Notas de aulas e programas

Francisco Moura

Contents

1 Queda livre com resistência do ar

O problema da queda livre com resistência do ar estende o estudo da física do movimento de corpos sob a gravidade, considerando a força de arrasto que o ar exerce sobre o objeto. A resistência do ar, que age na direção oposta ao movimento, depende de fatores como velocidade, densidade do ar, forma do objeto e sua área de seção transversal. Ela altera a trajetória, tornando-a diferente de uma parábola, como ocorre sem resistência. A resistência pode ser proporcional à velocidade ou ao quadrado dela. Esse modelo é essencial para simulações mais realistas e é aplicado em áreas como engenharia e design de veículos. A simulação computacional permite observar o impacto da resistência sobre a trajetória, variando parâmetros como o coeficiente de resistência.

1.1 Equação de Movimento

A equação da dinâmica para um corpo em queda livre com resistência do ar pode ser escrita como:

$$m\frac{dv}{dt} = mg - kv, \tag{1}$$

onde m é a massa do corpo, g é a aceleração gravitacional, k é o coeficiente de resistência do ar e v é a velocidade.

1.2 Solução Analítica

Separando as variáveis:

$$\frac{dv}{mg - kv} = \frac{dt}{m}. \tag{2}$$

Integramos ambos os lados:

$$\int \frac{dv}{mg - kv} = \int \frac{dt}{m}. \tag{3}$$

Fazendo a substituição $u = mg - kv$, temos $du = -kdv$, e reescrevemos a integral:

$$-\frac{1}{k}\int \frac{du}{u} = \frac{t}{m} + C. \tag{4}$$

A solução da integral é:

$$-\frac{1}{k}\ln|mg - kv| = \frac{t}{m} + C. \tag{5}$$

Isolando v:

$$v(t) = \frac{mg}{k}\left(1 - e^{-\frac{k}{m}t}\right). \tag{6}$$

A velocidade terminal é dada por:

$$v_T = \frac{mg}{k}. \tag{7}$$

1.3 Resolução Numérica

Para resolver a equação 10 numericamente, vamos aplicar o método de Euler. Este é um método simples e direto, baseado na aproximação das derivadas por diferenças finitas.

1.4 Passo 1: Reescrever a equação diferencial

Primeiro, podemos reescrever a equação de forma mais conveniente para o método de Euler. Dividindo ambos os lados por m, obtemos

$$\frac{dv}{dt} = g - \frac{k}{m}v. \tag{8}$$

Vamos definir $\beta = \frac{k}{m}$, então a equação se torna

$$\frac{dv}{dt} = g - \beta v. \tag{9}$$

1.5 Passo 2: Discretização da equação

No método de Euler, discretizamos o tempo em passos de tamanho Δt. Denotamos o tempo discreto por $t_n = n\Delta t$ e a velocidade v_n no tempo t_n. A aproximação de Euler para a derivada $\frac{dv}{dt}$ é dada por:

$$\frac{v_{n+1} - v_n}{\Delta t} = g - \beta v_n.$$

Rearranjando a equação, obtemos a fórmula de atualização para v_{n+1}:

$$v_{n+1} = v_n + \Delta t \left(g - \beta v_n\right).$$

Essa fórmula nos permite calcular a velocidade v_{n+1} a partir do valor da velocidade v_n no passo anterior.

1.6 Passo 3: Algoritmo de implementação

Agora, podemos implementar o algoritmo de Euler para resolver a equação diferencial. Abaixo está o pseudocódigo para a solução numérica:

```
Definir parâmetros:
    m = massa, g = gravidade, k = coeficiente de resistência, v0 = velocidade inicial, dt = passo de tempo, t_max = tempo total

Inicializar:
    t = 0, v = v0

Para cada passo de tempo n:
    Calcular v[n+1] = v[n] + dt * (g - (k/m) * v[n])
    Incrementar t por dt
```

1.7 Passo 4: Resultados numéricos

Ao iterar esse algoritmo, podemos calcular a velocidade $v(t)$ em vários instantes de tempo t. Esse método nos fornece uma aproximação da solução exata da equação diferencial. Vamos apresentar dois programas: o primeiro programa produz apenas um gráfico de velocidade versus tempo; o segundo programa produz uma animação envolvendo este gráfico.

Primeiro programa:

```
# Importação das bibliotecas necessárias
import numpy as np  # Importa o numpy para trabalhar com arrays e realizar cálculos numéricos
import matplotlib.pyplot as plt  # Importa o matplotlib.pyplot para criar gráficos

# Parâmetros físicos do problema
m = 1.0       # Massa do objeto (kg)
g = 9.81      # Aceleração gravitacional (m/s²)
k = 1.5       # Coeficiente de resistência do ar (kg/s)
dt = 0.01     # Passo de tempo (s)
t_max = 30    # Tempo total de simulação (s)

# Inicialização dos arrays
t_values = np.arange(0, t_max, dt)     # Vetor de tempos
v_values = np.zeros_like(t_values)    # Vetor para armazenar velocidades
v = 0.0                                # Velocidade inicial

# Método de Euler para resolver dv/dt = (mg - kv)/m
for i in range(1, len(t_values)):
    dv = (m * g - k * v) / m * dt      # Calcula variação de velocidade
    v += dv                            # Atualiza a velocidade
    v_values[i] = v                    # Armazena no vetor

# Plot do gráfico (sem animação)
plt.figure(figsize=(10, 6))  # Tamanho da figura
plt.plot(t_values, v_values, 'b-', lw=2, label='Velocidade (m/s)')  # Plota a curva
plt.axhline(m * g / k, color='r', linestyle='--', label='Velocidade Terminal')  # Linha da velocidade terminal
plt.xlabel("Tempo (s)")
plt.ylabel("Velocidade (m/s)")
plt.title("Queda Livre com Resistência do Ar")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Segundo programa:

```
# Importação das bibliotecas necessárias
import numpy as np  # Importa o numpy para trabalhar com arrays e realizar cálculos numéricos
import matplotlib.pyplot as plt  # Importa o matplotlib.pyplot para criar gráficos
import matplotlib.animation as animation  # Importa o matplotlib.animation para criar animações

# Parâmetros físicos do problema
m = 1.0  # Define a massa do objeto em quilogramas (kg)
g = 9.81  # Define a aceleração gravitacional em metros por segundo ao quadrado (m/s²)
k = 1.0  # Define o coeficiente de resistência do ar (kg/s)
dt = 0.01  # Define o passo de tempo para a simulação em segundos (s)
t_max = 30  # Define o tempo total de simulação em segundos (s)

# Inicialização dos arrays para armazenar os resultados
t_values = np.arange(0, t_max, dt)  # Cria um array de tempos, que vai de 0 até t_max, com intervalos de dt
v_values = np.zeros_like(t_values)  # Cria um array para armazenar as velocidades em cada instante de tempo
v = 0.0  # Define a velocidade inicial como 0 metros por segundo (m/s)
```

```
# Método de Euler para resolver a equação diferencial dv/dt = (mg - kv)/m numericamente
for i in range(1, len(t_values)):  # Itera sobre todos os índices dos tempos, começando do segundo (índice 1)
    # Calcula a variação da velocidade (dv) usando a equação de movimento:
    # dv/dt = (mg - kv)/m, onde m é a massa, g é a aceleração gravitacional, k é o coeficiente de resistência,
    # e v é a velocidade atual.
    dv = (m * g - k * v) / m * dt  # Calcula a variação de velocidade para o intervalo de tempo dt
    v += dv  # Atualiza a velocidade somando a variação calculada à velocidade anterior
    v_values[i] = v  # Armazena a nova velocidade no array v_values para o tempo correspondente

# Configuração da animação
fig, ax = plt.subplots()  # Cria uma figura e um eixo para o gráfico
ax.set_xlim(0, t_max)  # Define o limite do eixo X (tempo), de 0 até t_max
ax.set_ylim(0, 1.1 * (m * g / k))  # Define o limite do eixo Y (velocidade), ajustado para incluir a velocidade terminal
ax.set_xlabel("Tempo (s)")  # Define o rótulo do eixo X como "Tempo (s)"
ax.set_ylabel("Velocidade (m/s)")  # Define o rótulo do eixo Y como "Velocidade (m/s)"
ax.set_title("Queda Livre com Resistência do Ar")  # Define o título do gráfico

# Criação de uma linha vazia no gráfico para ser preenchida durante a animação
line, = ax.plot([], [], "bo-", lw=2)  # Cria a linha de pontos azuis conectados por uma linha (bo-), com espessura de linha 2

# Função de inicialização da animação
def init():
    line.set_data([], [])  # Limpa os dados da linha antes de iniciar a animação
    return line,  # Retorna a linha, necessário para a animação

# Função que atualiza o gráfico em cada quadro da animação
def update(frame):
    # Atualiza os dados da linha com os pontos correspondentes ao número do quadro atual
    line.set_data(t_values[:frame], v_values[:frame])  # Define os dados da linha com tempos e velocidades até o quadro atual
    return line,  # Retorna a linha, necessário para a animação

# Criação da animação
ani = animation.FuncAnimation(fig, update, frames=len(t_values), init_func=init, blit=True, interval=5)
# ‘update‘ é chamada para cada quadro da animação, ‘init_func‘
#é chamada uma vez no início para configurar a animação
# ‘blit=True‘ melhora o desempenho ao atualizar apenas as partes
#alteradas do gráfico, e ‘interval=5‘ define o intervalo de atualização em milissegundos

# Exibe a animação
plt.show()  # Exibe a animação criada
```

2 Sistema Massa-Mola com resistência do ar

O problema da massa-mola com resistência do ar descreve o movimento de um objeto ligado a uma mola, enquanto é afetado pela resistência do ar. A resistência do ar age como uma força dissipativa, contrariando o movimento do objeto e modificando o comportamento do sistema. Em um cenário sem resistência, o movimento seria oscilatório simples, mas com a resistência do ar, ele se torna amortecido, com a amplitude das oscilações diminuindo ao longo do tempo. Esse modelo é amplamente utilizado para estudar sistemas reais, onde a resistência do meio influencia diretamente a dinâmica. A solução desse problema envolve a análise das equações diferenciais resultantes, que geralmente requerem métodos numéricos para serem resolvidas.

2.1 Equação de Movimento

A equação da dinâmica para um sistema massa-mola com resistência do ar pode ser escrita como:

$$m\frac{d^2x}{dt^2} = -kx - \gamma\frac{dx}{dt}, \tag{10}$$

onde m é a massa, k é a constante elástica da mola, γ é o coeficiente de resistência do ar, e $x(t)$ é o deslocamento da massa.

2.2 Solução Analítica

Esta equação é uma equação diferencial de segunda ordem. A solução da equação depende do valor do discriminante da equação característica associada a ela. Para resolver essa equação diferencial, tentamos uma solução da forma $x(t) = e^{\lambda t}$, onde λ é uma constante a ser determinada. Substituindo esta solução na equação diferencial, obtemos a equação característica:

$$m\lambda^2 + \gamma\lambda + k = 0.$$

Esta é uma equação quadrática, e a solução para λ pode ser obtida através da fórmula de Bhaskara. O discriminante dessa equação é dado por:

$$\Delta = \gamma^2 - 4mk.$$

Dependendo do valor de Δ, as soluções podem ser diferentes. Vamos considerar o caso onde ($\Delta < 0$). Se $\Delta < 0$, temos raízes complexas conjugadas, dada por:

$$\lambda = -\frac{\gamma}{2m} \pm i\omega,$$

onde $\omega = \frac{\sqrt{4mk-\gamma^2}}{2m}$ é a frequência angular das oscilações. A solução é então:

$$x(t) = e^{-\frac{\gamma}{2m}t}\left(C_1\cos(\omega t) + C_2\sin(\omega t)\right).$$

2.3 Resolução Numérica

Para resolver a equação de movimento numericamente, vamos aplicar o método de Euler. Este é um método simples baseado em diferenças finitas.

2.4 Passo 1: Reescrever a equação diferencial

Podemos reescrever a equação de segunda ordem em um sistema de duas equações de primeira ordem. Definimos a velocidade $v = \frac{dx}{dt}$, assim a equação se torna:

$$\frac{dv}{dt} = -\frac{k}{m}x - \frac{\gamma}{m}v.$$

Agora temos o sistema:

$$\frac{dx}{dt} = v,$$

$$\frac{dv}{dt} = -\frac{k}{m}x - \frac{\gamma}{m}v.$$

2.5 Passo 2: Discretização da equação

No método de Euler, discretizamos o tempo em passos de tamanho Δt. Denotamos o tempo discreto por $t_n = n\Delta t$ e as variáveis x_n e v_n para o deslocamento e velocidade, respectivamente. As atualizações para o método de Euler são:

$$x_{n+1} = x_n + v_n \Delta t,$$

$$v_{n+1} = v_n + \left(-\frac{k}{m}x_n - \frac{\gamma}{m}v_n\right)\Delta t.$$

2.6 Passo 3: Algoritmo de implementação

Agora podemos implementar o algoritmo de Euler para resolver a equação diferencial. O pseudocódigo seria:

