Francisco Anacleto Barros Fidelis de Moura

Fundamentos de Python para Física Computacional: Uma Introdução Essencial

ISBN 978-65-01-41093-7

Fundamentos de Python para Física Computacional: Uma Introdução Essencial

Francisco Anacleto Barros Fidelis de Moura

Prefácio

Nas últimas décadas, a computação consolidou-se como uma ferramenta fundamental no avanço da física moderna. Através dela, tornou-se possível modelar sistemas complexos, analisar grandes volumes de dados e resolver, com eficiência, problemas matemáticos que desafiam abordagens analíticas tradicionais. Nesse cenário, a programação ocupa um papel central, e entre as linguagens disponíveis, o Python tem se destacado por sua sintaxe clara, seu vasto ecossistema de bibliotecas científicas e sua crescente popularidade na comunidade acadêmica.

Este livro oferece uma introdução acessível e didática ao uso do Python na física computacional. Partindo dos fundamentos da linguagem, conduzimos o leitor por uma jornada que inclui a implementação de algoritmos numéricos, a resolução de equações diferenciais e a simulação de fenômenos físicos diversos. Cada capítulo é acompanhado por exemplos comentados, exercícios práticos e explicações detalhadas que facilitam a compreensão e a aplicação dos conceitos apresentados.

Destinado a estudantes de graduação, pós-graduação e pesquisadores em física e áreas afins, o livro não exige experiência prévia em programação. Seu objetivo é justamente fornecer as bases necessárias para que o leitor possa utilizar a computação como uma poderosa aliada na investigação científica e no aprofundamento da compreensão dos fenômenos físicos.

Ao longo das próximas páginas, esperamos não apenas ensinar técnicas e ferramentas, mas também estimular a curiosidade, o pensamento crítico e a autonomia na exploração computacional da natureza.

Agradecimentos

Agradeço à minha esposa, Martha, pelo amor e companheirismo constantes, e ao meu filho, Miguel, cuja curiosidade e alegria são fontes diárias de inspiração. Aos meus pais, Sr. Fidelis e Dona Daia, devo não apenas apoio incondicional, mas também os valores que me guiaram ao longo da jornada acadêmica. Sou profundamente grato aos meus alunos e colegas do Instituto de Física da UFAL, cujas discussões instigantes e entusiasmo contagiante foram fundamentais para a construção deste material. Cada troca de ideias e cada desafio compartilhado enriqueceram este trabalho de maneira inestimável. Agradeço também à FAPEAL pelo suporte à pesquisa e à educação, viabilizando projetos como este e fortalecendo o desenvolvimento científico. Sem esse apoio, muitas dessas iniciativas não seriam possíveis.

Índice

1 Introdução . . . 1
2 Instalação do Python no Linux . . . 3
2.1 Instalação no OpenSUSE . . . 3
2.2 Instalação em Outras Distribuições . . . 3
2.3 Executando Programas em Python . . . 4
3 Conceitos Básicos de Programação em Python . . . 5
3.1 Variáveis e Tipos de Dados . . . 5
3.2 Entrada e Saída de Dados . . . 6
3.3 Criando Arquivos de Saída . . . 6
3.4 Operações Matemáticas Básicas . . . 7
3.5 Estruturas Condicionais (`if`, `elif`, `else`) . . . 8
3.6 Laços de Repetição (`for` e `while`) . . . 9
3.7 Funções . . . 11
3.8 Listas e Manipulação de Coleções . . . 11
3.9 Manipulação de Strings . . . 11
3.10 Tratamento de Erros (try, except) . . . 12
3.11 Números Pseudo-Aleatórios em Python . . . 12
4 Visualização de Dados . . . 17
4.1 Código para Geração de Gráficos de números aleatório . . . 17
4.2 Figuras de Lissajous e Animações em Python . . . 20
4.3 Animação do Movimento de um Pêndulo Simples . . . 23
5 Aprofundando os Conceitos de Python com Novos Exemplos . . . 27
5.1 Operações Matemáticas Avançadas com Dois Números . . . 27
5.2 Mais exemplos sobre o uso do Comando `for` em Python . . . 28
5.3 Soma dos Quadrados de Números Ímpares . . . 29
5.4 Soma dos Números Espaçados por 3 . . . 29
5.5 Explicando a importância dos `Loops` (comparando C e Python) . . . 30
5.6 Usando o Comando `while` para somar números de 1 até 10 . . . 31
5.7 Mais exemplos sobre manipulação de Vetores . . . 32
5.8 Mais informações sobre o uso de Condicionais em Python . . . 33
5.9 Ordenando três valores usando comandos "if" . . . 35
5.10 Exemplo: Ordenando quatro valores usando comandos "if" . . . 35
5.11 Ordenação de Números em Python(funções internas) . . . 36
5.12 Ordenando um vetor e mantendo a correspondência com outro vetor . . . 37
5.13 Exemplo de Programa em Python para Cálculo de Funções Trigonométricas . . . 38
6 Cálculo do Histograma Normalizado . . . 41
7 Solução numérica de alguns problemas de mecânica clássica . . . 45
7.1 Exemplo de Aplicação do Método de Euler na Equação $\frac{d^2x}{dt^2} = -kx$ . . . 45
7.2 Resolução Numérica do Oscilador Harmônico Amortecido $\frac{d^2x}{dt^2} = -kx - bv$ . . . 49
7.3 Queda livre com resistência do ar . . . 50
7.4 Pêndulo Simples com Resistência do Ar . . . 54
7.5 Simulação da Queda Livre com Efeito Magnus usando o Método de Euler . . . 57
7.6 Simulação da Trajetória de um Projétil com Resistência do Ar . . . 60
7.7 Simulação do Movimento de um Corpo em um Plano Inclinado com Resistência do Ar . . . 62
8 Derivação e integração numérica . . . 65
8.1 Derivação Numérica: Aproximação de Derivadas com Diferenças Finitas . . . 65
8.2 Integração Numérica: Aproximação de Integrais com Somatórios . . . 68
9 Problemas gerais de Física contemporânea . . . 72

9.1 Vibração de uma corda com extremidades fixas: solução numérica da equação de onda 73
9.2 Equação de Langevin, Movimento Browniano e Método Numérico de Heun 74
9.3 Modelo SIR: Dinâmica de Epidemias 76
9.4 O Mapa Logístico e o Comportamento Caótico 79
9.5 O Mapa de Lozi: Teoria, Aplicações e Implementação Computacional 80
9.6 Modelo de Neurônio Integrate-and-Fire: Formulação e Simulação com o Método de Euler . . . 82
9.7 Modelo de Hodgkin-Huxley 84
9.8 Introdução ao Conjunto de Mandelbrot 86
9.9 Integração de Monte Carlo 88
9.10 Decaimento Radioativo com Relaxação Temporal: Uma Abordagem Numérica Geral 91
9.11 Evolução temporal no modelo de Anderson 1D via método de Runge-Kutta 4ª ordem 93
10 Considerações Finais 96

1 Introdução

Python é uma linguagem de programação moderna, versátil e de alto nível, que vem se consolidando como uma das principais ferramentas para o desenvolvimento de aplicações científicas e técnicas. Sua sintaxe clara e intuitiva, aliada a uma extensa comunidade de desenvolvedores e uma rica coleção de bibliotecas, faz de Python uma escolha ideal tanto para iniciantes quanto para pesquisadores e profissionais experientes. Atualmente, Python é amplamente utilizada em áreas como ciência de dados, inteligência artificial, aprendizado de máquina, modelagem matemática, simulações numéricas, bioinformática e, de maneira especial, na física computacional.

Este livro tem como objetivo introduzir os conceitos fundamentais da linguagem Python com foco em aplicações na física computacional. Ao longo dos capítulos, o leitor será guiado desde os aspectos mais básicos da linguagem — como entrada e saída de dados, variáveis, estruturas de repetição e controle de fluxo, manipulação de listas e arrays — até tópicos mais avançados, como a implementação de algoritmos numéricos, visualização de dados e a construção de simulações computacionais de fenômenos físicos. A proposta é unir teoria e prática, por meio de exemplos comentados, exercícios e aplicações concretas, de modo a consolidar o aprendizado da linguagem ao mesmo tempo em que se desenvolvem habilidades em modelagem e análise de sistemas físicos.

Neste livro, abordamos diversos tópicos essenciais da física computacional, incluindo:

- Derivação numérica: Cálculo de derivadas por diferenças finitas, análise de erros e aplicações na cinemática e dinâmica de sistemas clássicos.
- Integração numérica: Métodos de integração como o trapézio e Simpson, com aplicações em cálculos de área, trabalho e energia.
- Soluções numéricas de equações diferenciais: Implementação de métodos como Euler para descrever a evolução temporal de sistemas físicos.
- Problemas de mecânica clássica: Simulação do movimento de projéteis, pêndulos simples e compostos, sistemas massa-mola, colisões e leis de conservação.
- Modelagem matemática aplicada: Estudo do modelo epidemiológico SIR, dinâmica populacional, crescimento logístico e outros sistemas modelados por equações diferenciais ordinárias.
- Métodos de Monte Carlo: Geração de distribuições aleatórias, simulação de sistemas estocásticos e estimação de integrais.
- Visualização científica: Criação de gráficos, animações e representações visuais interativas para facilitar a compreensão de fenômenos físicos.

Para isso, utilizamos poderosas bibliotecas da linguagem Python, tais como:
- **NumPy**: Para manipulação eficiente de arrays e realização de operações matemáticas vetoriais e matriciais.
- **Matplotlib**: Para a criação de gráficos estáticos, dinâmicos e interativos.
- **SymPy**: Para álgebra simbólica e manipulação de expressões matemáticas analíticas.
- **SciPy**: Para tarefas científicas avançadas, como resolução de equações diferenciais, otimização e transformadas.
- **Pygame**: Para desenvolvimento de simulações interativas, animações e jogos educativos com aplicações físicas.

A física computacional representa uma ponte entre a modelagem teórica e a experimentação, possibilitando o estudo de sistemas complexos e não lineares, muitas vezes inacessíveis por métodos analíticos tradicionais. Com Python, podemos explorar desde os fundamentos da física até aplicações modernas, como a simulação da propagação de epidemias, a dinâmica de partículas sob campos externos e o comportamento coletivo de sistemas caóticos ou estocásticos.

Nosso objetivo é proporcionar uma base sólida e acessível, capacitando o leitor a aplicar Python de forma criativa e eficiente na resolução de problemas físicos. Esperamos, com este livro, despertar o interesse pela programação científica e incentivar a autonomia no desenvolvimento de projetos computacionais na física e em outras áreas do conhecimento.

2 Instalação do Python no Linux

O Python pode ser instalado em distribuições Linux de diferentes maneiras. A maioria das distribuições já inclui uma versão do Python por padrão, mas, caso seja necessário instalar ou atualizar, siga os passos abaixo.

2.1 Instalação no OpenSUSE

No OpenSUSE, o Python pode ser instalado via gerenciador de pacotes extttzypper com o seguinte comando:

```
sudo zypper install python3
```

Para instalar bibliotecas como o `matplotlib` (usada para visualização gráfica) e outras dependências que você pode usar nos seus programas Python, primeiro instale o `pip`, o gerenciador de pacotes do Python:

```
sudo zypper install python3-pip
```

Depois de instalar o `pip`, você pode instalar o `matplotlib` e outras bibliotecas com o seguinte comando:

```
pip3 install matplotlib
```

Caso você precise instalar outras bibliotecas, como `numpy` ou `scipy`, basta usar o `pip` da mesma maneira:

```
pip3 install numpy scipy
```

2.2 Instalação em Outras Distribuições

Para outras distribuições Linux, os comandos variam:

- Ubuntu/Debian:

  ```
  sudo apt update && sudo apt install python3
  ```

- Fedora:

  ```
  sudo dnf install python3
  ```

- Arch Linux:

  ```
  sudo pacman -S python
  ```

- Linux Mint: O Linux Mint é baseado no Ubuntu/Debian e geralmente já vem com o Python instalado. No entanto, caso precise instalar ou atualizar, siga os passos abaixo:

  1. **Atualizar o sistema:** Antes de instalar qualquer pacote, execute:

```
sudo apt update && sudo apt upgrade -y
```

2. **Instalar o Python:** Para instalar a versão mais recente disponível:

   ```
   sudo apt install python3
   ```

3. **Verificar a instalação:** Para checar a versão instalada:

   ```
   python3 --version
   ```

4. **Instalar o gerenciador de pacotes extttpip (opcional):** Caso precise instalar bibliotecas adicionais:

   ```
   sudo apt install python3-pip
   ```

2.3 Executando Programas em Python

Após escrever seu código Python e salvá-lo em um arquivo, por exemplo, `script.py`, você pode executá-lo diretamente pelo terminal (ou prompt de comando) utilizando o seguinte comando:

```
python3 script.py
```

Esse comando solicita ao interpretador Python que execute o conteúdo do arquivo especificado. Certifique-se de estar no mesmo diretório onde o arquivo está salvo, ou forneça o caminho completo até ele.

