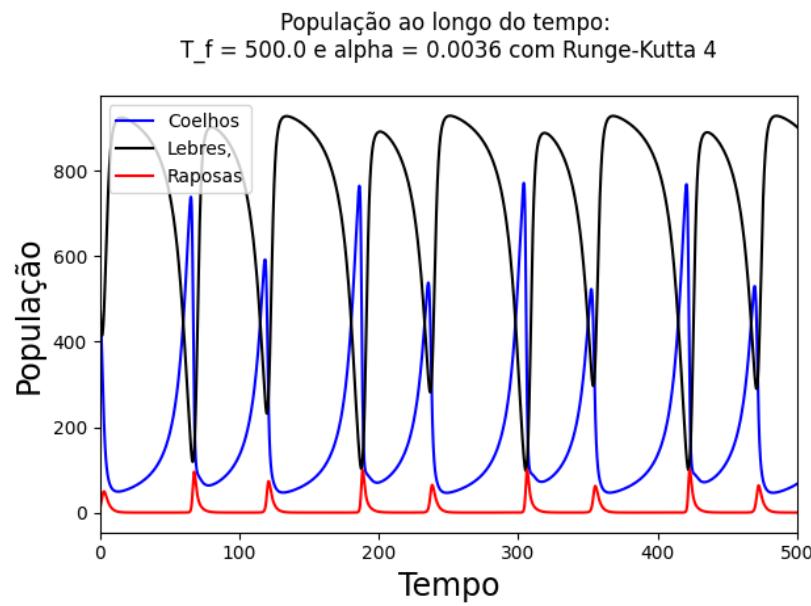


Figura 73: Exercício 3 - Caso 4 - Runge-Kutta 4 - Gráfico de tamanho da população pelo tempo

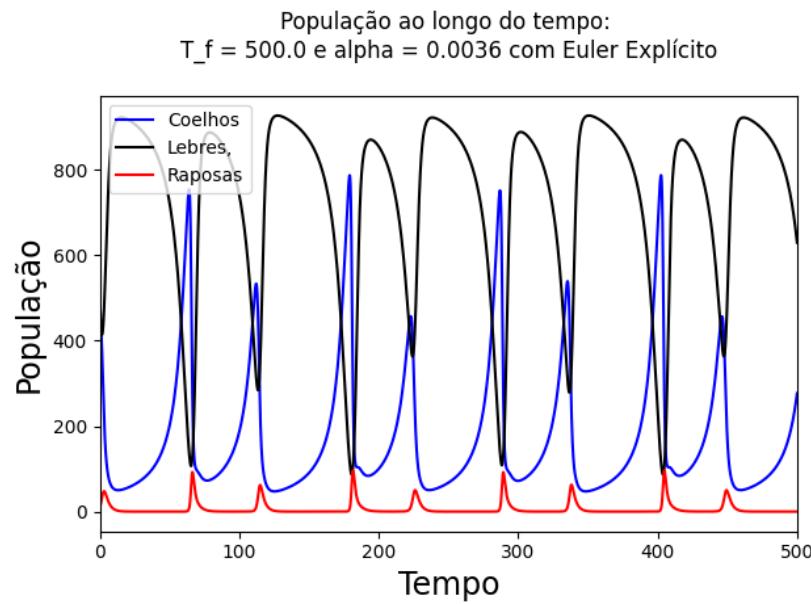


Figura 74: Exercício 3 - Caso 4 - Euler Explícito - Gráfico de tamanho da população pelo tempo

## Caso 5

Retrato de fase para  $T_f = 2000.0$  e  $\alpha = 0.005$  com Runge-Kutta 4

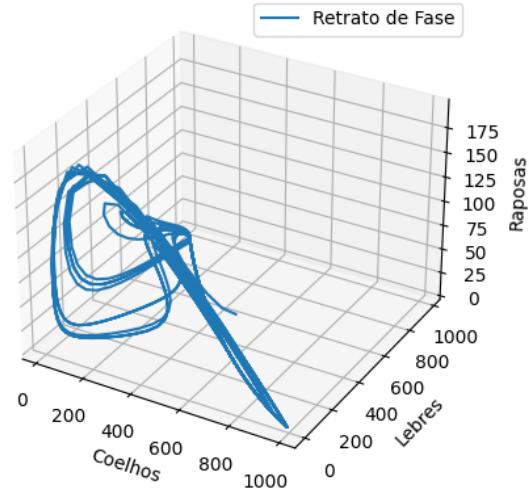


Figura 75: Exercício 3 - Caso 5 - Runge-Kutta 4 - Gráfico de Retrato de Fase 3D

Retrato de fase para  $T_f = 2000.0$  e  $\alpha = 0.005$  com Euler Explícito

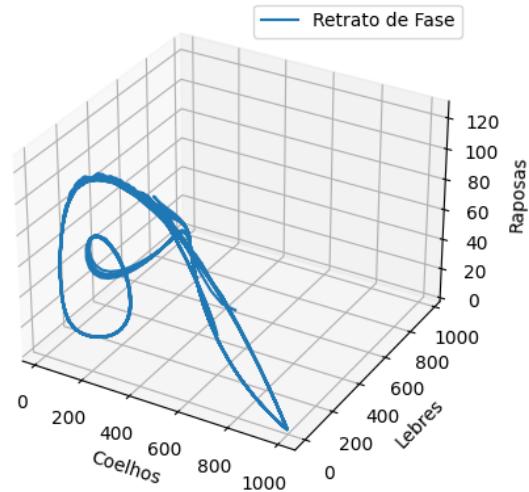


Figura 76: Exercício 3 - Caso 5 - Euler Explícito - Gráfico de Retrato de Fase 3D

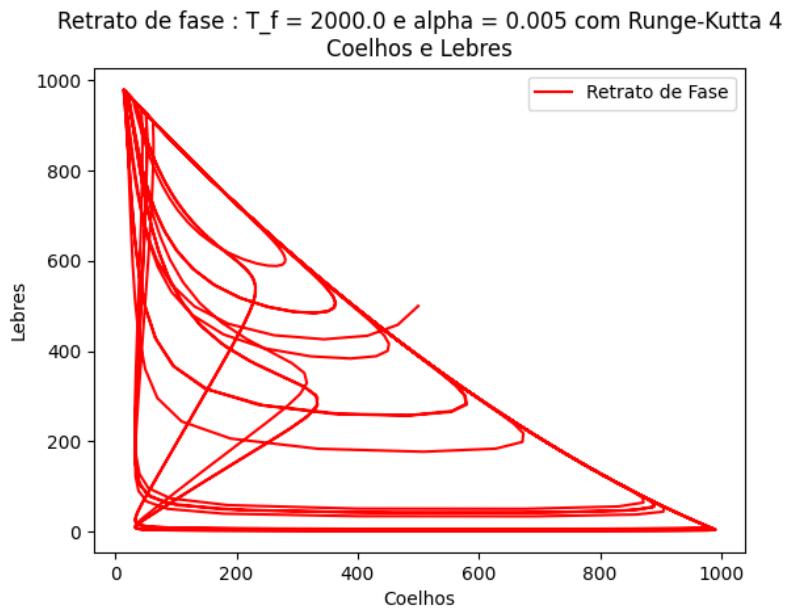


Figura 77: Exercício 3 - Caso 5 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

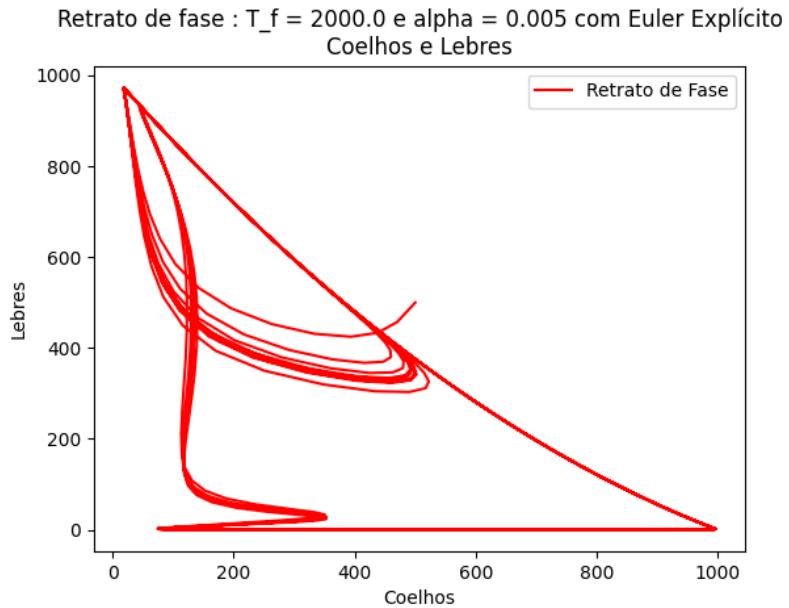


Figura 78: Exercício 3 - Caso 5 - Euler Explícito - Gráfico de Retrato de Fase 2D

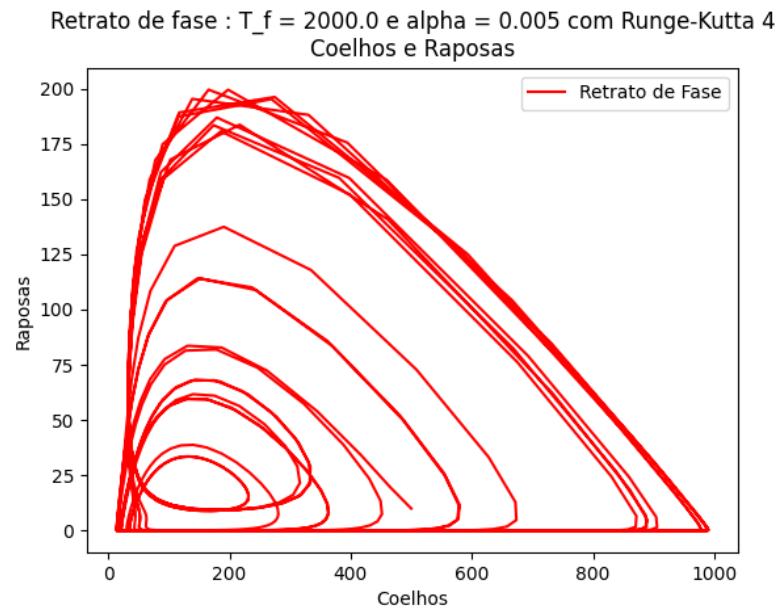


Figura 79: Exercício 3 - Caso 5 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

