

Relatório da Tarefa 3

Erik Davino Vincent - Artur Ortiz

NUSP: 10736584 - 10734071

BMAC - Turma 54

September 24, 2019



IME - USP

1 Introdução

O objetivo desse relatório é a análise dos métodos de Euler Implícito e o Método do Trapézio, assim como encontrar a solução numérica para o seguinte sistema presa-predador de Lotka-Volterra, analisando sua convergência através de gráficos:

$$\begin{cases} \dot{x}(t) = 0.87x(t) - 0.27x(t)y(t) \\ \dot{y}(t) = -0.38y(t) + 0.25y(t)x(t) \\ x(0) = 3.5 \\ y(0) = 2.7 \end{cases}$$

Para tal, antes de mais nada, implementamos os métodos em Python e utilizamos o seguinte sistema de EDOs com solução conhecida no intervalo $[t_0, t_f] = [1, 2]$ para verificar a consistência dos métodos (i.e. verificar a convergência do erro para 0 e verificar a convergência da ordem dos métodos):

$$\begin{cases} \dot{x}(t) = tx(t) \\ \dot{y}(t) = ty(t) \\ x(1) = 1 \\ y(1) = 1 \end{cases}$$

1.1 Métodos e implementação

O método de Euler Implícito pode ser escrito da forma:

$$\begin{aligned} x(t_0) &= x_0 \\ t_{k+1} &= t_k + \Delta t \\ x_{k+1} &\approx x_k + \Delta t f(t_{k+1}, x_{k+1}) \end{aligned} \tag{1}$$

que pode ser reescrito como:

$$\begin{aligned} x(t_0) &= x_0 \\ t_{k+1} &= t_k + \Delta t \\ x_{k+1} &\approx \text{RAIZ}(x - x_k - \Delta t f(t_{k+1}, x) = 0) \end{aligned} \tag{2}$$

onde $f(t, x)$ é uma função de passo do método (para nosso caso, o sistema de equações apresentado). Definimos: $[t_0, t_f] \in \text{Dom}(x(t))$, $\Delta t = \frac{t_f - t_0}{n}$, n o número de passos do método. Como estamos falando de um problema bidimensional, podemos reescrever nosso problema como:

$$\begin{aligned} \vec{X}(t_0) &= \vec{X}_0 = \begin{bmatrix} x(t_0) = x_0 \\ y(t_0) = y_0 \end{bmatrix} \\ t_{k+1} &= t_k + \Delta t \\ \vec{X}_{k+1} &= \begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} \text{RAIZ}(x - x_k - \Delta t \dot{x}(t_{k+1}) = 0) \\ \text{RAIZ}(y - y_k - \Delta t \dot{y}(t_{k+1}) = 0) \end{bmatrix} \end{aligned} \tag{3}$$

Note que para encontrar o próximo ponto \vec{X}_{k+1} , devemos encontrar as raízes na equação (3). Para tal, utilizamos o Método de Newton em duas dimensões.

O método do Trapézio pode ser escrito da forma:

$$\begin{aligned}
x(t_0) &= x_0 \\
t_{k+1} &= t_k + \Delta t \\
x_{k+1} &\approx x_k + \frac{\Delta t}{2}(f(t_{k+1}, x_{k+1}) + f(t_k, x_k))
\end{aligned} \tag{4}$$

que pode ser reescrito como:

$$\begin{aligned}
x(t_0) &= x_0 \\
t_{k+1} &= t_k + \Delta t \\
x_{k+1} &\approx \text{RAIZ}(x - x_k - \frac{\Delta t}{2}(f(t_{k+1}, x) + f(t_k, x_k)) = 0)
\end{aligned} \tag{5}$$

onde $f(t, x)$ é uma função de passo do método (para nosso caso, o sistema de equações apresentado). Definimos: $[t_0, t_f] \in \text{Dom}(x(t))$, $\Delta t = \frac{t_f - t_0}{n}$, n o número de passos do método. Como estamos falando de um problema bidimensional, podemos reescrever nosso problema como:

$$\begin{aligned}
\vec{X}(t_0) &= \vec{X}_0 = \begin{bmatrix} x(t_0) = x_0 \\ y(t_0) = y_0 \end{bmatrix} \\
t_{k+1} &= t_k + \Delta t \\
\vec{X}_{k+1} &= \begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} \text{RAIZ}(x - x_k - \frac{\Delta t}{2}(\dot{x}(t_{k+1}) + \dot{x}(t_k)) = 0) \\ \text{RAIZ}(y - y_k - \frac{\Delta t}{2}(\dot{y}(t_{k+1}) + \dot{y}(t_k)) = 0) \end{bmatrix}
\end{aligned} \tag{6}$$

Note que para encontrar o próximo ponto \vec{X}_{k+1} , devemos encontrar as raízes na equação (6). Para tal, utilizamos o Método de Newton em duas dimensões.