```
Definir parâmetros:
    m = massa, k = constante elástica,  = coeficiente de resistência, v0 = velocidade inicial, x0 = deslocamento inicial, dt = passo de tempo, t_max = tempo total

Inicializar:
    t = 0, x = x0, v = v0

Para cada passo de tempo n:
    Calcular x[n+1] = x[n] + v[n] * dt
    Calcular v[n+1] = v[n] + (-k/m * x[n] - /m * v[n]) * dt
    Incrementar t por dt
```

2.7 Passo 4: Resultados numéricos

Abaixo estão os código Python que implementam o algoritmo de Euler para resolver numericamente a equação de movimento. Vamos apresentar novamente dois códigos, um sem animação e outro com animação mostrando o deslocamento $x(t)$ ao longo do tempo.

Primeiro programa:

```
# Importação das bibliotecas necessárias
import numpy as np
import matplotlib.pyplot as plt

# Parâmetros físicos do problema
m = 1.0        # Massa (kg)
k = 1.0        # Constante da mola (N/m)
gamma = 0.1    # Coeficiente de resistência do ar (kg/s)
dt = 0.01      # Passo de tempo (s)
t_max = 30     # Tempo total de simulação (s)

# Inicialização dos arrays
t_values = np.arange(0, t_max, dt)
x_values = np.zeros_like(t_values)
v_values = np.zeros_like(t_values)

# Condições iniciais
x = 1.0  # Deslocamento inicial (m)
v = 0.0  # Velocidade inicial (m/s)

# Método de Euler para resolver a equação de movimento
for i in range(1, len(t_values)):
    x_values[i] = x
    v_values[i] = v
    x += v * dt
    v += (-k * x / m - gamma * v / m) * dt

# Plot do gráfico x(t)
plt.figure(figsize=(10, 5))
plt.plot(t_values, x_values, label="x(t)", color="blue")
plt.xlabel("Tempo (s)")
plt.ylabel("Deslocamento (m)")
plt.title("Movimento Massa-Mola com Resistência do Ar")
plt.grid(True)
plt.legend()
plt.show()
```

Segundo programa:

```
# Importação das bibliotecas necessárias
import numpy as np  # Importa o numpy para trabalhar com arrays e realizar cálculos numéricos
import matplotlib.pyplot as plt  # Importa o matplotlib.pyplot para criar gráficos
import matplotlib.animation as animation  # Importa o matplotlib.animation para criar animações

# Parâmetros físicos do problema
m = 1.0  # Define a massa do objeto em quilogramas (kg)
k = 1.0  # Define a constante elástica da mola (N/m)
gamma = 0.1  # Define o coeficiente de resistência do ar (kg/s)
dt = 0.01  # Define o passo de tempo para a simulação em segundos (s)
t_max = 30  # Define o tempo total de simulação em segundos (s)

# Inicialização dos arrays para armazenar os resultados
```

```
t_values = np.arange(0, t_max, dt)  # Cria um array de tempos, que vai de 0 até t_max, com intervalos de dt
x_values = np.zeros_like(t_values)  # Cria um array para armazenar os deslocamentos em cada instante de tempo
v_values = np.zeros_like(t_values)  # Cria um array para armazenar as velocidades em cada instante de tempo
x = 1.0  # Define o deslocamento inicial (m)
v = 0.0  # Define a velocidade inicial (m/s)

# Método de Euler para resolver a equação de movimento numericamente
for i in range(1, len(t_values)):  # Itera sobre todos os índices dos tempos, começando do segundo (índice 1)
    x_values[i] = x  # Armazena o deslocamento no array x_values
    v_values[i] = v  # Armazena a velocidade no array v_values
    x += v * dt  # Atualiza o deslocamento com a fórmula de Euler
    v += (-k * x / m - gamma * v / m) * dt  # Atualiza a velocidade com a fórmula de Euler

# Configuração da animação
fig, ax = plt.subplots()  # Cria uma figura e um eixo para o gráfico
ax.set_xlim(0, t_max)  # Define o limite do eixo X (tempo), de 0 até t_max
ax.set_ylim(min(x_values) - 1, max(x_values) + 1)  # Define o limite do eixo Y (deslocamento), ajustado para os valores de x
ax.set_xlabel("Tempo (s)")  # Define o rótulo do eixo X como "Tempo (s)"
ax.set_ylabel("Deslocamento (m)")  # Define o rótulo do eixo Y como "Deslocamento (m)"
ax.set_title("Massa-Mola com Resistência do Ar")  # Define o título do gráfico

# Criação de uma linha vazia no gráfico para ser preenchida durante a animação
line, = ax.plot([], [], "bo-", lw=2)  # Cria a linha de pontos azuis conectados por uma linha (bo-), com espessura de linha 2

# Função de inicialização da animação
def init():
    line.set_data([], [])  # Limpa os dados da linha antes de iniciar a animação
    return line,  # Retorna a linha, necessário para a animação

# Função que atualiza o gráfico em cada quadro da animação
def update(frame):
    # Atualiza os dados da linha com os pontos correspondentes ao número do quadro atual
    line.set_data(t_values[:frame], x_values[:frame])  # Define os dados da linha com tempos e deslocamentos até o quadro atual
    return line,  # Retorna a linha, necessário para a animação

# Criação da animação
ani = animation.FuncAnimation(fig, update, frames=len(t_values), init_func=init, blit=True, interval=5)

# Exibe a animação
plt.show()  # Exibe a animação criada
```

Estes programas simulam o movimento do sistema massa-mola com resistência do ar e gera uma animação que mostra o deslocamento da massa em função do tempo. A simulação foi realizada utilizando o método de Euler, e a animação foi criada com a biblioteca `matplotlib.animation`, proporcionando uma visualização clara da dinâmica do sistema.

3 Pêndulo Simples com Resistência do Ar

O pêndulo simples com resistência do ar é uma modificação do modelo clássico de pêndulo, onde a força de resistência do ar é considerada durante o movimento. No modelo tradicional, o pêndulo oscila com uma frequência constante, mas a presença do ar provoca uma força de atrito que reduz gradualmente a amplitude das oscilações. A resistência do ar depende da velocidade do pêndulo e das características do meio, alterando o comportamento do sistema, tornando-o amortecido. Este modelo é importante para descrever sistemas reais, onde a resistência do ar não pode ser negligenciada. A análise desse problema geralmente requer soluções numéricas devido à complexidade das equações envolvidas. Consideramos o movimento de um pêndulo simples de comprimento L e massa m com resistência do ar. O ângulo de oscilação é denotado por $\theta(t)$, e a força de resistência do ar é dada por $-K\frac{d\theta}{dt}$, onde K é o coeficiente de atrito. A equação diferencial para o movimento do pêndulo é dada por:

$$mL\frac{d^2\theta}{dt^2} + K\frac{d\theta}{dt} + mg\sin(\theta) = 0.$$

Aqui:

- m é a massa do pêndulo,
- L é o comprimento do pêndulo,
- K é o coeficiente de resistência do ar,
- g é a aceleração devido à gravidade,
- $\theta(t)$ é o ângulo do pêndulo em função do tempo.

Esta equação é não linear devido ao termo $\sin(\theta)$, mas pode ser resolvida numericamente. A solução analítica para este tipo de equação diferencial não pode ser expressa em termos de funções elementares devido à não linearidade introduzida pelo termo $\sin(\theta)$. No entanto, é possível resolver a equação linearizada (para pequenos ângulos, onde $\sin(\theta) \approx \theta$) para obter uma solução aproximada. Neste caso, a equação se torna:

$$mL\frac{d^2\theta}{dt^2} + K\frac{d\theta}{dt} + mg\theta = 0.$$

Esta é uma equação diferencial linear com solução que depende do discriminante:

$$\Delta = K^2 - 4mLmg.$$

Para $\Delta > 0$, a solução é do tipo amortecida, com raízes reais e distintas. Para $\Delta = 0$, temos uma solução criticamente amortecida, e para $\Delta < 0$, a solução é oscilatória amortecida.

3.1 Método Numérico de Euler

Para resolver numericamente a equação diferencial não linear, podemos usar o método de Euler. A equação para o ângulo e sua velocidade angular são dadas por:

$$\frac{d\theta}{dt} = \omega, \quad \frac{d\omega}{dt} = -\frac{K}{mL}\omega - \frac{g}{L}\sin(\theta).$$

Onde $\omega = \frac{d\theta}{dt}$ é a velocidade angular. O algoritmo de Euler para discretizar essas equações é:

$$\theta_{n+1} = \theta_n + \omega_n \Delta t,$$
$$\omega_{n+1} = \omega_n + \left(-\frac{K}{mL}\omega_n - \frac{g}{L}\sin(\theta_n)\right)\Delta t.$$

3.2 Código em Python

O código Python abaixo resolve numericamente a equação do pêndulo com resistência do ar usando o método de Euler. Ele também gera uma animação do ângulo $\theta(t)$ em função do tempo.