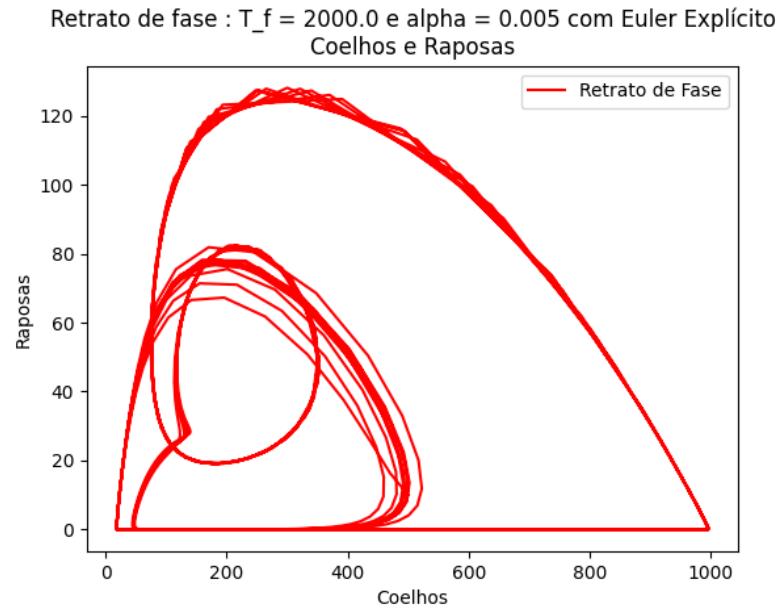


Figura 80: Exercício 3 - Caso 5 - Euler Explícito - Gráfico de Retrato de Fase 2D

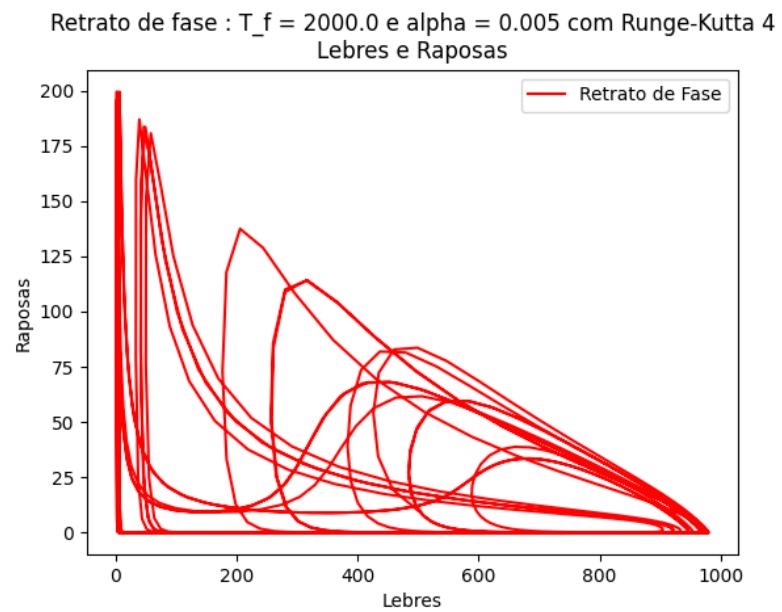


Figura 81: Exercício 3 - Caso 5 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

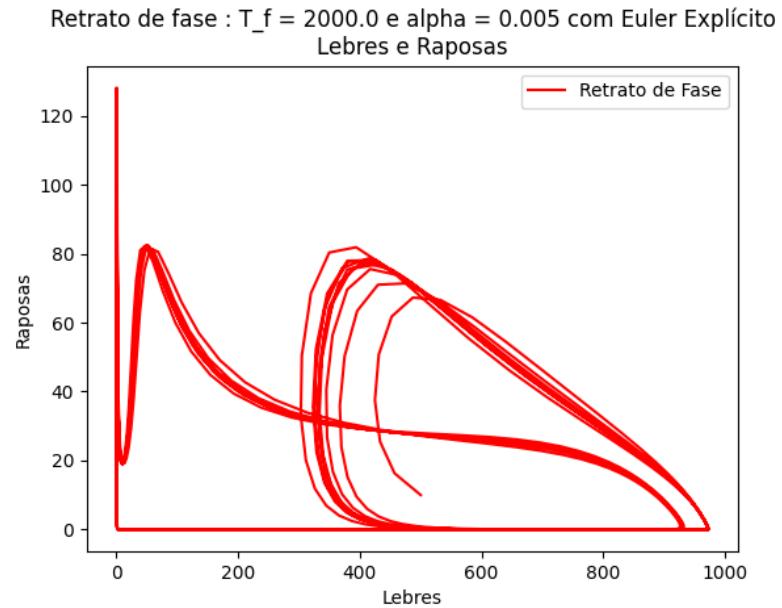


Figura 82: Exercício 3 - Caso 5 - Euler Explícito - Gráfico de Retrato de Fase 2D

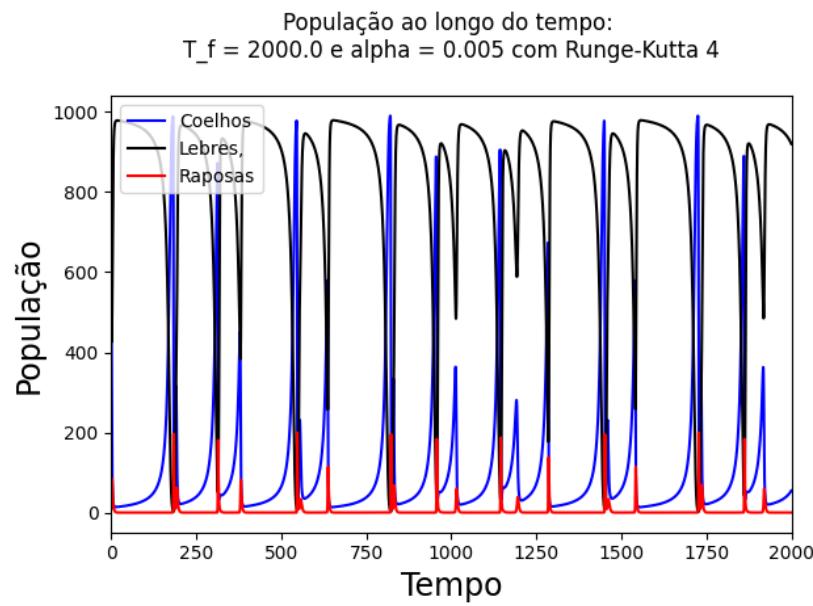


Figura 83: Exercício 3 - Caso 5 - Runge-Kutta 4 - Gráfico de tamanho da população pelo tempo

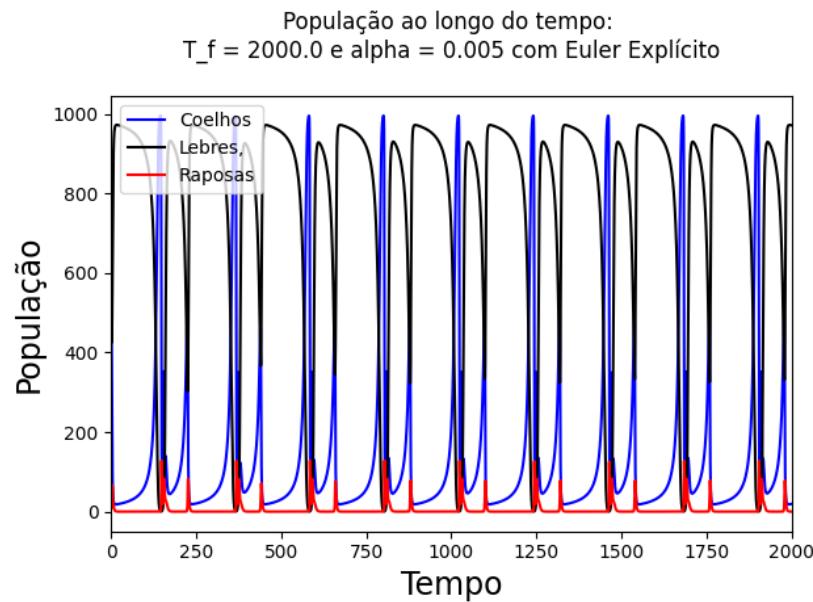


Figura 84: Exercício 3 - Caso 5 - Euler Explícito - Gráfico de tamanho da população pelo tempo

## Caso 6

Retrato de fase para  $T_f = 2000.0$  e  $\alpha = 0.0055$  com Runge-Kutta 4

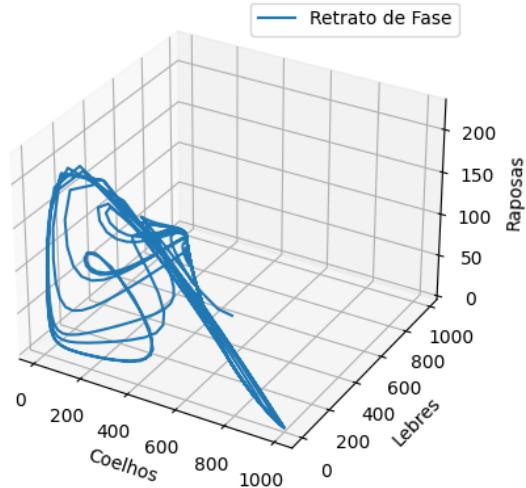


Figura 85: Exercício 3 - Caso 6 - Runge-Kutta 4 - Gráfico de Retrato de Fase 3D

Retrato de fase para  $T_f = 2000.0$  e  $\alpha = 0.0055$  com Euler Explícito

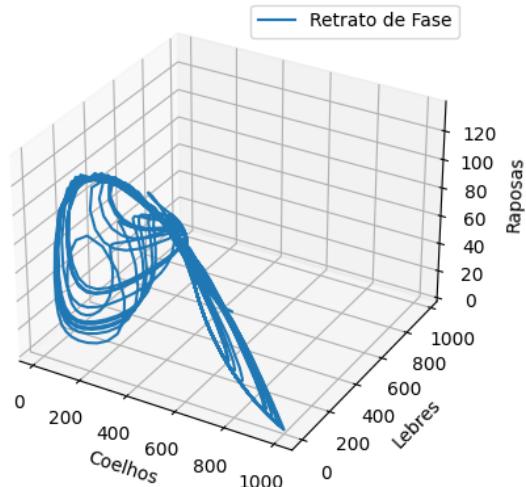


Figura 86: Exercício 3 - Caso 6 - Euler Explícito - Gráfico de Retrato de Fase 3D

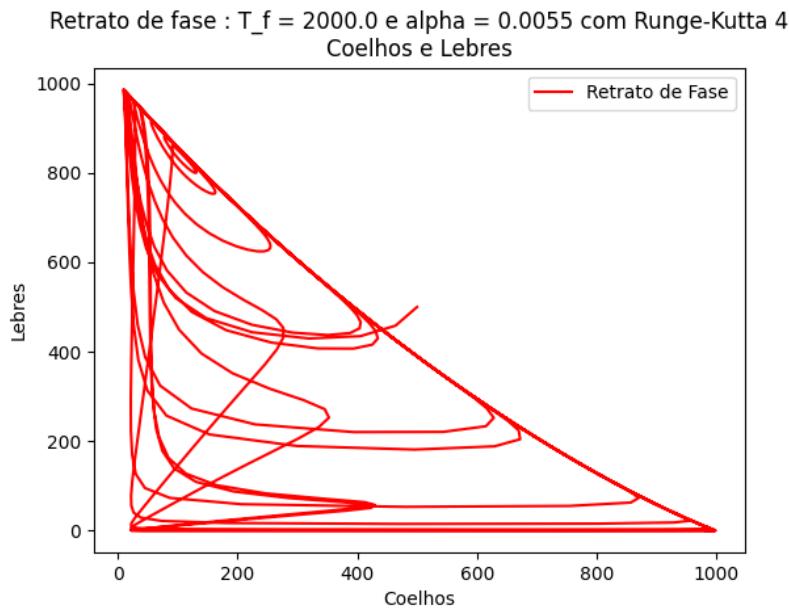


Figura 87: Exercício 3 - Caso 6 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