Além disso, o Python também pode ser utilizado de forma interativa. Para isso, basta digitar no terminal:

```
python3
```

e pressionar `Enter`. Isso abrirá o interpretador interativo (também conhecido como *modo REPL* — Read, Evaluate, Print, Loop), onde você pode digitar comandos Python linha por linha e ver os resultados imediatamente. Esse modo é útil para testar pequenos trechos de código, realizar cálculos rápidos ou explorar funcionalidades da linguagem.

Para sair do interpretador interativo, utilize o atalho `Ctrl+D` (em sistemas Unix) ou `Ctrl+Z` seguido de `Enter` (em Windows).

3 Conceitos Básicos de Programação em Python

Nesta seção, apresentaremos os conceitos fundamentais da programação em Python, de maneira clara, acessível e voltada especialmente para estudantes de graduação que estão dando seus primeiros passos na linguagem. Esses fundamentos são indispensáveis para a construção de programas simples e representam a base sobre a qual se desenvolvem habilidades mais avançadas em programação científica. Nosso objetivo é proporcionar uma introdução teórica e prática, que permita ao leitor compreender os princípios essenciais da linguagem Python e aplicá-los na resolução de problemas computacionais. Ao longo desta seção, abordaremos temas como a criação e uso de variáveis, a manipulação de dados, as estruturas de controle de fluxo (condicionais e laços de repetição), bem como outras funcionalidades essenciais que fazem de Python uma linguagem poderosa e versátil. Cada conceito será explicado de forma gradual, com exemplos práticos e códigos comentados, de modo que o leitor possa não apenas entender a teoria, mas também praticar e desenvolver suas próprias soluções. A proposta é construir uma base sólida para que você se sinta confortável em escrever seus primeiros programas e, posteriormente, enfrentar desafios mais complexos em áreas como física computacional, matemática aplicada e ciência de dados.

3.1 Variáveis e Tipos de Dados

Em Python, as variáveis são usadas para armazenar informações que podem ser manipuladas ao longo do programa. A linguagem é dinamicamente tipada, ou seja, não é necessário declarar o tipo da variável — o Python identifica automaticamente com base no valor atribuído. Os tipos de dados mais comuns incluem:

- `int` — números inteiros (ex: 10, -3, 0)
- `float` — números com casas decimais (ex: 3.14, -0.5)
- `str` — sequências de caracteres (ex: ”João”, ”Física”)
- `list` — listas de elementos (ex: [1, 2, 3])

Exemplo:

```
# Definindo variáveis com diferentes tipos de dados
numero = 10              # Variável do tipo inteiro
pi = 3.14                # Variável do tipo float
nome = "João"            # Variável do tipo string
valores = [1, 2, 3, 4]   # Variável do tipo lista

# Exibindo os valores na tela
print("Número inteiro:", numero)
print("Número decimal:", pi)
print("Nome:", nome)
print("Lista de valores:", valores)
```

Neste exemplo, usamos quatro tipos diferentes de dados. O comando `print()` serve para exibir os valores armazenados nas variáveis.

3.2 Entrada e Saída de Dados

Em Python, a comunicação com o usuário é feita principalmente por meio de duas funções:

- `input()`: permite que o usuário digite dados pelo teclado. O valor retornado é sempre do tipo `str` (string), então, quando necessário, deve-se converter para o tipo desejado, como `int` ou `float`.
- `print()`: exibe informações na tela. Pode ser usada para mostrar textos fixos, variáveis, resultados de expressões, entre outros.

Exemplo 1: Exibindo o número digitado pelo usuário

```
# Recebe um número do usuário e exibe na tela
numero = input("Digite um número: ")
print("Você digitou o número:", numero)
```

Neste exemplo, o programa solicita que o usuário digite um número. O valor digitado é armazenado na variável `numero` e, em seguida, exibido na tela usando a função `print()`. Note que o valor é tratado como texto (string), já que não realizamos nenhuma conversão ou operação com ele.

Exemplo 2: Saudação personalizada

```
# Solicita o nome do usuário e exibe uma mensagem de boas-vindas
nome = input("Digite seu nome: ")
print("Olá,", nome + "! Seja bem-vindo(a)!")
```

Neste segundo exemplo, a função `input()` captura o nome do usuário como uma string. Em seguida, a função `print()` é usada para montar e exibir uma mensagem personalizada com o nome fornecido.

Dica: para concatenar textos (strings), pode-se usar o símbolo `+`, e para separar itens automaticamente com espaço, basta usar vírgulas dentro do `print()`, como mostrado acima.

Essas funções são essenciais para tornar o programa interativo e permitir a troca de informações com o usuário.

3.3 Criando Arquivos de Saída

Em Python, é possível criar e escrever arquivos de texto com a função `open()`. Isso é muito útil para salvar dados, resultados ou mensagens geradas durante a execução do programa.
A forma geral para criar e escrever em um arquivo é:

- `open("nome_do_arquivo.txt", "w")`: abre (ou cria) o arquivo em modo de escrita.
- `write("texto")`: escreve o conteúdo dentro do arquivo.
- `close()`: fecha o arquivo e garante que os dados sejam salvos corretamente.

Exemplo 1: Criando um arquivo com uma mensagem simples

```
# Cria um arquivo e escreve frases dentro dele
arquivo = open("saida.txt", "w")
arquivo.write("Este é um arquivo de saída.\n")
arquivo.write("Podemos guardar dados aqui.\n")
arquivo.close()
```

Neste exemplo, o programa cria um arquivo chamado `saida.txt` e escreve duas linhas de texto. O caractere `\n` serve para quebrar a linha (pular para a próxima).

Exemplo 2: Salvando o nome digitado pelo usuário em um arquivo

```
# Solicita o nome do usuário e salva em um arquivo
nome = input("Digite seu nome: ")

arquivo = open("nome_usuario.txt", "w")
arquivo.write("Nome digitado pelo usuário: " + nome + "\n")
arquivo.close()
```

Nesse exemplo, o programa recebe o nome do usuário e grava esse nome dentro do arquivo `nome_usuario.txt`.

Exemplo 3: O nome do arquivo depende da entrada do usuário

```
# O usuário digita um número, e o nome do arquivo depende desse número
numero = input("Digite um número qualquer: ")

nome_arquivo = "dados_" + numero + ".txt"

arquivo = open(nome_arquivo, "w")
arquivo.write("Você digitou o número: " + numero + "\n")
arquivo.close()
```

Neste último exemplo, o nome do arquivo é criado dinamicamente com base no número digitado pelo usuário. Por exemplo, se o usuário digitar `7`, o arquivo gerado se chamará `dados_7.txt`. Dentro dele será escrita a frase com o número informado.

Dica: Usar nomes de arquivos que indicam o conteúdo pode ajudar a organizar melhor os resultados gerados pelos programas.

Importante: Sempre finalize com `close()` para garantir que as informações sejam salvas corretamente no arquivo.

3.4 Operações Matemáticas Básicas

Python permite realizar facilmente operações matemáticas fundamentais como:

- `+` soma
- `-` subtração
- `*` multiplicação
- `/` divisão (resultado com ponto flutuante)
- `//` divisão inteira (descarta o valor decimal)
- `%` módulo (resto da divisão)
- `**` exponenciação (potência)

Exemplo 1: Operações básicas entre dois números

```
# Definindo dois números
a = 10
b = 3

# Realizando operações matemáticas
soma = a + b         # Soma
subtracao = a - b    # Subtração
multiplicacao = a * b  # Multiplicação
divisao = a / b      # Divisão com resultado decimal
resto = a % b        # Resto da divisão (módulo)
div_inteira = a // b # Divisão inteira (sem casas decimais)
potencia = a ** b    # Exponenciação (a elevado à b)

# Exibindo os resultados
print("Soma:", soma)
print("Subtração:", subtracao)
print("Multiplicação:", multiplicacao)
print("Divisão:", divisao)
print("Resto da divisão:", resto)
print("Divisão inteira:", div_inteira)
print("Potência:", potencia)
```

Exemplo 2: Operações com números fornecidos pelo usuário

```
# Solicita dois números do usuário
x = float(input("Digite o primeiro número: "))
y = float(input("Digite o segundo número: "))

# Realiza operações
print("Soma:", x + y)
print("Subtração:", x - y)
print("Multiplicação:", x * y)
print("Divisão:", x / y)
print("Resto da divisão:", x % y)
print("x elevado à y:", x ** y)
```

Essas operações são fundamentais para qualquer programa que lide com cálculos. O Python segue a precedência de operadores padrão da matemática, ou seja, parênteses primeiro, depois exponenciação, multiplicações/divisões, e por fim somas/subtrações. Para garantir a ordem correta das operações, use sempre parênteses quando necessário.

3.5 Estruturas Condicionais (`if`, `elif`, `else`)

As estruturas condicionais são fundamentais para que um programa possa tomar decisões com base em determinadas condições. Em Python, utilizamos as palavras-chave `if`, `elif` (abreviação de "else if") e `else` para controlar o fluxo de execução.

- `if`: executa um bloco de código se a condição for verdadeira.
- `elif`: permite testar múltiplas condições, caso a condição anterior seja falsa.
- `else`: executa um bloco de código se todas as condições anteriores forem falsas.

Exemplo 1: Verifica se um número é positivo, negativo ou zero

```
# Solicita um número ao usuário
numero = float(input("Digite um número: "))

# Verifica se o número é positivo, negativo ou zero
if numero > 0:
    print("O número é positivo.")
elif numero < 0:
    print("O número é negativo.")
else:
    print("O número é zero.")
```

Exemplo 2: Verifica se um número é par ou ímpar

```
# Solicita um número inteiro ao usuário
numero = int(input("Digite um número inteiro: "))

# Usa o operador % (módulo) para verificar o resto da divisão por 2
if numero % 2 == 0:
    print("O número é par.")
else:
    print("O número é ímpar.")
```

Exemplo 3: Verifica se uma pessoa é maior de idade

```
# Solicita a idade do usuário
idade = int(input("Digite sua idade: "))

# Verifica se a idade é maior ou igual a 18
if idade >= 18:
    print("Você é maior de idade.")
else:
    print("Você é menor de idade.")
```

Exemplo 4: Classificação de nota

```
# Solicita a nota do aluno
nota = float(input("Digite a nota do aluno (0 a 10): "))

# Classifica a nota com base em faixas
if nota >= 9.0:
    print("Excelente")
elif nota >= 7.0:
    print("Bom")
elif nota >= 5.0:
    print("Regular")
else:
    print("Insuficiente")
```

Esses exemplos mostram como as estruturas condicionais permitem que um programa reaja de maneiras diferentes dependendo das entradas fornecidas. É essencial indentar corretamente os blocos de código após cada condição para que o Python interprete corretamente as instruções.

3.6 Laços de Repetição (`for` e `while`)

Os **laços de repetição** são estruturas fundamentais em programação, permitindo que blocos de código sejam executados várias vezes. Em Python, os dois tipos principais de laços são:

- `for`: usado para percorrer sequências (como listas, tuplas, strings ou intervalos numéricos).
- `while`: executa o bloco de código enquanto uma condição lógica for verdadeira.

Laço `for` com `range()`:

O comando `range(início, fim)` gera uma sequência de números inteiros. O valor do `fim` não é incluído na sequência. Veja um exemplo:

```
# Laço for: imprime os números de 1 a 5
for i in range(1, 6):
    print(i)
```

É possível também adicionar um terceiro parâmetro ao `range`, que define o **passo** da contagem:

```
# Imprime números ímpares de 1 a 9
for i in range(1, 10, 2):
    print(i)
```

O laço `for` pode ser usado para percorrer elementos de uma lista ou string:

```
# Percorre elementos de uma lista
nomes = ["Ana", "Bruno", "Carlos"]
for nome in nomes:
    print("Olá,", nome)

# Percorre cada caractere de uma string
for letra in "Física":
    print(letra)
```

Laço `while`:

O laço `while` continua executando o bloco de código enquanto a condição especificada for verdadeira.

```
# Imprime números de 1 a 5 usando while
i = 1
while i <= 5:
    print(i)
    i = i + 1
```

É importante garantir que a condição do `while` eventualmente se torne falsa, para evitar laços infinitos.

```
# Exemplo com entrada do usuário
senha = ""
while senha != "python123":
    senha = input("Digite a senha: ")
print("Acesso permitido.")
```

Resumo:
Os laços `for` e `while` são essenciais para automatizar tarefas repetitivas, percorrer dados e controlar fluxos em programas. A escolha entre eles depende da situação: use `for` quando souber exatamente quantas vezes deseja repetir, e `while` quando a repetição depender de uma condição.

