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:

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

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

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

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:

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

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

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

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):

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

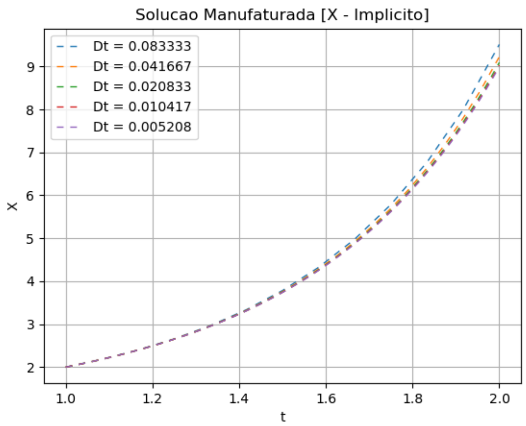
onde a função SOLUÇÃO resolve o sistema linear formado pela matriz J e o vetor $\begin{bmatrix} f_x(x_k, y_k) \\ f_y(x_k, y_k) \end{bmatrix}$

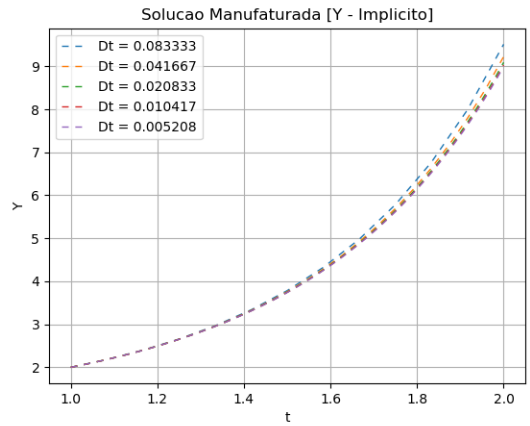
No caso especifico, utilizamos a função da biblioteca numpy: numpy.linalg.solve(A,b).

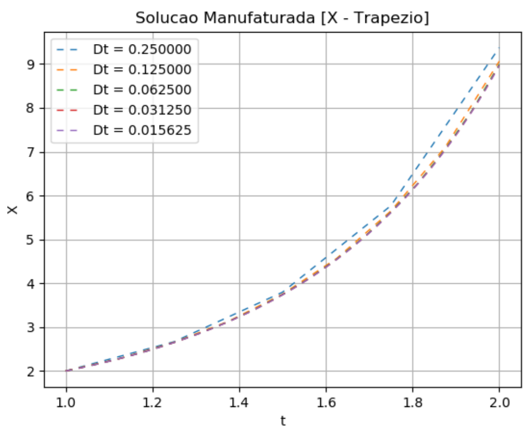
Para melhor visualização da implementação do três métodos mencionados acima em Python, ver a seção "Algorítimos" ao final do relatório ou os arquivos anexados Tarefa3.py e Testes.py. Nota-se que os parâmetros dos métodos devem ser alterados no próprio arquivo do programa, i.e. não recebe inputs de usuário.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

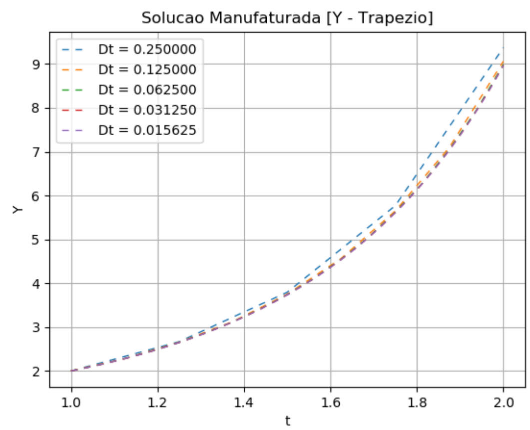
2 Verificação dos métodos

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

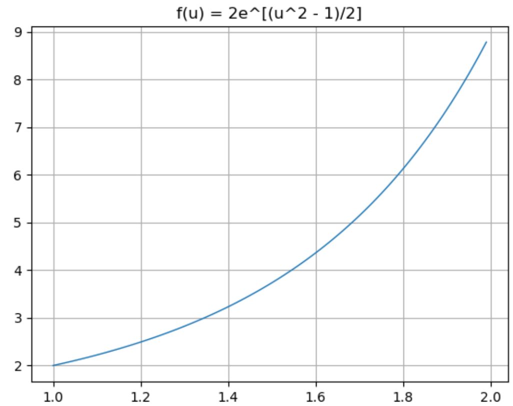








Veja também o gráfico para solução exata:



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 $^{k+1}$	$\frac{\text{Erro}^k}{\text{Erro}^{k+1}}$	$p \approx \log_{\frac{\Delta t^k}{\Delta t^{k+1}}}\left(\left\|\frac{\text{Erro}^k}{\text{Erro}^{k+1}}\right\|\right)$
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 $^{k+1}$	$\frac{\text{Erro}^k}{\text{Erro}^{k+1}}$	$p \approx \log_{\frac{\Delta t^k}{\Delta t^{k+1}}}\left(\left\|\frac{\text{Erro}^k}{\text{Erro}^{k+1}}\right\|\right)$
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

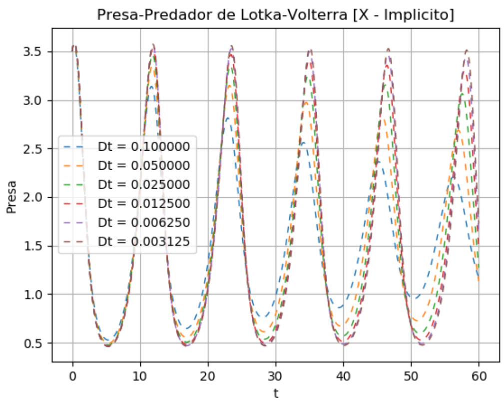
Table 2: Tabela de convergência do erro e de p para Método do Trapézio.

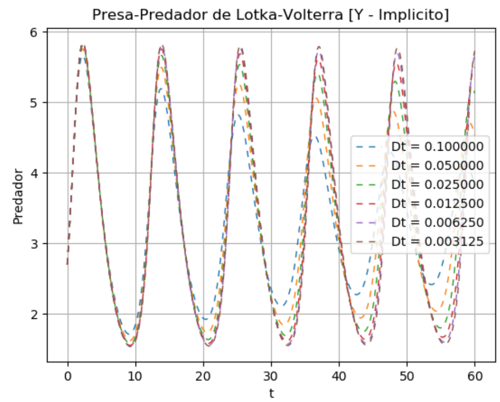
Obs: O erro foi calculado utilizando a diferença absoluta da Norma$_2$ da solução numérica com a Norma$_2$ da solução exata ($\sqrt{f_x^2(2) + f_y^2(2)} = \sqrt{9^2 + 9^2}$).

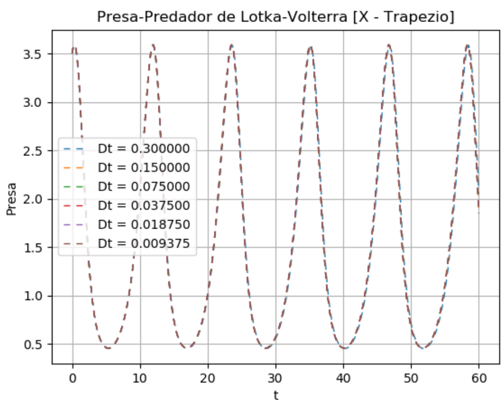
Como pode-se perceber, as estimativas de p estão convergindo de forma correta para cada método; Euler Implícito possui ordem $p = 1$ e Trapézio possui ordem $p = 2$. O erro também converge para 0, em ambos os métodos.