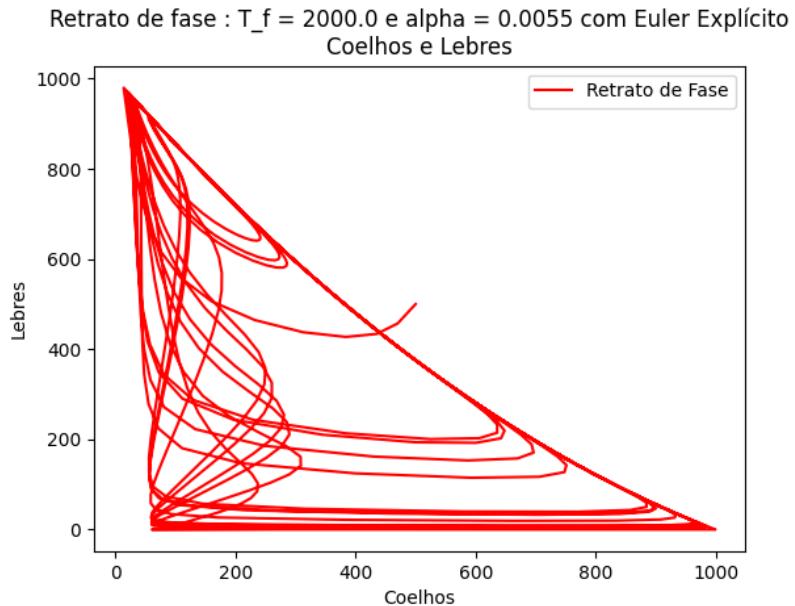


Figura 88: Exercício 3 - Caso 6 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 2000.0$  e  $\alpha = 0.0055$  com Runge-Kutta 4  
Coelhos e Raposas

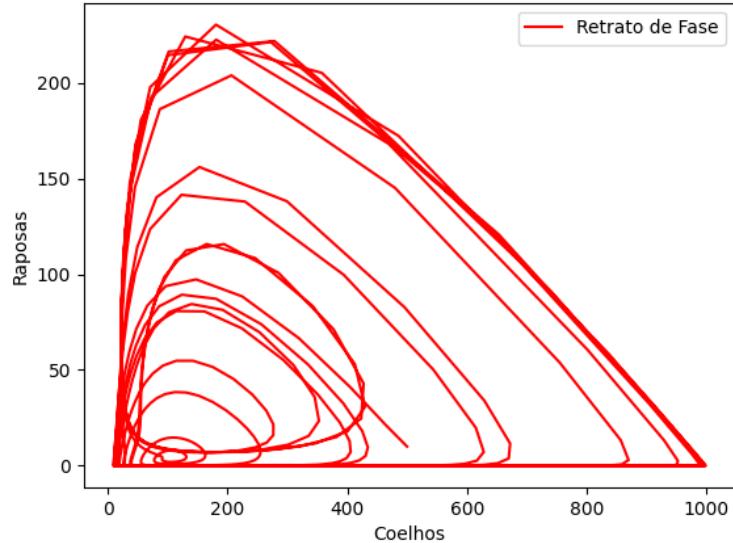


Figura 89: Exercício 3 - Caso 6 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 2000.0$  e  $\alpha = 0.0055$  com Euler Explícito  
Coelhos e Raposas

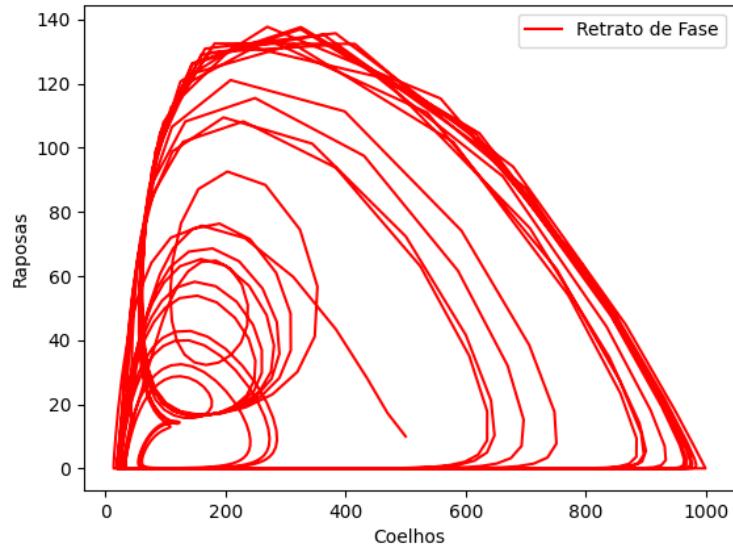


Figura 90: Exercício 3 - Caso 6 - Euler Explícito - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 2000.0$  e  $\alpha = 0.0055$  com Runge-Kutta 4  
Lebres e Raposas

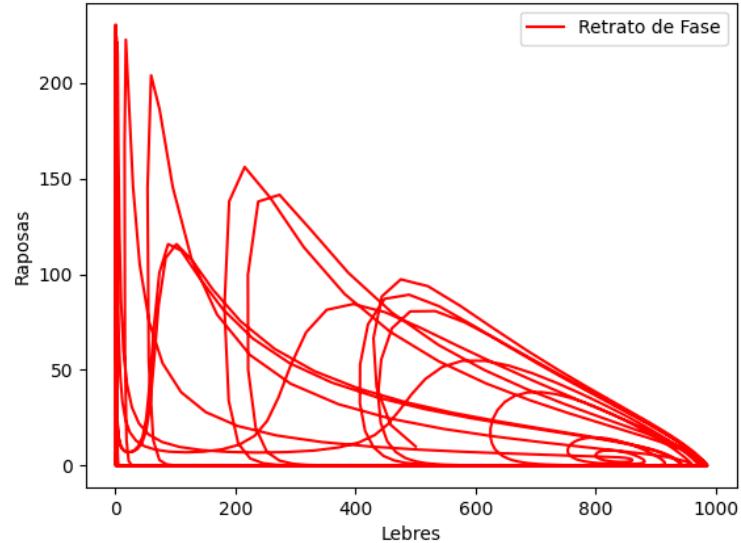


Figura 91: Exercício 3 - Caso 6 - Runge-Kutta 4 - Gráfico de Retrato de Fase 2D

Retrato de fase :  $T_f = 2000.0$  e  $\alpha = 0.0055$  com Euler Explícito  
Lebres e Raposas

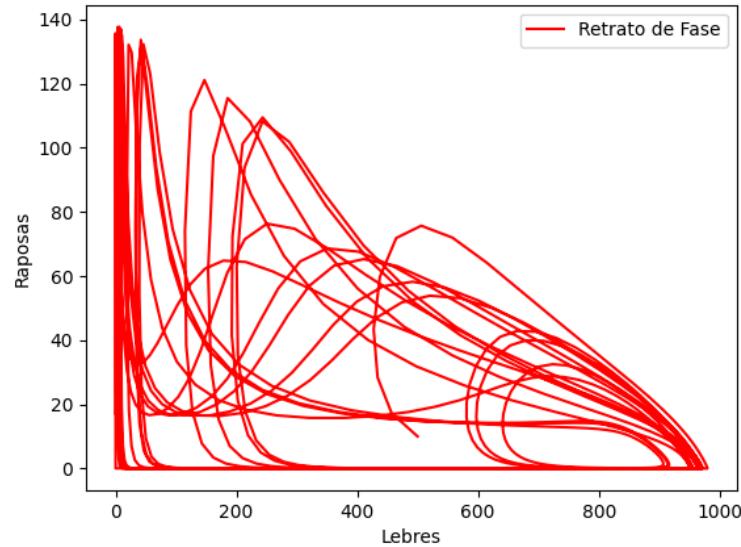


Figura 92: Exercício 3 - Caso 6 - Euler Explícito - Gráfico de Retrato de Fase 2D

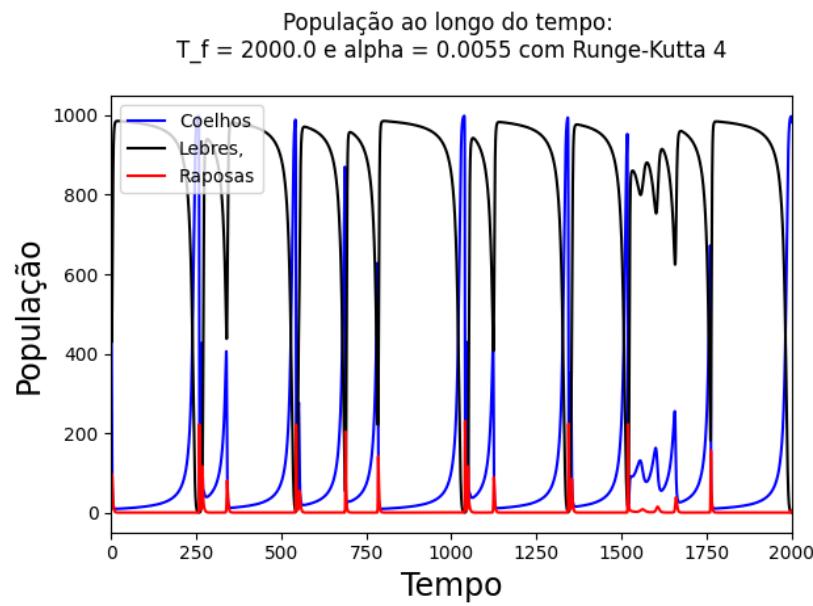


Figura 93: Exercício 3 - Caso 6 - Runge-Kutta 4 - Gráfico de tamanho da população pelo tempo

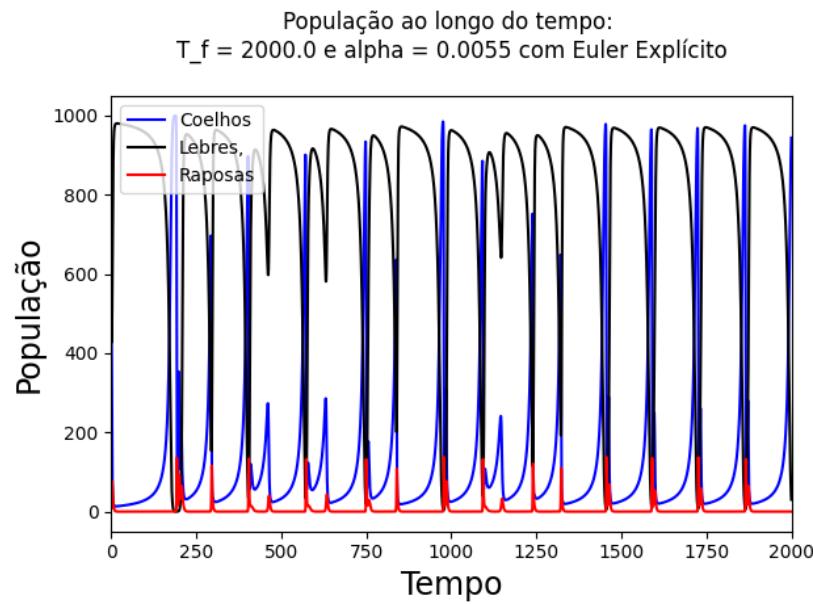


Figura 94: Exercício 3 - Caso 6 - Euler Explícito - Gráfico de tamanho da população pelo tempo

## Comentários

Ao analisar o diferente comportamento das equações para diferentes valores de  $\alpha$  e  $t_f$ , temos que essa mudança do sistema de equações forneceu resultados bem diferentes na simulação dos valores. Observando a equação original deste problema, podemos interpretar que a variável  $\alpha$  é o que determina o quanto as raposas conseguem sobreviver de um determinado número de coelhos. Quanto maior o  $\alpha$ , mais raposas sobreviverão para dado número de coelhos.

