UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

GABRIEL LUENEBERG - 14746439
GUILHERME MAGALHÃES VIANA - 14614394
MARIA BEATRIZ DE MENDONCA MAZETTI 14781471
RAFAEL CRISCOULO DE CARVALHO - 12728272

Determinação da Trajetória de Pêndulos Múltiplos

São Carlos
2024

GABRIEL LUENEBERG - 14746439
GUILHERME MAGALHÃES VIANA - 14614394
MARIA BEATRIZ DE MENDONCA MAZETTI 14781471
RAFAEL CRISCOULO DE CARVALHO - 12728272

Determinação da Trajetória de Pêndulos Múltiplos

Relatório apresentado a disciplina de Cálculo Numérico (SME300), da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte da avaliação da mesma.

Orientador(a): Prof(a). Dr(a). Livia Souza Freire Grion

São Carlos
2024

Sumário

1 INTRODUÇÃO

Uma equação diferencial ordinária é uma equação que envolve uma ou mais derivadas de uma função incógnita. Se todas as derivadas estão em relação a uma única variável, essa equação é chamada de equação diferencial ordinária (EDO). Além disso, a ordem de uma EDO é igual à ordem da maior derivada presente na equação (QUARTERONI, 2006).

Por meio das EDOs, uma vasta gama de fenômenos das mais diversas áreas são modelados (QUARTERONI, 2006). Entretanto, muitas vezes as EDOs obtidas para modelar esses fenômenos são muito complexas, de modo que obter uma solução analítica para essas equações não é possível. Nesse sentido, é necessário utilizar métodos numéricos para determinar uma solução numérica para a EDO (CHAPRA, 2011).

Determinar uma solução numérica para uma EDO consiste em encontrar pontos que estão próximos ao gráfico da função solução. Isso implica que as soluções numéricas de uma EDO são aproximações da solução analítica, e a aproximação será mais próxima da solução analítica quanto maior for a precisão e estabilidade do método empregado (QUARTERONI, 2006)(CHAPRA, 2011).

De modo geral, os métodos numérico para resolução de EDOs são aplicados a problemas de valor inicial (PVI), em que são conhecidas as derivadas da função em todos os pontos do domínio e um ponto pertencente a função solução da EDO, chamado de condição inicial (CHAPRA, 2011).Com base nessas informações, os métodos podem estimar uma aproximação para a solução da EDO. No entanto, as diferentes abordagens adotadas por cada método na realização dessa estimativa resultam na diversidade de métodos numéricos existentes para a resolução de EDOs.

Com base nisso, o presente relatório tem como objetivo analisar a eficácia dos métodos de Euler Explícito e Euler Implícito na determinação da trajetória de um pêndulo multíplo, comparando os resultados obtidos com o método de Runge-Kutta de ordem 4.

De forma resulmida, aplicará-se os métodos de Euler implícito e explícito a um pêndulo de múltiplos pontos, sendo que as coordenadas de cada ponto serão determinadas por esses métodos aliados à função da Lagrangiana de maneira a analisá-los, além disso, será adicionada uma parte de código utilizando o método de Runge Kutta com o intuito de comparar um método de alta precisão com os sorteados. Além disso, por meio dessa possibilidade de aplicação, nota-se a importância do pêndulo como parâmetro de análise dos métodos numéricos, sendo possível visualizar suas falhas em termos específicos como localidade, globalidade, tamanho dos passos, estabilidade e custo, o que antecipa a escolha deles em aplicações como a aerodinâmica de um espaço, ou um gráfico caracterizante de um aerofólio ou a determinação de pontos de coordenadas em robô médico com diversos graus de liberdade, o pêndulo também pode ser um pré determinante para o entendimento do conceito de turbulência, como um sistemas mais simples, devido à sua natureza caótica.

2 METODOLOGIA

Nesta seção são apresentadas as definições dos métodos de Euler Explícito, Euler Implícito e Runge-Kutta de ordem 4, bem como a explicação teórica do funcionamento do pêndulo múltiplo e a metodologia adotada para resolver numericamente a EDO que descreve o comportamento do pêndulo múltiplo.

2.1 Métodos Numéricos

Nesta subseção é apresentada a definição dos métodos numéricos utilizados para modelar e simular o comportamento do pêndulo múltiplo.

2.1.1 Euler Explícito

O método de Euler Explícito é uma técnica simples para determinar a aproximação numérica da solução de uma EDO. Este método consiste em aproximar a solução da EDO por meio de incrementos calculados com base na taxa de variação da função. A partir de uma condição inicial, isto é, um ponto que percentece a solução da EDO, o método de Euler calcula o próximo valor da função por meio da seguinte fórmula de recorrência de passo único

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{1}$$

em que y_n representa a solução aproximada no instante t_n, h é o passo de tempo, e $f(t_n, y_n)$ é a derivada da função incógnita $y(t)$ (QUARTERONI, 2006). A Figura 1 mostra o algoritmo desenvolvido para implementar o método de Euler Explicíto em MATLAB.

Figura 1 – Método de Euler Explícito implementado em MATLAB

```
function [t, y] = euler_explicito(f, y0, t0, tf, h)

    t = t0:h:tf;
    n = length(t);

    y = zeros(n, 1);
    y(1) = y0; % Condição inicial

    % Implementação do Método de Euler Explícito
    for i = 1:n-1
        y(i+1) = y(i) + h * f(t(i), y(i));
    end
end
```

Fonte: Elaboração própria (2024).

Embora a implementação do método de Euler Explicito seja simples, como visto na Figura 1, em problemas com alta variação ou oscilações rápidas, esse método pode introduzir erros significativos nestes casos, devido à alta sensibilidade do método a variações de h. Dessa forma, o Método de Euler Explícito é pouco eficaz para resolver equações rígidas ou sistemas complexos (CHAPRA, 2011).

No que diz respeito à precisão, o método de Euler Explicito é classificado como um método de primeira ordem, pelo fato de possuir erro global da ordem 1, isto é, $\mathcal{O}(h)$. Além disso, este método possui erro de truncamento local de ordem 2 ($\mathcal{O}(h)$). Matematicamente, o erro de truncamento do método de Euler explicito é dado por

$$\tau_n(h) = \frac{Mh}{2} \tag{2}$$

em que $M = \max_{t\in[t_0,T]} |y''(t)|$ (QUARTERONI, 2006).

2.1.2 Euler Implícito

O método de Euler Implícito é uma técnica estimar numericamente a solução de uma EDO. Este método consiste em aproximar a solução da EDO por meio de incrementos calculados com base na taxa de variação da função. Diferentemente, do método de Euler Explicíto, em que a derivada é avaliada no instante atual de tempo, o Euler Implicíto estima a solução futura usando a derivada do próximo instante de tempo.