```
import numpy as np  # Importa a biblioteca NumPy para cálculos numéricos
import matplotlib.pyplot as plt  # Importa a biblioteca Matplotlib para gráficos
from matplotlib.animation import FuncAnimation  # Importa o módulo para animações

# Definindo os parâmetros do pêndulo
m = 1.0  # Massa do pêndulo (kg)
L = 1.0  # Comprimento do pêndulo (m)
K = 0.25  # Coeficiente de resistência do ar (proporcional à velocidade angular)
g = 9.81  # Aceleração devido à gravidade (m/s^2)
theta0 = 0.5  # Ângulo inicial (rad)
omega0 = 0.0  # Velocidade angular inicial (rad/s)
dt = 0.01  # Passo de tempo para a simulação (s)
t_max = 30  # Tempo total de simulação (s)

# Inicializando os arrays para armazenar os valores de t,  (ângulo) e  (velocidade angular)
t_values = np.arange(0, t_max, dt)  # Vetor de tempo de 0 até t_max com intervalo dt
theta_values = np.zeros_like(t_values)  # Vetor para armazenar os valores de  (ângulo)
omega_values = np.zeros_like(t_values)  # Vetor para armazenar os valores de  (velocidade angular)
theta_values[0] = theta0  # Inicializa o primeiro valor de
omega_values[0] = omega0  # Inicializa o primeiro valor de

# Método de Euler para resolver numericamente as equações diferenciais
for i in range(1, len(t_values)):  # Loop sobre todos os passos de tempo
    # Atualiza o ângulo  usando o valor anterior de
    theta_values[i] = theta_values[i-1] + omega_values[i-1] * dt
    # Atualiza a velocidade angular  com base na equação de movimento do pêndulo
    omega_values[i] = omega_values[i-1] + (-K / (m * L*) * omega_values[i-1] - (g / L) * np.sin(theta_values[i-1])) * dt

# Criando a animação do movimento do pêndulo
fig, ax = plt.subplots()  # Cria um gráfico
line, = ax.plot([], [], 'o-', lw=2)  # Cria uma linha para o pêndulo (vazia inicialmente)
ax.set_xlim(-L, L)  # Define os limites do eixo X (em função do comprimento L)
ax.set_ylim(-L, L)  # Define os limites do eixo Y (em função do comprimento L)
ax.set_aspect('equal')  # Garante que os eixos X e Y tenham a mesma escala
ax.set_title(r'$\theta(t)$ vs. $t$', fontsize=15)  # Título do gráfico (ângulo vs. tempo)
ax.set_xlabel('x (m)', fontsize=12)  # Rótulo do eixo X (posição horizontal)
ax.set_ylabel('y (m)', fontsize=12)  # Rótulo do eixo Y (posição vertical)

# Função de inicialização para a animação (inicializa a linha vazia)
def init():
    line.set_data([], [])  # Define os dados da linha como vazios inicialmente
    return line,  # Retorna a linha para a animação

# Função de atualização da animação a cada frame
def update(frame):
    # Calcula a posição x e y do pêndulo com base no ângulo
    x = L * np.sin(theta_values[frame])  # Posição horizontal do pêndulo
    y = -L * np.cos(theta_values[frame])  # Posição vertical do pêndulo
    line.set_data([0, x], [0, y])  # Atualiza os dados da linha para a animação
```

```
    return line,  # Retorna a linha atualizada para a animação

# Criando a animação com o método FuncAnimation
ani = FuncAnimation(fig, update, frames=len(t_values), init_func=init, blit=True, interval=dt*1000)

# Exibe a animação
plt.show()
```

A animação gerada pelo código Python mostra o movimento do pêndulo simples com resistência do ar. O ângulo $\theta(t)$ diminui ao longo do tempo devido ao amortecimento, até que o pêndulo se estabiliza na posição de equilíbrio.

4 Simulação da Queda Livre com Efeito Magnus usando o Método de Euler

A queda livre com efeito Magnus refere-se ao movimento de um objeto, como uma esfera, que cai sob a ação da gravidade, enquanto sofre a influência do efeito Magnus. Esse efeito ocorre quando um objeto em rotação interage com o ar, gerando uma força perpendicular à direção do movimento devido à diferença de pressão criada ao redor do objeto. A rotação do objeto altera o fluxo do ar em torno dele, resultando em uma força lateral que pode desviar sua trajetória. No caso da queda livre, a força de Magnus pode causar um desvio horizontal na trajetória do objeto, modificando o comportamento esperado da queda. Esse fenômeno é relevante em esportes como o futebol e o tênis, e também em estudos de aerodinâmica. A modelagem desse efeito adiciona complexidade ao problema, pois envolve a combinação de forças de gravidade, resistência do ar e a força de Magnus. Neste cálculo, vamos simular a trajetória de uma bola em queda livre com o efeito Magnus. A bola parte de uma altura inicial de $y = 20\,\mathrm{m}$ e uma posição inicial $x = 0\,\mathrm{m}$, com velocidade inicial de translação zero, velocidade de rotação inicial diferente de zero , aceleração da gravidade g e a força de Magnus.

4.1 Equações de Movimento

O movimento da bola é governado por equações diferenciais de segunda ordem, levando em consideração as principais forças que atuam sobre ela: a força gravitacional, a força de arrasto e a força de Magnus.

- **Força gravitacional**: A gravidade é uma força constante que atua verticalmente para baixo. A aceleração gerada por essa força é dada por:

$$\mathbf{F}_g = -mg\hat{y}$$

  onde $g = 9.81\,\mathrm{m/s^2}$ é a aceleração gravitacional e $\hat{y}$ é o vetor unitário na direção vertical.

- **Força de arrasto**: Representa a resistência do ar ao movimento da bola. Sua direção é sempre oposta à da velocidade e sua intensidade depende do módulo da velocidade. A força de arrasto é dada por:

$$\mathbf{F}_D = -\frac{1}{2} C_d \cdot \rho \cdot A \cdot \|\mathbf{v}\| \cdot \mathbf{v}$$

  onde:

  - $C_d = 0.47$ é o coeficiente de arrasto para uma esfera,
  - $\rho = 1.225\,\mathrm{kg/m^3}$ é a densidade do ar,
  - $A = \pi r^2$ é a área da seção transversal da bola, com $r = 0.1\,\mathrm{m}$,
  - $\mathbf{v} = (v_x, v_y)$ é o vetor velocidade da bola,
  - $\|\mathbf{v}\| = \sqrt{v_x^2 + v_y^2}$ é o módulo da velocidade.

- **Força de Magnus**: Essa força surge devido à rotação da bola em movimento dentro de um fluido (neste caso, o ar). Ela atua perpendicularmente tanto ao vetor velocidade quanto ao vetor de rotação $\boldsymbol{\omega}$, sendo responsável pelo desvio da trajetória da bola. Sua expressão vetorial é:

$$\mathbf{F}_M = S \cdot \rho \cdot \|\mathbf{v}\| \cdot \boldsymbol{\omega} \times \mathbf{v}$$

  onde:

  - $\boldsymbol{\omega} = (0, 0, \omega_z)$ é o vetor de rotação da bola (assumido constante, na direção z, perpendicular ao plano de movimento),
  - $\omega_z = 1\,\mathrm{rad/s}$,
  - $\mathbf{v} = (v_x, v_y, 0)$ é o vetor velocidade da bola no plano xy,

– S é uma constante de proporcionalidade, frequentemente escrita como $S = c_M \cdot A$, onde c_M é o coeficiente de Magnus,
– O produto vetorial $\boldsymbol{\omega} \times \mathbf{v}$ resulta em uma força no plano xy, de direção perpendicular à velocidade.

Calculando explicitamente o produto vetorial:

$$\boldsymbol{\omega} \times \mathbf{v} = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ 0 & 0 & \omega_z \\ v_x & v_y & 0 \end{vmatrix} = (-\omega_z v_y, \omega_z v_x, 0)$$

Portanto, a força de Magnus tem componentes:

$$\mathbf{F}_M = S \cdot \rho \cdot \|\mathbf{v}\| \cdot (-\omega_z v_y, \omega_z v_x)$$

Assim, as equações diferenciais do movimento da bola nas direções x e y, considerando a massa m da bola, são:

$$\frac{d^2x}{dt^2} = -\frac{1}{2m} C_d \rho A \|\mathbf{v}\| v_x - \frac{S\rho}{m} \|\mathbf{v}\| \omega_z v_y$$

$$\frac{d^2y}{dt^2} = -\frac{1}{2m} C_d \rho A \|\mathbf{v}\| v_y + \frac{S\rho}{m} \|\mathbf{v}\| \omega_z v_x - g$$

Essas equações descrevem a aceleração da bola em função de sua velocidade, levando em conta os efeitos do arrasto e da rotação (força de Magnus), além da gravidade. Elas são não lineares devido à presença de $\|\mathbf{v}\|$ e podem ser resolvidas numericamente para obter a trajetória da bola.

4.2 Método Numérico - Método de Euler

Para resolver as equações diferenciais de segunda ordem, utilizamos o **método de Euler**. Este é um método explícito de integração numérica, simples e fácil de implementar, mas com menor precisão quando comparado a métodos de ordem superior como o de Runge-Kutta.

O método de Euler é aplicado para cada uma das equações diferenciais de 1ª ordem que resultam da transformação das equações de 2ª ordem. Considerando que o sistema de equações diferenciais de 1ª ordem é:

$$\frac{dx}{dt} = v_x, \quad \frac{dy}{dt} = v_y$$

$$\frac{dv_x}{dt} = \frac{F_{Dx}}{m} + \frac{F_{Mx}}{m}, \quad \frac{dv_y}{dt} = \frac{F_{Dy}}{m} + \frac{F_{My}}{m} - g$$

A solução numérica pelo método de Euler consiste em atualizar as variáveis a cada passo de tempo Δt, utilizando as seguintes fórmulas:

$$x(t + \Delta t) = x(t) + v_x(t)\Delta t$$

$$y(t + \Delta t) = y(t) + v_y(t)\Delta t$$

$$v_x(t + \Delta t) = v_x(t) + \frac{F_{Dx}(t)}{m}\Delta t + \frac{F_{Mx}(t)}{m}\Delta t$$

$$v_y(t + \Delta t) = v_y(t) + \left(\frac{F_{Dy}(t)}{m} + \frac{F_{My}(t)}{m} - g \right) \Delta t$$

Esse processo é repetido até que a bola atinja o solo, ou seja, até que $y(t) \leq 0$. A precisão do método de Euler depende do valor de Δt; valores menores de Δt resultam em maior precisão, mas exigem mais cálculos.

4.3 Implementação Numérica

O problema é resolvido numericamente através de um loop de iteração que atualiza as variáveis $x(t)$, $y(t)$, $v_x(t)$ e $v_y(t)$ ao longo do tempo usando o método de Euler. Vamos apresentar dois programas; o primeiro deles apresenta um gráfico de xt. Sem a força de Magnun este gráfico seria uma função constante ; com a força de Magnus este gráfico apresenta distorções. O segundo programa é uma versão mais sofisticada que faz uma animação da queda livre com o efeito Magnus.

Primeiro programa : monitorar a posição horizontal x durante a queda livre

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

# === Constantes físicas do problema ===
g = 9.81           # Gravidade (m/s²)
r = 0.1            # Raio da bola (m)
m = 0.5            # Massa da bola (kg)
rho = 1.225        # Densidade do ar (kg/m³)
Cd = 0.1           # Coeficiente de arrasto (esfera)
S = np.pi * r**2   # Área da seção transversal da bola (m²)
omega_z = 1.0      # Velocidade angular em torno do eixo z (rad/s)

# === Condições iniciais ===
x0 = 0
y0 = 20
vx0 = 0
vy0 = 0

# === Parâmetros da simulação ===
t_final = 2.5
num_steps = 300
dt = t_final / num_steps

# === Inicialização dos vetores ===
x_vals = [x0]
y_vals = [y0]
vx_vals = [vx0]
vy_vals = [vy0]
t_vals = [0]

# === Simulação com método de Euler ===
for step in range(1, num_steps):
    # Velocidades atuais
    vx = vx_vals[-1]
    vy = vy_vals[-1]
    v_mod = np.sqrt(vx**2 + vy**2)

    # === Força de arrasto ===
    F_drag_x = -0.5 * rho * Cd * S * v_mod * vx
    F_drag_y = -0.5 * rho * Cd * S * v_mod * vy

    # === Força de Magnus ===
    # omega_z × v = (0, 0, omega_z) × (vx, vy, 0) = (omega_z * vy, omega_z * vx, 0)
```

```
    F_magnus_x = S * v_mod * (-omega_z * vy)
    F_magnus_y = S * v_mod * ( omega_z * vx)

    # === Força peso ===
    F_grav_y = -m * g

    # === Acelerações ===
    ax = (F_drag_x + F_magnus_x) / m
    ay = (F_drag_y + F_magnus_y + F_grav_y) / m

    # === Atualiza velocidades ===
    vx_new = vx + ax * dt
    vy_new = vy + ay * dt

    # === Atualiza posições ===
    x_new = x_vals[-1] + vx * dt
    y_new = y_vals[-1] + vy * dt

    # Armazena os valores
    vx_vals.append(vx_new)
    vy_vals.append(vy_new)
    x_vals.append(x_new)
    y_vals.append(y_new)
    t_vals.append(t_vals[-1] + dt)

    # Interrompe se tocar o solo
    if y_new <= 0:
        break

# === Gráfico x(t) ===
plt.figure(figsize=(8, 5))
plt.plot(t_vals, x_vals, label='Posição x(t)', color='blue')
plt.xlabel('Tempo (s)')
plt.ylabel('Posição x (m)')
plt.title('Deslocamento Horizontal com Efeito Magnus (Forma Expandida)')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Segundo programa : monitorar a posição horizontal x durante a queda livre entretanto, neste programa usei funções especiais para calcular $\omega \times v$

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

# === Constantes físicas do problema ===
g = 9.81             # Gravidade (m/s²)
r = 0.1              # Raio da bola (m)
m = 0.5              # Massa da bola (kg)
rho = 1.225          # Densidade do ar (kg/m³)
Cd = 0.1             # Coeficiente de arrasto (esfera)
```

```
S = np.pi * r**2     # Área da seção transversal da bola (m²)
omega_z = 1.0        # Velocidade angular (rad/s), no eixo z (efeito Magnus)

# === Condições iniciais ===
x0, y0 = 0, 20        # Posição inicial (m)
vx0, vy0 = 0, 0       # Velocidade inicial (m/s)

# === Parâmetros da simulação ===
t_final = 2.5         # Tempo total de simulação (s)
num_steps = 300       # Número de passos
dt = t_final / num_steps  # Intervalo de tempo (s)

# === Função que calcula a força de Magnus (dividida por m, retorna aceleração) ===
def magnus_accel(v):
    omega = np.array([0, 0, omega_z])           # Vetor rotação no eixo z
    cross = np.cross(omega, np.append(v, 0))    # omega × v
    return S * np.linalg.norm(v) * cross[:2] / m  # Retorna componente x e y

# === Função que calcula aceleração total (drag + Magnus + gravidade) ===
def total_acceleration(v):
    drag = -0.5 * rho * Cd * S * np.linalg.norm(v) * v / m
    magnus = magnus_accel(v)
    gravity = np.array([0, -g])
    return drag + magnus + gravity

# === Inicialização dos vetores para posição, velocidade e tempo ===
x_vals = [x0]
y_vals = [y0]
vx_vals = [vx0]
vy_vals = [vy0]
t_vals = [0]

# === Loop de simulação usando método de Euler ===
for step in range(1, num_steps):
    # Velocidade atual
    v = np.array([vx_vals[-1], vy_vals[-1]])

    # Calcula aceleração
    ax, ay = total_acceleration(v)

    # Atualiza velocidade
    vx_new = vx_vals[-1] + ax * dt
    vy_new = vy_vals[-1] + ay * dt

    # Atualiza posição
    x_new = x_vals[-1] + vx_vals[-1] * dt
    y_new = y_vals[-1] + vy_vals[-1] * dt

    # Armazena os valores
    vx_vals.append(vx_new)
    vy_vals.append(vy_new)
    x_vals.append(x_new)
    y_vals.append(y_new)
```

```
    t_vals.append(t_vals[-1] + dt)

    # Para se a bola atingir o solo
    if y_new <= 0:
        break

# === Plot do resultado: posição x vs tempo ===
plt.figure(figsize=(8, 5))
plt.plot(t_vals, x_vals, label='Posição x(t)', color='blue')
plt.xlabel('Tempo (s)')
plt.ylabel('Posição x (m)')
plt.title('Deslocamento Horizontal com Efeito Magnus')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Terceiro programa : Produz uma animação da queda livre com efeito Magnus no plano $x - y$

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

# Constantes
g = 9.81  # Aceleração da gravidade (m/s^2)
r = 0.1  # Raio da bola (m)
m = 0.5  # Massa da bola (kg)
rho = 1.225  # Densidade do ar (kg/m^3)
Cd = 0.1  # Coeficiente de arrasto para esfera
S = np.pi * r**2  # Área da seção transversal (m^2)

# Condições iniciais
v0 = 0  # Sem velocidade inicial na direção x (começando a cair de repouso)
omega_val = 1  # Velocidade angular (rad/s)

# Velocidade inicial na direção y (simulando uma queda livre)
vx0 = 0  # Velocidade inicial em x
vy0 = 0  # Sem velocidade inicial em y (a bola começa a cair)

# Função de aceleração de Magnus
def magnus_force(velocity, omega, rho, r, S):
    cross_product = np.cross(omega, velocity)
    return S * np.linalg.norm(velocity) * cross_product

# Função para as equações de movimento
def derivatives(t, state):
    x, y, vx, vy = state
    v = np.array([vx, vy])
    omega = np.array([0, 0, omega_val])  # Vetor de rotação no eixo z (rotação da bola)

    # Forças
    drag_force = -0.5 * rho * np.linalg.norm(v) * v * Cd * S / m
```

```
    magnus = magnus_force(v, omega, rho, r, S)

    # Aceleração
    ax = drag_force[0] + magnus[0]
    ay = drag_force[1] + magnus[1] - g  # Aceleração devido à gravidade

    return [vx, vy, ax, ay]

# Estado inicial [x, y, vx, vy]
state0 = [0, 20, vx0, vy0]  # Começando de y = 10 metros, sem movimento inicial

# Tempo de simulação
t_span = (0, 2.07)  # 10 segundos para simular a queda
t_eval = np.linspace(0, 2.07, 200)  # Avaliar a cada ponto no tempo

# Simulação usando o método de Euler
dt = t_eval[1] - t_eval[0]  # Passo de tempo
num_steps = len(t_eval)

# Arrays para armazenar as posições e velocidades
x_vals = np.zeros(num_steps)
y_vals = np.zeros(num_steps)
vx_vals = np.zeros(num_steps)
vy_vals = np.zeros(num_steps)

# Condições iniciais
x_vals[0] = state0[0]
y_vals[0] = state0[1]
vx_vals[0] = state0[2]
vy_vals[0] = state0[3]

# Simulação usando o método de Euler
for i in range(1, num_steps):
    # Calculando as derivadas
    dxdt, dydt, dvxdt, dvydt = derivatives(t_eval[i-1], [x_vals[i-1], y_vals[i-1], vx_vals[i-1], vy_vals[i-1]])

    # Atualizando as variáveis com o método de Euler
    x_vals[i] = x_vals[i-1] + dxdt * dt
    y_vals[i] = y_vals[i-1] + dydt * dt
    vx_vals[i] = vx_vals[i-1] + dvxdt * dt
    vy_vals[i] = vy_vals[i-1] + dvydt * dt

    # Se a bola tocar o solo (y <= 0), parar a animação
    if y_vals[i] <= 0:
        y_vals[i] = 0  # Garantir que a bola pare exatamente em y = 0
        x_vals[i] = x_vals[i-1]  # Garantir que a posição x não mude quando a bola tocar o solo
        break

# Criando o gráfico e a animação
fig, ax = plt.subplots()
ax.set_xlim(-8, 8)  # Limitar o eixo x entre -7 e 7 metros
ax.set_ylim(0, 20)  # Limitar o eixo y entre 0 e 10 metros
ax.set_xlabel('Posição x (m)')
```

```
ax.set_ylabel('Posição y (m)')

line, = ax.plot([], [], 'ro', markersize=8)


# Função de atualização da animação
def update(frame):
    # Verifica se a posição y é maior que zero para continuar a animação
    if y_vals[frame] > 0:
        line.set_data(x_vals[frame], y_vals[frame])
    else:
        # Quando y <= 0, fixa a posição no solo
        line.set_data(x_vals[frame-2], y_vals[frame-2])
    return line,

# Criando a animação, mas só executa enquanto y > 0
#ani = FuncAnimation(fig, update, frames=len(x_vals), interval=20, blit=True)
ani = FuncAnimation(fig, update, frames=range(len(x_vals)), interval=20, blit=True)

# Exibir o vídeo
plt.title('Simulação de Queda Livre com Efeito Magnus')
plt.show()
```