1.2 Método de Newton Bidimensional

O Método de Newton é um método para encontrar a raiz de uma função f em um intervalo $[a, b]$, dado que:

- Há somente uma raiz no intervalo dado;
- $f(a)f(b) \leq 0$
- $f''(x) \leq 0 \vee f''(x) \geq 0$, $x \in [a, b]$

Se as condições satisfazem, executamos o seguinte loop i vezes, dado algum $x_0 \in [a, b]$:

$$\begin{aligned}
\text{Se } x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \in [a, b], \text{ então:} \\
x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \\
k = k + 1 \\
\text{Se } k = i, \text{ então:} \\
RAIZ(f[a, b]) = x_{k+1}
\end{aligned} \tag{7}$$

Para o resolver um problema bidimensional, basta utilizar a matriz jacobiana da f bidimensional. Dessa forma, se:

$$f = \begin{bmatrix} f_x(x_k, y_k) \\ f_y(x_k, y_k) \end{bmatrix} \tag{8}$$

então a Jacobiana de f é:

$$J = \begin{bmatrix} \frac{df_x}{dx} & \frac{df_x}{dy} \\ \frac{df_y}{dx} & \frac{df_y}{dy} \end{bmatrix} \tag{9}$$

e assim o Método de Newton pode ser reescrito substituindo:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

por:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - J^{-1} \begin{bmatrix} f_x(x_k, y_k) \\ f_y(x_k, y_k) \end{bmatrix}$$

Alternativamente, é possível executar o método da seguinte forma, que pode impedir erros de inversão de matriz (nem todas as matrizes são inversíveis, por exemplo, matrizes singulares):

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \end{bmatrix} - SOLUCAO \left(J, \begin{bmatrix} f_x(x_k, y_k) \\ f_y(x_k, y_k) \end{bmatrix} \right)$$

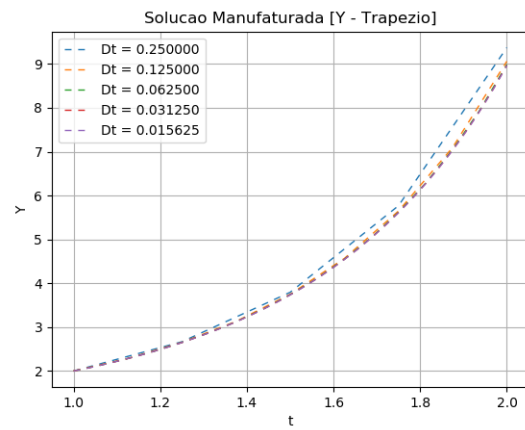
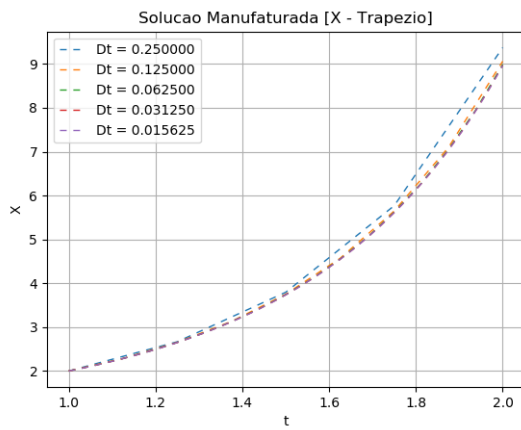
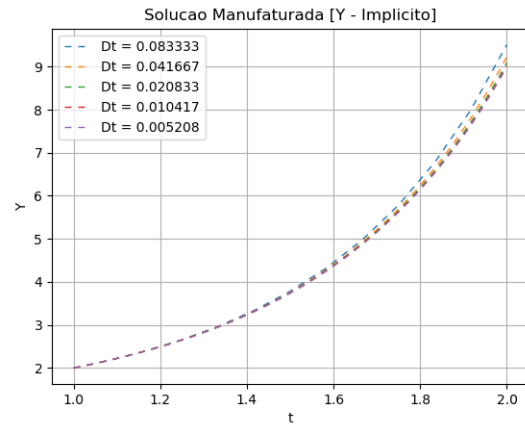
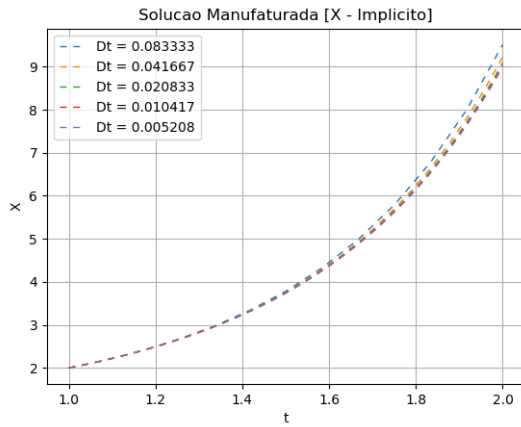
onde a função SOLUÇÃO resolve o sistema linear formado pela matriz J e o vetor $\begin{bmatrix} f_x(x_k, y_k) \\ f_y(x_k, y_k) \end{bmatrix}$

No caso específico, utilizamos a função da biblioteca numpy: `numpy.linalg.solve(A,b)`.