Desta forma, o método de Euler Implícito estima a solução numérica da EDO por meio da seguinte relação de recorrência

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}) \tag{3}$$

em que h é o passo de tempo e $t_{n+1} = t_n + h$. A equação 3 é classificada como não linear, pois y_{n+1}, que é a incógnita a ser determinada, aparece nos dois lados da equação. (QUARTERONI, 2006).

Pelo fato da equação 3 ser não linear, resolver-se-á interativamente, por meio de um método numérico interativo como por exemplo o método de como o método de Newton-Raphson. A Figura 2 mostra o algoritmo desenvolvido para implementar o método de Euler Implicíto em MATLAB.

Figura 2 – Método de Euler Implícito implementado em MATLAB

```
function [t, y] = euler_implicito(f, y0, t0, tf, h)

    t = t0:h:tf;
    n = length(t);

    % Inicialização do vetor de solução
    y = zeros(n, 1);
    y(1) = y0; % Condição inicial

    % Implementação do Método de Euler Implícito
    for i = 1:n-1
        g = @(z) z - y(i) - h * f(t(i+1), z);
        y(i+1) = fsolve(g, y(i));
    end
end
```

Fonte: Elaboração própria (2024).

Apesar do método de Euler Implícito ser um método de primeira ordem, cujo erro associado depende da derivada da função, assim como o método de Euler Explícito, o método de Euler Implícito apresenta uma estabilidade maior e consequentemente uma maior precisão do que o método de Euler Explícito. Isso se deve ao fato de que o método implícito, ao considerar a derivada no próximo instante de tempo, acaba por suavizar as oscilações e minimizar as instabilidades numéricas observadas nos métodos explícitos (TRINDADE, 2024).

De modo geral, o método de Euler Explícito é mais simples de ser implementado e é mais rápido computacionalmente quando comparado com o método de Euler Implícito. Contudo, o método de Euler Implícito, ao considerar a derivada no próximo instante de tempo, tende a ser mais robusto, proporcionando soluções mais estáveis mesmo para passos de tempo maiores. Por outro lado, seu custo computacional tende a ser maior pelo fato da necessidade de resolver uma equação não linear a cada passo de tempo por meio de um método interativo.

2.1.3 Runge-Kutta de ordem 4

O método de Runge-Kutta de quarta ordem é um método explícito que oferece um bom equilíbrio entre precisão, estabilidade e custo computacional, sendo por conta disso amplamente utilizado na engenharia e ciências exatas para realizar simulações numéricas e modelagem computacional de sistemas dinâmicos. (TRINDADE, 2024). Comparado aos métodos de Euler, o método de Runge-Kutta de quarta ordem é significativamente mais preciso e estável, pois apresenta erro de truncamento local de ordem 5 e erro global de

ordem 4, mas requer mais recursos computacionais (CHAPRA, 2011). Do ponto de vista numérico, o método de Runge-Kutta de ordem 4 determinar a solução numérica da EDO calculando um conjunto de fórmulas de recorrência. As fórmulas de recorrência do método de Runge-Kutta 4 são

$$\begin{aligned} k_1 &= hf(t_n, u_n), \\ k_2 &= hf\left(t_n + \frac{h}{2}, u_n + \frac{k_1}{2}\right), \\ k_3 &= hf\left(t_n + \frac{h}{2}, u_n + \frac{k_2}{2}\right), \\ k_4 &= hf(t_n + h, u_n + k_3). \end{aligned} \tag{4}$$

Já a solução numerica da EDO é dada por

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{5}$$

De modo geral, essas as equações 4 e 5 envolvem calcular quatro gradientes (ou derivadas) ponderados para obter uma estimativa mais precisa da variação da função.

2.2 Problema

Nesta subseção é apresentado o embasamento teórico necessário para modelar e simular computacionalmente o comportamento do pêndulo múltiplo.

2.2.1 Mecânica Lagrangiana

A aplicação da mecânica lagrangiana em um pêndulo acoplado simplifica significativamente a análise em comparação com o método tradicional de decomposição de forças. Enquanto o método tradicional exige a consideração de todas as forças atuantes e suas componentes em diferentes direções, a abordagem lagrangiana utiliza a energia potencial e cinética para formular as equações de movimento de maneira mais direta e eficiente. Essa simplificação se deve ao uso das coordenadas generalizadas, que permitem descrever o sistema com menos variáveis e sem a necessidade de resolver equações vetoriais. Portanto, embora seja possível resolver o problema do pêndulo não acoplado usando decomposição de forças, a mecânica lagrangiana torna o processo mais intuitivo e menos trabalhoso.

A função principal na mecânica lagrangiana é a *Lagrangiana* L, que é definida como a diferença entre a energia cinética T e a energia potencial V do sistema:

$$L = T - V \tag{6}$$

Para um sistema de coordenadas generalizadas $\{q_i\}$, as equações de movimento são obtidas a partir das equações de Euler-Lagrange:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = 0 \tag{7}$$

2.2.1.1 Energia Cinética e Potencial

A energia cinética T de um sistema de partículas pode ser expressa como:

$$T = \frac{1}{2}\sum_{i=1}^{n} m_i v_i^2 \tag{8}$$

A energia potencial V pode ser expressa como:

$$V = \sum_{i=1}^{n} m_i g y_i \tag{9}$$

2.2.2 Exemplo: Pêndulo Simples

Para ilustrar, será considerado um pêndulo simples de comprimento l e massa m, onde o ângulo θ descreve a posição do pêndulo. A energia cinética e potencial do sistema são dadas por:

$$T = \frac{1}{2} m l^2 \dot{\theta}^2 \tag{10}$$

$$V = mgl(1 - \cos\theta) \tag{11}$$

Assim, a Lagrangiana é:

$$L = T - V = \frac{1}{2} m l^2 \dot{\theta}^2 - mgl(1 - \cos\theta) \tag{12}$$

Aplicando a equação de Euler-Lagrange, obtemos a equação de movimento do pêndulo:

$$\frac{d}{dt}\left(ml^2\dot{\theta}\right) + mgl\sin\theta = 0 \tag{13}$$

Ou, simplificando:

$$\ddot{\theta} + \frac{g}{l}\sin\theta = 0 \tag{14}$$

Esta é a mesma equação diferencial que se obtém ao aplicar o método tradicional de decomposição de forças ao pêndulo simples, demonstrando que ambos os métodos fornecem resultados equivalentes.

2.2.3 Caso de n pêndulos acoplados

Como demonstrado, a mecânica lagrangiana funciona eficientemente para descrever o comportamento de um pêndulo simples. Com base nessa fundamentação, podemos agora generalizar essa abordagem para um sistema mais complexo, envolvendo n pêndulos acoplados. Para a simplificação dos cálculos desse problema, considerou-se um sistema sem fricção, com cordas de comprimento $1m$ e sem massa, e massas do pêndulo de $1kg$. Além disso, considerou-se $g = 9.8$ m/s^2.