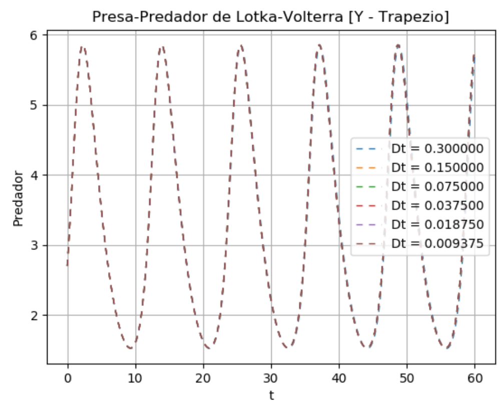
3 Resultados obtidos (Presa-Predador)

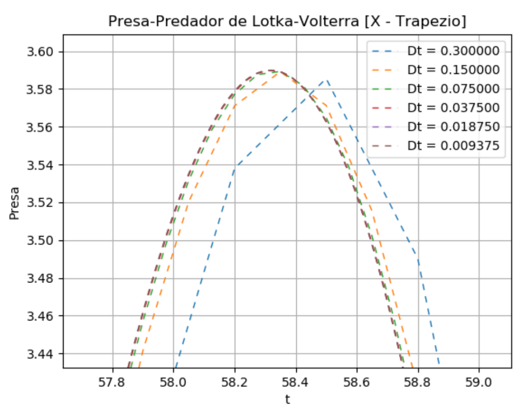
Vejamos abaixo os resultados obtidos para o modelo Presa-Predador, com parâmetro $t \in [0, 60]$:











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.

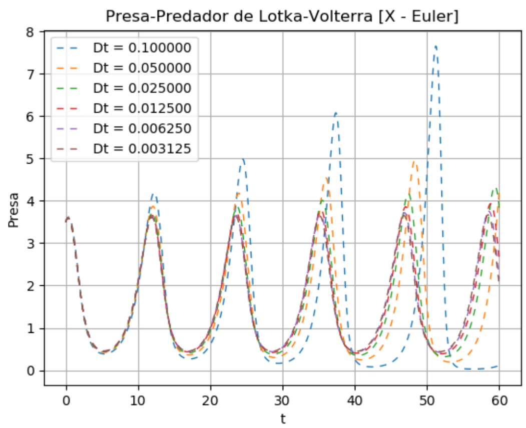
Os resultados obtidos para o instante $t = 60$ para Euler Implícito e Método do Trapézio foram respectivamente:

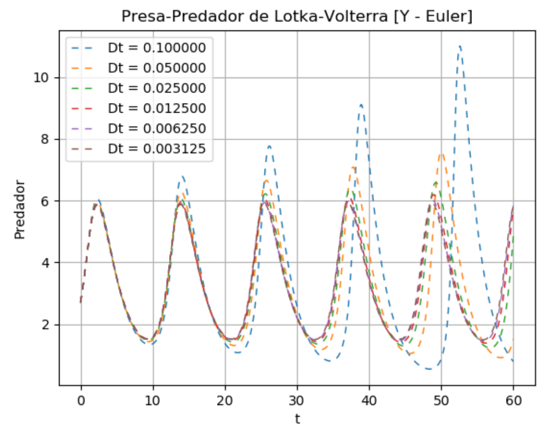
$$x(60) = 1.7262920385868656, \; y(60) = 5.729842119285323$$

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

Euler Explicito e Tempo computacional

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





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

Pode ser interessante comparar o comportamento de convergência do método explicito 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 traz". O método de Euler Explicito converge "de cima para baixo", ou, como menciona o nome em inglês "Forward Euler", "de traz para frente". Isso explica por que para valores grandes de Δt o método Implícito aparenta convergir e o método Explicito 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 Δts 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 Δts grandes, o método explicito acaba sendo muito mais rápido de executar.

Algorítimo

Abaixo o código utilizado para obter os resultados acima;