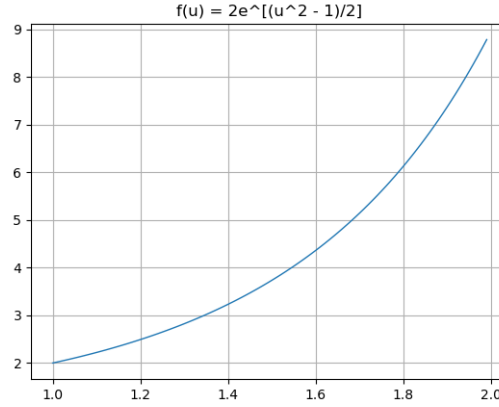
Para melhor visualização da implementação do três métodos mencionados acima em Python, ver a seção "Algoritmos" ao final do relatório ou os arquivos anexados Tarefa3.py e Testes.py. Nota-se que os parâmetros dos métodos devem ser alterados no próprio arquivo do programa, i.e. não recebe inputs de usuário. As mudanças devem ser "hardcoded" no código.

2 Verificação dos métodos

Vejamos abaixo os resultados do sistema de equações com solução manufaturada no intervalo $[1, 2]$. Note que $Dt = \Delta t$.



Veja também o gráfico para solução exata:



Pelos gráficos podemos notar que de fato as equações estão convergindo para a solução exata (é fácil perceber que os gráficos para x e para y são os mesmos. Isso ocorre pois x e y são independentes e suas EDOs são equivalentes). Além disso, pelas tabelas abaixo vemos que a estimativa ordem dos métodos está convergindo para a verdadeira ordem:

Δt	$\ \vec{X}_{k+1}\ _2$	Erro $^{k+1}$	$\frac{\text{Erro}^k}{\text{Erro}^{k+1}}$	$p \approx \log_{\frac{\Delta t^k}{\Delta t^{k+1}}} \left(\frac{\text{Erro}^k}{\text{Erro}^{k+1}} \right)$
0.0833333333	13.4508696131			
0.0416666667	13.0449601047	0.3688291735	2.1005352547	1.0707569998
0.0208333333	12.8562617047	0.1801307735	2.0475633692	1.033908102
0.0104166667	12.7651653229	0.0890343917	2.0231594788	1.0166100471
0.0052083333	12.7203951504	0.0442642192	2.0114302996	1.0082217463

Table 1: Tabela de convergência do erro e de p para Euler Implícito.

Δt	$\ \vec{X}_{k+1}\ _2$	Erro $^{k+1}$	$\frac{\text{Erro}^k}{\text{Erro}^{k+1}}$	$p \approx \log_{\frac{\Delta t^k}{\Delta t^{k+1}}} \left(\frac{\text{Erro}^k}{\text{Erro}^{k+1}} \right)$
0.25	13.2538989368			
0.125	12.8143889764	0.1382580452	4.1789105631	2.0631268825
0.0625	12.7103022971	0.0341713659	4.0460204541	2.0165036133
0.03125	12.6846495575	0.0085186263	4.0113704643	2.0040952103
0.015625	12.6782590799	0.0021281486	4.0028342537	2.001021879