Como as cordas têm comprimento de $1m$, é possível determinar as posições x_i e y_i da i-ésima massa do pendulo somente em termos do ângulo com a vertical θ. Com essas considerações, tem-se que:

$$x_i = \sum_{j=1}^{i} \sin(\theta_j) \tag{15}$$

$$y_i = -\sum_{j=1}^{i} \cos(\theta_j) \tag{16}$$

Para achar a velocidade, basta derivar em relação ao tempo, de forma que se encontra a seguinte relação:

$$\dot{x}_i = \sum_{j=1}^{i} \dot{\theta}_j \cos(\theta_j) \tag{17}$$

$$\dot{y}_i = \sum_{j=1}^{i} \dot{\theta}_j \sin(\theta_j) \tag{18}$$

Com esses termos, encontra-se a energia cinética do sistema em função de θ e $\dot{\theta}$ por meio de:

$$T = \frac{1}{2} \sum_{i=1}^{n} m_i v_i^2 \tag{19}$$

Lembrando que considerou-se as massas do pêndulo como sendo $1\,\text{kg}$, e que a velocidade pode ser decomposta em dois termos de velocidade perpendiculares $\dot{x}_i$ e $\dot{y}_i$, tem-se que:

$$T = \frac{1}{2} \sum_{i=1}^{n} (\dot{x}_i^2 + \dot{y}_i^2) \tag{20}$$

Substituindo as Equações 17 e 18 na Equação 20, chega-se a:

$$T = \frac{1}{2} \sum_{i=1}^{n} \left(\left[\sum_{j=1}^{i} \dot{\theta}_j \cos(\theta_j) \right]^2 + \left[\sum_{j=1}^{i} \dot{\theta}_j \sin(\theta_j) \right]^2 \right) \tag{21}$$

Expandindo os termos ao quadrado e reorganizando a expressão, encontra-se que:

$$T = \frac{1}{2} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \dot{\theta}_j^2 + \sum_{j \neq k} 2\dot{\theta}_j \dot{\theta}_k \cos(\theta_j) \cos(\theta_k) + \sum_{j \neq k} 2\dot{\theta}_j \dot{\theta}_k \sin(\theta_j) \sin(\theta_k) \right) \tag{22}$$

Para simplificar ainda mais a equação (22), usa-se a identidade trigonométrica:

$$\cos(A - B) = \cos A \cos B + \sin A \sin B \tag{23}$$

o que permite escrever a equação de energia cinética na forma:

$$T = \frac{1}{2} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \frac{1}{2} \dot{\theta}_j^2 + \sum_{j \neq k} \dot{\theta}_j \dot{\theta}_k \cos(\theta_j - \theta_k) \right) \tag{24}$$

Com a energia cinética calculada, parte-se para encontrar a energia potencial do sistema, usando:

$$V = \sum_{i=1}^{n} m_i g y_i \tag{25}$$

Substituindo a Equação 16 na Equação 25, tem-se que:

$$V = -g \sum_{i=1}^{n} \sum_{j=1}^{i} \cos(\theta_j) \tag{26}$$

que pode ser reescrito como:

$$V = -g \sum_{i=1}^{n} (n - i + 1) \cos(\theta_j) \tag{27}$$

Com a energia potencial e cinética calculada, e lembrando da definição da Lagrangiana como sendo $L = T - V$, tem-se que:

$$L = \frac{1}{2} \sum_{i=1}^{n} \left(\sum_{j=1}^{n} \frac{1}{2} \dot{\theta}_j^2 + \sum_{j \neq k} \dot{\theta}_j \dot{\theta}_k \cos(\theta_j - \theta_k) \right) + g \sum_{i=1}^{n} (n - i + 1) \cos(\theta_j) \tag{28}$$

Usando a Lagrangiana calculada, aplica-se as equações de Euler-Lagrange:

$$\frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\theta}_i} \right) - \frac{\partial L}{\partial \theta_i} = 0 \tag{29}$$

Para calcular o primeiro termo, tira-se a derivada da Lagrangiana com respeito a $\dot{\theta}$:

$$\frac{\partial L}{\partial \dot{\theta}_i} = \sum_{j=1}^{n} c(i,j) \dot{\theta}_j \cos(\theta_i - \theta_j), \quad c(i,j) = n - \max(i,j) + 1 \tag{30}$$

Derivando no tempo, tem-se que:

$$\frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\theta}_i} \right) = \sum_{j=1}^{n} c(i,j) \left[\ddot{\theta}_j \cos(\theta_i - \theta_j) - \dot{\theta}_i \dot{\theta}_j \sin(\theta_i - \theta_j) + \theta_j^2 \sin(\theta_i - \theta_j) \right] \tag{31}$$

Para encontrar o segundo termo, computa-se a derivada da Lagrangiana com respeito a θ:

$$\frac{\partial L}{\partial \theta_i} = -g(n - i + 1) \sin(\theta_i) - \sum_{j=1}^{n} c(i,j) \dot{\theta}_i \dot{\theta}_j \sin(\theta_i - \theta_j) \tag{32}$$

Substituindo os termos encontrados na equação de Euler-Lagrange e reorganizando termos, encontra-se que:

$$\sum_{j=1}^{n} c(i,j) \dot{\theta}_i \dot{\theta}_j \cos(\theta_i - \theta_j) = -g(n - i + 1) \sin(\theta_i) - \sum_{j=1}^{n} c(i,j) \dot{\theta}_i \dot{\theta}_j \sin(\theta_i - \theta_j) \tag{33}$$

Observa-se que esta equação pode ser interpretada como um sistema do tipo $\mathbf{Ax} = \mathbf{b}$, onde A é representada por $c(i,j) \cos(\theta_i - \theta_j)$, x corresponde a $\ddot{\theta}_i$, e b é dado por $-g(n - i + 1) \sin(\theta_i) - \sum_{j=1}^{n} c(i,j) \dot{\theta}_i \dot{\theta}_j \sin(\theta_i - \theta_j)$.

2.3 Código

No apêndice A é apresentado o código desenvolvido para modelar o comportamento do pêndulo múltiplo a partir das equações que descrevem seu movimento e dos métodos de Euler Explícito, Euler Implícito e Runge-Kutta.

De forma geral, para desenvolver o código da implementação do pêndulo multiplo, foi utilizada a programação orientada a objetos para simplificar a organização e a estrutura do código. Foi criada uma classe `Pendulo` para armazenar informações importantes do sistema, como o número de massas acopladas e a posição atual do pêndulo.