Para valores de  $\alpha$  pequenos, as raposas estão bem prejudicadas, e por isso entram em extinção rapidamente. Além disso, como os coelhos ocupam o mesmo nicho das lebres, e por não terem predadores, estes seres dominam o ecossistema, sendo assim, os predominantes, enquanto o número de lebres e de raposas decaem muito.

Conforme  $\alpha$  aumenta, mais estável fica a população de raposas, fazendo com que a população de coelhos se mantenha mais sob controle. Desta forma, como a população de coelhos e de raposas se comportam de forma periódica, é notável esse movimento circular no retrato de fase, que indica essa periodicidade de condições.

É notável que, para valores baixos de  $\alpha$ , o retrato de fase é quase linear, já que a extinção das raposas e das lebres se dá diretamente.

No entanto, para valores medianos de  $\alpha$ , essa extinção se dá de forma gradual, com algumas "ondulações" no meio, formando uma espécie de espiral.

Já para determinados valores de  $\alpha$ , onde o equilíbrio biológico é considerável, os retratos de fase apresentam formas quase cíclicas (curvas que quase se fecham), apresentando que condições muito próximas às iniciais são atingidas – fornecendo assim um ciclo mais estável.

Contudo, é possível visualizar que, ao longo do tempo (se aumentarmos o período de análise  $t_f$ ), é possível notar que estes ciclos não são perfeitos, eventualmente trazendo um desbalanço e desviando bastante das condições iniciais. Desta forma, é fácil concluir que o ciclo não é perfeito.

## 7.5 Teste de Sensibilidade

Os testes de sensibilidades foram feitos conforme o enunciado, ou seja quando  $\alpha = 0.005$ , executando o modelo até  $T_f = 400$  iniciando com 37 coelhos, 75 lebres e 137 raposas. Seguem aqui os resultados:

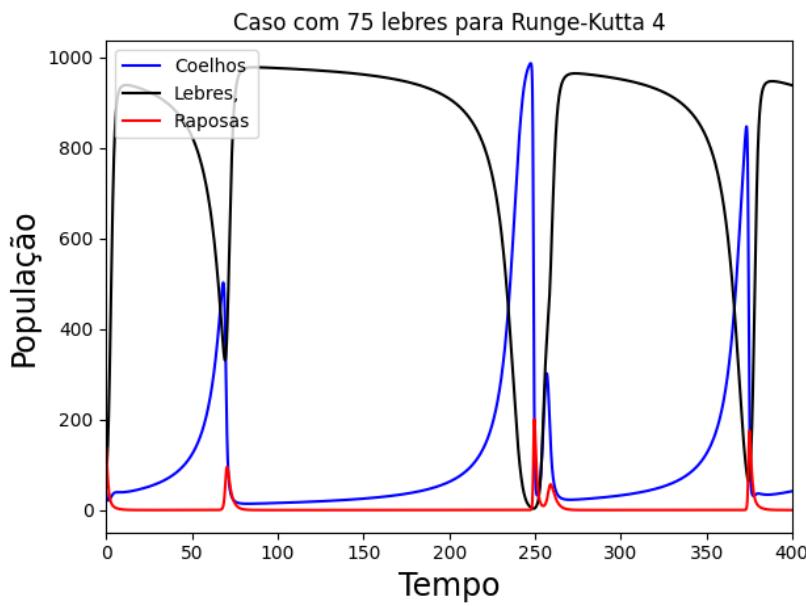


Figura 95: Exercício 3 - Teste de Sensibilidade - Runge-Kutta 4 - Valor inicial de lebres=75

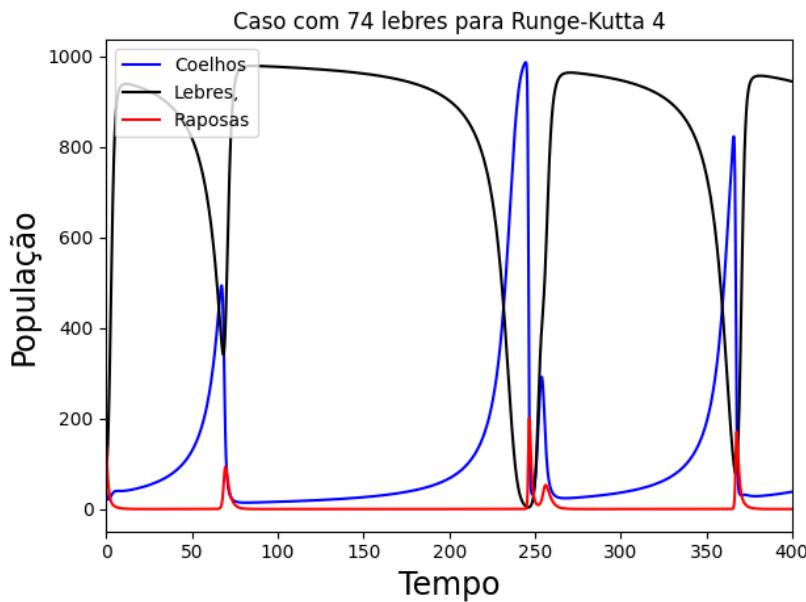


Figura 96: Exercício 3 - Teste de Sensibilidade - Runge-Kutta 4 - Valor inicial de lebres=74

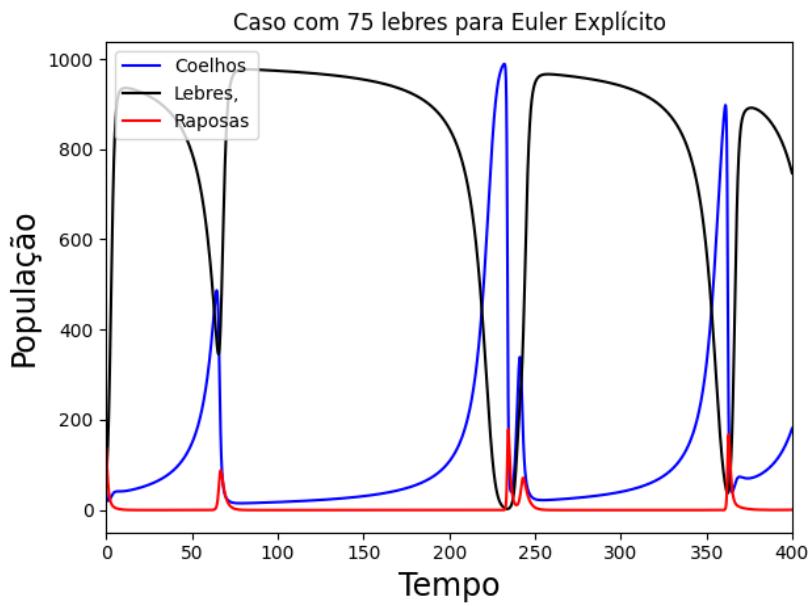


Figura 97: Exercício 3 - Teste de Sensibilidade - Euler Explícito - Valor inicial de lebres=75

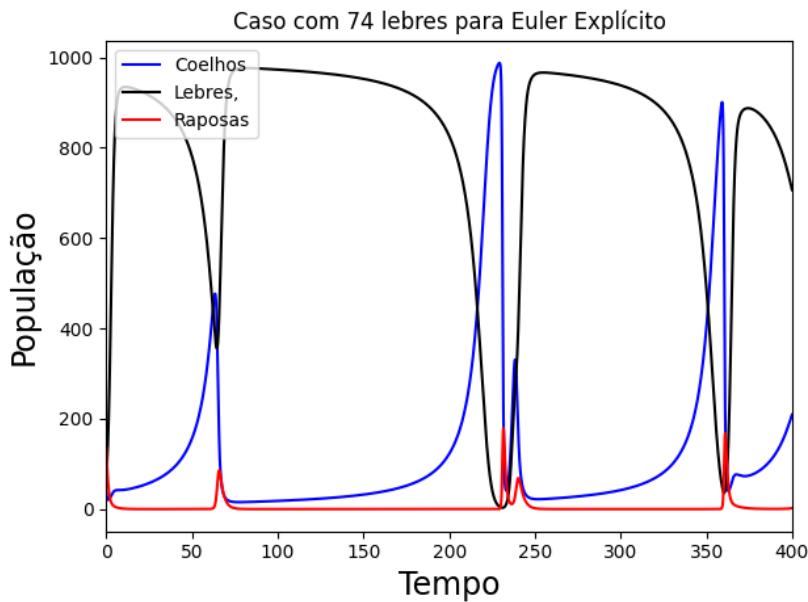


Figura 98: Exercício 3 - Teste de Sensibilidade - Euler Explícito - Valor inicial de lebres=74

Além disso, segue uma comparação dos valores finais de cada espécie para a variação do número inicial de lebres de 75 para 74:

Exercício 3 - Teste de Sensibilidade					
Caso: 1	Método: Runge-Kutta 4	Raposas: 0.009800311325792746	Lebres: 938.7186576838427	Coelhos: 41.67168765660642	
Caso: 2	Método: Runge-Kutta 4	Raposas: 0.0004472238773870274	Lebres: 943.942103746914	Coelhos: 38.07035127746362	
As raposas cresceram: -95.43663601573543% , as Lebres cresceram: 0.5564442573198011%, e os coelhos cresceram: -8.642165896470148%.					
Exercício 3 - Teste de Sensibilidade					
Caso: 1	Método: Euler Explícito	Raposas: 0.6576075210120298	Lebres: 747.061240083793	Coelhos: 180.7344859947337	
Caso: 2	Método: Euler Explícito	Raposas: 2.0714943280615676	Lebres: 706.6969062389971	Coelhos: 208.955357198327	
As raposas cresceram: 215.0046588386986% , as Lebres cresceram: -5.403082328333411%, e os coelhos cresceram: 15.614546968316615%.					

Figura 99: Exercício 3 - Teste de Sensibilidade - Saída com os Resultados

## Comentários

A partir desses comentários, é possível notar que há uma considerável sensibilidade do sistema às entradas iniciais : as raposas, que estavam praticamente extintas, ao se alterar uma unidade na população das lebres, cresceram drasticamente.