Em linhas gerais a simulação numérica da queda livre da bola com efeito Magnus permite observar como a rotação da bola desvia sua trajetória horizontalmente, fazendo com que a posição final x ao atingir o solo seja diferente de zero. Esse desvio é uma consequência direta do efeito Magnus, que age perpendicularmente à direção do movimento da bola.

5 Simulação da Trajetória de um Projétil com Resistência do Ar

Neste exercício, abordamos a simulação da trajetória de um projétil lançado de forma oblíqua, levando em consideração a resistência do ar. O modelo utilizado para a simulação assume que o projétil é influenciado por dois efeitos principais: a aceleração gravitacional e a resistência do ar. A resistência do ar é modelada como uma força proporcional à velocidade do projétil. O movimento do projétil é descrito pelas seguintes equações diferenciais de segunda ordem:

$$\frac{d^2x}{dt^2} = -cv_x$$
$$\frac{d^2y}{dt^2} = -g - cv_y$$

Onde:
- $x(t)$ e $y(t)$ são as posições do projétil nos eixos horizontal e vertical, respectivamente;
- v_x e v_y são as componentes da velocidade nos eixos x e y;
- $g = 9.81\,\mathrm{m/s^2}$ é a aceleração devido à gravidade;
- c é o coeficiente de resistência do ar, que depende da densidade do ar, da forma do projétil e de outras propriedades;
- $v_x = \frac{dx}{dt}$ e $v_y = \frac{dy}{dt}$ são as velocidades nos eixos x e y.

Para resolver numericamente as equações diferenciais, utilizamos o método de diferenças finitas, discretizando o tempo com um passo Δt. A discretização das equações de movimento é dada por:

$$v_x(t + \Delta t) = v_x(t) + a_x(t)\Delta t$$
$$v_y(t + \Delta t) = v_y(t) + a_y(t)\Delta t$$
$$x(t + \Delta t) = x(t) + v_x(t)\Delta t$$
$$y(t + \Delta t) = y(t) + v_y(t)\Delta t$$

Onde as acelerações a_x e a_y são dadas por:

$$a_x(t) = -cv_x(t)$$
$$a_y(t) = -g - cv_y(t)$$

O algoritmo continua até que a altura $y(t)$ seja zero, o que indica que o projétil atingiu o solo. Agora vamos apresentar dois programas distintos para a simulação do modelo: o primeiro realiza a visualização das trajetórias sem animação, enquanto o segundo inclui uma animação das trajetórias ao longo do tempo. Ambos os programas são implementados em Python, utilizando a biblioteca `matplotlib` para a geração dos gráficos e, no caso do segundo código, para a criação da animação. A simulação é realizada para diferentes valores do coeficiente de resistência c, permitindo observar como esse parâmetro afeta a dinâmica do sistema. O trecho de código a seguir ilustra como a simulação é conduzida e como as trajetórias são visualizadas, com ou sem animação.