Tarefa3.py:

```
import numpy as np
import matplotlib.pyplot as plt
import time

# Parametros da phi(X)
al = 0.87
be = 0.27
ga = 0.38
de = 0.25

# Phi(X) leva somente o parametro x, nesse caso. O t esta implicito. Alem disso, X eh o vetor (x,y)
def phi(X):
    x = X[0]
    y = X[1]
    return [al*x - be*x*y, -ga*y + de*y*x]

def Hphi(X, H):
    x = X[0]
    y = X[1]
    return [H*(al*x - be*x*y), H*(-ga*y + de*y*x)]

####################################################################################################

# funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo Implicito
def X_prox(X, Xk, H):
    v = X[0] - Xk[0] - H*phi(X)[0]
    return [X[0] - Xk[0] - H*phi(X)[0], X[1] - Xk[1] - H*phi(X)[1]]

# Jacobiana da funcao acima
def JX_prox(X, Xk, H):
    x = X[0]
    y = X[1]
    return [[1 - H*(al-be*y),-H*(-be*x)], [-H*(de*y),1 - H*(-ga+de*x)]]

####################################################################################################

# funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo do Trapezio
def Y_prox(X, Xk, H):
    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])]

# Jacobiana da funcao acima
def JY_prox(X, Xk, H):
```

```python
    x = X[0]
    y = X[1]
    xk = Xk[0]
    yk = Xk[1]
    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)]]

#####################################################################################################

# Implementacao para o metodo de newton em 2D. Recebe os intervalos para encontrar a raiz para x, Ix e para y, Iy.
# Recebe o X anterior Xk. A funcao e encerrada apos it iteracoes.
def newton_2D(Ix, Iy, Xk, H, it, JX, X_prox):

    xo = Ix[0]
    xf = Ix[1]
    yo = Iy[0]
    yf = Iy[1]

    if xf <= xo or yf <= yo:
        return False

    # verifica qual dos extremos dos intervalos deve estar mais proximo da raiz
    raiz = [(xf-xo)/2, (yf-yo)/2]
    raiz_prox = []
    raiz_prox = np.array(raiz) - np.linalg.solve(np.array(JX(raiz, Xk, H)),np.array(X_prox(raiz, Xk, H)))

    if raiz_prox[0] < raiz[0]:
        raiz[0] = xo
    else:
        raiz[0] = xf
    if raiz_prox[1] < raiz[1]:
        raiz[1] = yo
    else:
        raiz[1] = yf

    # loop do metodo. Utilizando o np.linalg.solve aceleramos o programa e evitamos problemas no momento de inverter a jacobiana (erro de matriz si
    for _ in range(it):
        raiz_prox = np.array(raiz) - np.linalg.solve(np.array(JX(raiz, Xk, H)),np.array(X_prox(raiz, Xk, H)))
        raiz[:] = raiz_prox

    return raiz

# metodo implicito
def euler_imp(Xo, to, tf, n):
    T = [to]
    res = [Xo]
    H = (tf-to)/n

    for _ in range(n):
        X_pro = []
        X_pro = newton_2D([0,200], [0,200], res[-1], H, 10, JX_prox, X_prox)
        res.append(X_pro)
        T.append(T[-1]+H)
    return [T, res]

# metodo do trapezio
def trap(Xo, to, tf, n):
    T = [to]
    res = [Xo]
    H = (tf-to)/n

    for _ in range(n):
        X_prox = []
        X_prox = newton_2D([0,50], [0,50], res[-1], H, 10, JY_prox, Y_prox)
        res.append(X_prox)
        T.append(T[-1]+H)
    return [T, res]

# metodo de euler, para comparacao
def euler(Xo, to, tf, n):
    T = [to]
    res = [Xo]
    H = (tf-to)/n

    for _ in range(n):
        X_prox = []
        X_prox = np.array(res[-1]) + np.array(Hphi(res[-1], H))
        res.append(X_prox)
        T.append(T[-1]+H)

    return [T, res]