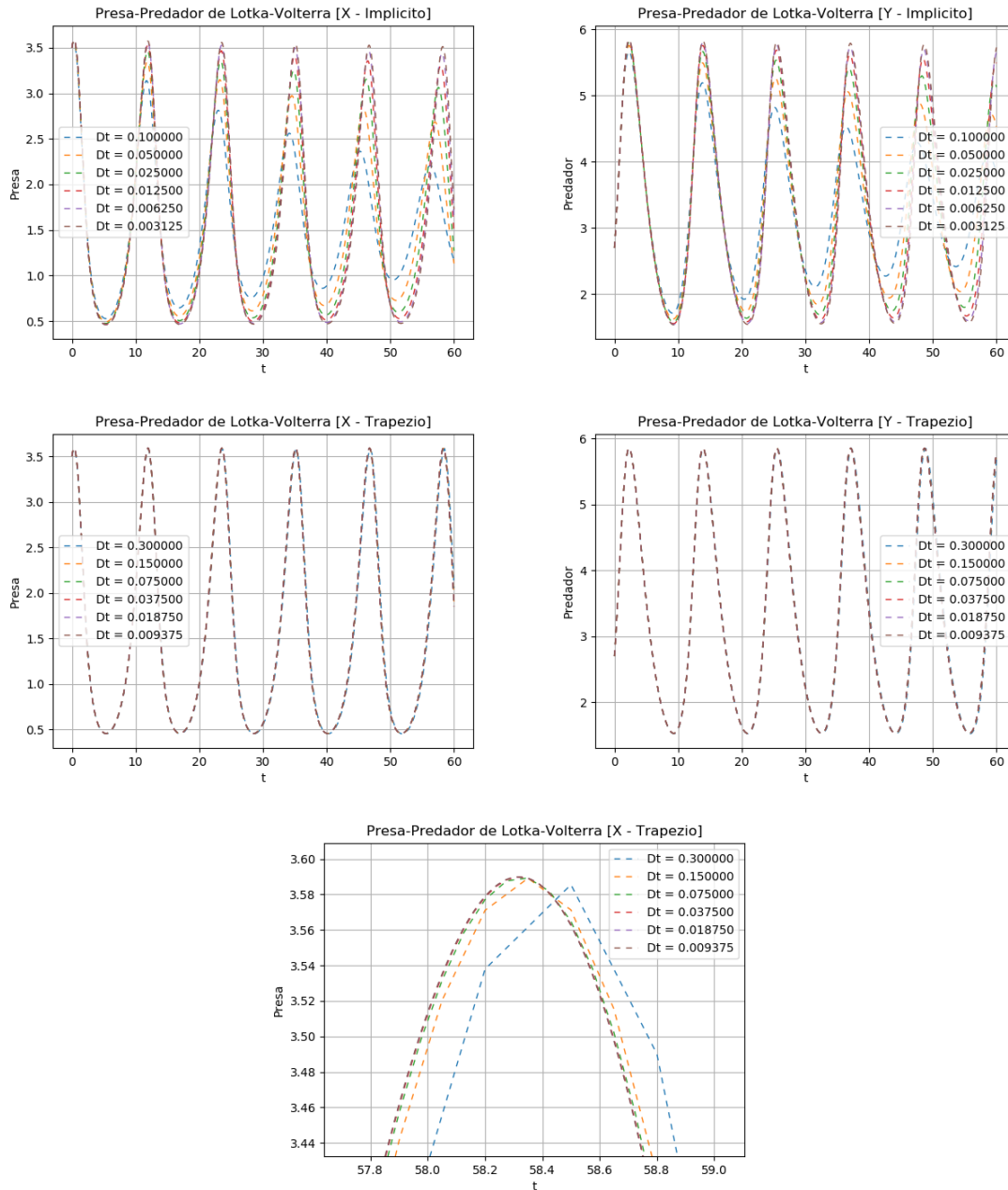
Table 2: Tabela de convergência do erro e de p para Método do Trapézio.

Obs: O erro foi calculado utilizando a diferença absoluta da Norma₂ da solução numérica com a Norma₂ da solução exata ($\sqrt{f_x^2(2) + f_y^2(2)} = \sqrt{9^2 + 9^2}$).

Como pode-se perceber, as estimativas de p estão convergindo de forma correta para cada método; Euler Implícito possui ordem $p = 1$ e Trapézio possui ordem $p = 2$. O erro também converge para 0, em ambos os métodos.

3 Resultados obtidos (Presa-Predador)

Vejamos abaixo os resultados obtidos para o modelo Presa-Predador, com parâmetro $t \in [0, 60]$:



Podemos ver a convergência de forma gráfica ao observar as imagens acima (a ultima imagem é um zoom do método do trapézio, o qual converge tao rapidamente que é difícil verificar as aproximações sucessivas). Podemos ver também que apresentam o resultado esperado do modelo, i.e. a oscilação da população de predadores, descompassada com a oscilação da população de presas.

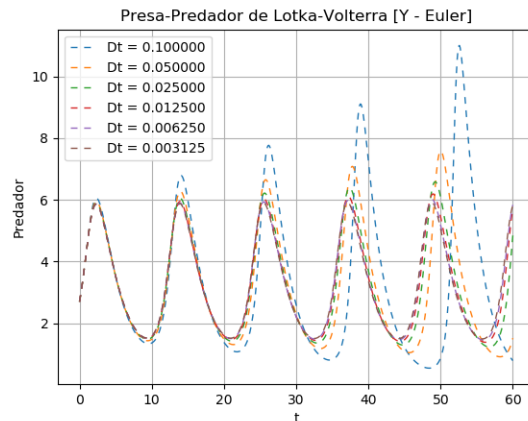
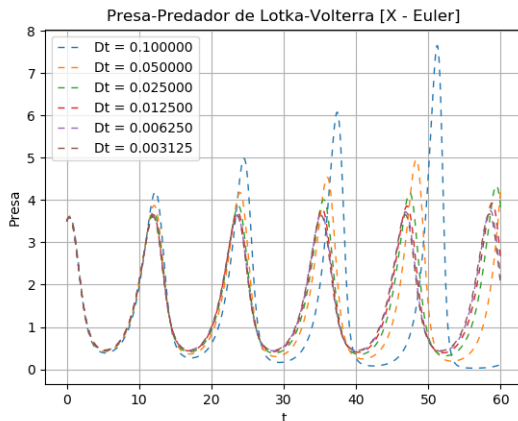
Os resultados obtidos para o instante $t = 60$ para Euler Implícito e Método do Trapézio foram respectivamente:

$$x(60) = 1.7262920385868656, \quad y(60) = 5.729842119285323$$

$$x(60) = 1.8482821968829244, \quad y(60) = 5.787519361222033$$

Euler Explícito e Tempo computacional

Abaixo, os resultados obtidos com o método de Euler Explícito:



$$x(60) = 1.9923000368045625, \quad y(60) = 5.825862292917863$$

Pode ser interessante comparar o comportamento de convergência do método explícito com o método implícito. Graficamente podemos notar que o método de Euler Implícito converge "de baixo para cima", ou, como menciona o nome do método em inglês, "Backward Euler", "de frente para trás". O método de Euler Explícito converge "de cima para baixo", ou, como menciona o nome em inglês "Forward Euler", "de trás para frente". Isso explica por que para valores grandes de Δt o método Implícito aparenta convergir e o método Explícito aparenta divergir.

Seguindo das analogias feitas acima, podemos entender o Método do Trapézio como um método que converge por cima e por baixo ao mesmo tempo, ou uma "média" entre os resultados dos outros dois métodos.

Vejamos a seguir o tempo computacional em segundos que cada método necessitou para Δt s cada vez menores:

Método Implícito:

0.1 - Tempo: 0.3281223773956299
 0.05 - Tempo: 0.51163649559021
 0.025 - Tempo: 0.9484648704528809
 0.0125 - Tempo: 1.478050708770752
 0.00625 - Tempo: 2.785555839538574

Método do Trapézio:

0.3 - Tempo: 0.07579731941223145
 0.15 - Tempo: 0.13264155387878418
 0.075 - Tempo: 0.26030802726745605
 0.0375 - Tempo: 0.5574913024902344
 0.01875 - Tempo: 1.0332555770874023

Método Explícito:

0.1 - Tempo: 0.013962030410766602
 0.05 - Tempo: 0.008980512619018555
 0.025 - Tempo: 0.01399087905883789
 0.0125 - Tempo: 0.024933338165283203
 0.00625 - Tempo: 0.047899723052978516

Apesar da vantagem dos métodos implícitos de funcionarem para Δt s grandes, o método explícito acaba sendo muito mais rápido de executar.

Algoritmo

Abaixo o código utilizado para obter os resultados acima;

Tarefa3.py:

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3  import time
4
5  # Parametros da phi(X)
6  al = 0.87
7  be = 0.27
8  ga = 0.38
9  de = 0.25
10
11 # Phi(X) leva somente o parametro x, nesse caso. O t esta implicito. Alem disso, X eh o vetor (x,y)
12 def phi(X):
13     x = X[0]
14     y = X[1]
15     return [al*x - be*x*y, -ga*y + de*y*x]
16
17 def Hphi(X, H):
18     x = X[0]
19     y = X[1]
20     return [H*(al*x - be*x*y), H*(-ga*y + de*y*x)]
21
22 #####
23
24 # funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo Implicito
25 def X_prox(X, Xk, H):
26     v = X[0] - Xk[0] - H*phi(X)[0]
27     return [X[0] - Xk[0] - H*phi(X)[0], X[1] - Xk[1] - H*phi(X)[1]]
28
29 # Jacobiana da funcao acima
30 def JX_prox(X, Xk, H):
31     x = X[0]
32     y = X[1]
33     return [[1 - H*(al-be*y), -H*(-be*x)], [-H*(de*y), 1 - H*(-ga+de*x)]]
34
35 #####
36
37 # funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo do Trapezio
38 def Y_prox(X, Xk, H):
39     return [X[0] - Xk[0] - (H/2)*(phi(X)[0]+phi(Xk)[0]), X[1] - Xk[1] - (H/2)*(phi(X)[1]+phi(Xk)[1])]
40
41 # Jacobiana da funcao acima
42 def JY_prox(X, Xk, H):

```