Primeiro programa

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

# Parâmetros do problema
g = 9.81  # Gravidade (m/s²)
v0 = 40   # Velocidade inicial (m/s)
theta = 45  # Ângulo em graus
dt = 0.01  # Passo de tempo (s)

# Ângulo em radianos e componentes iniciais da velocidade
theta_rad = np.radians(theta)
vx0 = v0 * np.cos(theta_rad)
vy0 = v0 * np.sin(theta_rad)

# Lista de coeficientes de resistência
coeficientes = [0.0, 0.1, 0.2]

# Criar a figura com 3 subplots empilhados
fig, axs = plt.subplots(3, 1, figsize=(8, 10), sharex=True)

# Laço para calcular e plotar cada caso
for idx in range(3):
    c = coeficientes[idx] # escolhendo o valor de c na lista

    # Listas para posições
    x = [0.0]
    y = [0.0]
    vx = vx0
    vy = vy0

    # Simulação com Euler
    while y[-1] >= 0:
        ax = -c * vx
        ay = -g - c * vy

        vx += ax * dt
        vy += ay * dt

        x.append(x[-1] + vx * dt)
        y.append(y[-1] + vy * dt)

    # Plotar no subplot correspondente
    axs[idx].plot(x, y, color='black')
    axs[idx].set_ylabel("Altura (m)")
    axs[idx].set_title(f"Trajetória com c = {c}")
    axs[idx].grid(True)

# Configuração final do gráfico
axs[2].set_xlabel("Distância horizontal (m)")
plt.tight_layout()
plt.show()
```

Segundo programa contendo animações

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

# Parâmetros globais
g = 9.81  # Aceleração da gravidade (m/s²)
v0 = 40  # Velocidade inicial (m/s)
theta = 45  # Ângulo de lançamento (graus)
dt = 0.01  # Intervalo de tempo (s)

# Função para simular a trajetória
def simulacao(c):
    theta_rad = np.radians(theta)  # Conversão para radianos
    vx, vy = v0 * np.cos(theta_rad), v0 * np.sin(theta_rad)  # Componentes da velocidade
    x, y = [0], [0]  # Posições iniciais

    while y[-1] >= 0:  # Enquanto o projétil não tocar o chão
        ax = -c * vx  # Aceleração no eixo x
        ay = -g - c * vy  # Aceleração no eixo y

        # Atualizar velocidades
        vx += ax * dt
        vy += ay * dt

        # Atualizar posições
        x.append(x[-1] + vx * dt)
        y.append(y[-1] + vy * dt)

    return x, y

# Gerar trajetórias para diferentes valores de c
x1, y1 = simulacao(c=0)       # Sem resistência do ar
x2, y2 = simulacao(c=0.1)    # Resistência do ar moderada
x3, y3 = simulacao(c=0.2)     # Resistência do ar alta

# Encontrar o número máximo de frames
max_frames = max(len(x1), len(x2), len(x3))

# Função para ajustar os tamanhos das listas (para animação sincronizada)
def ajustar_tamanhos(x, y, tamanho):
    while len(x) < tamanho:
        x.append(x[-1])
        y.append(0)
    return x, y

x1, y1 = ajustar_tamanhos(x1, y1, max_frames)
x2, y2 = ajustar_tamanhos(x2, y2, max_frames)
x3, y3 = ajustar_tamanhos(x3, y3, max_frames)

# Configuração do gráfico
```

```
fig, ax = plt.subplots(figsize=(10, 6))
ax.set_xlim(0, max(max(x1), max(x2), max(x3)) + 10)
ax.set_ylim(0, max(max(y1), max(y2), max(y3)) + 10)
ax.set_title("Trajetória de Lançamento Oblíquo com Resistência do Ar", fontsize=14)
ax.set_xlabel("Distância Horizontal (m)", fontsize=12)
ax.set_ylabel("Altura Vertical (m)", fontsize=12)
ax.grid(True)

# Linhas para as três trajetórias
linha1, = ax.plot([], [], 'k-', label="c = 0 (Sem resistência)")
linha2, = ax.plot([], [], 'b-', label="c = 0.1")
linha3, = ax.plot([], [], 'r-', label="c = 0.2")
ax.legend(fontsize=12)

# Função de atualização para a animação
def update(frame):
    linha1.set_data(x1[:frame], y1[:frame])
    linha2.set_data(x2[:frame], y2[:frame])
    linha3.set_data(x3[:frame], y3[:frame])
    return linha1, linha2, linha3

# Criar a animação
ani = FuncAnimation(fig, update, frames=max_frames, interval=5, blit=True)

# Exibir a animação
plt.show()

# (Opcional) Salvar como vídeo
# ani.save('lancamento_3_curvas.mp4', writer='ffmpeg', fps=30)
```

⋆ Explicação do Código com animação

1. Definição de Parâmetros:
- g representa a aceleração devido à gravidade;
- $v0$ é a velocidade inicial do projétil;
- θ é o ângulo de lançamento, convertido para radianos na função de simulação;
- dt é o intervalo de tempo utilizado para discretizar as equações diferenciais.

2. Função de Simulação: A função `simulacao(c)` simula a trajetória de um projétil, levando em consideração a resistência do ar, para um valor específico de c. A cada iteração, as velocidades e posições são atualizadas de acordo com as equações discretizadas.
3. Animação: A função `FuncAnimation` do `matplotlib` é usada para animar as trajetórias do projétil para diferentes valores de c, permitindo visualizar como a resistência do ar influencia a trajetória.

4. Ajuste dos Tamanhos: Para garantir que todas as trajetórias tenham o mesmo número de pontos, a função `ajustar_tamanhos` preenche as listas de coordenadas com valores nulos, de modo a sincronizar a animação.
Este modelo computacional fornece uma visão clara sobre como a resistência do ar afeta o movimento de um projétil lançado de forma oblíqua. O

código implementado é capaz de simular diferentes cenários, variando o coeficiente de resistência do ar, e visualizar as trajetórias resultantes em uma animação. Isso permite uma análise mais intuitiva dos efeitos da resistência do ar no alcance e na altura máxima atingida pelo projétil.

6 Simulação do Movimento de um Corpo em um Plano Inclinado com Resistência do Ar

Neste problema, consideramos um corpo movendo-se ao longo de um plano inclinado com resistência do ar. O movimento é influenciado pela gravidade e pelo arraste do ar, que é modelado como uma força proporcional à velocidade.

6.1 Esquema do Problema

A seguir, apresentamos um diagrama esquemático do problema. O corpo parte da base do plano inclinado com uma velocidade inicial v_0. O eixo x está orientado ao longo do plano, enquanto o eixo y é perpendicular ao plano. A resistência do ar atua contra o movimento do corpo e é proporcional à velocidade.

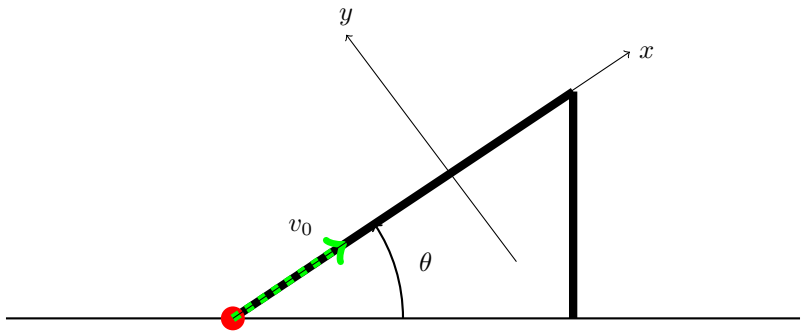

Figure 1: Esquema do movimento do corpo em um plano inclinado com resistência do ar. O corpo parte com velocidade inicial v_0 na base do plano. O eixo x está paralelo ao plano inclinado e o eixo y é perpendicular ao plano. O ângulo θ é o ângulo de inclinação do plano.

6.2 Equações de Movimento

A equação do movimento do corpo é obtida a partir das forças que atuam sobre ele. Agora, a força de resistência do ar é proporcional à velocidade ($F_{\text{resist}} = -Cv$). Portanto, a equação do movimento ao longo do plano inclinado é:

$$m\frac{d^2x}{dt^2} = -mg\sin(\theta) - Cv$$

onde:

- m é a massa do corpo,
- g é a aceleração gravitacional,
- θ é o ângulo de inclinação do plano,

- C é a constante de resistência ao movimento,
- v é a velocidade do corpo.

A equação para a velocidade, que descreve a aceleração do corpo, é dada por:

$$\frac{dv}{dt} = -g\sin(\theta) - \frac{C}{m}v$$

onde a resistência do ar é proporcional à velocidade do corpo. A segunda lei de Newton aplicada à direção do movimento pode ser simplificada dessa forma.

6.3 Simulação Numérica

Para resolver essa equação numericamente com o método de Euler, precisamos discretizar o tempo. O método de Euler para uma equação diferencial $\frac{dv}{dt}$ é dado por:

$$v_{n+1} = v_n + \Delta t \cdot \frac{dv}{dt}$$

Substituímos a expressão de $\frac{dv}{dt}$ que obtivemos anteriormente:

$$v_{n+1} = v_n + \Delta t \left(-g\sin(\theta) - \frac{C}{m}v_n\right)$$

Este é o passo de atualização da velocidade no método de Euler, onde: - v_n é a velocidade no passo n, - Δt é o passo de tempo, - v_{n+1} é a velocidade no próximo passo de tempo.

Essa equação nos dá a velocidade do corpo em cada passo de tempo, levando em consideração tanto a gravidade quanto a resistência do ar proporcional à velocidade. O código apresentado realiza uma simulação numérica do movimento de um corpo sujeito à gravidade e à resistência do ar, utilizando o método de Euler. O objetivo é investigar como diferentes valores do coeficiente de arrasto, C_d, afetam a altura máxima atingida pelo corpo.
O código começa com a definição dos parâmetros do sistema:
- $m = 1.0\,\text{kg}$: a massa do corpo,
- $g = 9.81\,\text{m/s}^2$: a aceleração gravitacional,
- $\theta = 30°$: o ângulo de inclinação do plano,
- $v_0 = 10.0\,\text{m/s}$: a velocidade inicial,
- $t_{\max} = 30.0\,\text{s}$: o tempo máximo de simulação,
- $\Delta t = 0.01\,\text{s}$: o passo de tempo.

O ângulo de inclinação θ é convertido de graus para radianos utilizando a função `np.radians`.

A função `simulate(C_d)` é responsável por rodar a simulação para um valor específico de C_d. Dentro dessa função:
1. As variáveis v (velocidade), t (tempo) e h (altura) são inicializadas com os valores iniciais.
2. As listas `velocities`, `heights` e `times` são criadas para armazenar os resultados ao longo da simulação.
3. Um loop `while` é usado para realizar a simulação enquanto a ditância x_h ao longo do plano inclinado for maior ou igual a zero e o tempo t for menor ou igual ao tempo máximo $t_{\max}$.
4. Dentro do loop, a aceleração a é calculada com base na gravidade e na resistência do ar.
5. A velocidade v e a distância x_h são atualizadas utilizando o método de Euler, com o passo de tempo Δt:

$$v = v + a\Delta t$$
$$x_h = x_h + v\Delta t$$

6. O tempo t é incrementado por Δt, e os resultados são armazenados nas listas correspondentes.
7. A simulação é interrompida quando a velocidade v se torna negativa ou zero, indicando que o corpo parou de subir.

O código testa diferentes valores para o coeficiente de arrasto C_d e plota a distância x_h em função do tempo t para cada valor de C_d. Os valores testados são: $C_d = 0.01, 0.1, 1.0, 2.0$. O gráfico resultante mostra como a altura máxima atingida pelo corpo diminui à medida que o coeficiente de resistência do ar aumenta. Com um maior C_d, a resistência do ar se torna mais significativa, impedindo o corpo de alcançar grandes alturas. Adicionalmente, o código também imprime a altura máxima atingida para cada valor de C_d.