```

```
##############################

# Input dos parametros iniciais

Xo = [3.5, 2.7] # xo e yo
to = 0 # tempo inicial
tf = 60 # instante final
n = 300 # numero de iteracoes inicial dos metodos
it = 6 # numero de vezes em que o n sera multiplicado

m = n


# Abaixo estao apenas loops para plottar os graficos

print("Metodo Implicito")
print()
plt.figure()
for i in range(it):
    t = time.time()
    print(i+1, end = " - Tempo: ")
    n *= 2
    eu = euler_imp(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
    print(time.time()-t)
plt.xlabel("t")
plt.ylabel("Presa")
plt.title("Presa-Predador de Lotka-Volterra [X - Implicito]")
plt.legend()
plt.grid(True)

print()

n = m

plt.figure()
for i in range(it):
    t = time.time()
    print(i+1, end = " - Tempo: ")
    n *= 2
    eu = euler_imp(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
    print(time.time()-t)
plt.xlabel("t")
plt.ylabel("Predador")
plt.title("Presa-Predador de Lotka-Volterra [Y - Implicito]")
plt.legend()
plt.grid(True)

print()

x,y = eu[-1][0],eu[-1][1]
print(x,y)

print()

n = int(m/3)

print("Metodo do Trapezio")
print()
plt.figure()
for i in range(it):
    t = time.time()
    print(i+1, end = " - Tempo: ")
    n *= 2
    eu = trap(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
    print(time.time()-t)
plt.xlabel("t")
plt.ylabel("Presa")
plt.title("Presa-Predador de Lotka-Volterra [X - Trapezio]")
plt.legend()
plt.grid(True)

print()
```

```

n = int(m/3)

plt.figure()
for i in range(it):
    t = time.time()
    print(i+1, end = " - Tempo: ")
    n *= 2
    eu = trap(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
    print(time.time()-t)
plt.xlabel("t")
plt.ylabel("Predador")
plt.title("Presa-Predador de Lotka-Volterra [Y - Trapezio]")
plt.legend()
plt.grid(True)

print()

print(eu[-1][0],eu[-1][1])

print()

n = m

print("Metodo Explicito")
print()
plt.figure()
for i in range(it):
    t = time.time()
    print(i+1, end = " - Tempo: ")
    n *= 2
    eu = euler(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
    print(time.time()-t)
plt.xlabel("t")
plt.ylabel("Presa")
plt.title("Presa-Predador de Lotka-Volterra [X - Euler]")
plt.legend()
plt.grid(True)

print()

n = m

plt.figure()
for i in range(it):
    t = time.time()
    print(i+1, end = " - Tempo: ")
    n *= 2
    eu = euler(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
    print(time.time()-t)
plt.xlabel("t")
plt.ylabel("Predador")
plt.title("Presa-Predador de Lotka-Volterra [Y - Euler]")
plt.legend()
plt.grid(True)

print()

print(eu[-1][0],eu[-1][1])

print()

print("Media explicito com implicito = ", (eu[-1][0] + x)/2,(eu[-1][1] + y)/2)


plt.show()
```

Testes.py:

```python
import numpy as np
import matplotlib.pyplot as plt

def fun(x):
    return 2*np.exp((x**2 -1)/2)

# Phi(X) leva somente o parametro x, nesse caso. O t esta implicito. Alem disso, X eh o vetor (x,y)
def phi(X, t):
    x = X[0]
    y = X[1]
    return [x*t, y*t]

####################################################################################################

# funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo Implicito
def X_prox(X, Xk, H, t, t_prox):
    return [X[0] - Xk[0] - H*phi(X, t)[0], X[1] - Xk[1] - H*phi(X, t)[1]]

# Jacobiana da funcao acima
def JX_prox(X, Xk, H , t, t_prox):
    x = X[0]
    y = X[1]
    return [[1 - H*t, 0], [0, 1 - H*t]]

####################################################################################################

# funcao para qual a raiz precisa ser encontrada no metodo de newton, para metodo do Trapezio
def Y_prox(X, Xk, H, t, t_prox):
    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])]

# Jacobiana da funcao acima
def JY_prox(X, Xk, H, t, t_prox):
    x = X[0]
    y = X[1]
    xk = Xk[0]
    yk = Xk[1]
    return[[1 - (H/2)*(t_prox + t), 0], [0, 1 - (H/2)*(t_prox + t)]]

####################################################################################################

# Implementacao para o metodo de newton em 2D. Recebe os intervalos para encontrar a raiz para x, Ix e para y, Iy.
# Recebe o X anterior Xk. A funcao e encerrada apos it iteracoes.
def newton_2D(Ix, Iy, Xk, H, it, JX, X_prox, t, t_prox):

    xo = Ix[0]
    xf = Ix[1]
    yo = Iy[0]
    yf = Iy[1]

    if xf <= xo or yf <= yo:
        return False

    # verifica qual dos extremos dos intervalos deve estar mais proximo da raiz
    raiz = [(xf-xo)/2, (yf-yo)/2]
    raiz_prox = []
    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)))

    if raiz_prox[0] < raiz[0]:
        raiz[0] = xo
    else:
        raiz[0] = xf
    if raiz_prox[1] < raiz[1]:
        raiz[1] = yo
    else:
        raiz[1] = yf

    # loop do metodo. Utilizando o np.linalg.solve aceleramos o programa e evitamos problemas no momento de inverter a jacobiana (erro de matriz si
    for _ in range(it):
        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)))
        raiz[:] = raiz_prox

    return raiz

# metodo implicito
def euler_imp(Xo, to, tf, n):
    T = [to]
    res = [Xo]
    H = (tf-to)/n