3.7 Funções

Funções são blocos de código reutilizáveis que executam tarefas específicas. Elas ajudam a organizar melhor o programa, evitar repetições e facilitar a manutenção do código. Uma função pode receber parâmetros, realizar cálculos ou operações e retornar um valor.

Exemplo:

```
# Função que soma dois números
def soma(a, b):
    return a + b

# Chamando a função
resultado = soma(3, 5)
print("A soma é:", resultado)
```

Neste exemplo, a função `soma` recebe dois parâmetros, realiza a adição e retorna o resultado. Em seguida, o valor retornado é armazenado na variável `resultado` e impresso na tela.

3.8 Listas e Manipulação de Coleções

Listas são estruturas de dados que armazenam coleções ordenadas de elementos. Elas permitem acesso direto a seus itens por índice, além de operações como inserção, remoção, ordenação e iteração.

Exemplo:

```
# Lista de números
numeros = [1, 2, 3, 4]

# Adicionando um item à lista
numeros.append(5)

# Acessando e imprimindo um item
print("Primeiro número:", numeros[0])

# Imprimindo a lista atualizada
print("Lista atualizada:", numeros)
```

O código acima cria uma lista de inteiros chamada `numeros`, adiciona um novo elemento ao final da lista usando `append`, acessa o primeiro item utilizando índice zero, e imprime a lista atualizada.

3.9 Manipulação de Strings

Strings são sequências de caracteres amplamente utilizadas para representar textos. Em Python, é possível realizar diversas operações com strings, como concatenação, fatiamento, substituição, busca por substrings, entre outras.

Exemplo:

```
# Manipulando strings
nome = "João"
sobrenome = "Silva"

# Concatenando strings
nome_completo = nome + " " + sobrenome
```

```
# Exibindo o nome completo
print("Nome completo:", nome_completo)
```

Neste exemplo, duas variáveis do tipo string são concatenadas com um espaço entre elas para formar o nome completo. O resultado é armazenado na variável `nome_completo` e exibido na tela.

3.10 Tratamento de Erros (try, except)

Durante a execução de um programa, podem ocorrer situações inesperadas, como entradas inválidas ou operações que falham. O tratamento de erros com `try` e `except` permite capturar essas exceções e lidar com elas de forma apropriada, evitando que o programa seja interrompido abruptamente.

Exemplo:

```
# Tratamento de erro
try:
    numero = int(input("Digite um número inteiro: "))
    print("Você digitou:", numero)
except ValueError:
    print("Erro: Você não digitou um número inteiro!")
```

Neste exemplo, o programa tenta converter a entrada do usuário para um número inteiro usando a função `int()`. Se o usuário digitar algo inválido (como letras ou símbolos), ocorrerá uma exceção do tipo `ValueError`. O bloco `except` captura esse erro e exibe uma mensagem amigável, mantendo o programa em funcionamento.

O tratamento de erros é uma prática essencial na escrita de programas robustos e seguros. Com ele, é possível prever falhas, informar o usuário adequadamente e evitar que o programa seja finalizado de forma inesperada.

Esses conceitos apresentados — estruturas de dados simples, funções, manipulação de coleções e strings, e tratamento de erros — são fundamentais para qualquer pessoa que esteja começando a aprender Python. Eles constituem a base para a construção de programas funcionais e a exploração dos recursos mais avançados da linguagem.

3.11 Números Pseudo-Aleatórios em Python

Vamos iniciar agora o estudo dos números aleatórios em python. Vamos aprender inicialmente sobre a implementação um gerador de números aleatórios pseudo-aleatórios bem famoso na literatura chamado ” Gerador Congruente Linear” (LCG). Este é um dos métodos mais simples e conhecidos para gerar sequências de números aleatórios. Vamos explicar o algoritmo, seu funcionamento e como implementá-lo em Python de forma simples e direta.

Método dos Geradores Congruentes Lineares (LCG)

O **Gerador Congruente Linear** é um algoritmo que gera números pseudo-aleatórios a partir de uma fórmula recursiva, dada por:

$$X_{n+1} = (aX_n + c) \mod m$$

Onde:

- X_n é o número aleatório gerado na n-ésima iteração;

- a é o multiplicador;
- c é o incremento;
- m é o módulo;
- X_0 é a semente inicial (ou valor de partida).

O gerador começa com um valor inicial X_0 (semente) e, em cada iteração, gera um novo número X_{n+1} com base no valor anterior X_n. O processo é repetido para gerar uma sequência de números pseudo-aleatórios. O sucesso de um Gerador Congruente Linear depende da escolha adequada dos parâmetros a, c, e m. Vamos discutir a escolha de cada um deles:

- **Módulo** m : Geralmente, escolhe-se m como uma grande potência de 2, pois isso facilita o cálculo do módulo em sistemas digitais. O valor m deve ser grande o suficiente para garantir que o gerador tenha um bom período. A operação "mod" que aparece na equação é basicamente o resto da divisão de (a * X + c) por 'm'.
- **Multiplicador** a: O valor de a deve ser escolhido de modo que a sequência de números aleatórios gerados seja bem distribuída. Certos valores de a são conhecidos por fornecer boas distribuições.
- **Incremento** c: O valor de c também deve ser escolhido de forma que ajude a distribuir bem os números gerados. Normalmente, escolhe-se c de forma arbitrária, mas sempre maior que zero.
- **Semente** X_0: A semente é o valor inicial. Para obter diferentes sequências de números aleatórios, basta variar a semente.

O objetivo de um bom gerador é maximizar o período, ou seja, o número máximo de iterações até que a sequência de números se repita. A escolha adequada de a, c, e m pode garantir um bom período. Agora, vamos implementar um gerador simples de números aleatórios utilizando o método do Gerador Congruente Linear em Python. Abaixo está o código para o gerador:

```
# Definindo os parâmetros do gerador
m = 2**32         # Módulo (geralmente uma potência de 2)
a = 1664525       # Multiplicador
c = 1013904223    # Incremento
X0 = 42           # Semente inicial

# Função para gerar um número aleatório
#usando o método congruente linear (LCG)
def gerar_aleatorio(X):
# A equação usada segue a forma padrão do gerador congruente linear:
# X_{n+1} = (a * X_n + c) % m
# Onde:
# - 'a' é o multiplicador
# - 'c' é o incremento
# - 'm' é o módulo (limite superior dos valores gerados)
# - 'X' é o valor atual da sequência (semente inicial ou valor anterior)
# O operador '%' (módulo) retorna o resto da divisão de (a * X + c) por 'm'.
# Isso garante que os valores gerados estejam sempre no intervalo [0, m-1].

    return (a * X + c) % m
```

```
# Gerar 10 números aleatórios
X = X0
for i in range(10):
    X = gerar_aleatorio(X)
    print(X)
```

O código acima implementa um gerador congruente linear simples em Python. A seguir, explicamos cada parte do código:

- Definição dos parâmetros:

  ```
  m = 2**32         # Módulo (geralmente uma potência de 2)
  a = 1664525       # Multiplicador
  c = 1013904223    # Incremento
  X0 = 42           # Semente inicial
  ```

  Aqui, definimos o valor do módulo m como 2^{32}, uma potência de 2. O multiplicador a, o incremento c, e a semente X_0 são valores arbitrários, mas escolhidos de maneira a garantir uma boa distribuição dos números gerados.

- Função geradora:

  ```
  def gerar_aleatorio(X):
      return (a * X + c) % m
  ```

  A função **gerar_aleatorio(X)** recebe um valor X e retorna o próximo número gerado na sequência utilizando a fórmula $X_{n+1} = (aX_n + c) \mod m$.

- Gerando números aleatórios:

  ```
  X = X0
  for i in range(10):
      X = gerar_aleatorio(X)
      print(X)
  ```

  Inicializamos a variável X com o valor da semente X_0 e, em seguida, usamos um laço 'for' para gerar e imprimir 10 números aleatórios. A cada iteração, o valor de X é atualizado utilizando a função **gerar_aleatorio**.

Embora o Gerador Congruente Linear seja simples e fácil de implementar, ele possui algumas limitações em termos de qualidade dos números aleatórios gerados, como a possibilidade de repetição e a falta de aleatoriedade verdadeira. Esse tipo de gerador é útil em situações em que a simplicidade e a eficiência são mais importantes do que a qualidade dos números aleatórios, como em algumas simulações simples ou jogos. No entanto, para aplicações que exigem uma aleatoriedade de alta qualidade, como simulações científicas avançadas ou criptografia, métodos mais sofisticados, como o Mersenne Twister, são frequentemente utilizados. Este exemplo, embora simples, demonstra a importância de entender os fundamentos dos geradores de números aleatórios e como esses algoritmos podem ser implementados de maneira direta. Portanto, em linhas gerais, o código anterior implementa um gerador de números pseudo-aleatórios utilizando o método Linear Congruential Generator

(LCG). A sequência gerada é determinística, baseada na semente inicial X_0 e nos parâmetros m, a e c. Os números gerados são inteiros no intervalo de 0 até $m-1$, o que resulta em uma sequência de números pseudo-aleatórios. Para que os números gerados fiquem no intervalo de 0 a 1, basta dividir os valores gerados por m, o que transformará os números inteiros em números reais no intervalo $[0, 1)$. O codigo seguinte faz esta implementação :

```
# Definindo os parâmetros do gerador
m = 2**32         # Módulo (geralmente uma potência de 2)
a = 1664525       # Multiplicador
c = 1013904223    # Incremento
X0 = 42           # Semente inicial

# Função para gerar números aleatórios
def gerar_aleatorio(X):
    return (a * X + c) % m

# Gerar 10 números aleatórios no intervalo [0, 1)
X = X0
for i in range(10):
    X = gerar_aleatorio(X)
    print(X / m)
# Dividindo pelo módulo para obter um número
# real no intervalo [0, 1)
```

Gerador Uniforme em Python

O Python possui uma biblioteca chamada `random`, que inclui um gerador de números pseudo-aleatórios com distribuição uniforme. A função `uniform(a, b)` é usada para gerar números aleatórios de uma distribuição uniforme contínua no intervalo $[a, b]$. A função `uniform(a, b)` retorna um número aleatório x tal que:

$$a \leq x \leq b$$

onde a é o limite inferior e b é o limite superior. O valor gerado será um número decimal (float). Aqui está um exemplo simples de como usar o gerador uniforme em Python para gerar números aleatórios entre 0 e 1:

```
import random

# Definir a semente do gerador
seed = 42
random.seed(seed)

# Gerar e imprimir 10 números aleatórios entre 0 e 1
for _ in range(10):
    numero = random.uniform(0, 1)
    print(numero)
```

Este código utiliza a biblioteca ‘random‘ do Python para gerar números pseudo-aleatórios no intervalo $[0, 1]$. A função ‘random.seed(seed)‘ permite definir uma semente inicial, garantindo que a sequência de números gerados seja reproduzível. Em seguida, um loop é utilizado para gerar e imprimir 10 números utilizando ‘random.uniform(0, 1)‘, que retorna valores reais dentro do intervalo especificado. Diferentemente dos geradores congruentes lineares (LCG), que operam com uma fórmula recursiva do tipo $X_{n+1} = (aX_n + c) \mod m$, o gerador usado neste código faz parte de algoritmos internos mais sofisticados, como o Mersenne Twister, oferecendo uma melhor distribuição

estatística e maior período antes de repetir a sequência. Além disso, enquanto o LCG gera números inteiros e precisa ser normalizado, ‘random.uniform(0,1)‘ já retorna números reais diretamente no intervalo desejado.

4 Visualização de Dados

A visualização de dados é uma parte fundamental na análise de dados, pois permite uma compreensão visual das distribuições e padrões dos dados. Gráficos são frequentemente usados para resumir informações complexas de maneira intuitiva, facilitando a análise e interpretação dos resultados. No contexto de programação e análise computacional, uma das ferramentas mais utilizadas para a criação de gráficos é a biblioteca `matplotlib`.

A biblioteca `matplotlib` no Python oferece uma interface simples para criar uma grande variedade de gráficos. Entre os tipos mais comuns de gráficos, destacam-se: - Gráficos de linha (para mostrar a relação entre duas variáveis). - Histogramas (para exibir a distribuição de um conjunto de dados). - Gráficos de dispersão (para mostrar como os pontos de dados estão distribuídos em um espaço bidimensional).