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

# Definição dos parâmetros
m = 1.0  # massa do corpo em kg
g = 9.81  # aceleração gravitacional em m/s^2
theta = 30.0  # ângulo de inclinação em graus
v_0 = 10.0  # velocidade inicial em m/s
t_max = 30.0  # tempo máximo de simulação em segundos
dt = 0.01  # passo de tempo em segundos

# Conversão do ângulo de graus para radianos
theta = np.radians(theta)

# Função para rodar a simulação para um dado C_d
def simulate(C_d):
    # Inicialização das variáveis
    v = v_0  # velocidade inicial
    t = 0.0  # tempo inicial
    xh = 0.0  # distância  inicial
    velocities = [v]  # lista para armazenar as velocidades
    xheights = [xh]  # lista para armazenar as distâncias
    times = [t]  # lista para armazenar os tempos

    # Loop de simulação (método de Euler)
```

```
    while xh >= 0 and t <= t_max:
        # Aceleração devido à gravidade e ao arrasto do ar
        a = -g * np.sin(theta) - C_d * v

        # Atualização da velocidade e da distância
        v = v + a * dt
        xh = xh + v * dt
        t += dt

        # Armazenamento dos resultados
        velocities.append(v)
        xheights.append(xh)  # Usando xheights aqui
        times.append(t)

        # Condição de parada: quando a velocidade for zero ou negativa (o corpo parar de subir)
        if v <= 0:
            break

    return times, velocities, xheights  # Retornando xheights

# Valores de C_d a serem testados
C_d_values = [0.01, 0.1, 1.0, 2.]

# Plotagem dos resultados
plt.figure(figsize=(10, 6))

for C_d in C_d_values:
    times, velocities, xheights = simulate(C_d)
    plt.plot(times, xheights, label=f'C_d = {C_d}')

plt.xlabel("Tempo (s)")
plt.ylabel("Distância (m)")
plt.legend()
plt.title("Comparação da Distância Máxima com Diferentes Coeficientes de Arrasto (C_d)")
plt.tight_layout()
plt.show()

# Mostrar altura máxima para cada valor de C_d
for C_d in C_d_values:
    _, _, xheights = simulate(C_d)
    print(f"Distância  máxima com C_d = {C_d}: {xheights[-1]:.2f} metros")
```

A simulação permite que visualizemos como a resistência do ar afeta o movimento do corpo. A distância máxima atingida depende diretamente do valor da constante de resistência C. Se o valor de C for maior, a resistência será mais significativa, diminuindo a velocidade do corpo mais rapidamente e reduzindo a distância que o corpo percorre ao longo do plano inclinado.

7 Vibração de uma corda com extremidades fixas

A vibração de uma corda de violão presa em ambas as extremidades pode ser modelada pela equação de onda unidimensional:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2 \frac{\partial^2 u(x,t)}{\partial x^2},$$

onde $u(x,t)$ representa o deslocamento transversal da corda no ponto x, no instante t, e c é a velocidade de propagação da onda ao longo da corda, determinada pelas propriedades físicas da mesma (como tensão e densidade linear de massa).

7.1 Condições de contorno e iniciais

Para simular a corda com extremidades fixas, impomos as condições de contorno:

$$u(0,t) = u(L,t) = 0,$$

onde L é o comprimento total da corda. Como condição inicial, consideramos que a corda está inicialmente deformada em uma forma de pulso (por exemplo, um triângulo) e em repouso:

$$u(x,0) = f(x), \quad \frac{\partial u}{\partial t}(x,0) = 0.$$

7.2 Esquema de diferenças finitas

Discretizando o espaço com passo Δx e o tempo com passo Δt, usamos o seguinte esquema explícito para resolver numericamente a equação de onda:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + r^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right),$$

onde $r = \frac{c\Delta t}{\Delta x}$. Esse esquema é estável se $r \leq 1$ (condição de Courant-Friedrichs-Lewy). A seguir, apresentamos um programa em Python que simula esse fenômeno e visualiza a propagação do pulso ao longo do tempo. O pulso inicial é triangular, localizado no centro da corda.

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

# Parâmetros físicos e numéricos
L = 1.0         # comprimento da corda
c = 1.0         # velocidade da onda
nx = 100        # número de pontos espaciais
dx = L / (nx - 1)
dt = 0.005      # passo temporal
nt = 600        # número de passos no tempo
r = c * dt / dx # parâmetro CFL
```

```
# Estabilidade
assert r <= 1.0, "Condição de estabilidade violada: r deve ser <= 1"

# Inicialização
u = np.zeros((nt, nx))

# Condição inicial: pulso triangular no centro
for i in range(nx):
    x = i * dx
    if 0.4 <= x <= 0.6:
        u[0, i] = 1 - abs(x - 0.5) * 10  # pico no centro
    else:
        u[0, i] = 0

# Primeira iteração usando condição de repouso
u[1, 1:-1] = u[0, 1:-1] + 0.5 * r**2 * (u[0, 2:] - 2*u[0, 1:-1] + u[0, :-2])

# Loop no tempo
for n in range(1, nt-1):
    u[n+1, 1:-1] = 2*u[n, 1:-1] - u[n-1, 1:-1] + \
                   r**2 * (u[n, 2:] - 2*u[n, 1:-1] + u[n, :-2])

# Visualização com animação
fig, ax = plt.subplots()
line, = ax.plot(np.linspace(0, L, nx), u[0])
ax.set_ylim(-1.2, 1.2)
ax.set_xlabel('x')
ax.set_ylabel('u(x,t)')
ax.set_title('Vibração de uma corda com extremidades fixas')

def animate(n):
    line.set_ydata(u[n])
    return line,

ani = animation.FuncAnimation(fig, animate, frames=nt, interval=20)
plt.show()
```

A simulação mostra claramente a propagação do pulso inicial em ambas as direções, seguido pela reflexão nas extremidades da corda. O padrão de interferência que emerge é consequência da superposição das ondas refletidas. Esse comportamento é típico de ondas em sistemas confinados com condições de contorno fixas, como em instrumentos musicais. O modelo numérico é simples, mas captura com precisão a essência da propagação ondulatória, oferecendo uma ferramenta poderosa para estudo e visualização de fenômenos físicos reais.

8 Modelo SIR: Dinâmica de Epidemias

O modelo **SIR** é um modelo matemático utilizado para descrever a propagação de doenças infecciosas em uma população. Ele divide a população em três grupos:

- **Susceptíveis (S)**: Indivíduos que ainda não foram infectados, mas estão suscetíveis à infecção.
- **Infectados (I)**: Indivíduos que estão infectados e podem transmitir a doença.
- **Recuperados (R)**: Indivíduos que foram infectados, mas já se recuperaram e não podem mais ser infectados nem transmitir a doença.

As variáveis $S(t)$, $I(t)$, e $R(t)$ representam a fração de indivíduos em cada uma dessas classes no tempo t.

8.1 Equações do Modelo SIR

As equações diferenciais que governam o modelo SIR são dadas por:

$$\frac{dS}{dt} = -\beta \cdot S \cdot I \tag{11}$$

$$\frac{dI}{dt} = \beta \cdot S \cdot I - \gamma \cdot I \tag{12}$$

$$\frac{dR}{dt} = \gamma \cdot I \tag{13}$$

Onde:

- β é a taxa de transmissão, que determina a probabilidade de uma pessoa suscetível ser infectada por uma pessoa infectada.
- γ é a taxa de recuperação, que indica a proporção de infectados que se recuperam a cada unidade de tempo.

A taxa de variação de $S(t)$ é negativa porque, à medida que os indivíduos se infectam, o número de suscetíveis diminui. Por outro lado, o número de infectados $I(t)$ aumenta à medida que mais indivíduos se infectam, mas diminui à medida que os infectados se recuperam.

8.2 Método de Euler para o Modelo SIR

O modelo SIR consiste em três equações diferenciais que descrevem a variação de suscetíveis, infectados e recuperados ao longo do tempo. Como essas equações não possuem uma solução analítica simples, podemos resolvê-las numericamente utilizando o *método de Euler* para resolver equações diferenciais de primeira ordem. Para aplicar o método de Euler, substituímos as derivadas por diferenças finitas para cada uma das variáveis:
1. Para $S(t)$:

$$S(t + \Delta t) = S(t) - \beta \cdot S(t) \cdot I(t) \cdot \Delta t$$

2. Para $I(t)$:

$$I(t + \Delta t) = I(t) + (\beta \cdot S(t) \cdot I(t) - \gamma \cdot I(t)) \cdot \Delta t$$

3. Para $R(t)$:

$$R(t + \Delta t) = R(t) + \gamma \cdot I(t) \cdot \Delta t$$

Essas equações são resolvidas iterativamente, começando com as condições iniciais $S(0)$, $I(0)$ e $R(0)$. Para cada passo de tempo Δt, calculamos os novos valores de S, I e R, com base nos valores anteriores, e repetimos até o tempo final. Esse processo é a base da implementação numérica do modelo SIR usando o método de Euler, que permite simular a evolução da epidemia ao longo do tempo.

8.3 Simulação e Animação do Modelo SIR

A seguir, apresentamos dois códigos em Python que simulam a dinâmica do modelo SIR utilizando o método de Euler. O primeiro programa gera um gráfico estático com a evolução temporal das três populações — suscetíveis, infectados e recuperados. O segundo programa, por sua vez, inclui uma animação que ilustra de forma dinâmica a propagação da epidemia ao longo do tempo. Ambos os programas permitem observar o comportamento do sistema epidemiológico a partir das mesmas condições iniciais e parâmetros de transmissão e recuperação.

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

# Parâmetros do modelo SIR
beta = 0.3  # Taxa de transmissão
gamma = 0.1  # Taxa de recuperação

# Condições iniciais
S0 = 0.99  # Fração inicial de suscetíveis
I0 = 0.01  # Fração inicial de infectados
R0 = 0.0   # Fração inicial de recuperados
t_max = 160  # Número de passos de tempo

# Passo de tempo para o método de Euler
dt = 1

# Arrays para armazenar os valores de S, I e R
S_vals = np.zeros(t_max)
I_vals = np.zeros(t_max)
R_vals = np.zeros(t_max)

# Condições iniciais
S_vals[0] = S0
I_vals[0] = I0
R_vals[0] = R0

# Função de atualização das variáveis S, I e R usando o método de Euler
for t in range(1, t_max):
    dS = -beta * S_vals[t-1] * I_vals[t-1] * dt
```

```
    dI = (beta * S_vals[t-1] * I_vals[t-1] - gamma * I_vals[t-1]) * dt
    dR = gamma * I_vals[t-1] * dt

    S_vals[t] = S_vals[t-1] + dS
    I_vals[t] = I_vals[t-1] + dI
    R_vals[t] = R_vals[t-1] + dR

# Criando o gráfico das curvas S, I e R
tempo = np.arange(t_max)
plt.plot(tempo, S_vals, label="Suscetíveis", color="blue")
plt.plot(tempo, I_vals, label="Infectados", color="red")
plt.plot(tempo, R_vals, label="Recuperados", color="green")

# Ajustes do gráfico
plt.xlabel("Tempo")
plt.ylabel("Proporção da população")
plt.title("Modelo SIR - Simulação de Epidemia")
plt.ylim(0, 1)
plt.legend()
plt.grid(True)
plt.show()

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

# Parâmetros do modelo SIR
beta = 0.3  # Taxa de transmissão
gamma = 0.1  # Taxa de recuperação

# Condições iniciais
S0 = 0.99  # Fração inicial de suscetíveis
I0 = 0.01  # Fração inicial de infectados
R0 = 0.0   # Fração inicial de recuperados
t_max = 160  # Número de passos de tempo

# Passo de tempo para o método de Euler
dt = 1

# Arrays para armazenar os valores de S, I e R
S_vals = np.zeros(t_max)
I_vals = np.zeros(t_max)
R_vals = np.zeros(t_max)

# Condições iniciais
S_vals[0] = S0
I_vals[0] = I0
R_vals[0] = R0

# Função de atualização das variáveis S, I e R usando o método de Euler
for t in range(1, t_max):
    dS = -beta * S_vals[t-1] * I_vals[t-1] * dt
    dI = (beta * S_vals[t-1] * I_vals[t-1] - gamma * I_vals[t-1]) * dt
    dR = gamma * I_vals[t-1] * dt
```

```
    S_vals[t] = S_vals[t-1] + dS
    I_vals[t] = I_vals[t-1] + dI
    R_vals[t] = R_vals[t-1] + dR

# Criando o gráfico e a animação
fig, ax = plt.subplots()
ax.set_xlim(0, t_max)
ax.set_ylim(0, 1)
ax.set_xlabel('Tempo')
ax.set_ylabel('Proporção da população')
line_S, = ax.plot([], [], label="Suscetíveis", color="blue")
line_I, = ax.plot([], [], label="Infectados", color="red")
line_R, = ax.plot([], [], label="Recuperados", color="green")
ax.legend()