Inicialmente, foram desenvolvidos os métodos `A()` e `b()` para a classe, que criam as matrizes A e b descritas pelas equações de movimento previamente encontradas. É importante notar que a matriz A depende apenas dos valores da matriz contendo os ângulos θ, enquanto b depende tanto dos ângulos θ quanto das velocidades $\dot{\theta}$.

Em seguida, foi definido o método `f()`, que retorna a solução da matriz do sistema linear $Ax = b$, contendo as acelerações do sistema, e também retorna as velocidades sem modificá-las.

Com os valores de aceleração angular e velocidade angular obtidos, é possível calcular as novas posições e velocidades angulares do sistema usando métodos numéricos. Isso permite resolver novamente o sistema $Ax = b$, criando um loop que permite mapear todos os possíveis estados do sistema dentro de um intervalo de tempo, a partir dos valores do estado inicial do pendulo. Em seguida, plota-se todos os estados do pêndulo em sequencia.

Para calcular esses estados a partir da aceleração e velocidade, utilizou-se um método de resolução de EDO. Para isto, adaptou-se o código desenvolvendo de modo que essas EDOs fossem resolvidas numericamente respectivamente pelos métodos de Runge-Kutta de quarta ordem, Euler Explícito e Euler Implícito. As Figuras de 3 a 5 mostram os métodos de Runge-Kutta de ordem 4, Euler Explícito e Euler Implícito impletentados em Python para calcular as EDOs associadas ao movimento do pêndulo múltiplo.

Figura 3 – Implementação do Método de Runge-Kutta de quarta ordem, utilizado para calcular as EDOs associadas ao movimento do pêndulo múltiplo.

```
def RK4(self, dt, thetas, thetaDots):
    k1 = self.f(thetas, thetaDots)
    k2 = self.f(thetas + 0.5 * dt * k1[0], thetaDots + 0.5 * dt * k1[1])
    k3 = self.f(thetas + 0.5 * dt * k2[0], thetaDots + 0.5 * dt * k2[1])
    k4 = self.f(thetas + dt * k3[0], thetaDots + dt * k3[1])

    thetaDeltas = (k1[0] + 2 * k2[0] + 2 * k3[0] + k4[0]) * dt / 6
    thetaDotDeltas = (k1[1] + 2 * k2[1] + 2 * k3[1] + k4[1]) * dt / 6

    return [thetas + thetaDeltas, thetaDots + thetaDotDeltas]

def tick(self, dt):
    newState = self.RK4(dt, self.thetas, self.thetaDots)
    self.thetas = newState[0]
    self.thetaDots = newState[1]
```

Fonte: Elaboração própria (2024).

Figura 4 – Implementação de Euler Explícito, utilizado para calcular as EDOs associadas ao movimento do pêndulo múltiplo.

```
def euler_explicito(self, dt):
    thetas_next = np.zeros(self.n)
    thetaDots_next = np.zeros(self.n)
    for i in range(self.n):
        f_i = self.f(self.thetas, self.thetaDots)
        thetas_next[i] = self.thetas[i] + self.thetaDots[i] * dt
        thetaDots_next[i] = self.thetaDots[i] + f_i[1][i] * dt
    self.thetas = thetas_next
    self.thetaDots = thetaDots_next

def tick(self, dt):
    self.euler_explicito(dt)
```

Fonte: Elaboração própria (2024).

Figura 5 – Implementação de Euler Implícito, utilizado para calcular as EDOs associadas ao movimento do pêndulo múltiplo.

```
def euler_implicito(self, dt):
    def newton_step(thetas_guess, thetaDots_guess):
        thetas_next = thetas_guess
        thetaDots_next = thetaDots_guess
        f_i = self.f(thetas_next, thetaDots_next)
        thetas_residual = thetas_next - self.thetas - dt * thetaDots_next
        thetaDots_residual = thetaDots_next - self.thetaDots - dt * f_i[1]
        return np.concatenate((thetas_residual, thetaDots_residual))

    tolerance = 1e-10
    max_iter = 100
    initial_guess = np.concatenate((self.thetas, self.thetaDots))
    for _ in range(max_iter):
        residual = newton_step(initial_guess[:self.n], initial_guess[self.n:])
        jacobian = self.jacobian(initial_guess[:self.n], initial_guess[self.n:], dt)
        delta = np.linalg.solve(jacobian, -residual)
        initial_guess += delta
        if np.linalg.norm(delta) < tolerance:
            break
    self.thetas = initial_guess[:self.n]
    self.thetaDots = initial_guess[self.n:]
def jacobian(self, thetas, thetaDots, dt):
    eps = 1e-8
    size = 2 * self.n
    jacobian = np.zeros((size, size))
    for i in range(size):
        perturb = np.zeros(size)
        perturb[i] = eps
        perturbed_residual = self.residual(thetas + perturb[:self.n], thetaDots + perturb[self.n:], dt)
        original_residual = self.residual(thetas, thetaDots, dt)
        jacobian[:, i] = (perturbed_residual - original_residual) / eps

    return jacobian
def residual(self, thetas, thetaDots, dt):
    f_i = self.f(thetas, thetaDots)
    thetas_residual = thetas - self.thetas - dt * thetaDots
    thetaDots_residual = thetaDots - self.thetaDots - dt * f_i[1]
    return np.concatenate((thetas_residual, thetaDots_residual))
def tick(self, dt):
    self.euler_implicito(dt)
```

Fonte: Elaboração própria (2024).

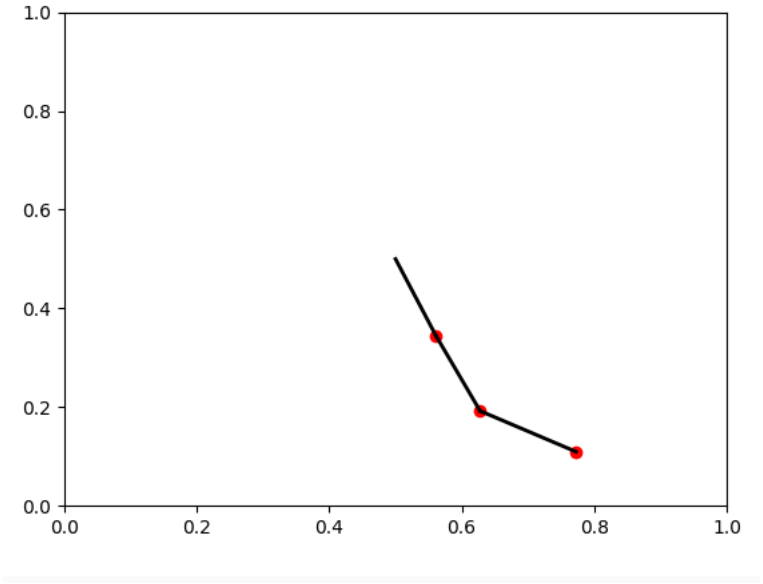
A partir da análise das Figuras de 3 a 5, observa-se que os métodos foram implementados em conformidade com suas definições, com algumas pequenas alterações para se adequarem ao problema em análise e à linguagem de programação utilizada. Além disso, no método de Euler Implícito, foi implementado também o método de Newton-Raphson para resolver a equação não linear.