4.1 Código para Geração de Gráficos de números aleatório

Nesta subseção, vamos explorar como gerar gráficos a partir de números aleatórios gerados por dois métodos diferentes: o Gerador Congruente Linear (LCG) e o comando `random.uniform` da biblioteca `random`. A seguir, apresentamos os gráficos que serão gerados com esses dados:

- Gráfico 1: Números aleatórios x_i gerados pelo LCG.
- Gráfico 2: Números aleatórios x_i gerados pelo comando `random.uniform`.
- Gráfico 3: Histograma dos números aleatórios gerados pelo LCG.
- Gráfico 4: Histograma dos números aleatórios gerados pelo comando `random.uniform`.

Os gráficos gerados serão salvos no formato `EPS` para garantir a qualidade das figuras e a capacidade de edição em outros softwares gráficos. Abaixo está o código em Python que gera os quatro gráficos mencionados, salva as figuras no formato `EPS` e exibe os gráficos na tela:

```
import random
# Importa o módulo 'random', que fornece funções
# para gerar números aleatórios no Python.


import numpy as np
# Importa a biblioteca NumPy, que oferece suporte
# para operações matemáticas eficientes, como
# arrays e funções de álgebra linear.

import matplotlib.pyplot as plt
# Importa a sub-biblioteca 'pyplot' do Matplotlib, que é usada para
# criar gráficos e visualizações em Python.


# Parâmetros do gerador congruente linear (LCG)
m = 2**32
a = 1664525
c = 1013904223
X0 = 42

# Definir a semente do gerador (pode ser alterada pelo usuário)
random.seed(X0)

# Função para gerar números aleatórios usando LCG
def lcg(n, seed):
```

```
    X = seed
    numeros = []
    for _ in range(n):
        X = (a * X + c) % m
        numeros.append(X / m)  # Normaliza para [0,1]
    return numeros

# Definir número de amostras e bins
N = 1000000
bins = 50

# Gerar números aleatórios com LCG
lcg_numbers = lcg(N, X0)

# Gerar números aleatórios com random.uniform
random_numbers = [random.uniform(0, 1) for _ in range(N)]

# Plot 1: Números aleatórios gerados pelo LCG

plt.figure()  # Cria uma nova figura para o gráfico

# Plota os números aleatórios gerados pelo LCG como pontos azuis ('b.')
plt.plot(lcg_numbers, 'b.')

# Define o título do gráfico
plt.title('Números Aleatórios Gerados pelo LCG')

# Define o rótulo do eixo X (índice dos números gerados)
plt.xlabel('Índice')

# Define o rótulo do eixo Y (valores gerados pelo LCG)
plt.ylabel('Valor Aleatório')

# Salva o gráfico no formato EPS

plt.savefig('lcg_numbers.eps', format='eps')


plt.show()  # Exibe o gráfico na tela


# Plot 2: Números aleatórios gerados pelo random.uniform

plt.figure()  # Cria uma nova figura para o gráfico

# Plota os números aleatórios gerados pelo
#random.uniform como pontos vermelhos ('r.')
plt.plot(random_numbers, 'r.')


# Define o título do gráfico
plt.title('Números Aleatórios Gerados pelo random.uniform')

# Define o rótulo do eixo X (índice dos números gerados)
plt.xlabel('Índice')

# Define o rótulo do eixo Y (valores gerados pelo random.uniform)
plt.ylabel('Valor Aleatório')
```

```
# Salva o gráfico no formato EPS (para qualidade de impressão)
plt.savefig('random_numbers.eps', format='eps')


plt.show()  # Exibe o gráfico na tela


# Plot 3: Histograma dos números aleatórios gerados pelo LCG

plt.figure()  # Cria uma nova figura para o gráfico


plt.hist(lcg_numbers, bins=bins, density=True, alpha=0.6, color='g')

# Plota o histograma dos números gerados pelo LCG
# 'bins' define o número de intervalos (bins) no histograma
# 'density=True' normaliza o histograma, transformando-o
#  em uma densidade de probabilidade
# 'alpha=0.6' ajusta a transparência das barras do histograma
# (quanto mais próximo de 1, mais opaco)
# 'color='g'' define a cor das barras como verde


# Define o título do gráfico
plt.title('Histograma de Números Aleatórios (LCG)')


# Define o rótulo do eixo X (valores dos números aleatórios gerados)
plt.xlabel('Valor Aleatório')

# Define o rótulo do eixo Y (densidade de probabilidade do histograma)
plt.ylabel('Densidade de Probabilidade')

# Salva o gráfico no formato EPS (para qualidade de impressão)
plt.savefig('lcg_histogram.eps', format='eps')


plt.show()  # Exibe o gráfico na tela


# Plot 4: Histograma dos números aleatórios gerados pelo random.uniform

plt.figure()  # Cria uma nova figura para o gráfico

# Plota o histograma dos números gerados pelo random.uniform
plt.hist(random_numbers, bins=bins, density=True, alpha=0.6, color='m')

# 'bins' define o número de intervalos (bins) no histograma
# 'density=True' normaliza o histograma, transformando-o em
# uma densidade de probabilidade
# 'alpha=0.6' ajusta a transparência das barras do histograma
# (quanto mais próximo de 1, mais opaco)
# 'color='m'' define a cor das barras como magenta



# Define o título do gráfico
plt.title('Histograma de Números Aleatórios (random.uniform)')
```

```
# Define o rótulo do eixo X (valores dos números aleatórios gerados)
plt.xlabel('Valor Aleatório')

# Define o rótulo do eixo Y (densidade de probabilidade do histograma)
plt.ylabel('Densidade de Probabilidade')

# Salva o gráfico no formato EPS (para qualidade de impressão)
plt.savefig('random_histogram.eps', format='eps')


plt.show()  # Exibe o gráfico na tela
```

Vamos apresentar mais alguns detalhes acerca do programa acima:

- `import matplotlib.pyplot as plt`: Esta linha importa a biblioteca `matplotlib.pyplot`, que é utilizada para gerar gráficos em Python. A função `plt` é o ponto de entrada para a maioria dos comandos gráficos.
- `plt.figure()`: Cria uma nova figura para o gráfico. Esta linha deve ser chamada antes de qualquer gráfico ser plotado, para garantir que os gráficos sejam desenhados em figuras separadas.
- `plt.plot(...)`: Plota os dados em um gráfico de linha. No caso dos gráficos de números aleatórios, estamos utilizando o estilo de pontos (`'.'`) para representar os valores gerados ao longo do índice.
- `plt.hist(...)`: Plota o histograma dos números aleatórios gerados. A função `plt.hist` recebe o conjunto de dados, o número de intervalos (`bins`) e a opção `density=True` para normalizar o histograma.
- `plt.title(...)`: Define o título do gráfico.
- `plt.xlabel(...)` e `plt.ylabel(...)`: Definem os rótulos dos eixos x e y, respectivamente.
- `plt.savefig(...)`: Salva a figura gerada no formato especificado (no caso, `EPS`). O parâmetro `format='eps'` garante que o arquivo seja salvo com a extensão correta.
- `plt.show()`: Exibe o gráfico gerado na tela.

4.2 Figuras de Lissajous e Animações em Python

As figuras de Lissajous são curvas paramétricas descritas pelo movimento de um ponto cujas coordenadas variam de acordo com funções harmônicas ortogonais. Essas curvas são definidas pelas equações:

$$x = A\sin(at), \quad y = B\sin(bt + \delta) \tag{1}$$

onde A e B representam as amplitudes nos eixos x e y, respectivamente, a e b são as frequências angulares associadas a cada direção, e δ é o deslocamento de fase entre os movimentos oscilatórios. O estudo das figuras de Lissajous é de grande relevância em diversas áreas da física, matemática e engenharia, pois elas descrevem padrões característicos de sistemas oscilatórios acoplados. Essas curvas surgem naturalmente na análise de sinais periódicos, como em osciloscópios analógicos, onde são utilizadas para comparar a frequência e a fase de dois sinais elétricos ortogonais. Além disso, desempenham um papel fundamental na teoria de ressonância, na análise de sistemas dinâmicos não lineares e no estudo de fenômenos de interferência e batimentos em ondas mecânicas e eletromagnéticas. Na engenharia e na acústica, as figuras de Lissajous ajudam a visualizar a relação entre diferentes frequências de vibração, sendo úteis na caracterização de sistemas de transmissão

de sinais e na análise de estabilidade de estruturas sujeitas a oscilações. Em óptica, essas curvas também aparecem na descrição do comportamento de feixes de luz polarizados elipticamente. Do ponto de vista matemático, as figuras de Lissajous ilustram relações entre funções trigonométricas e suas combinações, fornecendo exemplos visuais de conceitos como periodicidade, fase e ressonância. A generalização dessas curvas para três dimensões permite modelar movimentos mais complexos, como trajetórias de partículas carregadas em campos eletromagnéticos ou padrões de interferência tridimensionais em ondas acústicas. Dessa forma, o estudo das figuras de Lissajous não apenas fornece uma ferramenta gráfica intuitiva para a compreensão de fenômenos oscilatórios, mas também desempenha um papel essencial em diversas aplicações científicas e tecnológicas.
Nesta subseção vamos explicar como fazer uma implementação interessante destas figuras. A implementação do programa em Python consiste em duas partes principais. Primeiramente, o código gera a figura de Lissajous em um gráfico estático, representando a trajetória completa da curva no espaço bidimensional . Esse gráfico permite visualizar a forma característica da curva, que depende das amplitudes, frequências e do deslocamento de fase entre os movimentos oscilatórios nos eixos coordenados. Em seguida, o programa exibe uma animação dinâmica de um ponto percorrendo a curva ao longo do tempo, demonstrando a evolução do movimento oscilatório de maneira interativa. Para isso, utiliza-se a biblioteca `matplotlib`, que permite tanto a plotagem da figura estática quanto a criação da animação por meio da função `FuncAnimation`. Essa abordagem facilita a compreensão do comportamento da curva de Lissajous, ilustrando de forma clara como a combinação de oscilações perpendiculares resulta nos padrões característicos dessas figuras. A seguir, detalhamos cada etapa do código e sua função dentro do programa. O primeiro passo é importar as bibliotecas necessárias:

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

A biblioteca `numpy` é usada para cálculos matemáticos e geração de arrays. A `matplotlib.pyplot` permite criar gráficos e a `matplotlib.animation` gerencia a animação.
Os parâmetros da curva de Lissajous são definidos para determinar a amplitude e a frequência da oscilação:

```
A = 1  # Amplitude no eixo x
B = 1  # Amplitude no eixo y
freq_x = 3  # Frequência no eixo x
freq_y = 2  # Frequência no eixo y
```

Esses valores determinam a forma da curva de Lissajous. Criamos uma função para calcular os pontos da curva com base no tempo:

```
def lissajous(t):
    x = A * np.sin(freq_x * t)
    y = B * np.sin(freq_y * t)
    return x, y
```

Dividimos a figura em dois subgráficos: um para a imagem estática e outro para a animação.

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 8))
```

O primeiro gráfico exibe a curva completa no plano $x - y$:

```
t = np.linspace(0, 2 * np.pi, 1000)
x_static, y_static = lissajous(t)
ax1.plot(x_static, y_static, 'b')
ax1.set_xlim(-A, A)
ax1.set_ylim(-B, B)
ax1.set_xlabel("X")
ax1.set_ylabel("Y")
ax1.set_title("Figura de Lissajous Estática")
```

O segundo gráfico exibe a animação de um ponto percorrendo a curva:

```
ax2.set_xlim(-A, A)
ax2.set_ylim(-B, B)
ax2.set_xlabel("X")
ax2.set_ylabel("Y")
ax2.set_title("Animação da Figura de Lissajous")
point, = ax2.plot([], [], 'ro', markersize=5)
```

Definimos duas funções auxiliares:

```
def init():
    point.set_data([], [])
    return point,
```

Esta função inicializa a animação sem pontos visíveis. A função de atualização move o ponto ao longo da curva:

```
def update(frame):
    t = frame * 0.05  # Ajuste do tempo
    x, y = lissajous(t)
    point.set_data(x, y)
    return point,
```

A animação é criada e exibida da seguinte forma:

```
ani = animation.FuncAnimation(fig, update, frames=200, init_func=init, blit=True, interval=30)
plt.tight_layout()
plt.show()
```

A função `FuncAnimation`, da biblioteca `matplotlib.animation`, é responsável por criar a animação ao chamar repetidamente a função `update` em intervalos regulares de tempo. Essa abordagem permite atualizar dinamicamente os elementos gráficos, proporcionando uma visualização contínua da trajetória de um ponto ao longo da curva de Lissajous. A geração da figura estática é realizada através da função `plot` da biblioteca `matplotlib`, que recebe as coordenadas x e y obtidas a partir da parametrização matemática da curva. Para a animação, a função `FuncAnimation` executa sucessivas chamadas à função `update`, responsável por modificar dinamicamente a posição do ponto animado. O método `set_data` é utilizado para atualizar as coordenadas x e y do ponto ao longo do tempo. Para garantir que a curva completa permaneça visível durante a animação, a trajetória é plotada previamente, e a cada quadro apenas a posição do ponto animado é alterada, sem necessidade de redesenhar toda a figura. Esse procedimento evita a limpeza do gráfico a cada iteração, garantindo um desempenho mais eficiente e uma transição visual mais suave. A biblioteca `matplotlib` permite a criação de visualizações interativas e didáticas, sendo amplamente utilizada para a representação de curvas paramétricas e fenômenos físicos. A separação entre a geração da curva estática e a animação facilita a compreensão do comportamento das figuras de Lissajous no plano, permitindo explorar sua dependência em relação aos parâmetros envolvidos.

4.3 Animação do Movimento de um Pêndulo Simples

O pêndulo simples é um sistema oscilatório amplamente estudado na física, pois representa um exemplo clássico de movimento harmônico para pequenas oscilações. A equação do movimento para um pêndulo ideal, assumindo um fio inextensível e sem resistência do ar, é dada por:

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\sin(\theta) = 0 \tag{2}$$

onde:

- θ é o ângulo de oscilação em relação à posição de equilíbrio,
- g é a aceleração da gravidade,
- L é o comprimento do fio.

Para pequenos ângulos ($\theta \ll 1$), podemos aproximar $\sin(\theta) \approx \theta$, o que leva a uma equação diferencial linear cuja solução exata é:

$$\theta(t) = \theta_0 \cos\left(\sqrt{\frac{g}{L}}t\right) \tag{3}$$

onde θ_0 é o ângulo inicial. Essa equação descreve um movimento harmônico simples com período:

$$T = 2\pi\sqrt{\frac{L}{g}} \tag{4}$$

Vamos apresentar um código em Python que implementa a animação do movimento de um pêndulo simples utilizando a biblioteca `matplotlib`. A implementação segue uma abordagem sistemática para representar graficamente a oscilação do pêndulo ao longo do tempo, respeitando a solução da equação diferencial do sistema. O processo de construção da animação pode ser descrito pelos seguintes passos:

1. Definição dos parâmetros físicos: Inicialmente, são estabelecidos os valores para a aceleração da gravidade g, o comprimento do fio L e o ângulo inicial θ_0. O período de oscilação do pêndulo também é calculado com base na equação $T = 2\pi\sqrt{L/g}$, garantindo que a animação tenha uma duração adequada.

2. Discretização do tempo: O tempo é discretizado em uma sequência de instantes igualmente espaçados para que a posição do pêndulo seja calculada de forma contínua. Isso permite que a animação seja fluida e bem definida.

3. Cálculo da posição do pêndulo: Utilizando a solução exata da equação do movimento para pequenos ângulos, a posição angular $\theta(t)$ do pêndulo ao longo do tempo é determinada. Em seguida, essa posição angular é convertida em coordenadas cartesianas (x, y), facilitando sua representação no gráfico.

4. Configuração do ambiente gráfico: Uma figura é criada utilizando `matplotlib`, onde são definidos os limites dos eixos, os rótulos e o título do gráfico. O pêndulo é representado por dois elementos gráficos: uma linha conectando o ponto de suspensão à massa e um ponto representando a própria massa do pêndulo.

5. Criação da função de atualização: A animação é gerada através da função `update`, que recebe um índice de quadro como entrada e ajusta a posição do pêndulo conforme o tempo avança. A linha e a massa do pêndulo são atualizadas a cada quadro para refletir a nova posição calculada.

6. Uso da função `FuncAnimation`: A animação é construída com a função `FuncAnimation` da biblioteca `matplotlib.animation`, que chama a função de atualização repetidamente para gerar a movimentação do pêndulo. O intervalo entre quadros é ajustado para manter a fluidez da animação.

   Esse método permite visualizar o comportamento oscilatório do pêndulo de maneira dinâmica, proporcionando uma ferramenta interativa para o estudo de sistemas físicos oscilatórios. O código completo é apresentado abaixo:

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

# Parâmetros físicos do pêndulo
g = 9.81   # Aceleração da gravidade (m/s²)
L = 1.0    # Comprimento do fio (m)
theta_0 = np.radians(30)  # Ângulo inicial em radianos
T = 2 * np.pi * np.sqrt(L / g)  # Período do pêndulo

# Definição do tempo
t = np.linspace(0, 2*T, 300)  # 300 quadros para 2 períodos

# Cálculo da posição angular ao longo do tempo
theta = theta_0 * np.cos(np.sqrt(g / L) * t)

# Conversão para coordenadas cartesianas
x = L * np.sin(theta)
y = -L * np.cos(theta)

# Criando a figura e o eixo
fig, ax = plt.subplots(figsize=(5, 5))
ax.set_xlim(-L, L)
ax.set_ylim(-1.1 * L, 0.1 * L)
ax.set_xlabel("X (m)")
ax.set_ylabel("Y (m)")
ax.set_title("Animação do Pêndulo Simples")

# Criando o fio e a massa do pêndulo
line, = ax.plot([], [], 'k-', lw=2)  # Fio
mass, = ax.plot([], [], 'ro', markersize=10)  # Massa

# Função de atualização da animação
def update(frame):
    line.set_data([0, x[frame]], [0, y[frame]])  # Atualiza o fio
    mass.set_data(x[frame], y[frame])  # Atualiza a posição da massa
    return line, mass

# Criando a animação
ani = animation.FuncAnimation(fig, update, frames=len(t), interval=20, blit=True)

plt.show()
```

   A seguir, explicamos cada bloco de código em detalhes. Primeiro, importamos as bibliotecas necessárias:

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

- `numpy` é utilizado para manipulação matemática e cálculo da posição do pêndulo.
- `matplotlib.pyplot` é usado para gerar os gráficos.
- `matplotlib.animation` permite criar a animação do pêndulo.

Definimos os valores físicos do pêndulo:

```
g = 9.81   # Aceleração da gravidade (m/s²)
L = 1.0    # Comprimento do fio (m)
theta_0 = np.radians(30)  # Ângulo inicial em radianos
T = 2 * np.pi * np.sqrt(L / g)  # Período do pêndulo
```

- `g` representa a aceleração da gravidade.
- `L` é o comprimento do fio do pêndulo.
- `theta_0` é o ângulo inicial do pêndulo convertido para radianos.
- `T` é o período de oscilação do pêndulo, calculado pela equação $T = 2\pi\sqrt{L/g}$.

Criamos um array de tempo e calculamos a posição angular:

```
t = np.linspace(0, 2*T, 300)  # 300 quadros para 2 períodos
theta = theta_0 * np.cos(np.sqrt(g / L) * t)
```

- O vetor `t` contém 300 pontos distribuídos ao longo de dois períodos do pêndulo.
- A função $\theta(t) = \theta_0 \cos(\sqrt{g/L} \cdot t)$ representa a solução da equação de movimento para pequenas oscilações.

Para representar a posição do pêndulo em um gráfico, convertemos a posição angular para coordenadas cartesianas:

```
x = L * np.sin(theta)
y = -L * np.cos(theta)
```

- `x` e `y` representam as coordenadas da massa do pêndulo.

Configuramos o gráfico onde a animação será exibida:

```
fig, ax = plt.subplots(figsize=(5, 5))
ax.set_xlim(-L, L)
ax.set_ylim(-1.1 * L, 0.1 * L)
ax.set_xlabel("X (m)")
ax.set_ylabel("Y (m)")
ax.set_title("Animação do Pêndulo Simples")
```

- Criamos a figura e os eixos do gráfico.
- Definimos os limites dos eixos `x` e `y`.
- Rotulamos os eixos e adicionamos um título.

Criamos os elementos gráficos que representam o pêndulo:

```
line, = ax.plot([], [], 'k-', lw=2)  # Fio
mass, = ax.plot([], [], 'ro', markersize=10)  # Massa
```

- `line` representa o fio do pêndulo, inicialmente vazio.
- `mass` representa a massa do pêndulo, exibida como um ponto vermelho.

Criamos a função que atualiza a animação a cada quadro:

```
def update(frame):
    line.set_data([0, x[frame]], [0, y[frame]])  # Atualiza o fio
    mass.set_data(x[frame], y[frame])  # Atualiza a posição da massa
    return line, mass
```

- A função `update` recebe um índice `frame` e atualiza a posição do fio e da massa.

Utilizamos `FuncAnimation` para gerar a animação:

```
ani = animation.FuncAnimation(fig, update, frames=len(t), interval=20, blit=True)
```

- `frames=len(t)` define o número total de quadros na animação.
- `interval=20` controla a velocidade da animação.
- `blit=True` melhora o desempenho da animação.

Por fim, exibimos a animação:

```
plt.show()
```

Este código cria uma animação na qual o pêndulo oscila suavemente de um lado para o outro, respeitando as equações do movimento. A biblioteca `FuncAnimation` é usada para atualizar a posição da massa do pêndulo em cada quadro, garantindo um movimento contínuo e realista. A implementação apresentada pode ser expandida para incluir efeitos como amortecimento e forças externas, tornando-se uma ferramenta útil para o ensino de sistemas oscilatórios e dinâmica de sistemas físicos.

5 Aprofundando os Conceitos de Python com Novos Exemplos

Na seção anterior, exploramos os conceitos fundamentais da programação em Python, compreendendo desde a criação de variáveis até estruturas básicas de controle de fluxo. Agora, daremos um passo adiante, aplicando esses conceitos em novos exemplos que ilustram sua utilidade em diferentes contextos. Nos próximos tópicos, apresentaremos programas que utilizam e combinam os conceitos previamente aprendidos, permitindo uma compreensão mais sólida da lógica de programação. Inicialmente, os exemplos serão simples, reforçando os fundamentos já discutidos. À medida que avançamos, os desafios aumentam gradativamente, introduzindo novas possibilidades e exigindo uma aplicação mais criativa das ferramentas que Python oferece. O objetivo desta seção é ajudar você a consolidar seus conhecimentos, desenvolvendo uma maior familiaridade com a linguagem e sua sintaxe. Além disso, exploraremos situações comuns na programação, incentivando o pensamento lógico e a capacidade de resolver problemas de forma estruturada. Com a prática contínua, será possível não apenas compreender a teoria, mas também aplicá-la de maneira eficiente na construção de programas mais complexos. Vamos, então, aprofundar nossos estudos e expandir nossas habilidades em Python com exemplos práticos um pouco mais desafiadores.

5.1 Operações Matemáticas Avançadas com Dois Números

Nesta seção, apresentamos um programa em Python que realiza operações matemáticas mais avançadas com dois números reais fornecidos pelo usuário. Vamos explorar algumas funções úteis do módulo `math`, incluindo:

- Potência: A^B,
- Exponencial: e^A, e^B,
- Logaritmo natural: $\ln A$, $\ln B$,
- Fatorial (se aplicável),
- Valor absoluto.

Para utilizar essas funções, é necessário importar o módulo `math`, que contém uma variedade de operações matemáticas.

Código Completo:

```
import math  # Importa o módulo matemático

# Solicita dois valores reais ao usuário
A = float(input("Digite o valor de A: "))
B = float(input("Digite o valor de B: "))

# Cálculo de potência A^B
potencia = math.pow(A, B)

# Exponenciais
exp_A = math.exp(A)
exp_B = math.exp(B)

# Logaritmos naturais (se positivos)
if A > 0:
    log_A = math.log(A)
else:
    log_A = "indefinido (A <= 0)"
```

```
if B > 0:
    log_B = math.log(B)
else:
    log_B = "indefinido (B <= 0)"

# Fatoriais (valores inteiros e não-negativos)
if A >= 0 and A.is_integer():
    fat_A = math.factorial(int(A))
else:
    fat_A = "indefinido (A negativo ou não inteiro)"

if B >= 0 and B.is_integer():
    fat_B = math.factorial(int(B))
else:
    fat_B = "indefinido (B negativo ou não inteiro)"


# Valor absoluto de A - B
abs_diff = abs(A - B)


# Exibe todos os resultados
print("\nRESULTADOS:")
print("A ^ B =", potencia)
print("exp(A) =", exp_A)
print("exp(B) =", exp_B)
print("ln(A) =", log_A)
print("ln(B) =", log_B)
print("fatorial de A =", fat_A)
print("fatorial de B =", fat_B)
print("|A - B| =", abs_diff)
```

Notas importantes:

- A função `math.pow(A, B)` calcula A^B.
- As funções `math.exp(x)` e `math.log(x)` calculam e^x e $\ln x$, respectivamente. Para o logaritmo, o valor de entrada deve ser positivo.
- A função `math.factorial(n)` só funciona para números inteiros não-negativos. O programa verifica essa condição.
- `abs()` é uma função nativa do Python.