```

43     x = X[0]
44     y = X[1]
45     xk = Xk[0]
46     yk = Xk[1]
47     return [[1 - (H/2)*(al-be*y + al-be*yk), -(H/2)*(-be*x - be*xk)], [-(H/2)*(de*y + de*yk), 1 - (H/2)*(-ga + de*x -ga + de*xk)]]
48
49 #####
50
51 # Implementacao para o metodo de newton em 2D. Recebe os intervalos para encontrar a raiz para x, Ix e para y, Iy.
52 # Recebe o X anterior Xk. A funcao e encerrada apos it iteracoes.
53 def newton_2D(Ix, Iy, Xk, H, it, JX, X_prox):
54
55     xo = Ix[0]
56     xf = Ix[1]
57     yo = Iy[0]
58     yf = Iy[1]
59
60     if xf <= xo or yf <= yo:
61         return False
62
63     # verifica qual dos extremos dos intervalos deve estar mais proximo da raiz
64     raiz = [(xf-xo)/2, (yf-yo)/2]
65     raiz_prox = []
66     raiz_prox = np.array(raiz) - np.linalg.solve(np.array(JX(raiz, Xk, H)), np.array(X_prox(raiz, Xk, H)))
67
68     if raiz_prox[0] < raiz[0]:
69         raiz[0] = xo
70     else:
71         raiz[0] = xf
72     if raiz_prox[1] < raiz[1]:
73         raiz[1] = yo
74     else:
75         raiz[1] = yf
76
77     # loop do metodo. Utilizando o np.linalg.solve aceleramos o programa e evitamos problemas no momento de inverter a jacobiana (erro de matriz singular)
78     for _ in range(it):
79         raiz_prox = np.array(raiz) - np.linalg.solve(np.array(JX(raiz, Xk, H)), np.array(X_prox(raiz, Xk, H)))
80         raiz[:] = raiz_prox
81
82     return raiz
83
84 # metodo implicito
85 def euler_imp(Xo, to, tf, n):
86     T = [to]
87     res = [Xo]
88     H = (tf-to)/n
89
90     for _ in range(n):
91         X_pro = []
92         X_pro = newton_2D([0,200], [0,200], res[-1], H, 10, JX_prox, X_prox)
93         res.append(X_pro)
94         T.append(T[-1]+H)
95     return [T, res]
96
97 # metodo do trapezio
98 def trap(Xo, to, tf, n):
99     T = [to]
100     res = [Xo]
101     H = (tf-to)/n
102
103     for _ in range(n):
104         X_prox = []
105         X_prox = newton_2D([0,50], [0,50], res[-1], H, 10, JY_prox, Y_prox)
106         res.append(X_prox)
107         T.append(T[-1]+H)
108     return [T, res]
109
110 # metodo de euler, para comparacao
111 def euler(Xo, to, tf, n):
112     T = [to]
113     res = [Xo]
114     H = (tf-to)/n
115
116     for _ in range(n):
117         X_prox = []
118         X_prox = np.array(res[-1]) + np.array(Hphi(res[-1], H))
119         res.append(X_prox)
120         T.append(T[-1]+H)
121
122     return [T, res]
123

```

```

124 #####
125
126 # Input dos parametros iniciais
127
128 Xo = [3.5, 2.7] # xo e yo
129 to = 0 # tempo inicial
130 tf = 60 # instante final
131 n = 300 # numero de iteracoes inicial dos metodos
132 it = 6 # numero de vezes em que o n sera multiplicado
133
134 m = n
135
136
137 # Abaixo estao apenas loops para plottar os graficos
138
139 print("Metodo Implicito")
140 print()
141 plt.figure()
142 for i in range(it):
143     t = time.time()
144     print(i+1, end = " - Tempo: ")
145     n *= 2
146     eu = euler_imp(Xo, to, tf, n)
147     T = eu[0]
148     eu = eu[1]
149     plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
150     print(time.time()-t)
151 plt.xlabel("t")
152 plt.ylabel("Presa")
153 plt.title("Presa-Predador de Lotka-Volterra [X - Implicito]")
154 plt.legend()
155 plt.grid(True)
156
157 print()
158
159 n = m
160
161 plt.figure()
162 for i in range(it):
163     t = time.time()
164     print(i+1, end = " - Tempo: ")
165     n *= 2
166     eu = euler_imp(Xo, to, tf, n)
167     T = eu[0]
168     eu = eu[1]
169     plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
170     print(time.time()-t)
171 plt.xlabel("t")
172 plt.ylabel("Predador")
173 plt.title("Presa-Predador de Lotka-Volterra [Y - Implicito]")
174 plt.legend()
175 plt.grid(True)
176
177 print()
178
179 x,y = eu[-1][0],eu[-1][1]
180 print(x,y)
181
182 print()
183
184 n = int(m/3)
185
186 print("Metodo do Trapezio")
187 print()
188 plt.figure()
189 for i in range(it):
190     t = time.time()
191     print(i+1, end = " - Tempo: ")
192     n *= 2
193     eu = trap(Xo, to, tf, n)
194     T = eu[0]
195     eu = eu[1]
196     plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
197     print(time.time()-t)
198 plt.xlabel("t")
199 plt.ylabel("Presa")
200 plt.title("Presa-Predador de Lotka-Volterra [X - Trapezio]")
201 plt.legend()
202 plt.grid(True)
203
204 print()

```