3 RESULTADOS

A partir da execução dos três códigos apresentados nos apêndices A, B e C, foram geradas animações do movimento do pêndulo múltiplo. As Figuras 6 a 8 ilustram diferentes momentos da animação obtida pelo código que utiliza o método de Runge-Kutta de ordem 4, enquanto as Figuras 9 a 11 mostram instantes diferentes da animação obtida pelo código que utiliza o método de Euler Explícito. Por fim, as Figuras 12 a 14 apresentam instantes variados da animação obtida pelo código que utiliza o método de Euler Implícito. Além disso, as animações do movimento do pêndulo multiplo estão disponeis no link abaixo:

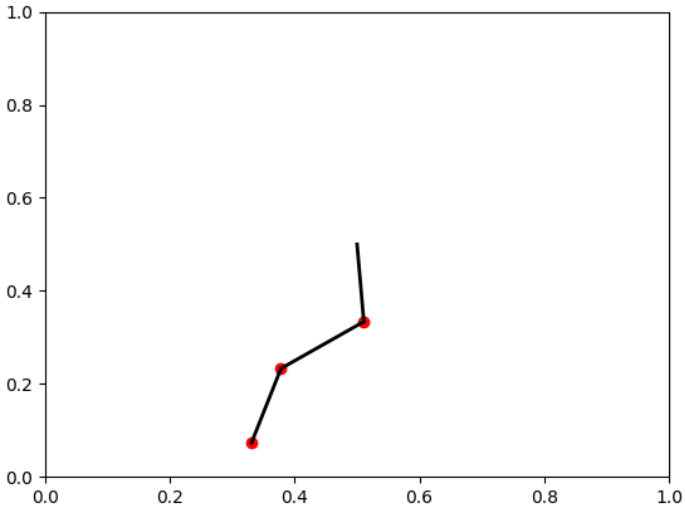
https://drive.google.com/drive/folders/1Z9zkYtXKqd$_{y}nV2ixu8RpgZmyilHepFJ?usp = sharing$

Figura 6 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Runge-Kutta de ordem 4 - Instante 1.



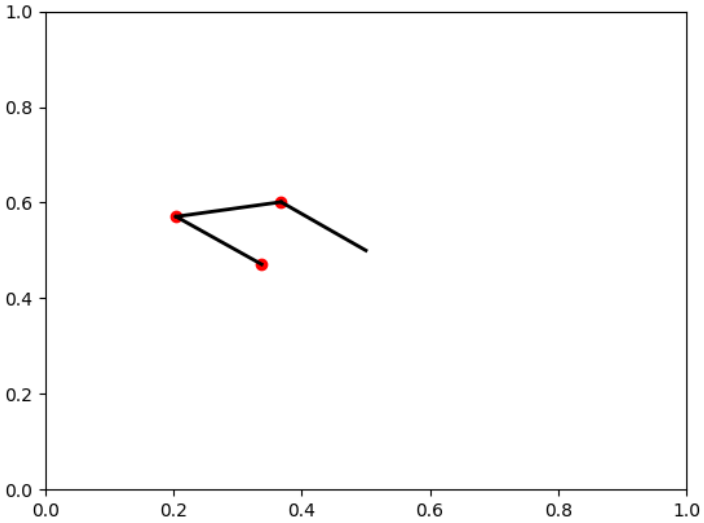
Fonte: Elaboração própria (2024).

Figura 7 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Runge-Kutta de ordem 4 - Instante 2.



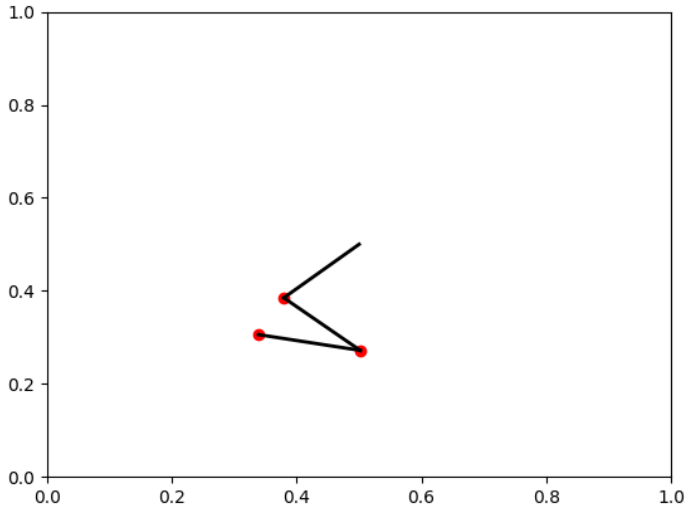
Fonte: Elaboração própria (2024).

Figura 8 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Runge-Kutta de ordem 4 - Instante 3.



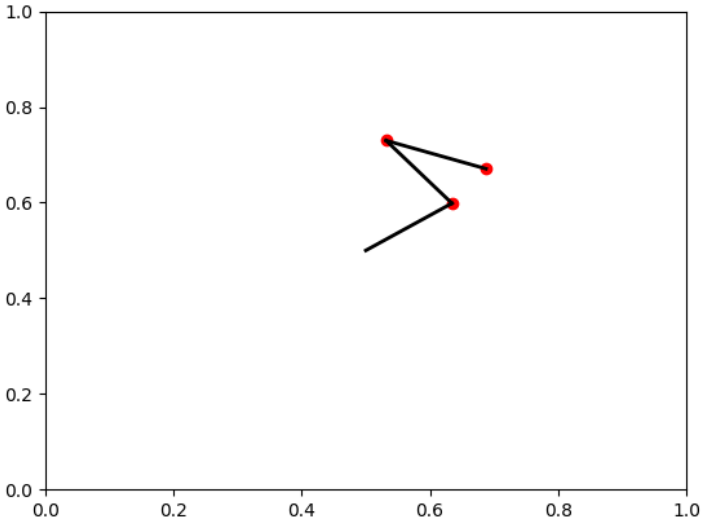
Fonte: Elaboração própria (2024).

Figura 9 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Euler Explícito - Instante 1.