Esse efeito era esperado (não a sensibilidade do sistema, mas que as raposas fossem beneficiadas), já que a diminuição do número de lebres também acarreta num aumento do número de coelhos, que são o alimento principal das raposas. Desta forma, com as lebres em uma desvantagem, era esperado que as raposas e os coelhos (que ocupam o mesmo nicho ecológico das lebres) fossem beneficiados.

Já sobre a sensibilidade do sistema, é possível dizer que as raposas sofrem mais com essa sensibilidade do que as lebres e os coelhos – o que também era esperado: em um ecossistema real, quanto mais instável for o ecossistema, menor a presença dos predadores, pois eles sobrevivem de outras populações – e se essas populações flutuarem muito, menos garantida será a sobrevivência dos predadores.

Já quanto as presas, é normal que sejam menos sensíveis, principalmente à uma alteração no número inicial de presas - seus números são usualmente maiores (logo uma unidade equivale a uma diferença menor) e existe uma maior estabilidade em suas quantias – já que os maiores fatores reguladores são a presença dos próprios predadores e a competição do nicho ecológico.

## 8 Conclusão e Comentários Finais

Neste exercício computacional, aplicaram-se métodos para cálculo das soluções de equações diferenciais ordinárias, e também de sistemas desse tipo de equação. Buscou-se, primeiro no exercício 1, testar os métodos desenvolvidos – em específico o método de Runge-Kutta de ordem 4 e o método de Euler implícito.

Com a aplicação destes métodos em **Python**, buscou-se ao máximo modularizar as funções desenvolvidas, de forma a que os métodos resolvessem diferentes tipos de sistemas de equações: sejam elas com uma variável, duas variáveis, ou  $n$  variáveis. Isto se provou ter sido de grande proveito, já que se utilizaram as mesmas funções para diferentes seções do exercício programa.

Com estas aplicações, estudou-se um modelo de comportamento populacional de diferentes espécies em um ecossistema. Utilizando as equações providas pelo enunciado e resolvendo-as com os métodos fornecidos, foi possível apresentar resultados deste modelo ao longo do tempo, e estudando a relação entre as populações por meio dos retratos de fase.

Desta forma, utilizaram-se os métodos estudados para o tratamento de EDOs (e também um dos métodos para encontrar raízes de funções, o método de Newton) para resolver modelos que se assemelham à realidade, provando a utilidade destes métodos numéricos para outras ciências, não só a matemática – mas também a biologia, a física, e muitas outras.

# 9 Apêndice

## 9.1 Código completo plotter.py

```
1 #!/usr/bin/env python3
2
3 ######
4 # File : exercicio1.py
5 # Project : MAP3122 - EP02 - Metodos numericos para
6 #             resolucao de EDOs
7 # Date : April/2021
8 #####
9 # Author : Felipe Bagni
10 # Email : febagni@usp.br
11 # NUSP : 11257571
12 #####
13 # Author : Gabriel Yugo Kishida
14 # Email : gabriel.kishida@usp.br
15 # NUSP : 11257647
16 #####
17
18 import matplotlib.pyplot as plt
19 from mpl_toolkits import mplot3d
20
21 def get_time_array(t0, tf, n):
22     '''
23         Brief : Funcao que recebe o tempo inicial e o final, o numero
24             de iteracoes
25             e devolve o vetor com todos os pontos do tempo que
26             serao analisados.
27         Parameters: t0 - Tempo inicial,
28                     tf - Tempo final,
29                     n - N mero de iteracoes.
30         Returns: time_array - Vetor com os valores de tempo a ser
31             analisados.
32     '''
33     time_array = []
34     h = (tf-t0)/n
35     t = t0
36     for _ in range(1, n+1):
37         time_array.append(t)
38         t += h
```

```

36     return time_array
37
38 def plot_2d_1f(x, y, title, color_letter, labelx, labely, label_msg=""):
39     """
40     Brief :      Funcao que plota um gr fico 2D com uma funcao linear y
41     em funcao de x.
42     Parameters: x - Vetor com valores do eixo x,
43                 y - Vetor com valores do eixo y,
44                 title - T tulo do gr fico ,
45                 color_letter - Cor do gr fico ,
46                 labelx - Nome do Eixo X
47                 labely - Nome do Eixo Y,
48                 label_msg - Legenda do gr fico (pode ser nao existente
49                 ).
50     """
51     if label_msg=="":
52         plt.plot(x, y, color=color_letter)
53     else:
54         plt.plot(x, y, label=label_msg, color=color_letter)
55         plt.legend()
56     plt.xlabel(labelx)
57     plt.ylabel(labely)
58     plt.title(title)
59     plt.show()
60
61 def plot_multiple_graphs(x, y, title, n_rows, n_columns, legend, labelx,
62                         labely):
63     """
64     Brief :      Funcao que plota m ltiplos gr fico 2D com funcoes
65     lineares
66                 y em funcao de x (com diferentes y e um mesmo x).
67     Parameters: x - Vetor com valores do eixo x,
68                 y - Vetor com vetores de valores do eixo y,
69                 title - T tulo do gr fico ,
70                 n_rows - n mero de linhas de gr ficos ,
71                 n_columns - n mero de colunas de gr ficos ,
72                 legend - Legenda para a funcao
73                 labelx - Nome do Eixo X
74                 labely - Nome do Eixo Y,
75
76     for i in range(len(y)):

```

```

73     plt.subplot(n_rows, n_columns, i+1)
74     plt.plot(x,y[i])
75     plt.legend([legend], loc=2, fontsize=6)
76     plt.title("Aproximacao para x" + str(i) + "(t)")
77     plt.xlabel(labelx)
78     plt.ylabel(labely)
79     plt.subplots_adjust(hspace=0.4, wspace = 0.5)
80     plt.suptitle(title)
81     plt.show()
82
83
84 def distance_graph(ts, ys, t, yexact, I, legend1, legend2, labelx,
85                     labely, title):
86     """
87         Brief :      Funcao que plota dois graficos em um plano 2D. Chama-
88                     se distance_graph
89                     pois apresenta visualmente a distancia entre duas
90                     funcoes.
91                     Pode ser usado para comparar um y calculado e um y
92                     exato.
93                     Parameters: ts - Vetor com valores do tempo (eixo x),
94                     ys - Vetor com valores calculados do eixo y,
95                     yexact - Vetor com valores exatos do eixo y,
96                     I - Intervalo limite a ser analisado,
97                     legend1 - Legenda para a funcao ys,
98                     legend2 - Legenda para a funcao yexact,
99                     labelx - Nome do Eixo X,
100                    labely - Nome do Eixo Y,
101                    title - Titulo do grafico ,
102        """
103    plt.plot(ts, ys, 'r')
104    plt.plot(t, yexact, 'b')
105    plt.xlim(I[0], I[1])
106    plt.legend([legend1,
107               legend2], loc=2)
108    plt.xlabel(labelx, fontsize=17)
109    plt.ylabel(labely, fontsize=17)
110    plt.title(title)
111    plt.tight_layout()
112    plt.show()

```

```

111 def plot_multiple_distance_graphs(x, y, exact_y, title, n_rows,
112     n_columns, legend1, legend2, xlabel, ylabel):
113     """
114         Brief :      Funcao que plota diversos graficos de distancia em
115         s uma pagina.
116             Pode ser usado para comparar um y calculado e um y
117             exato.
118             Parameters: x - Vetor com valores do eixo x,
119                         y - Vetor com vetores de valores do eixo y,
120                         exact_y - Vetor com vetores de valores exatos do eixo y
121                         ,
122                         title - Titulo do grafico ,
123                         n_rows - numero de linhas de graficos ,
124                         n_columns - numero de colunas de graficos ,
125                         legend1 - Legenda para a funcao y,
126                         legend2 - Legenda para a funcao exact_y ,
127                         xlabelx - Nome do Eixo X,
128                         labely - Nome do Eixo Y,
129                         """
130
131     for i in range(len(y)):
132         plt.subplot(n_rows, n_columns, i+1)
133         plt.plot(x,y[i])
134         plt.plot(x,exact_y[i])
135         plt.legend([legend1, legend2], loc=2, fontsize=6)
136         plt.title("Aproximacao para x" + str(i) + "(t)")
137         plt.xlabel(xlabel)
138         plt.ylabel(ylabel)
139         plt.subplots_adjust(hspace=0.4, wspace = 0.5)
140     plt.suptitle(title)
141     plt.show()
142
143
144 def distance_graph_3(ts, xs, ys, zs, I, legend1, legend2, legend3,
145     title):
146     """
147         Brief :      Funcao que plota tres funcoes em um grafico .
148         Parameters: ts - Vetor com valores do tempo (eixo x),
149                         xs - Vetor com valores calculados de x para o eixo y,
150                         ys - Vetor com valores calculados de y para o eixo y,
151                         zs - Vetor com valores calculados de z para o eixo y,
152                         I - Intervalo limite a ser analisado ,
153                         legend1 - Legenda para a funcao xs ,
154                         legend2 - Legenda para a funcao ys ,

```

```

148         legend3 - Legenda para a funcao zs,
149         title - Título do gráfico,
150     """
151     plt.plot(ts, xs, "blue")
152     plt.plot(ts, ys, "black")
153     plt.plot(ts, zs, "red")
154     plt.xlim(I[0], I[1])
155     plt.legend([legend1,
156                 legend2,
157                 legend3], loc=2)
158     plt.xlabel("Tempo", fontsize=17)
159     plt.ylabel("População", fontsize=17)
160     plt.title(title)
161     plt.tight_layout()
162     plt.show()
163
164 """
165 def pair_distance_graph_3(ts, xs, ys, zs, I, legend1, legend2, legend3,
166                           labelx, labely, titles):
167     for i in range(2):
168         plt.subplot(1,2,i+1)
169         plt.plot(ts[i], xs[i], "blue")
170         plt.plot(ts[i], ys[i], "black")
171         plt.plot(ts[i], zs[i], "red")
172         plt.title(titles[i])
173         plt.xlim(I[0], I[1])
174         plt.legend([legend1,
175                     legend2,
176                     legend3], loc=2)
177         plt.xlabel(labelx, fontsize=17)
178         plt.ylabel(labely, fontsize=17)
179         plt.tight_layout()
180     plt.show()
181 """
182 def plot_3d_graph(xs, ys, zs, title, labelx, labely, labelz, legend):
183     """
184     Brief :      Função que plota uma curva paramétrica em 3D
185     Parameters: xs - Vetor com valores x do ponto,
186                 ys - Vetor com valores y do ponto,
187                 zs - Vetor com valores z do ponto,
188                 title - Título do gráfico,

```

```

189         xlabelx - Nome do Eixo X,
190         xlabely - Nome do Eixo Y,
191         xlabelz - Nome do Eixo Z,
192         I - Intervalo limite a ser analisado,
193         legend - Legenda para a curva.
194     """
195
196     ax = plt.axes(projection="3d")
197     ax.plot3D(xs,ys,zs)
198     plt.title(title)
199     ax.set_xlabel(labelx)
200     ax.set_ylabel(labely)
201     ax.set_zlabel(labelz)
202     plt.legend([legend])
203     plt.show()