Este programa é uma ótima introdução a funções matemáticas mais complexas, mostrando como usar condicionais para evitar erros e como fornecer resultados úteis e bem organizados ao usuário.

5.2 Mais exemplos sobre o uso do Comando `for` em Python

Como já debatemos, o comando `for` é uma estrutura de repetição usada para iterar sobre sequências, como listas, intervalos numéricos e strings. Ele permite executar um bloco de código múltiplas vezes, variando um contador de iteração automaticamente.

Laço de 1 a 100

Este exemplo usa um laço `for` para imprimir na tela os números de 1 a 100.

```
# Imprime os números de 1 a 100 usando um laço for
for i in range(1, 101):  # O intervalo vai de 1 até 100 (inclusive)
    print(i)  # Exibe o número atual
```

Uma breve explicação sobre o trecho de código acima:

- `range(1, 101)` gera números de 1 a 100.
- A variável `i` assume um valor de cada vez dentro desse intervalo.
- O comando `print(i)` exibe o valor atual de `i`.

5.3 Soma dos Quadrados de Números Ímpares

Este programa resolve o problema de calcular a soma simples e a soma dos quadrados de todos os números ímpares entre 1 e 100. A lógica do programa é baseada em um loop 'for' que percorre os números ímpares de 1 a 100, utilizando $range(1, 101, 2)$, que gera apenas valores ímpares. Dentro do loop, duas variáveis são atualizadas a cada iteração: "soma-simples", que acumula a soma dos números ímpares, e "soma-quadrados", que acumula a soma dos quadrados desses números. Como o loop avança sempre de 2 em 2, ele evita automaticamente os números pares, garantindo que apenas os ímpares sejam considerados. No final do processo, os resultados são exibidos com a função 'print()'.

```
# Inicializa as variáveis para armazenar as somas
soma_quadrados = 0
soma_simples = 0

# Loop para percorrer os números ímpares de 1 a 100
for i in range(1, 101, 2):  # i assume apenas valores ímpares
    soma_simples += i       # Soma simples dos números ímpares
    soma_quadrados += i ** 2  # Soma dos quadrados dos números ímpares

# Exibe os resultados
print("Soma dos quadrados dos números ímpares:", soma_quadrados)
print("Soma simples dos números ímpares:", soma_simples)
```

5.4 Soma dos Números Espaçados por 3

Este programa resolve o problema de calcular a soma de todos os números naturais entre 1 e 1000 que estão espaçados por 3, ou seja, a sequência $1, 4, 7, 10, \ldots, 1000$. A lógica do programa é baseada em um loop 'for' que percorre esses números utilizando 'range(1, 1001, 3)', que gera automaticamente os valores com espaçamento de 3. Dentro do loop, uma variável 'soma' é atualizada a cada iteração, acumulando a soma dos números sucessivos. Como o loop avança sempre de 3 em 3, ele seleciona apenas os números da sequência desejada sem a necessidade de verificações adicionais. No final do processo, o resultado total da soma é exibido com a função 'print()'.

```
# Inicializa a variável para armazenar a soma
soma = 0

# Loop para percorrer os números de 1 a 1000 com passo 3
for i in range(1, 1001, 3):
    soma += i  # Soma os valores sucessivos

# Exibe o resultado
print("Soma dos números espaçados por 3 de 1 até 1000:", soma)
```

Esses exemplos mostram como usar o comando `for` para iterar sobre sequências numéricas e listas, permitindo realizar operações repetitivas de forma eficiente.

5.5 Explicando a importância dos `Loops` (comparando C e Python)

Para explicar melhor a importância dos `loops` em programação e as potencialidades desta ferramenta vamos considerar, por exemplo, a soma dos números de 1 até 10. Antes de usar um loop, vamos primeiro tentar resolver o problema de forma "manual", sem usar qualquer estrutura de repetição, apenas fazendo cada operação separadamente. Em seguida, mostraremos como podemos simplificar esse processo usando um loop.

Soma sem usar loops

Comecemos com a abordagem de "força bruta", onde realizamos cada operação separadamente:

```
i = 1
soma = i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
soma = soma + i
i = i + 1
```

Neste caso, estamos somando $1 + 2 + 3 + 4 + 5 + \ldots + 10$ de maneira explícita. Como podemos ver, são necessárias 10 linhas para realizar essa operação.

Soma usando um loop

Agora, vamos utilizar um *loop* para realizar a mesma operação de forma mais eficiente. O loop é uma estrutura que repete um bloco de código várias vezes, economizando o trabalho de escrever várias linhas de código para a mesma operação. Vejamos como isso é feito em C e Python . Em C, podemos usar um loop `for` para iterar de 1 até 10 e somar os valores. O código em C seria o seguinte:

```
#include <stdio.h>

int main() {
    int soma = 0;

    for (int i = 1; i <= 10; i++) {
        soma += i;
    }

    printf("A soma de 1 até 10 é: %d\n", soma);
    return 0;
}
```

Aqui, o loop `for` começa com `i = 1`, e em cada iteração, a variável `i` é incrementada até atingir 10. O valor de `i` é somado à variável `soma` a cada iteração. Após o loop, a soma total é exibida.

Programa em Python

Em Python, podemos usar o loop `for` de maneira semelhante. O código em Python seria:

```
soma = 0

for i in range(1, 11):
    soma += i

print(f'A soma de 1 até 10 é: {soma}')
```

No Python, usamos a função `range(1, 11)` para gerar os números de 1 até 10. O loop `for` itera sobre esse intervalo e soma os valores de `i` à variável `soma`. O resultado é então impresso. Como podemos observar, ao usar um loop, conseguimos reduzir significativamente o número de linhas de código e aumentar a legibilidade. A estrutura de repetição `for` nos permite automatizar a tarefa de somar os números de 1 até 10 de maneira eficiente e sem a necessidade de escrever várias linhas de código repetitivas. Em resumo, loops são ferramentas poderosas para realizar tarefas repetitivas de forma eficiente em C e Python.

5.6 Usando o Comando `while` para somar números de 1 até 10

Como já foi debatido anteriormente, o comando `while` é uma estrutura de repetição em Python que permite executar um bloco de código enquanto uma condição for verdadeira. A sintaxe básica do `while` é a seguinte:

```
while condição:
    # Bloco de código a ser executado
```

Neste caso, a condição é testada antes da execução de cada iteração. Se a condição for `True`, o bloco de código dentro do `while` será executado. O comando `while` é comumente usado quando o número de iterações não é previamente conhecido e depende de uma condição dinâmica. No programa abaixo , o objetivo é somar os números de 1 até 10. Vamos analisar o código em detalhes:

```
soma = 0  # Inicializando a variável que armazenará a soma
numero = 1  # Inicializando a variável número, que começará de 1

while numero <= 10:  # Enquanto o número for menor ou igual a 10
    soma += numero  # Adiciona o valor do número à soma
    numero += 1  # Incrementa o número em 1

print("A soma dos números de 1 a 10 é:", soma)  # Exibe o resultado
```

Explicação do programa:

1. `soma = 0`: Inicializamos a variável `soma` com o valor 0. Ela será usada para armazenar a soma dos números de 1 a 10.

2. `numero = 1`: Inicializamos a variável `numero` com o valor 1, que será o primeiro número a ser somado.

3. `while numero <= 10:`: O laço `while` começa a execução enquanto a condição `numero <= 10` for verdadeira. Ou seja, enquanto o valor de `numero` for menor ou igual a 10, o bloco de código dentro do `while` será executado.

4. `soma += numero`: A cada iteração do laço, o valor de `numero` é somado à variável `soma`. A operação `+=` é uma forma compacta de escrever `soma = soma + numero`.

5. `numero += 1`: Após cada iteração, o valor de `numero` é incrementado em 1, movendo-se para o próximo número a ser somado.

6. `print("A soma dos números de 1 a 10 é:", soma)`: Após a execução do laço `while`, o valor final de `soma` (que contém a soma de 1 até 10) é impresso.

O programa termina quando `numero` chega a 11, momento em que a condição `numero <= 10` não é mais satisfeita e o laço `while` é interrompido. O resultado final da soma dos números de 1 até 10 é:

$$1 + 2 + 3 + \cdots + 10 = 55$$

Portanto, a saída do programa será:

```
A soma dos números de 1 a 10 é: 55
```

5.7 Mais exemplos sobre manipulação de Vetores

Nesta subseção vamos resolver computacionalmente o problema de calcular a soma e o produto de uma sequência de números de 1 a 20. Vamos criar um vetor que vai guardar os números. O procedimento computacional vai utilizar um loop para iterar sobre cada número do vetor e realizar as operações. A sequência é representada por um vetor (ou lista), no qual os números são armazenados em ordem crescente, começando de 1 até 20. O objetivo é demonstrar como realizar operações em um vetor utilizando um índice que varia de 1 a 20, em vez de usar funções prontas como ‘sum‘ e ‘prod‘. A lógica básica do programa é a seguinte: primeiro, o vetor é gerado contendo os números de 1 a 20. Depois, duas variáveis são inicializadas: uma para armazenar a soma e outra para o produto dos elementos. Dentro do loop, a cada iteração, o valor correspondente do vetor é somado à variável ‘soma‘ e multiplicado à variável ‘produto‘. O loop percorre o vetor de forma sequencial, e as operações são realizadas para cada elemento, até o fim do vetor. O programa utiliza a técnica de indexação do vetor para acessar seus elementos, ajustando o índice do loop para corresponder corretamente à posição no vetor (considerando que a indexação em Python começa em 0). Dessa forma, o cálculo da soma e do produto é feito manualmente, sem utilizar funções agregadas.

```
# Criação do vetor contendo os números de 1 a 20
vetor = list(range(1, 21))  # Cria uma lista de números de 1 a 20

# Inicializa as variáveis de soma e produto
soma = 0
produto = 1

# Loop para calcular a soma e o produto
for i in range(1, 21):  # O indexador i vai de 1 a 20
    soma += vetor[i - 1]  # Adiciona o valor do vetor[i-1] à soma
    produto *= vetor[i - 1]  # Multiplica o valor do vetor[i-1] ao produto

# Exibe os resultados
print("Soma dos elementos do vetor:", soma)
print("Produto dos elementos do vetor:", produto)
```

O próximo programa que vamos apresentar faz exatamente a mesma coisa que o programa anterior (ou seja: cria um vetor contendo os números de 1 a 20 e depois calcula a soma e o produto de todos os elementos desse vetor). Entretanto, para calcular a soma dos elementos do vetor, o programa utiliza a função embutida 'sum(vetor)', que automaticamente soma todos os valores da lista e armazena o resultado na variável 'soma'. Em seguida, o cálculo do produto dos elementos é feito manualmente dentro de um loop. A variável 'produto' é inicializada com o valor 1 (porque qualquer número multiplicado por 1 é ele mesmo), e o loop percorre todos os números no vetor. A cada iteração, o número atual é multiplicado pelo valor acumulado de 'produto', resultando no produto total de todos os números de 1 a 20. Por fim, os resultados da soma e do produto são exibidos usando a função 'print()', mostrando o valor da soma de todos os elementos e o produto de todos os números do vetor.

```
# Este programa cria um vetor de números de 1 a 20
#e realiza operações com ele

# Criação do vetor contendo os números de 1 a 20
# Cria uma lista de números de 1 a 20
vetor = list(range(1, 21))

# Calcula a soma dos elementos do vetor
soma = sum(vetor)

# Calcula o produto dos elementos do vetor
produto = 1  # Inicializa a variável produto
for numero in vetor:
    produto *= numero
# Multiplica cada elemento pelo produto acumulado


# Exibe os resultados
print("Soma dos elementos do vetor:", soma)
print("Produto dos elementos do vetor:", produto)
```

Uma breve explicação do código pode ser apresentada de forma resumida abaixo:

- `range(1, 21)` gera uma sequência de números inteiros de 1 a 20.
- `list(range(1, 21))` converte essa sequência em uma lista.
- `sum(vetor)` calcula a soma de todos os elementos do vetor.
- O laço `for` percorre o vetor multiplicando todos os seus elementos para calcular o produto total.

5.8 Mais informações sobre o uso de Condicionais em Python

Em Python, utilizamos os comandos condicionais para executar diferentes blocos de código dependendo de uma condição lógica. O comando `if` permite avaliar expressões e decidir qual código será executado com base em seu resultado.

Estrutura do Comando `if`

A estrutura básica de um condicional em Python é:

```
if condicao:
# Código executado se a condição for verdadeira
```

```
elif outra_condicao:
# Código executado se a primeira condição for falsa, mas esta for verdadeira
else:
# Código executado se nenhuma das condições anteriores for verdadeira
```

Agora, vejamos mais alguns exemplos práticos utilizando esse conceito.