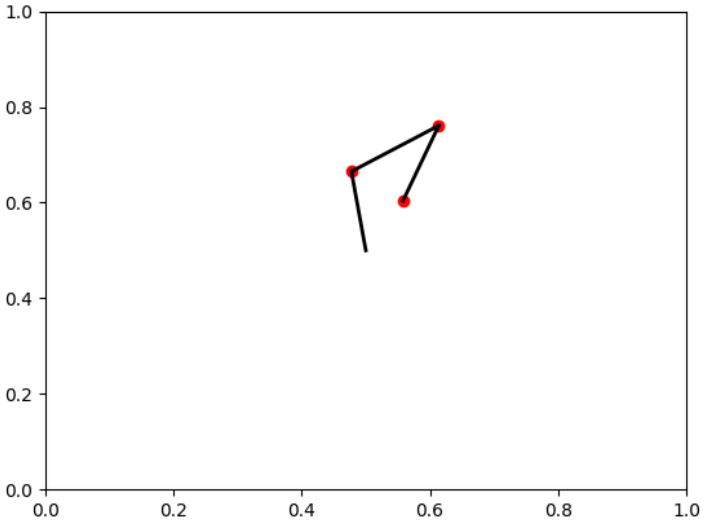
Fonte: Elaboração própria (2024).

Figura 10 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Euler Explícito - Instante 2.



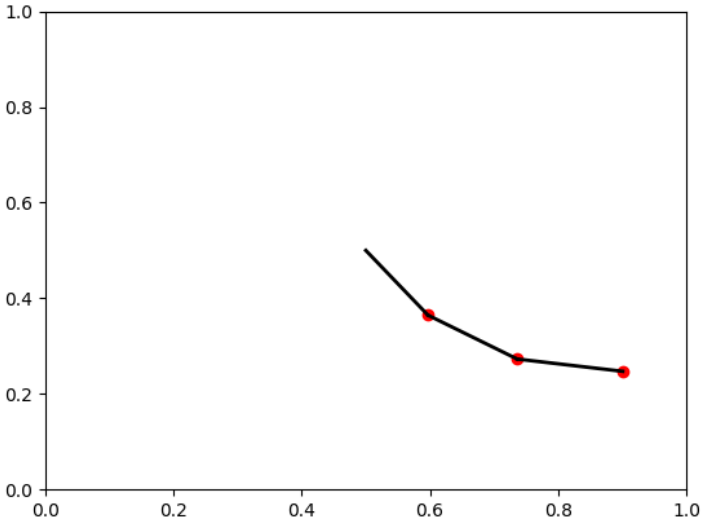
Fonte: Elaboração própria (2024).

Figura 11 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Euler Explícito - Instante 3.



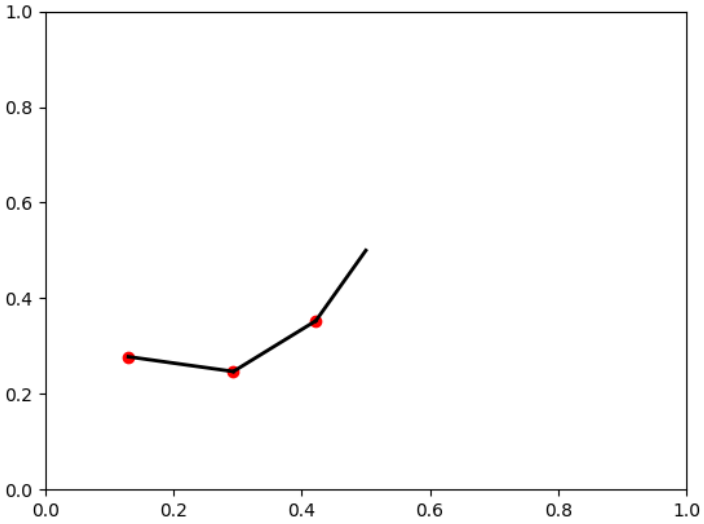
Fonte: Elaboração própria (2024).

Figura 12 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Euler Implícito - Instante 1.



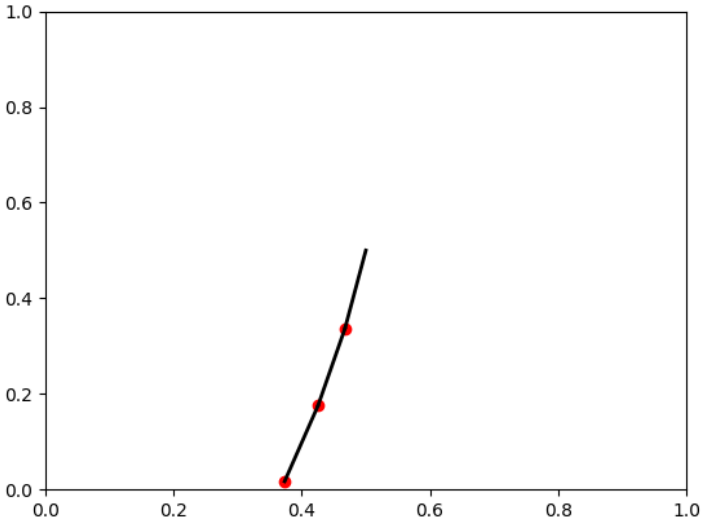
Fonte: Elaboração própria (2024).

Figura 13 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Euler Implícito - Instante 2.



Fonte: Elaboração própria (2024).

Figura 14 – Animação do movimento do pêndulo múltiplo obtido por meio do método de Euler Implícito - Instante 3.



Fonte: Elaboração própria (2024).

A partir da análise do movimento realizado pelo pêndulo múltiplo nos três casos, observou-se que o método de Runge-Kutta de ordem 4 foi o que mais se aproximou do movimento real do pêndulo múltiplo. Além disso, como esperado, os métodos de Euler não foram capazes de descrever com precisão o movimento do pêndulo simples, pelo fato de que esses métodos não apresentam muita eficiência na análise de sistemas físicos caóticos, como o pêndulo múltiplo.

Entretanto, embora fosse esperado que o código utilizando o método de Euler Implícito não pudesse reproduzir perfeitamente o movimento do pêndulo múltiplo, esperava-se que os resultados fossem pelo menos um pouco mais próximos dos obtidos pelo método de Runge-Kutta. No entanto, ao analisar o movimento do pêndulo obtido pelo método implícito, observa-se um comportamento que sugere amortecimento, o que não deveria ocorrer, uma vez que o modelo matemático do pêndulo múltiplo não especifica nenhum amortecimento, sendo este erro observado ainda que alterado o passo de tempo adotado. Por conta disso, é provável que possa haver algum erro na impplentação do método de Euler Implícito ou isto está ocorre apenas pelas características inerentes do método. Pois, embora o método de Euler seja conhecido por sua estabilidadade em relação ao tamanho do passo, este método pode introduzir erros significativos, especialmente em sistemas que requerem alta precisão na simulação do movimento, como é o caso do pêndulo múltiplo. A subestimação da taxa de mudança em cada passo de tempo pelo método pode levar a uma perda de energia artificial ao longo do tempo, simulando erroneamente um amortecimento que não está presente no modelo físico real.