```

Listing 4: Código completo plotter.py

## 9.2 Código completo euler\_backward.py

```

1 #!/usr/bin/env python3
2
3 ######
4 # File      : euler_backward.py
5 # Project   : MAP3122 - EP02 - Metodos numericos para
6 #                  resolucao de EDOs
7 # Date      : April/2021
8 #####
9 # Author    : Felipe Bagni
10 # Email     : febagni@usp.br
11 # NUSP      : 11257571
12 #####
13 # Author    : Gabriel Yugo Kishida
14 # Email     : gabriel.kishida@usp.br
15 # NUSP      : 11257647
16 #####
17
18 import matplotlib.pyplot as plt
19 import numpy as np
20
21
22 def partial_derivative(t, u, partial_in, f, step):

```

```

23 """
24     Brief : Essa fun ao calcula a derivada parcial de uma das funcoes
25     de f (sendo
26         f um vetor de funcoes f1, f2, ...). A derivada parcial
27     derivada na
28         variavel de indice "partial_in" e     feita de forma
29     num rica (para um step
30         pequeno).
31     Par metros: t - valor de tempo,
32             u - variaveis do sistema,
33             partial_in - indice (de "u") da variavel sobre a qual
34     ser    derivada f
35             f - vetor de funcoes, uma das quais ser    derivada,
36             step - O tamanho do passo que define a precisao da
37     derivada numerica.
38     Retorna:    Valor da derivada parcial para os determinados valores
39     de "input".
40 """
41
42     step_u = u.copy()
43     step_u[partial_in] += step
44     return (f(t,step_u) - f(t,u))/step
45
46
47 def newton_iter(t, u, f, h, last_u):
48     """
49     Brief : Essa funcao calcula uma iteracao do m todo de aproximacao
50     de Newton para
51         obter U_k+1 - com o objetivo de se apicar o m todo
52     implementado de Euler.
53     Par metros: t - valor de tempo,
54             u - U_k+1 (antes da iteracao) variaveis do sistema,
55             f - vetor de funcoes, uma das quais ser    derivada,
56             h - tamanho do passo temporal percorrido a cada
57     iteracao ,
58             last_u - U_k vari veis do sistema.
59     Retorna:    Valor de U_k+1 ap s a iteracao de Newton
60     """
61
62     n = len(u)
63     jacobian = np.identity(n)
64     G = np.zeros(len(u))
65     new_t = t + h
66     for i in range(n):
67         for j in range(n):
68

```

```

56         jacobian[i][j] -= h*partial_derivative(new_t, u, j, f[i],
57                                         0.001)
58     inv_jacobian = np.linalg.inv(jacobian)
59     for i in range(len(u)):
60         G[i] = u[i] - h * f[i](t,u) - last_u[i]
61     return u - np.matmul(inv_jacobian, G) # == new_u
62
63 def implicit_euler_iter(u, t, h, f, newton_iter_num):
64     """
65     Brief : Essa funcao calcula uma iteracao do m todo de aproximacao
66     de Newton para
67         obter U_k+1 - com o objetivo de se apicar o m todo
68     impl cito de Euler.
69     Par metros: u - U_k (antes da iteracao) variaveis do sistema,
70                 t - valor de tempo,
71                 h - tamanho do passo temporal percorrido a cada
72     iteracao,
73                 f - vetor de funcoes, uma das quais ser derivada,
74                 newton_iter - numeros de iteracao de newton que sera feitas.
75     Retorna:    Valor de U_k+1 para a iteracao de Euler Impl cito
76     """
77
78     newton_u = u.copy()
79     euler_u = u.copy()
80     for _ in range(newton_iter_num) :
81         newton_u = newton_iter(t, newton_u, f, h, u)
82     for i in range(len(u)):
83         euler_u[i] = u[i] + h * f[i](t, newton_u)
84     return euler_u
85
86 def implicit_euler_system(u, f, t0, tf, n, newton_iter_num):
87     """
88     Brief : Essa funcao calcula iteracoes do m todo de Euler
89     impl cito para um
90         sistema de funcoes.
91     Par metros: u - U_0 valores iniciais do sistema,
92                 f - vetor de funcoes que criam o sistema,
93                 t0 e tf - tempos iniciais e finais da iteracao,
94                 n - numero de iteracoes do m todo de Euler a serem
95     realizadas
96                 newton_iter_num - numeros de iteracao de newton que
97     sera feitas.

```

```

90     Retorna : Valores de u para cada passo temporal calculado.
91     """
92     u_values = []
93     new_u = u.copy()
94     u_values.append(new_u)
95     h = (tf-t0)/n
96     t = t0
97
98     tsol = np.empty(0) # Creates an empty array for t
99     tsol = np.append(tsol, t) # Fills in the first element of tsol
100
101    for _ in range(1, n+1):
102        new_u = implicit_euler_iter(new_u, t, h, f, newton_iter_num)
103        u_values.append(new_u)
104        t += h
105        tsol = np.append(tsol, t) # Saves it in the tsol array
106
107    return [tsol, u_values]

```

Listing 5: Código completo euler\_backward.py

### 9.3 Código completo euler\_forward.py

```

1 #!/usr/bin/env python3
2
3 ######
4 # File       : euler_forward.py
5 # Project    : MAP3122 - EP02 - Metodos numericos para
6 #                  resolucao de EDOs
7 # Date       : April/2021#!/usr/bin/env python3
8 #####
9 # Author     : Felipe Bagni
10 # Email      : febagni@usp.br
11 # NUSP       : 11257571
12 #####
13 # Author     : Gabriel Yugo Kishida
14 # Email      : gabriel.kishida@usp.br
15 # NUSP       : 11257647
16 #####
17
18 import numpy as np

```

```

19 from plotter import distance_graph
20
21 """
22 The Euler's forward method or explicit Euler's method
23 or Euler-Cauchy method is formulated as:
24      $y_{i+1} = y_i + h * f(x_i, y_i),$ 
25
26 where  $f(x_i, y_i)$  is the differential equation evaluated
27 at  $x_i$  and  $y_i$ .
28 """
29
30 def forwardEuler(f, yinit, I, n):
31     """
32         This function/module performs the forward Euler method steps.
33     """
34     m = len(yinit) # Number of ODEs
35
36     h = (I[1] - I[0])/n
37
38     x = I[0] # Initializes variable x
39     y = yinit.copy() # Initializes variable y
40
41     xsol = np.empty(0) # Creates an empty array for x
42     xsol = np.append(xsol, x) # Fills in the first element of xsol
43
44     ysol = []
45     ysol.append(y.copy()) # Fills in the initial conditions
46
47     for _ in range(n):
48
49         for j in range(m):
50             y[j] = y[j] + h*f[j](x, y) # Euler-forward eq.
51
52             x += h # Increase x-step
53             xsol = np.append(xsol, x) # Saves it in the xsol array
54             ysol.append(y.copy()) # Saves all new y's
55
56     return [xsol, ysol]

```

Listing 6: Código completo euler\_forward.py

## 9.4 Código completo rk4.py

```
1 #!/usr/bin/env python3
2
3 ######
4 # File : metodos.py
5 # Project : MAP3122 - EP02 - Metodos numericos para
6 #             resolucao de EDOs
7 # Date : April/2021
8 #####
9 # Author : Felipe Bagni
10 # Email : febagni@usp.br
11 # NUSP : 11257571
12 #####
13 # Author : Gabriel Yugo Kishida
14 # Email : gabriel.kishida@usp.br
15 # NUSP : 11257647
16 #####
17
18 import numpy as np
19
20 def rk4iter(u, t, h, f) :
21     """
22         Brief : Essa funao aplica o algoritmo Runge Kutta em uma unica
23         iteracao
24         Par metros: x - valor a ser iterado,
25                 t - valor de t a iterar,
26                 h - passo da funao ,
27                 f - funcao f para aplicar Runge Kutta.
28     """
29     K1 = np.zeros(len(f))
30     K2 = np.zeros(len(f))
31     K3 = np.zeros(len(f))
32     K4 = np.zeros(len(f))
33     for i in range(len(f)):
34         K1[i] = h*f[i](t,u)
35     for i in range(len(f)):
36         K2[i] = h*f[i](t + h/2,u + K1*0.5)
37     for i in range(len(f)):
38         K3[i] = h*f[i](t + h/2,u + K2*0.5)
39     for i in range(len(f)):
40         K4[i] = h*f[i](t+h,u + K3)
```

```

40     return u + (K1 + 2*K2 +2*K3 + K4)/6
41
42 def rk4system(u, f, t0, tf, n):
43     """
44     Brief: Essa fun ao aplica o algoritmo Runge Kutta resolvendo um
45     SPVI
46             sendo o sistema de forma linear. O sistema pode ser
47     fornecido
48             por meio da matriz A.
49     Parameters: u - valores iniciais,
50                 A - matriz do sistema linear,
51                 t0 - valor inicial de t,
52                 tf - valor final de t,
53                 n - numero de divisoes entre t0 e tf.
54     """
55     newx = np.zeros(len(f))
56     x = u.copy()
57     rk4values = []
58     rk4values.append(x)
59     tsol = []
60     h = (tf-t0)/n
61     t = t0
62     tsol.append(t)
63     for _ in range (1,n+1):
64         newx = rk4iter(x,t,h,f)
65         t = t + h
66         x = newx.copy()
67         tsol.append(t)
68         rk4values.append(x)
69     return [tsol, np.array(rk4values)]
70
71 def calc_error(t_solution, calc_solution, explicit_solution):
72     """
73     Brief : Essa fun ao      respons vel por calcular o erro de cada
74     itera ao
75             tendo as respostas expl citas.
76     Parameters: calc_solution - Os valores de x(t) calculados por algum
77                 m todo iterativo,
78                     explicit_solution - Fun oes de t que sao a solucao
79                     explicita do problema,
80                     t0 / tf - tempo inicial e tempo final, respectivamente,
81                     n - numero de iteracoes realizadas.