# Função de atualização da animação
def update(frame):
    line_S.set_data(np.arange(frame), S_vals[:frame])
    line_I.set_data(np.arange(frame), I_vals[:frame])
    line_R.set_data(np.arange(frame), R_vals[:frame])
    return line_S, line_I, line_R

# Criando a animação
ani = FuncAnimation(fig, update, frames=t_max, interval=50, blit=True)

# Exibir a animação
plt.title('Modelo SIR - Simulação de Epidemia')
plt.show()
```

Durante a pandemia de COVID-19, o modelo SIR foi amplamente utilizado para entender e prever a dinâmica da doença. As soluções numéricas dessas equações permitiram simular diferentes cenários de propagação da infecção, ajudando autoridades de saúde pública a implementar políticas de mitigação, como o distanciamento social e o uso de máscaras, bem como a prever os impactos de diferentes estratégias de vacinação. Embora o modelo SIR seja uma simplificação, ele fornece uma base fundamental para a análise da propagação de epidemias e serve como ponto de partida para modelos mais complexos que incluem fatores adicionais, como a distribuição espacial da doença, a heterogeneidade na população ou a introdução de novas variantes do patógeno. Em tempos de pandemia, o modelo SIR, junto com outras abordagens numéricas, desempenha um papel crucial em ajudar a orientar decisões em tempo real e a minimizar o impacto de surtos de doenças infecciosas.

9 O Mapa Logístico e o Comportamento Caótico

O mapa logístico é um modelo matemático que descreve o crescimento populacional de uma espécie com recursos limitados, sendo representado pela equação iterativa:

$$x_{n+1} = rx_n(1 - x_n),$$

onde x_n é a população na geração n e r é o parâmetro de taxa de crescimento. O comportamento deste sistema varia significativamente com o valor de r. Para valores pequenos de r, a população converge para um valor fixo. No entanto, à medida que r aumenta, o comportamento do sistema torna-se mais complexo, exibindo bifurcações periódicas e, em certos intervalos de r, o sistema entra em um regime caótico, caracterizado por uma alta sensibilidade às condições iniciais e a ausência de padrões previsíveis no longo prazo. O comportamento caótico é um fenômeno no qual pequenas alterações nas condições iniciais de um sistema podem levar a grandes variações nos resultados, tornando o sistema altamente imprevisível. Esse comportamento é uma das características dos sistemas não lineares dinâmicos, como o mapa logístico, e é amplamente estudado na teoria do caos. O mapa logístico é útil em várias áreas, incluindo a biologia, para modelar o crescimento de populações, a física, para estudar sistemas dinâmicos, e na economia, para modelar flutuações econômicas. Além disso, o mapa logístico também serve como uma ferramenta educativa para ilustrar conceitos de caos e dinâmica não linear.

9.1 Diagrama de Bifurcação

O diagrama de bifurcação do mapa logístico mostra os valores assintóticos de x_n para diferentes valores de r. À medida que r aumenta, observa-se a transição do comportamento ordenado para o caos.

9.2 Código em Python

O programa gera uma animação progressiva do diagrama de bifurcação do mapa logístico. Para cada valor do parâmetro r, iteramos a equação $x_{n+1} = rx_n(1 - x_n)$ até atingir um regime estacionário. Apenas os últimos 200 valores de cada r são armazenados para exibir as bifurcações corretamente. Durante a animação, os pontos são acumulados no gráfico, revelando gradualmente a transição do comportamento regular para o caos. Isso permite visualizar como pequenas mudanças em r levam a transições periódicas e, eventualmente, ao comportamento caótico.

```
import numpy as np  # Biblioteca para cálculos numéricos
import matplotlib.pyplot as plt  # Biblioteca para gráficos
import matplotlib.animation as animation  # Biblioteca para animação

# Função do mapa logístico
def logistic_map(x, r):
    """
    Calcula a próxima iteração do mapa logístico.

    Parâmetros:
    x (float) -> Valor atual de x
    r (float) -> Parâmetro de crescimento

    Retorna:
```

```
    float -> Próximo valor de x
    """
    return r * x * (1 - x)

# Configuração do espaço de parâmetros
r_values = np.linspace(2.5, 4.0, 400)  # Pegamos 400 valores de r para suavizar a animação
iterations = 1000  # Número total de iterações
last = 200  # Pegamos os últimos 200 valores de x para visualizar melhor

# Criamos a figura e os eixos do gráfico
fig, ax = plt.subplots(figsize=(10, 6))
ax.set_xlim(2.5, 4.0)  # Intervalo do eixo x (valores de r)
ax.set_ylim(0, 1)  # Intervalo do eixo y (valores de x)
ax.set_xlabel("r", fontsize=12)  # Nome do eixo x
ax.set_ylabel("x", fontsize=12)  # Nome do eixo y
ax.set_title("Construção Progressiva do Diagrama de Bifurcação", fontsize=14)  # Título

# Lista para armazenar os pontos ao longo da animação
all_points = []

# Criamos a linha que será animada
sc = ax.scatter([], [], s=0.1, color='black')  # Criamos um scatter vazio para atualizar

# Função de inicialização
def init():
    """
    Inicializa a animação com um gráfico vazio.
    """
    sc.set_offsets([])  # Sem pontos no início
    return (sc,)

# Função de atualização da animação
def update(frame):
    """
    Adiciona progressivamente pontos ao diagrama de bifurcação.

    Parâmetros:
    frame (int) -> Número do quadro da animação

    Retorna:
    tuple -> Gráfico atualizado
    """
    r = r_values[frame]  # Pegamos o valor atual de r
    x = 0.5  # Iniciamos com x_0 = 0.5

    # Iteramos o mapa logístico para estabilizar a órbita
    for i in range(iterations):
        x = logistic_map(x, r)
        if i >= (iterations - last):  # Apenas últimos valores são armazenados
            all_points.append([r, x])  # Acumulamos os pontos

    # Atualizamos os pontos no scatter plot
    sc.set_offsets(np.array(all_points))
```

```
    return (sc,)

# Criamos a animação
ani = animation.FuncAnimation(fig, update, frames=len(r_values), init_func=init, blit=False, interval=10)

plt.show()  # Exibe a animação
```

O mapa logístico é um exemplo clássico de como sistemas dinâmicos simples podem gerar comportamento complexo. Seu estudo fornece insights fundamentais sobre o caos determinístico e tem aplicações em diversas áreas da ciência.

10 O Mapa de Lozi: Teoria, Aplicações e Implementação Computacional

O Mapa de Lozi é um sistema dinâmico discreto que apresenta comportamento caótico, sendo uma versão linearizada do Mapa de Hénon. Ele é definido pelas equações:

$$x_{n+1} = 1 - a|x_n| + y_n, \tag{14}$$

$$y_{n+1} = bx_n. \tag{15}$$

Os parâmetros a e b controlam a dinâmica do sistema, e para certos valores, observa-se a presença de um atrator estranho.

10.1 Aplicações

O Mapa de Lozi tem diversas aplicações, incluindo:

- **Sistemas Dinâmicos e Caos**: Estudo de sensibilidade a condições iniciais e atratores estranhos.
- **Física e Processos Estocásticos**: Modelagem de sistemas físicos caóticos, como turbulência e plasmas.
- **Processamento de Sinais e Criptografia**: Uso na geração de números pseudoaleatórios e segurança da informação.
- **Engenharia Eletrônica e Controle**: Aplicações em circuitos osciladores caóticos e sistemas robóticos.
- **Neurociência e Redes Neurais**: Modelagem de padrões de ativação e aprendizado em redes neurais.

10.2 Implementação Computacional

A seguir, apresentamos um código em Python que gera um diagrama de bifurcação tridimensional para o Mapa de Lozi, com visualização animada:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from mpl_toolkits.mplot3d import Axes3D

# Parâmetros otimizados
a_min, a_max = 1.2, 1.8  # Intervalo de a
b = 0.5  # Mantemos b fixo
num_a_values = 100  # Reduzindo valores de a para acelerar
num_iter = 500  # Reduzindo número de iterações
transient = 300  # Iterações descartadas

# Valores de a para o diagrama
a_values = np.linspace(a_min, a_max, num_a_values)

# Configuração da figura
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.set_xlabel("a")
```

```
ax.set_ylabel("x")
ax.set_zlabel("y")
ax.set_title("Diagrama de Bifurcação 3D do Mapa de Lozi")
ax.set_xlim(a_min, a_max)
ax.set_ylim(-1.5, 1.5)
ax.set_zlim(-1.5, 1.5)

# Função do Mapa de Lozi
def lozi_map(x, y, a, b):
    x_new = 1 - a * np.abs(x) + y
    y_new = b * x
    return x_new, y_new

# Função para gerar os pontos do diagrama
def generate_bifurcation(a):
    x, y = 0.1, 0.1  # Condição inicial

    for _ in range(transient):  # Descartamos transiente
        x, y = lozi_map(x, y, a, b)

    points = np.zeros((num_iter, 2))
    for i in range(num_iter):
        x, y = lozi_map(x, y, a, b)
        points[i] = [x, y]
    return points

# Função para animação
def update(frame):
    a = a_values[frame]
    points = generate_bifurcation(a)
    x_vals, y_vals = points[:, 0], points[:, 1]
    ax.scatter([a] * len(x_vals), x_vals, y_vals, color='black', s=0.1)

    # Rotação da câmera
    ax.view_init(elev=20, azim=frame * 2)  # Ajuste de ângulos
    return ax

ani = animation.FuncAnimation(fig, update, frames=len(a_values), interval=30, blit=False)
ani.save("lozi_bifurcation.mp4", writer="ffmpeg", fps=20)

plt.show()
```

O Mapa de Lozi é um exemplo fascinante de dinâmica caótica, apresentando bifurcações complexas e atratores estranhos. Seu estudo é fundamental para diversas áreas da ciência e engenharia, sendo uma ferramenta poderosa para a modelagem de sistemas não-lineares.

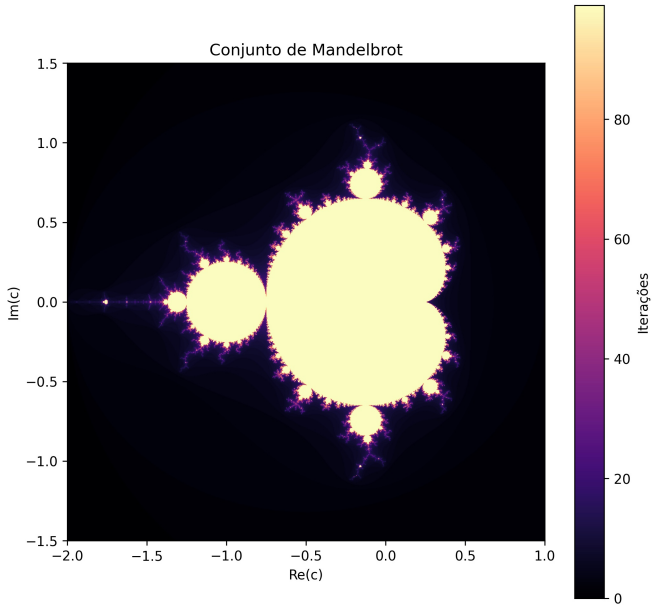

Figure 2: Conjunto de Mandelbrot gerado pelo programa.

11 Introdução ao Conjunto de Mandelbrot

O conjunto de Mandelbrot é um dos objetos mais famosos da matemática fractal. Ele é definido pela iteração da função quadrática complexa:

$$z_{n+1} = z_n^2 + c, \tag{16}$$

onde z e c são números complexos, e começamos com $z_0 = 0$. Um ponto c pertence ao conjunto de Mandelbrot se a sequência $\{z_n\}$ não divergir para o infinito. Caso contrário, consideramos que c não pertence ao conjunto. Esse conjunto é importante na matemática e na física porque ilustra conceitos como dinâmica não linear, caos e estruturas auto-semelhantes. Ele é também um exemplo clássico de como sistemas simples podem produzir padrões extremamente complexos.