Exemplo: Encontrando o Maior, o Menor e a Média de Quatro Números

Este programa solicita ao usuário que insira quatro números, identifica o maior e o menor deles e calcula a média.

```
# Solicita quatro números ao usuário
num1 = float(input("Digite o primeiro número: "))
num2 = float(input("Digite o segundo número: "))
num3 = float(input("Digite o terceiro número: "))
num4 = float(input("Digite o quarto número: "))

# Calcula a média dos números
media = (num1 + num2 + num3 + num4) / 4

# Determina o maior número
maior = num1  # Assume que num1 é o maior inicialmente
if num2 > maior:
    maior = num2
if num3 > maior:
    maior = num3
if num4 > maior:
    maior = num4

# Determina o menor número
menor = num1  # Assume que num1 é o menor inicialmente
if num2 < menor:
    menor = num2
if num3 < menor:
    menor = num3
if num4 < menor:
    menor = num4

# Exibe os resultados
print("Maior número:", maior)
print("Menor número:", menor)
print("Média dos números:", media)
```

Vamos debater um pouco os detalhes do código :

- Os valores são lidos e convertidos para números decimais com `float(input())`.
- A média é calculada somando os quatro números e dividindo o resultado por 4.
- Para encontrar o maior número, começamos assumindo que o primeiro número é o maior e depois comparamos com os outros três usando `if`.
- O mesmo procedimento é aplicado para encontrar o menor número.
- O programa imprime o maior e o menor número, além da média calculada.

O uso do `if` é essencial para tomada de decisões em programas. No caso acima, usamos apenas `if` encadeados para encontrar o maior e o menor número. Alternativamente, poderíamos usar as funções `max()` e `min()`, que fazem essa comparação automaticamente:

```
maior = max(num1, num2, num3, num4)
menor = min(num1, num2, num3, num4)
```

Esse método é mais eficiente e reduz a quantidade de código. Os comandos condicionais `if`, `elif` e `else` são fundamentais para controlar o fluxo de execução de um programa. Eles permitem criar lógicas de decisão, tornando os programas mais dinâmicos e interativos.

5.9 Ordenando três valores usando comandos "if"

Este programa apresenta uma abordagem simples para ordenar três números fornecidos pelo usuário em ordem crescente. O método consiste em solicitar os três valores e armazená-los em variáveis (‘x, y, z‘). Em seguida, utiliza uma sequência de comandos ‘if‘ para comparar e trocar os valores sempre que necessário, garantindo que os números fiquem organizados corretamente. A lógica empregada pode ser vista como uma versão simplificada do **Bubble Sort**, onde realizamos trocas diretas entre os valores até que a ordenação seja alcançada. Como estamos lidando com apenas três números, o número de comparações necessárias é reduzido, tornando desnecessário o uso de loops. No final, os números ordenados são exibidos na tela.

```
# Solicita os três números ao usuário
x = float(input("Digite o primeiro número: "))
y = float(input("Digite o segundo número: "))
z = float(input("Digite o terceiro número: "))

# Algoritmo de ordenação manual usando if
if x > y:
    x, y = y, x
if y > z:
    y, z = z, y
if x > y:
    x, y = y, x

# Exibe os números ordenados
print("Números em ordem crescente:", x, y, z)
```

5.10 Exemplo: Ordenando quatro valores usando comandos "if"

Este programa resolve o problema de ordenar quatro números fornecidos pelo usuário em ordem crescente. A lógica do programa começa solicitando quatro valores numéricos e armazenando-os em variáveis individuais (‘a, b, c, d‘). Em seguida, utiliza uma série de comparações ‘if‘ para organizar os números, trocando-os de posição sempre que um valor maior aparece antes de um menor. Esse método simula uma versão simplificada do **Bubble Sort**, onde as trocas são feitas repetidamente até que os números estejam na ordem correta. Como há apenas quatro números, o programa realiza um número adequado de comparações para garantir a ordenação, sem precisar de loops. Por fim, os números organizados são exibidos na tela. Esse método é simples e eficiente para poucos valores, servindo como um bom exercício para entender a lógica de ordenação manual.

```
# Solicita os quatro números ao usuário
a = float(input("Digite o primeiro número: "))
b = float(input("Digite o segundo número: "))
```

```
c = float(input("Digite o terceiro número: "))
d = float(input("Digite o quarto número: "))

# Algoritmo de ordenação manual usando if
if a > b:
    a, b = b, a
if b > c:
    b, c = c, b
if c > d:
    c, d = d, c
if a > b:
    a, b = b, a
if b > c:
    b, c = c, b
if a > b:
    a, b = b, a

# Exibe os números ordenados
print("Números em ordem crescente:", a, b, c, d)
```

5.11 Ordenação de Números em Python(funções internas)

Em Python, ordenar uma pequena quantidade de números, como 3 ou 4, pode ser feito de forma mais simples utilizando listas e o método `sort()`. O seguinte código exemplifica como ordenar três números dados pelo usuário.

```
def ordenar_numeros(a, b, c):
    lista = [a, b, c]
    lista.sort()  # Ordena a lista em ordem crescente
    return lista

# Solicita três números ao usuário
num1 = float(input("Digite o primeiro número: "))
num2 = float(input("Digite o segundo número: "))
num3 = float(input("Digite o terceiro número: "))

# Chama a função para ordenar
numeros_ordenados = ordenar_numeros(num1, num2, num3)

print("Números ordenados:", numeros_ordenados)
```

A função `ordenar_numeros` recebe três números como entrada, armazena-os em uma lista e utiliza o método `sort()` para ordená-los. A lista ordenada é então retornada e exibida ao usuário. Caso seja necessário ordenar quatro números, a função pode ser adaptada facilmente, conforme mostrado no seguinte exemplo:

```
def ordenar_numeros(a, b, c, d):
    lista = [a, b, c, d]
    lista.sort()  # Ordena a lista em ordem crescente
    return lista

# Solicita quatro números ao usuário
num1 = float(input("Digite o primeiro número: "))
num2 = float(input("Digite o segundo número: "))
num3 = float(input("Digite o terceiro número: "))
num4 = float(input("Digite o quarto número: "))
```

```
# Chama a função para ordenar
numeros_ordenados = ordenar_numeros(num1, num2, num3, num4)

print("Números ordenados:", numeros_ordenados)
```

A única diferença nesse segundo programa é a inclusão de um quarto número na lista. O método `sort()` continua funcionando da mesma forma, ordenando todos os elementos da lista em ordem crescente. Ordenar números é algo que se torna necessário em diversas atividades da física computacional. Por exemplo, supomos que exista uma dada coleção de dados aleatórios guardados em um dado vetor $v[]$. Em Python, isso pode ser feito de forma simples utilizando a função `sort()` ou a função embutida `sorted()`. A seguir, apresentamos um exemplo de código que cria um vetor com 20 números aleatórios e os ordena.

```
import random

def ordenar_vetor(v):
    v.sort()  # Ordena o vetor in-place (modifica o próprio vetor)
    return v

# Criando um vetor com 20 números aleatórios entre 0 e 100
v = [random.randint(0, 100) for _ in range(20)]

print("Vetor original:", v)

# Chamando a função para ordenar o vetor
ordenar_vetor(v)

print("Vetor ordenado:", v)
```

O código inicia gerando um vetor `v` com 20 números aleatórios entre 0 e 100 usando a função `random.randint`. Em seguida, a função `ordenar_vetor()` utiliza o método `sort()` para ordenar o vetor diretamente. Por fim, o programa exibe o vetor original e o vetor ordenado.

5.12 Ordenando um vetor e mantendo a correspondência com outro vetor

Suponha que temos dois vetores $x[i = 1, ..., 5]$ e $y[i = 1, ..., 5]$, onde cada par (x_i, y_i) representa uma relação entre os valores de x e y. Nosso objetivo é ordenar x do menor para o maior e reorganizar y de modo a manter a relação inicial entre os pares. Isso pode ser feito em Python utilizando a função 'zip()' para criar pares (x, y), ordenar com base em x, e depois separar novamente os vetores ordenados. O seguinte programa implementa esse processo:

```
import numpy as np

# Gerando dois vetores de exemplo
x = np.random.rand(5)  # Vetor x com 20 valores aleatórios
y = np.random.rand(5)  # Vetor y com 20 valores aleatórios

# Exibindo os vetores antes da ordenação
print("Vetor x original :", x)
print("Vetor y original :", y)


# Ordenando x e reorganizando y para manter a correspondência
```

```
# A função zip(x, y) cria um iterador que combina
#elementos das listas x e y em tuplas.
# O primeiro elemento de x é combinado com o primeiro de y,
# o segundo de x com o segundo de y, e assim  por diante.
# Exemplo: zip([1, 2, 3], [4, 5, 6]) resulta em
#[(1, 4), (2, 5), (3, 6)].


# Cria pares (x, y) e ordena pelo primeiro elemento (x)
pares_ordenados = sorted(zip(x, y))


# Separa os vetores ordenados
x_ordenado, y_ordenado = zip(*pares_ordenados)

# Exibindo os resultados
print("Vetor x ordenado:", x_ordenado)
print("Vetor y reorganizado:", y_ordenado)
```

Neste código, utilizamos a função 'zip()' para agrupar os vetores, a função 'sorted()' para ordenar os pares com base em x, e finalmente 'zip(*)' para separar os vetores ordenados. Assim, garantimos que y seja reorganizado corretamente, mantendo a relação inicial com x.

5.13 Exemplo de Programa em Python para Cálculo de Funções Trigonométricas

Neste exemplo, vamos criar um programa em Python que calcula as funções trigonométricas seno, cosseno e tangente para valores de θ variando de 0 até 2π. O programa irá:

- Calcular as funções seno, cosseno e tangente para cada valor de θ,
- Salvar os resultados em três arquivos de dados: `seno.dat`, `cos.dat` e `tg.dat`.

Passo 1: Importação dos Módulos Necessários

Primeiro, vamos importar os módulos necessários. O módulo `math` é utilizado para calcular o seno, cosseno e tangente, e o módulo `numpy` será usado para gerar os valores de θ. Também utilizaremos o módulo `matplotlib` para visualizar os gráficos posteriormente, embora a parte de visualização não seja parte do exemplo aqui.

```
import math  # Importa o módulo math para funções trigonométricas
import numpy as np  # Importa o numpy para gerar valores de theta
```

Passo 2: Definição do Intervalo de θ

A variável θ será variada de 0 até 2π. Para fazer isso de forma eficiente, usaremos a função `np.linspace()` para gerar valores de θ igualmente espaçados ao longo desse intervalo.

```
# Definir o intervalo de theta de 0 até 2*pi com 100 pontos
theta = np.linspace(0, 2 * math.pi, 100)
```

Neste código, a função `np.linspace(0, 2 * math.pi, 100)` gera 100 valores igualmente espaçados entre 0 e 2π. O número 100 é arbitrário e pode ser ajustado conforme necessário.

Passo 3: Cálculo das Funções Trigonométricas

Agora, vamos calcular as funções trigonométricas seno, cosseno e tangente para cada valor de θ gerado. Usaremos os métodos `math.sin()`, `math.cos()` e `math.tan()` para esses cálculos.

```
# Cálculo das funções trigonométricas
seno = np.sin(theta)  # Calcula o seno de cada valor de theta
coseno = np.cos(theta)  # Calcula o cosseno de cada valor de theta
tangente = np.tan(theta)  # Calcula a tangente de cada valor de theta
```

Aqui, `np.sin()`, `np.cos()` e `np.tan()` são funções vetorizadas que operam diretamente sobre o array `theta`, o que significa que as funções trigonométricas serão calculadas para todos os valores de θ simultaneamente.

Passo 4: Salvando os Resultados em Arquivos de Dados

Agora que calculamos as funções, vamos salvar os resultados em três arquivos separados. Cada arquivo conterá duas colunas: a primeira será o valor de θ, e a segunda será o valor da função trigonométrica correspondente.

```
# Salvando os resultados nos arquivos de dados
np.savetxt("seno.dat", np.column_stack((theta, seno)), header="theta, seno", fmt="%.6f")
np.savetxt("cos.dat", np.column_stack((theta, coseno)), header="theta, cosseno", fmt="%.6f")
np.savetxt("tg.dat", np.column_stack((theta, tangente)), header="theta, tangente", fmt="%.6f")
```

A função `np.savetxt()` é usada para salvar os dados em um arquivo de texto. A função `np.column_stack()` empilha as colunas de θ e a função trigonométrica correspondente. O parâmetro `header` define um cabeçalho para o arquivo de dados, e `fmt="%.6f"` especifica o formato numérico (6 casas decimais).