```

205
206 n = int(m/3)
207
208 plt.figure()
209 for i in range(it):
210     t = time.time()
211     print(i+1, end = " - Tempo: ")
212     n *= 2
213     eu = trap(Xo, to, tf, n)
214     T = eu[0]
215     eu = eu[1]
216     plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
217     print(time.time()-t)
218 plt.xlabel("t")
219 plt.ylabel("Predador")
220 plt.title("Presa-Predador de Lotka-Volterra [Y - Trapezio]")
221 plt.legend()
222 plt.grid(True)
223
224 print()
225
226 print(eu[-1][0],eu[-1][1])
227
228 print()
229
230 n = m
231
232 print("Metodo Explicito")
233 print()
234 plt.figure()
235 for i in range(it):
236     t = time.time()
237     print(i+1, end = " - Tempo: ")
238     n *= 2
239     eu = euler(Xo, to, tf, n)
240     T = eu[0]
241     eu = eu[1]
242     plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
243     print(time.time()-t)
244 plt.xlabel("t")
245 plt.ylabel("Presa")
246 plt.title("Presa-Predador de Lotka-Volterra [X - Euler]")
247 plt.legend()
248 plt.grid(True)
249
250 print()
251
252 n = m
253
254 plt.figure()
255 for i in range(it):
256     t = time.time()
257     print(i+1, end = " - Tempo: ")
258     n *= 2
259     eu = euler(Xo, to, tf, n)
260     T = eu[0]
261     eu = eu[1]
262     plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
263     print(time.time()-t)
264 plt.xlabel("t")
265 plt.ylabel("Predador")
266 plt.title("Presa-Predador de Lotka-Volterra [Y - Euler]")
267 plt.legend()
268 plt.grid(True)
269
270 print()
271
272 print(eu[-1][0],eu[-1][1])
273
274 print()
275
276 print("Media explicito com implicito = ", (eu[-1][0] + x)/2,(eu[-1][1] + y)/2)
277
278
279 plt.show()

```

Testes.py:

```

1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def fun(x):
5      return 2*np.exp((x**2 -1)/2)
6
7  # Phi(X) leva somente o parametro x, nesse caso. O t esta implicito. Alem disso, X eh o vetor (x,y)
8  def phi(X, t):
9      x = X[0]
10     y = X[1]
11     return [x*t, y*t]
12
13     #####
14
15     # funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo Implicito
16     def X_prox(X, Xk, H, t, t_prox):
17         return [X[0] - Xk[0] - H*phi(X, t)[0], X[1] - Xk[1] - H*phi(X, t)[1]]
18
19     # Jacobiana da funcao acima
20     def JX_prox(X, Xk, H, t, t_prox):
21         x = X[0]
22         y = X[1]
23         return [[1 - H*t, 0], [0, 1 - H*t]]
24
25     #####
26
27     # funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo do Trapezio
28     def Y_prox(X, Xk, H, t, t_prox):
29         return [X[0] - Xk[0] - (H/2)*(phi(X, t_prox)[0]+phi(Xk, t)[0]), X[1] - Xk[1] - (H/2)*(phi(X, t_prox)[1]+phi(Xk, t)[1])]
30
31     # Jacobiana da funcao acima
32     def JY_prox(X, Xk, H, t, t_prox):
33         x = X[0]
34         y = X[1]
35         xk = Xk[0]
36         yk = Xk[1]
37         return [[1 - (H/2)*(t_prox + t), 0], [0, 1 - (H/2)*(t_prox + t)]]
38
39     #####
40
41     # Implementacao para o metodo de newton em 2D. Recebe os intervalos para encontrar a raiz para x, Ix e para y, Iy.
42     # Recebe o X anterior Xk. A funcao e encerrada apos it iteracoes.
43     def newton_2D(Ix, Iy, Xk, H, it, JX, X_prox, t, t_prox):
44
45         xo = Ix[0]
46         xf = Ix[1]
47         yo = Iy[0]
48         yf = Iy[1]
49
50         if xf <= xo or yf <= yo:
51             return False
52
53         # verifica qual dos extremos dos intervalos deve estar mais proximo da raiz
54         raiz = [(xf-xo)/2, (yf-yo)/2]
55         raiz_prox = []
56         raiz_prox = np.array(raiz) - np.linalg.solve(np.array(JX(raiz, Xk, H, t, t_prox)), np.array(X_prox(raiz, Xk, H, t, t_prox)))
57
58         if raiz_prox[0] < raiz[0]:
59             raiz[0] = xo
60         else:
61             raiz[0] = xf
62         if raiz_prox[1] < raiz[1]:
63             raiz[1] = yo
64         else:
65             raiz[1] = yf
66
67         # loop do metodo. Utilizando o np.linalg.solve aceleramos o programa e evitamos problemas no momento de inverter a jacobiana (erro de matriz singular)
68         for _ in range(it):
69             raiz_prox = np.array(raiz) - np.linalg.solve(np.array(JX(raiz, Xk, H, t, t_prox)), np.array(X_prox(raiz, Xk, H, t, t_prox)))
70             raiz[:] = raiz_prox
71
72         return raiz
73
74     # metodo implicito
75     def euler_imp(Xo, to, tf, n):
76         T = [to]
77         res = [Xo]
78         H = (tf-to)/n
79

```