```

```
    for _ in range(n):
        X_pro = []
        X_pro = newton_2D([0,200], [0,200], res[-1], H, 10, JX_prox, X_prox, T[-1], T[-1]+H)
        res.append(X_pro)
        T.append(T[-1]+H)
    return [T, res]

# metodo do trapezio
def trap(Xo, to, tf, n):
    T = [to]
    res = [Xo]
    H = (tf-to)/n

    for _ in range(n):
        X_prox = []
        X_prox = newton_2D([0,50], [0,50], res[-1], H, 10, JY_prox, Y_prox, T[-1], T[-1]+H)
        res.append(X_prox)
        T.append(T[-1]+H)
    return [T, res]

###############################

# Input dos parametros iniciais

Xo = [2, 2] # xo e yo
to = 1 # tempo inicial
tf = 2 # instante final
n = 6 # numero de iteracoes inicial dos metodos
it = 5 # numero de vezes em que o n sera multiplicado

m = n

# Abaixo estao apenas loops para plottar os graficos

erro_ant = 1

print("Metodo Implicito")
print()
plt.figure()
for i in range(it):
    n *= 2
    eu = euler_imp(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)

    erro = abs(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2) - np.sqrt(2*fun(2)**2))
    div_err = erro_ant/erro
    print(round((tf - to)/n,10), " & " ,round(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2),10), " & ",\
    round(erro, 10), " & ", round(div_err,10), " & ",round(np.log(div_err)/np.log(2),10), "\\\\")
    erro_ant = erro
plt.xlabel("t")
plt.ylabel("X")
plt.title("Solucao Manufaturada [X - Implicito]")
plt.legend()
plt.grid(True)

print()

n = m

plt.figure()
for i in range(it):
    n *= 2
    eu = euler_imp(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[1] for i in eu],label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
plt.xlabel("t")
plt.ylabel("Y")
plt.title("Solucao Manufaturada [Y - Implicito]")
plt.legend()
plt.grid(True)

print()

x,y = eu[-1][0],eu[-1][1]
print(x,y)

print()

```

```
n = int(m/3)

erro_ant = 1

print("Metodo do Trapezio")
print()
plt.figure()
for i in range(it):
    n *= 2
    eu = trap(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[0] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)

    erro = abs(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2) - np.sqrt(2*fun(2)**2))
    div_err = erro_ant/erro
    print(round((tf - to)/n,10), " & " ,round(np.sqrt((eu[-1][0])**2 + (eu[-1][1])**2),10), " & ",\
    round(erro, 10), " & ", round(div_err,10), " & ",round(np.log(div_err)/np.log(2),10), "\\\\")
    erro_ant = erro
plt.xlabel("t")
plt.ylabel("X")
plt.title("Solucao Manufaturada [X - Trapezio]")
plt.legend()
plt.grid(True)

print()

n = int(m/3)

plt.figure()
for i in range(it):
    n *= 2
    eu = trap(Xo, to, tf, n)
    T = eu[0]
    eu = eu[1]
    plt.plot(T,[i[1] for i in eu], label = "Dt = %f" %((tf-to)/n), linestyle = (0,(5,5)), lw = 1)
plt.xlabel("t")
plt.ylabel("Y")
plt.title("Solucao Manufaturada [Y - Trapezio]")
plt.legend()
plt.grid(True)

print()

print(eu[-1][0],eu[-1][1])

print()

plt.figure()
u = np.arange(1.,2., 0.01)
plt.plot(u, fun(u), lw = 1)
plt.title("f(u) = 2e^[(u^2 - 1)/2]")
plt.grid(True)

plt.show()
```