11.1 Implementação Computacional

Abaixo, apresentamos um programa em Python para gerar e visualizar o conjunto de Mandelbrot. O código está amplamente comentado para explicar cada etapa do processo.

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

```
# Definição do tamanho da imagem (resolução em pixels)
width, height = 1000, 1000  # Resolução da imagem
max_iter = 100  # Número máximo de iterações por ponto

# Definição da área do plano complexo que será mapeada
xmin, xmax = -2.0, 1.0
ymin, ymax = -1.5, 1.5

# Criar a grade de pontos complexos
x = np.linspace(xmin, xmax, width)  # Discretiza o eixo real
y = np.linspace(ymin, ymax, height) # Discretiza o eixo imaginário
X, Y = np.meshgrid(x, y)  # Cria uma grade 2D
C = X + 1j * Y  # Converte para números complexos

# Inicializa a matriz de escape
Z = np.zeros(C.shape, dtype=complex)  # Matriz de valores complexos
mandelbrot_set = np.full(C.shape, max_iter)  # Matriz para armazenar as iterações

# Iteração sobre os pontos da grade
for i in range(max_iter):
    mask = np.abs(Z) < 2  # Identifica pontos que ainda não divergiram
    Z[mask] = Z[mask]**2 + C[mask]  # Atualiza os valores de Z
    mandelbrot_set[mask] = i  # Armazena o número de iterações até a divergência

# Criar a figura e exibir o conjunto de Mandelbrot
plt.figure(figsize=(8, 8))  # Define o tamanho do gráfico
plt.imshow(mandelbrot_set, extent=[xmin, xmax, ymin, ymax], cmap="magma")
plt.colorbar(label="Iterações")  # Barra de cores indicando iterações
plt.title("Conjunto de Mandelbrot")
plt.xlabel("Re(c)")  # Eixo x representando a parte real de c
plt.ylabel("Im(c)")  # Eixo y representando a parte imaginária de c

# Salvar a imagem gerada como 'mandel.jpg'
plt.savefig("mandel.jpg", dpi=300, bbox_inches='tight')

# Mostrar a imagem na tela
plt.show()
```

O conjunto de Mandelbrot demonstra a riqueza das estruturas fractais, revelando padrões belos e complexos a partir de uma simples equação iterativa. Sua implementação computacional permite a exploração visual dessas estruturas e sua aplicação em áreas como dinâmica não linear e teoria do caos.

12 Integração de Monte Carlo

A integração de Monte Carlo é uma técnica numérica utilizada para calcular integrais através de amostragem aleatória. Esse método é especialmente útil para integrais de alta dimensão, onde os métodos tradicionais, como a quadratura numérica, se tornam impraticáveis. A ideia central da integração de Monte Carlo é estimar a integral de uma função através da média dos valores da função em pontos escolhidos aleatoriamente dentro do domínio de integração.

12.1 Números Aleatórios

O método de Monte Carlo depende da geração de números aleatórios. Um número aleatório é um valor gerado de forma imprevisível, seguindo uma distribuição de probabilidade específica. Em computação, usamos *geradores de números pseudoaleatórios*, que produzem sequências de números que parecem aleatórias, mas são determinadas por um algoritmo.

Em Python, podemos gerar números aleatórios uniformemente distribuídos no intervalo $[a, b]$ usando a função `random.uniform(a, b)` do módulo `random`. Em nossa aplicação, utilizaremos esses números para amostrar pontos dentro do intervalo de integração.

12.2 Princípio da Integração de Monte Carlo

Se quisermos calcular a integral de uma função $f(x)$ em um intervalo $[a, b]$, ou seja,

$$I = \int_a^b f(x)\,dx$$

podemos aproximá-la usando Monte Carlo da seguinte maneira:

1. Geramos N números aleatórios x_i uniformemente distribuídos em $[a, b]$. 2. Calculamos o valor da função $f(x_i)$ nesses pontos. 3. Estimamos a integral pela média dos valores da função, multiplicada pelo comprimento do intervalo:

$$I \approx (b-a)\frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

12.3 Exemplo: Integração de $f(x) = x^2$ em $[0, 2]$

Vamos calcular numericamente a seguinte integral:

$$I = \int_0^2 x^2\,dx$$

Usando a fórmula de Monte Carlo, temos:

$$I \approx 2 \times \frac{1}{N} \sum_{i=1}^{N} x_i^2$$

onde x_i são N valores aleatórios uniformemente distribuídos em $[0, 2]$.

12.4 Implementação em Python

A seguir, apresentamos um programa em Python que implementa esse método.

```
# -*- coding: utf-8 -*-
import random  # Importa o módulo random para geração de números pseudoaleatórios

# Definição da função que queremos integrar
def f(x):
    return x**2  # Retorna o valor de x ao quadrado

# Definição dos parâmetros da integração
a, b = 0, 2  # Limites inferior e superior da integral
N = 10000    # Número de pontos amostrados (quanto maior, mais precisa a estimativa)

# Inicializa a variável que acumulará a soma dos valores da função
soma = 0

# Loop que gera N pontos aleatórios e soma os valores da função nesses pontos
for _ in range(N):
    x = random.uniform(a, b)  # Gera um número aleatório uniformemente distribuído entre a e b
    soma += f(x)  # Adiciona o valor de f(x) à soma total

# Estimativa da integral utilizando o método de Monte Carlo
# A integral é estimada pela média dos valores da função vezes o tamanho do intervalo
integral = (b - a) * (soma / N)

# Exibição do resultado da estimativa
print(f"Estimativa de I =  x^2 dx de {a} a {b}: {integral}")

# Explicação do cálculo:
# A integral  f(x) dx de a até b pode ser aproximada por:
#   I  (b - a) * (1/N) *  f(x_i), onde x_i são amostras aleatórias uniformes em [a, b]
# Aqui, estamos calculando a média dos valores f(x) e multiplicando pelo comprimento do intervalo.
```

12.5 Comparação com o Valor Exato

Podemos calcular a integral analiticamente:

$$I = \int_0^2 x^2\,dx = \left[\frac{x^3}{3}\right]_0^2 = \frac{8}{3} \approx 2.6667$$

Com um número suficientemente grande de amostras N, o valor obtido pelo método de Monte Carlo se aproxima do valor exato. Isso ilustra a eficiência da técnica para estimar integrais de forma probabilística. A integração de Monte Carlo é uma técnica poderosa, especialmente útil quando lidamos com dimensões elevadas ou domínios de integração complexos. Como vimos, ela se baseia na geração de números aleatórios e na média dos valores da função para estimar o resultado da integral. O método é simples de implementar e pode ser usado para resolver problemas de integração difíceis de tratar com métodos tradicionais.

12.6 Método da Amostragem por Rejeição

A amostragem por rejeição é uma técnica de Monte Carlo utilizada para calcular integrais de funções complexas. A ideia central desse método é gerar pontos aleatórios dentro de uma região e, em seguida, "rejeitar" pontos que não estejam sob a curva da função que desejamos integrar.

Para ilustrar o funcionamento desse método, vamos considerar novamente a integral :

$$I = \int_0^2 x^2\,dx$$

onde $f(x) = x^2$ e o intervalo de integração é $[0, 2]$.

12.6.1 Princípio do Método

1. Definir uma função de referência: Escolhemos uma função $g(x)$ que seja fácil de amostrar e que cubra $f(x)$ em todo o intervalo de integração. No caso de $f(x) = x^2$, podemos usar $g(x) = 4$, pois é uma constante que sempre será maior ou igual a x^2 para $x \in [0, 2]$.

2. Gerar pontos aleatórios: Geramos pontos aleatórios (x, y), onde x é uniformemente distribuído no intervalo $[0, 2]$ e y é uniformemente distribuído no intervalo $[0, 4]$ (a altura de $g(x) = 4$).

3. Aceitação e rejeição: Para cada ponto gerado (x, y), verificamos se $y \leq f(x)$. Se a condição for satisfeita, aceitamos o ponto, caso contrário, rejeitamos e geramos um novo ponto.
4. Estimativa da integral: A integral é estimada pela proporção de pontos aceitos, multiplicada pela área do retângulo onde os pontos são gerados. O valor de I será aproximado por:

$$I \approx \frac{2}{N} \sum_{i=1}^{N} \mathbb{1}_{\{y_i \leq f(x_i)\}}$$

onde $\mathbb{1}_{\{y_i \leq f(x_i)\}}$ é uma função indicadora que vale 1 se o ponto for aceito, e 0 caso contrário.

12.6.2 Implementação em Python

Abaixo segue o código Python para calcular a integral de $y = x^2$ usando o método de amostragem por rejeição.

```
import random
import matplotlib.pyplot as plt

# Definição da função f(x) = x^2
def f(x):
    return x**2

# Definindo os limites de integração
a, b = 0, 2
N = 10000  # Número de pontos a serem amostrados
g_max = 4  # Altura máxima da função de referência g(x) = 4

# Inicializa o contador de pontos aceitos
aceitos = 0

# Listas para armazenar os pontos (para visualização)
pontos_x = []
pontos_y = []

# Loop para gerar os pontos aleatórios
for _ in range(N):
    x = random.uniform(a, b)  # Gera um valor de x aleatório no intervalo [a, b]
    y = random.uniform(0, g_max)  # Gera um valor de y aleatório no intervalo [0, g_max]

    # Verifica se o ponto está abaixo da curva f(x)
    if y <= f(x):
        aceitos += 1  # Se o ponto é aceito, incrementa o contador
        pontos_x.append(x)
        pontos_y.append(y)

# Estima a integral usando a área do retângulo
area_retangulo = (b - a) * g_max
integral = (aceitos / N) * area_retangulo  # Correção da fórmula

# Exibe o resultado
print(f"Estimativa de I =  x^2 dx de {a} a {b}: {integral}")

# Visualização dos pontos aceitos
plt.scatter(pontos_x, pontos_y, color='blue', s=1, label='Pontos Aceitos')
plt.plot([x / 100 for x in range(201)], [f(x / 100) for x in range(201)], color='red', label='f(x) = x^2')
plt.title('Amostragem por Rejeição para  x^2 dx')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

12.6.3 Explicação do Código

1. Funções $f(x)$ e $g(x)$: A função $f(x) = x^2$ é a função que queremos integrar, e $g(x) = 4$ é a função de referência (um valor constante maior ou igual a x^2 para todo $x \in [0, 2]$).

2. Geração dos pontos: Para cada ponto, geramos um x aleatório dentro do intervalo $[0, 2]$ e um y aleatório dentro de $[0, 4]$.

3. Aceitação e Rejeição: Se $y \leq f(x)$, aceitamos o ponto; caso contrário, rejeitamos e geramos um novo ponto.

4. Estimativa da Integral: A integral é estimada pela proporção de pontos aceitos em relação ao número total de amostras, multiplicado pela área do retângulo $[0, 2] \times [0, 4]$.

5. Visualização: O gráfico mostra os pontos aceitos na amostragem (em azul) e a curva $f(x) = x^2$ (em vermelho).

12.6.4 Comparação com o Método da Média de Função

A estimativa obtida pelo método de amostragem por rejeição deve se aproximar do valor exato da integral, que é dado por:

$$I = \int_0^2 x^2 \, dx = \left[\frac{x^3}{3} \right]_0^2 = \frac{8}{3} \approx 2.6667$$

A vantagem desse método é que ele pode ser útil quando a função $f(x)$ tem regiões que são difíceis de amostrar diretamente, mas sua eficiência depende de uma boa escolha da função de referência $g(x)$.