Ademais, a instabilidade no movimento do pêndulo obtido a partir do método de Euler Explícito deve-se à alta sensibilidade do método às variações do passo, bem como à

caoticidade do problema em análise.

4 CONCLUSÃO

Portanto, por meio da visualização da animação dos pêndulos, é perceptível a diferença entre os métodos, sendo que, em ordem decrescente de precisão, os métodos ficam ordenados em Runge Kutta, Euler implícito, Euler explícito. Analisando os métodos de maneira individual, em primeiro lugar, o método de Euler explícito se demonstra como o menos próprio a essa simulação, embora sua simplicidade e baixo custo computacional, em razão ao fato de ser instável em grandes passos de tempo e impreciso em sistemas não lineares e com grandes períodos de duração, sendo que isso é demonstrado pela desordenação da animação do pêndulo resultante desse método. Já o método de Euler implícito se apresenta um movimento mais estável, porém distante do real movimento do pêndulo múltiplo, pelo fato de apresentar um amortecimento que não foi definido no modelo matemático. Isto ocorre possivelmente em razão da presença de erros numéricos que ficam mais evidentes conforme o passar do tempo, contribuindo para a dissipação de energia do sistema, e as características do método.

Por último, o método de Runge Kutta é visivelmente o mais adequado para essa prática devido à sua maior precisão mesmo em passos maiores de tempo, com um erro de $\mathcal{O}(h^4)$, além de sua melhor gestão com erros numéricos, que fica evidente com a ausência de um amortecimento artificial apresentado no método de Euler implícito, entretanto, vale destacar que seu custo computacional é maior, embora possa ser uma escolha comum na resolução de EDOs.

Portanto, nota-se que o método de Runge-Kutta de ordem 4 é considerado mais preciso e eficiente para simular o comportamento de sistemas dinâmicos caóticos. Os resultados obtidos destacam as limitações dos métodos numéricos de primeira ordem na análise de sistemas caóticos, evidenciando assim a importância de escolher os métodos numéricos adequados à complexidade do problema.

REFERÊNCIAS

CHAPRA, R. P. C. S. C. *Métodos Numéricos para Engenharia*. 5th ed.. ed. Porto Alegre: McGraw-Hil, 2011.

QUARTERONI, F. S. A. *Scientific Computing with MATLAB and Octave*. 2th ed.. ed. Milado: Springer, 2006.

TRINDADE, M. A. *Roteiro de práticas de Problemas em Engenharia Mecatrônica 2: Aproximação numérica de EDOs de 1ª ordem*. São Carlos: Universidade de São Paulo, 2024.

APÊNDICE A

Código de simulação do movimento do pêndulo multiplo considerando o método de Runge-Kutta de quarta ordem.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class Pendulo:
    def __init__(self, n=3, thetas=None, thetaDots=None, g=-9.8):
        self.n = n
        self.thetas = np.full(n, 0.5 * np.pi) if thetas is None else thetas
        self.thetaDots = np.zeros(n) if thetaDots is None else thetaDots
        self.g = -g

    def A(self, thetas):
        M = np.zeros((self.n, self.n))
        for i in range(self.n):
            for j in range(self.n):
                M[i, j] = (self.n - max(i, j)) * np.cos(thetas[i] - thetas[j])
        return M

    def b(self, thetas, thetaDots):
        v = np.zeros(self.n)
        for i in range(self.n):
            b_i = 0
            for j in range(self.n):
                b_i -=(self.n - max(i, j))*np.sin(thetas[i]-thetas[j])*thetaDots[j]** 2
            b_i -= self.g * (self.n - i) * np.sin(thetas[i])
            v[i] = b_i
        return v

    def f(self, thetas, thetaDots):
        A = self.A(thetas)
        b = self.b(thetas, thetaDots)
        return [thetaDots, np.linalg.solve(A, b)]

    def RK4(self, dt, thetas, thetaDots):
        k1 = self.f(thetas, thetaDots)
        k2 = self.f(thetas + 0.5 * dt * k1[0], thetaDots + 0.5 * dt * k1[1])
        k3 = self.f(thetas + 0.5 * dt * k2[0], thetaDots + 0.5 * dt * k2[1])
        k4 = self.f(thetas + dt * k3[0], thetaDots + dt * k3[1])

        thetaDeltas = (k1[0] + 2 * k2[0] + 2 * k3[0] + k4[0]) * dt / 6
        thetaDotDeltas = (k1[1] + 2 * k2[1] + 2 * k3[1] + k4[1]) * dt / 6

        return [thetas + thetaDeltas, thetaDots + thetaDotDeltas]

    def tick(self, dt):
        newState = self.RK4(dt, self.thetas, self.thetaDots)
        self.thetas = newState[0]
        self.thetaDots = newState[1]