```

```

77     Returns:      Um vetor com os valores de erro para cada iteracao.
78
79     """
80
81     errIter = []
82     for k in range (len(calc_solution)):
83         errIter.append(max([abs(explicit_solution(t_solution[k])[i] -
84             calc_solution[k][i]) for i in range (len(calc_solution[k]))]))
85
86     return errIter

```

Listing 7: Código completo rk4.py

## 9.5 Código completo exercício1.py

```

1 #!/usr/bin/env python3
2
3 ######
4 # File       :   exercicio1.py
5 # Project    :   MAP3122 - EP02 - Metodos numericos para
6 #                   resolucao de EDOs
7 # Date       :   April/2021
8 #####
9 # Author     :   Felipe Bagni
10 # Email      :   febagni@usp.br
11 # NUSP       :   11257571
12 #####
13 # Author     :   Gabriel Yugo Kishida
14 # Email      :   gabriel.kishida@usp.br
15 # NUSP       :   11257647
16 #####
17
18 import numpy as np
19 from plotter import plot_2d_1f, get_time_array, distance_graph,
20       plot_multiple_graphs, plot_multiple_distance_graphs
21 from rk4 import rk4system, calc_error
22 from euler_backward import implicit_euler_system
23 from euler_forward import forwardEuler
24
25
26 """
27 Brief : Essa funao fornece a solucao explicita para o item 1 do
primeiro exercicio,

```

```

28         dado determinado valor de t.
29     Parameters: t - valor de tempo.
30     Returns:    Um vetor com os valores de explicitos de x(t) para t
31     """
32
33     x_explicit = np.array([
34         np.exp(-t)*np.sin(t) + np.exp(-3*t)*np.cos(3*t),
35         np.exp(-t)*np.cos(t) + np.exp(-3*t)*np.sin(3*t),
36         -np.exp(-t)*np.sin(t) + np.exp(-3*t)*np.cos(3*t),
37         -np.exp(-t)*np.cos(t) + np.exp(-3*t)*np.sin(3*t)])
38
39     return x_explicit
40
41
42     """
43     Brief : Essa funao devolve os valores exatos de x dado um vetor
44     de tempo "time_array".
45     Parameters: time_array - Um vetor com os valores de tempo.
46     Returns:    Um vetor com os valores exatos de x(t) para cada tempo
47     fornecido
48     """
49
50     exact_x = []
51     for t in range(len(time_array)):
52         exact_x.append(explicit_solution(time_array[t]))
53
54     return exact_x
55
56
57     def plot_x (t,x,title):
58
59     """
60     Brief : Essa funao recebe o tempo t e os valores de x e plota os
61     graficos
62     necessrios para a analise dos resultados
63     Parameters: t - Vetor com todos os valores de tempo analisados
64             x - Matriz com todos os valores (x1,x2,x3... xn) de x
65             analisados para cada t
66             title - titulo do grafico
67     """
68
69     x = np.transpose(x)
70     exact_x = np.transpose(get_exact_x(t))
71     plot_multiple_graphs(t, x, title, 2, 2, "Valor Calculado x(t)", "t"

```

```

66     , "x") # Plota somente o gr fico com a solucoes calculadas
67 plot_multiple_distance_graphs(t, x, exact_x, title, 2, 2 , "Valor
68 Calculado x(t)", "Valor Exato x*(t)", "t", "x") # Plota o gr fico
69 comparativo com as solucoes calculadas e exatas
70
71
72
73 def exercise1_test1():
74     """
75     Brief : Essa fun ao representa o script para a solucao do
76     exercicio 1 - teste 1
77     """
78
79     print("Exerc cio 1 - Teste 1: ")
80
81     x0 = [1,1,1,-1]
82
83     f = []
84
85     # Esses appends foram calculados por meio da matriz A que foi
86     fornecida no enunciado do exerc cio
87     f.append(lambda t,x : -2*x[0] -1*x[1] -1*x[2] -2*x[3])
88     f.append(lambda t,x : 1*x[0] -2*x[1] + 2*x[2] -1*x[3])
89     f.append(lambda t,x : -1*x[0] -2*x[1] -2*x[2] -1*x[3])
90     f.append(lambda t,x : 2*x[0] -1*x[1] + 1*x[2] -2*x[3])
91
92     # Valores de n para os quais os dados serao analisados
93     n = [20, 40, 80, 160, 320, 640]
94
95     # Iterando sobre o vetor n, analisando valores para cada elemento
96     # do vetor
97     for i in range (len(n)-1):
98         [t_sol,x_t] = rk4system(x0, f, 0, 2, n[i]) ##
99         C lculo de RK4 para n_i
100        plot_x(t_sol,x_t,"Gr ficos de x(t) para n=" + str(n[i]))
101        [t_aux,x_aux] = rk4system(x0, f, 0, 2, n[i + 1]) ##
102        C lculo de RK4 para n_i+1
103        erro = calc_error(t_sol, x_t, explicit_solution,) ##
104        C lculo do erro para n_i
105        erro_aux = calc_error(t_aux, x_aux, explicit_solution) ##
106        C lculo do erro para n_i+1
107        R = max(erro)/max(erro_aux) ##
108        C lculo de R
109        print("O R para a iteracao i = " + str(i+1) + " : " + str(R))

```

```

)
97     plot_2d_1f(t_sol, erro, str("Gr fico de E_1(t) para n = " + str
98      (n[i])), 'r', 't', "E_1", "Erro E_1(t)")
99     print("O valor calculado x(Tf) para n = " + str(n[i]) + " : " +
100    str(x_t[n[i]]))
101    print("O valor exato x*(Tf) para n = " + str(n[i]) + " : " +
102      str(explicit_solution(t_sol[n[i]])))
103    print()
104    if i == len(n) - 2 : # ltimo caso: impressao dos dados para n
105      = 640
106      print("O valor calculado x(Tf) para n = " + str(n[i+1]) +
107      " : " + str(x_aux[n[i+1]]))
108      print("O valor exato x*(Tf) para n = " + str(n[i+1]) + " :
109      " + str(explicit_solution(t_aux[n[i+1]])))
110      plot_x(t_aux,x_aux,"Gr ficos de x(t) para n=" + str(n[i
111      +1]))
112      plot_2d_1f(t_aux, erro_aux, str("Gr fico de E(t) para n =" +
113      + str(n[i+1])), 'r', 't', "E_1", "Erro E_1(t)")

114 def exercice1_test2():
115   """
116   Brief : Essa fun ao representa o script para a solucao do
117   exercicio 1 - teste 2
118   """
119
120   print("Exerc cio 1 - Teste 2: ")
121   u_0 = np.array([-8.79])
122   f = []
123   I = np.array([1.1, 3.0])
124   n = 5000
125   newton_iter_num = 7
126   h = (I[1] - I[0])/n
127
128   # Funcao f(t,x) para a qual ser aplicada o metodo.
129   f.append(lambda t,x : 2*t + (x-t*t)*(x-t*t))
130
131   [ts, resposta_euler_implicito] = implicit_euler_system(u_0, f, I
132     [0], I[1], n, newton_iter_num)
133
134   #--- Calculates the exact solution, for comparison ---#
135   dt = int((I[1] - I[0]) / h)
136   t = [I[0]+i*h for i in range(dt+1)]

```

```

128     yexact = []
129     for i in range(dt+1):
130         ye = t[i]*t[i] + 1/(1-t[i])
131         yexact.append(ye)
132
133     plot_2d_if(ts, resposta_euler_implicito, str("Gráfico de Solução por Euler Implicito"), 'r', "t", "x", "solução pelo método todo x(t)")
134     plot_2d_if(t, yexact, str("Gráfico de Solução Explícita"), 'b', "t", "x", "solução exata explícita x*(t)")
135
136     distance_graph(ts, resposta_euler_implicito, t, yexact, I, "Valor Calculado", "Valor Exato", "t", "x", "Compara a solução com a solução explícita")
137
138     #Falta cálculo de E_2 e plots
139
140     E_2 = [abs(yexact[i] - resposta_euler_implicito[i]) for i in range(len(yexact))]
141
142     plot_2d_if(ts, E_2, str("Gráfico de E_2(t)"), 'r', "t", "E_2", "erro E_2(t)")
143
144     print("A solução pelo método de Euler implicito : ", resposta_euler_implicito[n])
145     print("A solução explícita : ", yexact[n])
146
147 def main():
148     """
149     Brief : Essa função eh a main do exercício 1
150     """
151     exercise1_test1()
152     print()
153     exercise1_test2()
154
155 main()

```

Listing 8: Código completo exercício1.py

## 9.6 Código completo exercício2.py

```

1 #!/usr/bin/env python3
2
3 ######
4 # File      :   exercicio2.py
5 # Project   :   MAP3122 - EP02 - Metodos numericos para
6 #                   resolucao de EDOs
7 # Date      :   April/2021
8 #####
9 # Author    :   Felipe Bagni
10 # Email     :   febagni@usp.br
11 # NUSP      :   11257571
12 #####
13 # Author    :   Gabriel Yugo Kishida
14 # Email     :   gabriel.kishida@usp.br
15 # NUSP      :   11257647
16 #####
17
18 import numpy as np
19 from plotter import plot_2d_1f, distance_graph
20 from euler_backward import implicit_euler_system
21 from euler_forward import forwardEuler
22 from rk4 import rk4system
23
24 def init():
25     """
26         Brief : Funcao inicializadora que fornece valores comuns
27             a n lises do exerc cio
28         Returns: Valores iniciais u_0 / Intervalo I = [t0, tf]
29     """
30     x_0 = 1.5
31     y_0 = 1.5
32     I = np.array([0., 10.0])
33     return x_0, y_0, I
34
35 def ODE_solver(n, method):
36     """
37         Brief : Essa fun ao resolve a EDO para diferentes m todos ,
38             aplicando "n" iteracoes.
39         Parameters: n - N mero de iteracoes a serem chamadas ,
40                     method - Nome do m todo a ser utilizado.
41         Returns: ts - Valores de tempo onde a solucao     analisada;
42                  ys - Valores da solu ao da EDO calculados

```

```

41 """
42     x_0, y_0, I = init()
43     u_0 = np.array([x_0, y_0])
44     f = []
45
46     f.append(lambda t,u : ((2/3)*u[0]) - ((4/3)*u[0]*u[1])) #=dx
47     f.append(lambda t,u : u[0]*u[1] - u[1]) #=dy
48
49     if method=="euler_forward":
50         [ts, ys] = forwardEuler(f, u_0, I, n)
51     elif method=="euler_backward":
52         [ts, ys] = implicit_euler_system(u_0, f, I[0], I[1], n,
53         newton_iter_num=7)
54     elif method=="rk4":
55         [ts, ys] = rk4system(u_0, f, I[0], I[1], n)
56     else:
57         print("Error: Method not found")
58
59     return [ts, ys]
60
61 def plot_graphs(ts, ys, method):
62 """
63     Brief : Essa funao plota os grficos em funo do tempo ts, e
64     tambm cria o retrato
65         de fase dos valores de ys.
66     Parameters: ts - Valores de tempo onde a soluao analisada;
67                 ys - Valores da soluao da EDO calculados
68 """
69     _, _, I = init()
70
71     if method=="euler_forward":
72         method_name = "Euler Expl cito"
73     elif method=="euler_backward":
74         method_name = "Euler Impl cito"
75     elif method=="rk4":
76         method_name = "Runge-Kutta 4"
77     else:
78         print("Error: Method not found")
79
80     x_sol = np.transpose(ys)[0]

```

```

81     y_sol = np.transpose(ys)[1]
82
83     distance_graph(ts, y_sol, ts, x_sol, I, "Raposas", "Coelhos", "tempo", "popula o", str("Tamanho da popula o ao longo do tempo", por " + method_name))
84
85     plot_2d_1f(x_sol, y_sol, str("Gr fico de Retrato de fase Coelhos X Raposas, por " + method_name), 'g', "coelhos", "raposas", "retrato de fase")
86
87 def ex2_1():
88     """
89     Brief : Essa fun ao resolve o que requerido no exerc cio 2.1:
90         Resolu ao da EDO utilizando o m todo de Euler expl cito
91     """
92     print("Exerc cio 2.1: ")
93
94     [ts, ys] = ODE_solver(n=500, method="euler_forward")
95
96     plot_graphs(ts, ys, method="euler_forward")
97
98 def ex2_2():
99     """
100    Brief : Essa fun ao resolve o que requerido no exerc cio 2.2
101        Resolu ao utilizando o m todo de Euler Impl cito
102    """
103    print("Exerc cio 2.2: ")
104
105    [ts, ys] = ODE_solver(n=1000, method="euler_backward")
106
107    plot_graphs(ts, ys, method="euler_backward")
108
109
110 def ex2_3():
111     """
112     Brief : Essa fun ao resolve o que requerido no exerc cio 2.3
113         C lculo das diferen as entre o m todo Impl cito e Expl cito
114     """
115     print("Exerc cio 2.3: ")
116     _, _, I = init()
117     n = [250, 500, 1000, 2000, 4000]