```

80     for _ in range(n):
81         X_pro = []
82         X_pro = newton_2D([0,200], [0,200], res[-1], H, 10, JX_prox, X_prox, T[-1], T[-1]+H)
83         res.append(X_pro)
84         T.append(T[-1]+H)
85     return [T, res]
86
87 # metodo do trapezio
88 def trap(Xo, to, tf, n):
89     T = [to]
90     res = [Xo]
91     H = (tf-to)/n
92
93     for _ in range(n):
94         X_prox = []
95         X_prox = newton_2D([0,50], [0,50], res[-1], H, 10, JY_prox, Y_prox, T[-1], T[-1]+H)
96         res.append(X_prox)
97         T.append(T[-1]+H)
98     return [T, res]
99
100 #####
101
102 # Input dos parametros iniciais
103
104 Xo = [2, 2] # xo e yo
105 to = 1 # tempo inicial
106 tf = 2 # instante final
107 n = 6 # numero de iteracoes inicial dos metodos
108 it = 5 # numero de vezes em que o n sera multiplicado
109
110 m = n
111
112 # Abaixo estao apenas loops para plottar os graficos
113
114 erro_ant = 1
115
116 print("Metodo Implicito")
117 print()
118 plt.figure()
119 for i in range(it):
120     n *= 2
121     eu = euler_imp(Xo, to, tf, n)
122     T = eu[0]
123     eu = eu[1]
124     plt.plot(T, [i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
125
126     erro = abs(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2) - np.sqrt(2*fun(2)**2))
127     div_err = erro_ant/erro
128     print(round((tf - to)/n,10), " & " ,round(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2),10), " & " ,\
129     round(erro, 10), " & " , round(div_err,10), " & " ,round(np.log(div_err)/np.log(2),10), "\\")
130     erro_ant = erro
131 plt.xlabel("t")
132 plt.ylabel("X")
133 plt.title("Solucao Manufaturada [X - Implicito]")
134 plt.legend()
135 plt.grid(True)
136
137 print()
138
139 n = m
140
141 plt.figure()
142 for i in range(it):
143     n *= 2
144     eu = euler_imp(Xo, to, tf, n)
145     T = eu[0]
146     eu = eu[1]
147     plt.plot(T, [i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
148 plt.xlabel("t")
149 plt.ylabel("Y")
150 plt.title("Solucao Manufaturada [Y - Implicito]")
151 plt.legend()
152 plt.grid(True)
153
154 print()
155
156 x,y = eu[-1][0],eu[-1][1]
157 print(x,y)
158
159 print()
160

```

```

161 n = int(m/3)
162
163 erro_ant = 1
164
165 print("Metodo do Trapezio")
166 print()
167 plt.figure()
168 for i in range(it):
169     n *= 2
170     eu = trap(Xo, to, tf, n)
171     T = eu[0]
172     eu = eu[1]
173     plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
174
175     erro = abs(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2) - np.sqrt(2*fun(2)**2))
176     div_err = erro_ant/erro
177     print(round((tf - to)/n,10), " & " ,round(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2),10), " & " ,\
178           round(erro, 10), " & " , round(div_err,10), " & " ,round(np.log(div_err)/np.log(2),10), "\\")
179     erro_ant = erro
180 plt.xlabel("t")
181 plt.ylabel("X")
182 plt.title("Solucao Manufaturada [X - Trapezio]")
183 plt.legend()
184 plt.grid(True)
185
186 print()
187
188 n = int(m/3)
189
190 plt.figure()
191 for i in range(it):
192     n *= 2
193     eu = trap(Xo, to, tf, n)
194     T = eu[0]
195     eu = eu[1]
196     plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
197 plt.xlabel("t")
198 plt.ylabel("Y")
199 plt.title("Solucao Manufaturada [Y - Trapezio]")
200 plt.legend()
201 plt.grid(True)
202
203 print()
204
205 print(eu[-1][0],eu[-1][1])
206
207 print()
208
209 plt.figure()
210 u = np.arange(1.,2., 0.01)
211 plt.plot(u, fun(u), lw = 1)
212 plt.title("f(u) = 2e^[(u^2 - 1)/2]")
213 plt.grid(True)
214
215 plt.show()

```