    @property
```

```
51      def coordinates(self):
52          x = 0
53          y = 0
54          coords = []
55          for i in range(len(self.thetas)):
56              theta = self.thetas[i]
57              x += np.sin(theta)
58              y += np.cos(theta)
59              coords.append((x, y))
60          return coords
61
62  pendulo = Pendulo()
63
64  fig, ax = plt.subplots()
65
66  def draw(i):
67      ax.clear()
68      coords = pendulo.coordinates
69      x1, y1 = 0.5, 0.5
70
71      for coord in coords:
72          x2 = 0.5 + coord[0] * 0.5 / pendulo.n
73          y2 = 0.5 - coord[1] * 0.5 / pendulo.n
74          ax.plot([x1, x2], [y1, y2], color='black', linewidth=2)
75          ax.scatter(x2, y2, color='red')
76
77          x1, y1 = x2, y2
78
79      ax.set_xlim(0, 1)
80      ax.set_ylim(0, 1)
81
82  def animate(i):
83      draw(i)
84      pendulo.tick(1/30)
85
86  ani = FuncAnimation(fig, animate, frames=100, interval=33)
87  plt.show()
```

APÊNDICE B

Código de simulação do movimento do pêndulo multiplo considerando o método de Euler Explícito.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class Pendulo:
    def __init__(self, n=3, thetas=None, thetaDots=None, g=-9.8):
        self.n = n
        self.thetas = np.full(n, 0.5 * np.pi) if thetas is None else thetas
        self.thetaDots = np.zeros(n) if thetaDots is None else thetaDots
        self.g = -g

    def A(self, thetas):
        M = np.zeros((self.n, self.n))
        for i in range(self.n):
            for j in range(self.n):
                M[i, j] = (self.n - max(i, j)) * np.cos(thetas[i] - thetas[j])
        return M

    def b(self, thetas, thetaDots):
        v = np.zeros(self.n)
        for i in range(self.n):
            b_i = 0
            for j in range(self.n):
                thetaDot_squared = np.clip(thetaDots[j], -100, 100) ** 2
                b_i -= (self.n - max(i, j)) * np.sin(thetas[i] - thetas[j]) * thetaDot_squared
            b_i -= self.g * (self.n - i) * np.sin(thetas[i])
            v[i] = b_i
        return v


    def f(self, thetas, thetaDots):
        A = self.A(thetas)
        b = self.b(thetas, thetaDots)
        return [thetaDots, np.linalg.solve(A, b)]

    def euler_explicito(self, dt):
        thetas_next = np.zeros(self.n)
        thetaDots_next = np.zeros(self.n)
        for i in range(self.n):
            f_i = self.f(self.thetas, self.thetaDots)
            thetas_next[i] = self.thetas[i] + self.thetaDots[i] * dt
            thetaDots_next[i] = self.thetaDots[i] + f_i[1][i] * dt
        self.thetas = thetas_next
        self.thetaDots = thetaDots_next

    def tick(self, dt):
        self.euler_explicito(dt)

    @property
    def coordinates(self):
```

```
51          x = 0
52          y = 0
53          coords = []
54          for i in range(len(self.thetas)):
55              theta = self.thetas[i]
56              x += np.sin(theta)
57              y += np.cos(theta)
58              coords.append((x, y))
59          return coords
60
61  pendulo = Pendulo()
62
63  fig, ax = plt.subplots()
64
65  def draw(i):
66      ax.clear()
67      coords = pendulo.coordinates
68      x1, y1 = 0.5, 0.5
69
70      for coord in coords:
71          x2 = 0.5 + coord[0] * 0.5 / pendulo.n
72          y2 = 0.5 - coord[1] * 0.5 / pendulo.n
73          ax.plot([x1, x2], [y1, y2], color='black', linewidth=2)
74          ax.scatter(x2, y2, color='red')
75
76          x1, y1 = x2, y2
77
78      ax.set_xlim(0, 1)
79      ax.set_ylim(0, 1)
80
81  def animate(i):
82      draw(i)
83      pendulo.tick(1/30)
84
85  ani = FuncAnimation(fig, animate, frames=100, interval=33)
86  plt.show()
87
```

APÊNDICE C

Figura A3: Código de simulação do movimento do pêndulo multiplo considerando o método de Euler Implícito.

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class Pendulo:
    def __init__(self, n=3, thetas=None, thetaDots=None, g=-9.8):
        self.n = n
        self.thetas = np.full(n, 0.5 * np.pi) if thetas is None else thetas
        self.thetaDots = np.zeros(n) if thetaDots is None else thetaDots
        self.g = -g

    def A(self, thetas):
        M = np.zeros((self.n, self.n))
        for i in range(self.n):
            for j in range(self.n):
                M[i, j] = (self.n - max(i, j)) * np.cos(thetas[i] - thetas[j])
        return M

    def b(self, thetas, thetaDots):
        v = np.zeros(self.n)
        for i in range(self.n):
            b_i = 0
            for j in range(self.n):
                thetaDot_squared = np.clip(thetaDots[j], -100, 100) ** 2
                b_i -= (self.n - max(i, j)) * np.sin(thetas[i] - thetas[j]) * thetaDot_squared
            b_i -= self.g * (self.n - i) * np.sin(thetas[i])
            v[i] = b_i
        return v

    def f(self, thetas, thetaDots):
        A = self.A(thetas)
        b = self.b(thetas, thetaDots)
        return [thetaDots, np.linalg.solve(A, b)]

    def euler_implicito(self, dt):
        def newton_step(thetas_guess, thetaDots_guess):
            thetas_next = thetas_guess
            thetaDots_next = thetaDots_guess
            f_i = self.f(thetas_next, thetaDots_next)
            thetas_residual = thetas_next - self.thetas - dt * thetaDots_next
            thetaDots_residual = thetaDots_next - self.thetaDots - dt * f_i[1]
            return np.concatenate((thetas_residual, thetaDots_residual))

        tolerance = 1e-10
        max_iter = 100
        initial_guess = np.concatenate((self.thetas, self.thetaDots))

        for _ in range(max_iter):
            residual = newton_step(initial_guess[:self.n], initial_guess[self.n:])
            jacobian = self.jacobian(initial_guess[:self.n], initial_guess[self.n:], dt)
```

```
51              delta = np.linalg.solve(jacobian, -residual)
52              initial_guess += delta
53              if np.linalg.norm(delta) < tolerance:
54                  break
55
56          self.thetas = initial_guess[:self.n]
57          self.thetaDots = initial_guess[self.n:]
58
59      def jacobian(self, thetas, thetaDots, dt):
60          eps = 1e-8
61          size = 2 * self.n
62          jacobian = np.zeros((size, size))
63
64          for i in range(size):
65              perturb = np.zeros(size)
66              perturb[i] = eps
67              perturbed_residual = self.residual(thetas + perturb[:self.n], thetaDots + perturb[s
68              original_residual = self.residual(thetas, thetaDots, dt)
69              jacobian[:, i] = (perturbed_residual - original_residual) / eps
70
71          return jacobian
72
73      def residual(self, thetas, thetaDots, dt):
74          f_i = self.f(thetas, thetaDots)
75          thetas_residual = thetas - self.thetas - dt * thetaDots
76          thetaDots_residual = thetaDots - self.thetaDots - dt * f_i[1]
77          return np.concatenate((thetas_residual, thetaDots_residual))
78
79      def tick(self, dt):
80          self.euler_implicito(dt)
81
82      @property
83      def coordinates(self):
84          x = 0
85          y = 0
86          coords = []
87          for i in range(len(self.thetas)):
88              theta = self.thetas[i]
89              x += np.sin(theta)
90              y += np.cos(theta)
91              coords.append((x, y))
92          return coords
93
94  pendulo = Pendulo()
95
96  fig, ax = plt.subplots()
97
98  def draw(i):
99      ax.clear()
100     coords = pendulo.coordinates
101     x1, y1 = 0.5, 0.5
102
103     for coord in coords:
104         x2 = 0.5 + coord[0] * 0.5 / pendulo.n
105         y2 = 0.5 - coord[1] * 0.5 / pendulo.n
106         ax.plot([x1, x2], [y1, y2], color='black', linewidth=2)
```

```
107             ax.scatter(x2, y2, color='red')
108 
109             x1, y1 = x2, y2
110 
111         ax.set_xlim(0, 1)
112         ax.set_ylim(0, 1)
113 
114     def animate(i):
115         draw(i)
116         pendulo.tick(1/30)
117 
118     ani = FuncAnimation(fig, animate, frames=100, interval=33)
119     plt.show()
120 
```