```

```

118     for n_i in n:
119         [ts_forward, ys_forward] = ODE_solver(n=n_i, method="euler_forward")
120         [ts_backward, ys_backward] = ODE_solver(n=n_i, method="euler_backward")
121
122         E = [np.asarray(j) - np.asarray(ys_forward[i]) for i,j in
123             enumerate(ys_backward)]
124
125         E_x_sol = np.transpose(E)[0]
126         E_y_sol = np.transpose(E)[1]
127
128         distance_graph(ts_backward, E_y_sol, ts_forward, E_x_sol, I, "E_y", "E_x", "t", "E", str("Erro em fun o do tempo para n = " +
129             str(n_i)))
130
131     def ex2_4():
132         """
133             Brief : Essa fun ao resolve o que      requerido no exerc cio 2.4
134             Resolu ao da EDO utilizando o m todo de Runge-Kutta de
135             ordem 4.
136         """
137         print("Exerc cio 2.4: ")
138
139     def main():
140         """
141             Brief : Essa fun ao eh a main do exercicio 2
142         """
143         ex2_1()
144         ex2_2()
145         ex2_3()
146         ex2_4()
147
148     main()

```

Listing 9: Código completo exercício2.py

## 9.7 Código completo exercício3.py

```
1 #!/usr/bin/env python3
2
3 ######
4 # File : exercicio3.py
5 # Project : MAP3122 - EP02 - Metodos numericos para
6 #             resolucao de EDOs
7 # Date : April/2021
8 #####
9 # Author : Felipe Bagni
10 # Email : febagni@usp.br
11 # NUSP : 11257571
12 #####
13 # Author : Gabriel Yugo Kishida
14 # Email : gabriel.kishida@usp.br
15 # NUSP : 11257647
16 #####
17
18 import numpy as np
19 from plotter import plot_2d_1f, distance_graph, plot_3d_graph,
20       distance_graph_3
21 from euler_backward import implicit_euler_system
22 from euler_forward import forwardEuler
23 from rk4 import rk4system
24
25 def init():
26     """
27         Brief : Funcao inicializadora que fornece valores comuns
28             as lises do exerc cio
29         Returns: N mero de iteracoes "n",
30                  Tempo inicial "t0",
31                  Vetor de constantes "B" (para o sistema de EDOs),
32                  Matriz de Constantes "A" (para o sistema de EDOs).
33     """
34     n = 5000
35     t0 = 0
36     B = np.array([1.0, 1.0, -1.0])
37     A = np.array([[0.0010, 0.001, 0.015],
38                  [0.0015, 0.001, 0.001],
39                  [0.000, -0.0005, 0.0]])
40
41     return n, t0, B, A
```

```

39
40 def get_f(A,B):
41     """
42         Brief :      Funcao que fornece as funcoes f do sistema de EDOs,
43                     dadas a matriz A
44                     e o vetor B.
45         Parameters: Matriz de Constantes "A",
46                     Vetor de Constantes "B".
47         Returns:    Vetor com as funcoes do Sistema.
48     """
49     f = []
50     #f.append(lambda t, u : u[0]*(1.0 - 0.001*u[0] - 0.001*u[1] -
51     #0.0015*u[2]))
52     #f.append(lambda t, u : u[1]*(1.0 - 0.0015*u[0] - 0.001*u[1] -
53     #0.001*u[2]))
54     #f.append(lambda t, u : u[2]*(-1.0 + 0.0033*u[0] - (-0.0005)*u[1] -
55     #0*u[2]))
56     #return f
57     for i in range(len(A)):
58         f.append(lambda t, u, i=i: u[i]*(B[i] - A[i][0]*u[0] - A[i][1]*u[1] -
59         A[i][2]*u[2]))
60     return f
61
62
63 def divide_sol(sol):
64     """
65         Brief :      Funcao que recebe as respostas de um sistema de EDOs
66                     com 3 variaveis
67                     e devolve os valores em tres vetores distintos x, y e
68                     z.
69         Parameters: Matriz de solucoes "sol".
70         Returns:    Vetores com respostas x, y e z
71     """
72     xs = np.transpose(sol)[0]
73     ys = np.transpose(sol)[1]
74     zs = np.transpose(sol)[2]
75     return xs, ys, zs
76
77
78 def ex3_methods(method):
79     """
80         Brief :      Funcao que calcula a solucao para o problema proposto
81                     pelo exercicio 3,
82                     para diferentes tipos de todos de resolucao de EDO.

```

```

    Al m de calcular
        a solucao, tamb m plota os gr ficos para a devida
        analise.
    Parameters: method - Nome do m todo a ser utilizado
    """
    n, t0, B, A = init()

    u_0 = np.array([500., 500., 10.])
    tf = np.array([100., 500., 2000.])
    alpha = np.array([0.001, 0.002, 0.0033, 0.0036, 0.005, 0.0055])

    for k in range(len(alpha)):
        A[2][0] = -alpha[k]
        tf_iter = tf[int(k/2)]
        f = get_f(A, B)
        I = [t0, tf_iter]

    if method == "rk4":
        [ts, sol] = rk4system(u_0, f, t0, tf_iter, n)
        method_name = "Runge-Kutta 4"
    elif method == "euler_forward":
        [ts, sol] = forwardEuler(f, u_0, I, n)
        method_name = "Euler Expl cito"
    else : print("Error: Method not found")
    xs, ys, zs = divide_sol(sol)

    time_title = "T_f = " + str(tf_iter)
    alpha_title = "alpha = " + str(alpha[k])
    title_complement = str(time_title + " e " + alpha_title + " com"
    " + method_name + "\n")

    plot_3d_graph(xs, ys, zs, "Retrato de fase para " +
title_complement, "Coelhos", "Lebres", "Raposas", "Retrato de Fase")

    plot_2d_1f(xs, ys, "Retrato de fase : " + title_complement + " "
Coelhos e Lebres", "red", "Coelhos", "Lebres", "Retrato de Fase")
    plot_2d_1f(xs, zs, "Retrato de fase : " + title_complement + " "
Coelhos e Raposas", "red", "Coelhos", "Raposas", "Retrato de Fase")
    plot_2d_1f(ys, zs, "Retrato de fase : " + title_complement + " "
Lebres e Raposas", "red", "Lebres", "Raposas", "Retrato de Fase")

    distance_graph_3(ts, xs, ys, zs, I, "Coelhos", "Lebres,", "

```

```

Raposas", "Popula o ao longo do tempo: \n" + title_complement)

108
109
110 def ex3_1():
111     """
112         Brief :      Funcao que Realiza os c lculos pedidos para a fun ao
113             de Runge Kutta 4
114                 e para o Euler expl cito.
115     """
116     print("Exerc cio 3.1")
117     ex3_methods(method="rk4")
118     ex3_methods(method="euler_forward")

119 def ex3_sensibility_test(method):
120     """
121         Brief :      Funcao que realiza o teste de sensibilidade s
122             diferencias das condicoes
123                 inicioais de um sistema de EDOs. O teste pode ser feito
124             em diferentes
125                 m todos. Al m disso, plota os gr ficos a serem
126             analisados.
127             Parameters: method - Nome do m todo de resoluao de EDO a ser
128             utilizado.
129     """
130     print("Exerc cio 3 - Teste de Sensibilidade")
131
132     caso1 = np.array([37., 75., 137.])
133     caso2 = np.array([37., 74., 137.])
134
135     n, t0, B, A = init()
136     tf = np.array([400])
137     alpha = np.array([0.005])
138
139     A[2][0] = -alpha[0]
140     tf_iter = tf[0]
141     f = get_f(A, B)
142     I = [t0, tf_iter]
143
144     if method == "rk4":
145         [ts1, sol1] = rk4system(caso1, f, t0, tf_iter, n)
146         [ts2, sol2] = rk4system(caso2, f, t0, tf_iter, n)
147         method_name = "Runge-Kutta 4"

```

```

144     elif method == "euler_forward":
145         [ts1, sol1] = forwardEuler(f, caso1, I, n)
146         [ts2, sol2] = forwardEuler(f, caso2, I, n)
147         method_name = "Euler Expl cito"
148     else : print("Error: Method not found")
149     print(" | Caso: 1 | Metodo: " + method_name + " | Raposas: " + str(
150         sol1[n][2]) + " | Lebres: " + str(sol1[n][1]) + " | Coelhos: " + str(
151         sol1[n][0]) + " | ")
152     print(" | Caso: 2 | Metodo: " + method_name + " | Raposas: " + str(
153         sol2[n][2]) + " | Lebres: " + str(sol2[n][1]) + " | Coelhos: " + str(
154         sol2[n][0]) + " | ")
155     print()
156     print("As raposas cresceram: " + str(((sol2[n][2]-sol1[n][2])/sol1[
157         n][2])*100) + "% , as Lebres cresceram: " + str(((sol2[n][1]-sol1[n]
158         ][1])/sol1[n][1])*100) + "%, e os coelhos cresceram: " + str(((sol2[
159         n][0]-sol1[n][0])/sol1[n][0])*100) + "% .")
160     print()
161     xs1, ys1, zs1 = divide_sol(sol1)
162     xs2, ys2, zs2 = divide_sol(sol2)
163
164     ts = [ts1, ts2]
165     xs = [xs1, xs2]
166     ys = [ys1, ys2]
167     zs = [zs1, zs2]
168
169     title_complement = " para " + method_name
170     titles = ["Caso com 75 lebres" + title_complement, "Caso com 74
171     lebres" + title_complement]
172
173     for i in range(2):
174         distance_graph_3(ts[i], xs[i], ys[i], zs[i], I, "Coelhos", "
175         Lebres", "Raposas", titles[i])
176
177 def ex3_2():
178     """
179     Brief :      Funcao que realiza o teste de sensibilidade para os
180     dois casos
181             propostos pelo exerc cio.
182     """
183     ex3_sensibility_test("rk4")
184     ex3_sensibility_test("euler_forward")
185

```

```
176 def main():
177     """
178     Brief : Essa funcao eh a main do exercicio 3
179     """
180     ex3_1()
181     print()
182     ex3_2()
183
184 main()
```

Listing 10: Código completo exercício3.py

## Referências

- [1] LAURAIN, A. ***EP2\_V1***. Moodle da Disciplina MAP3122 - 1o Quadrimestre 2021
- [2] LAURAIN, A. ***Métodos numéricos para EDOs***. Moodle da Disciplina MAP3122 - 1o Quadrimestre 2021
- [3] LAURAIN, A. ***Raízes de equações***. Moodle da Disciplina MAP3122 - 1o Quadrimestre 2021