Passo 5: Visualização dos Resultados (Opcional)

Embora o foco não seja a visualização aqui, podemos opcionalmente plotar as funções usando a biblioteca `matplotlib`. O código para plotar as três funções seria algo assim:

```
# Criando a figura com 3 subgráficos empilhados (3 linhas e 1 coluna)
fig, axs = plt.subplots(3, 1, figsize=(7, 8))  # 3 linhas, 1 coluna

# Plotando a função seno no primeiro subgráfico
axs[0].plot(theta, seno, label='Seno', color='blue')
#axs[0].set_title('Função Seno')
axs[0].set_xlabel(' (radianos)')
axs[0].set_ylabel('Valor')
axs[0].legend()

# Plotando a função cosseno no segundo subgráfico
axs[1].plot(theta, coseno, label='Cosseno', color='green')
#axs[1].set_title('Função Cosseno')
axs[1].set_xlabel(' (radianos)')
axs[1].set_ylabel('Valor')
axs[1].legend()

# Plotando a função tangente no terceiro subgráfico
axs[2].plot(theta, tangente, label='Tangente', color='red')
#axs[2].set_title('Função Tangente')
axs[2].set_xlabel(' (radianos)')
axs[2].set_ylabel('Valor')
# Limitar o eixo y para evitar valores muito grandes na tangente
axs[2].set_ylim(-10, 10)
axs[2].legend()

# Ajustar o layout para evitar sobreposição de labels
plt.tight_layout()

# Exibindo os gráficos
plt.show()
```

Neste código, usamos `plt.plot()` para criar os gráficos de cada função. A legenda é adicionada com `plt.legend()` e os rótulos dos eixos com `plt.xlabel()` e `plt.ylabel()`.

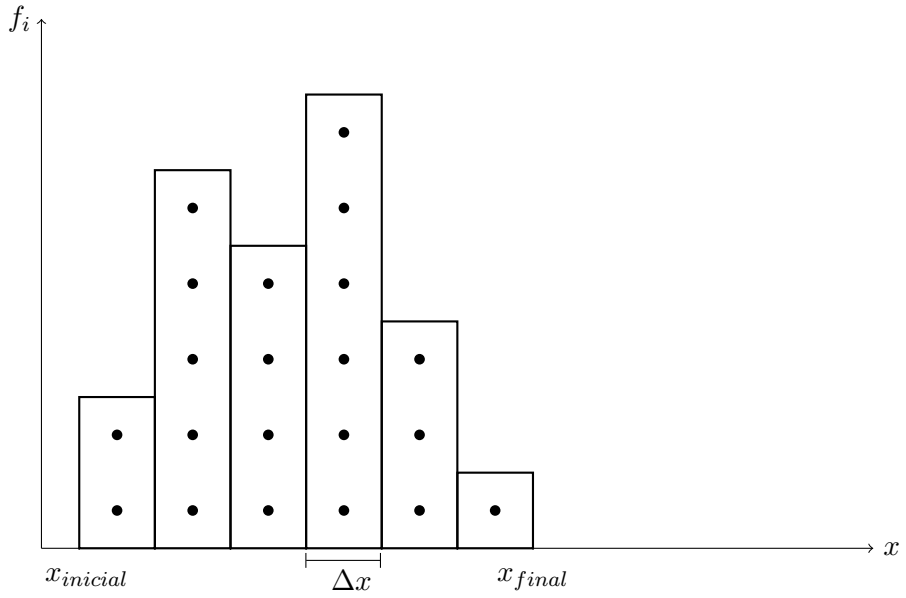
Agora vamos juntar todas estas etapas e apresentar o código completo para a solução numérica do problema em tela:

```
# Importa o módulo math para funções trigonométricas
import math

# Importa o numpy para gerar valores de theta
import numpy as np

# Importa o matplotlib para plotar os gráficos
import matplotlib.pyplot as plt

# Definir o intervalo de theta de 0 até 2*pi com 100 pontos
theta = np.linspace(0, 2 * math.pi, 100)

# Cálculo das funções trigonométricas
seno = np.sin(theta)  # Calcula o seno de cada valor de theta
coseno = np.cos(theta)  # Calcula o cosseno de cada valor de theta
tangente = np.tan(theta)  # Calcula a tangente de cada valor de theta

# Salvando os resultados nos arquivos de dados
np.savetxt("seno.dat", np.column_stack((theta, seno)), header="theta, seno", fmt="%.6f")
np.savetxt("cos.dat", np.column_stack((theta, coseno)), header="theta, cosseno", fmt="%.6f")
np.savetxt("tg.dat", np.column_stack((theta, tangente)), header="theta, tangente", fmt="%.6f")

# Criando a figura com 3 subgráficos empilhados (3 linhas e 1 coluna)
fig, axs = plt.subplots(3, 1, figsize=(7, 8))  # 3 linhas, 1 coluna

# Plotando a função seno no primeiro subgráfico
axs[0].plot(theta, seno, label='Seno', color='blue')
#axs[0].set_title('Função Seno')
axs[0].set_xlabel(' (radianos)')
axs[0].set_ylabel('Valor')
axs[0].legend()

# Plotando a função cosseno no segundo subgráfico
axs[1].plot(theta, coseno, label='Cosseno', color='green')
#axs[1].set_title('Função Cosseno')
axs[1].set_xlabel(' (radianos)')
axs[1].set_ylabel('Valor')
axs[1].legend()

# Plotando a função tangente no terceiro subgráfico
axs[2].plot(theta, tangente, label='Tangente', color='red')
#axs[2].set_title('Função Tangente')
axs[2].set_xlabel(' (radianos)')
axs[2].set_ylabel('Valor')
# Limitar o eixo y para evitar valores muito grandes na tangente
axs[2].set_ylim(-10, 10)
axs[2].legend()

# Ajustar o layout para evitar sobreposição de labels
plt.tight_layout()

# Exibindo os gráficos
plt.show()
```

Este programa calcula as funções seno, cosseno e tangente para valores de θ variando de 0 a 2π, e salva os resultados em arquivos de dados que podem ser usados para análise posterior. A visualização gráfica também é opcional, mas útil para verificar a forma das funções trigonométricas.

6 Cálculo do Histograma Normalizado

O histograma é uma ferramenta estatística usada para representar a distribuição de uma sequência de números. Em linhas gerais um **histograma** é uma representação gráfica da distribuição de um conjunto de dados. Ele consiste em dividir o intervalo dos valores em subintervalos (chamados de "bins" ou "caixas") e contar quantos valores pertencem a cada um desses subintervalos. Matematicamente, se tivermos um conjunto de dados $\{x_1, x_2, x_3, \ldots, x_N\}$ e dividirmos o intervalo $[x_{inicial}, x_{final}]$ em caixas de largura Δx, então a frequência absoluta f_i de cada caixa é o número de pontos que "caem" dentro de cada caixa. A seguir, ilustramos um histograma simples:



Cada bolinha acima de uma caixa representa a contagem dos valores de x que pertencem àquele intervalo. A altura de cada caixa corresponde à frequência absoluta f_i, que define o histograma. O histograma normalizado é obtido dividindo a contagem de cada *bin* pelo número total de elementos, garantindo que a soma das alturas das barras seja 1. Para calcular um histograma normalizado, seguimos os seguintes passos:

(a) Definir o número de *bins* desejado, que determina a resolução do histograma.

(b) Contar quantos valores pertencem a cada *bin*.

(c) Dividir a contagem de cada *bin* pelo número total de valores para obter a distribuição normalizada.

(d) Salvar os dados do histograma em um arquivo para análise posterior.

Abaixo, apresentamos um programa em Python que gera um conjunto de 10^6 números aleatórios utilizando dois métodos: um gerador congruente linear (LCG) e a função `random.uniform(0, 1)` do Python. Os histogramas normalizados são calculados e salvos nos arquivos `GCv1.dat` (para o gerador congruente) e `RUv1.dat` (para a função `random.uniform(0, 1)`).

```
import random  # Importa o módulo para geração de números aleatórios

# Parâmetros do gerador congruente linear (LCG)
m = 2**32
a = 1664525
c = 1013904223
X0 = 42  # Semente inicial
```

```
# Definir número de amostras e bins
N = 1000000  # Número de números aleatórios
bins = 50  # Número de intervalos (bins)

# Função para gerar números aleatórios usando LCG
def lcg(n, seed):
    X = seed  # Inicializa com a semente
    numeros = []  # Lista para armazenar os números gerados
    for _ in range(n):
        X = (a * X + c) % m  # Atualiza o valor de X usando a fórmula do LCG
        numeros.append(X / m)  # Normaliza para o intervalo [0,1]
    return numeros

# Gerar números aleatórios com LCG
lcg_numbers = lcg(N, X0)


# Gerar números aleatórios com random.uniform

random.seed(1) #semente 1

random_numbers = [random.uniform(0, 1) for _ in range(N)]

# Criar os limites dos bins manualmente
min_value = 0.0  # Valor mínimo da distribuição
max_value = 1.0  # Valor máximo da distribuição
bin_width = (max_value - min_value) / bins  # Largura de cada bin

# Limites dos bins
bin_edges = [min_value + i * bin_width for i in range(bins + 1)]

# Inicializar os histogramas com zeros
hist_lcg = [0] * bins
hist_random = [0] * bins

# Contagem dos números dentro de cada bin (frequência absoluta)
for number in lcg_numbers:
# Determina em qual bin o número pertence
    bin_index = int((number - min_value) / bin_width)
    if bin_index == bins:  # Para evitar erro com números no limite superior
        bin_index -= 1
    hist_lcg[bin_index] += 1  # Incrementa a contagem no bin correspondente

for number in random_numbers:
    bin_index = int((number - min_value) / bin_width)
    if bin_index == bins:
        bin_index -= 1
    hist_random[bin_index] += 1

# Normalização dos histogramas
# Normaliza pelo total de amostras e largura do bin
for i in range(bins):
    hist_lcg[i] /= (N * bin_width)
    hist_random[i] /= (N * bin_width)

# Salvar histogramas nos arquivos
with open("GCv1.dat", "w") as f_lcg:
    for i in range(bins):
```

```
        f_lcg.write(f"{bin_edges[i]} {hist_lcg[i]}\n")

with open("RUv1.dat", "w") as f_random:
    for i in range(bins):
        f_random.write(f"{bin_edges[i]} {hist_random[i]}\n")
```

Se faz relevante salientar que Python tem funções internas para calcular histogramas de forma eficiente, como a 'np.histogram()' da biblioteca NumPy. Essa função divide os dados em intervalos (bins), conta quantos valores pertencem a cada bin e, opcionalmente, normaliza os resultados. No programa a seguir, utilizaremos essas funções internas para calcular os histogramas das sequências de números aleatórios geradas pelos dois métodos geradores que já foram considerados (o Gerador Congruente Linear (LCG) e a função 'random.uniform()' do próprio Python). Assim, o código em questão demonstra como calcular e armazenar histogramas de maneira automatizada, sem a necessidade de implementar manualmente a contagem e normalização dos dados.

```
# Importa o módulo 'random', que fornece funções
#para geração de números aleatórios.
import random

# Importa a biblioteca NumPy, que oferece suporte para
#operações matemáticas e manipulação de arrays.
import numpy as np

# Parâmetros do gerador congruente linear (LCG)
m = 2**32
a = 1664525
c = 1013904223
X0 = 42

# Definir a semente do gerador
random.seed(X0)

# Função para gerar números aleatórios usando LCG
def lcg(n, seed):
    X = seed
    numeros = []
    for _ in range(n):
        X = (a * X + c) % m
        numeros.append(X / m)  # Normaliza para [0,1]
    return numeros

# Definir número de amostras e bins
N = 1000000
bins = 50

# Gerar números aleatórios com LCG

lcg_numbers = lcg(N, X0)


# Gerar números aleatórios com  random.uniform
random.seed(1) #semente 1
random_numbers = [random.uniform(0, 1) for _ in range(N)]
# Gera uma lista de 'N' números aleatórios no intervalo [0,1]
# usando a função random.uniform(0, 1).
# O loop for cria cada número e os armazena na lista 'random_numbers'.
# O '_' indica que a variável de iteração não será usada.
```

```
# Calcular histogramas normalizados
hist_lcg, bin_edges = np.histogram(lcg_numbers, bins=bins, density=True)
hist_random, bin_edges = np.histogram(random_numbers, bins=bins, density=True)

# A função np.histogram calcula o histograma de um conjunto de dados.
# - O primeiro argumento é o conjunto de números aleatórios gerados.
# - O parâmetro 'bins' define o número de intervalos (classes) no histograma.
# - O parâmetro 'density=True' normaliza o histograma,
# garantindo que a soma das áreas das barras seja igual a 1.
# - A função retorna dois valores:
#   1) hist_lcg (ou hist_random) -> Um array com os valores
# normalizados do histograma (densidade de probabilidade).
#   2) bin_edges -> Um array contendo os limites dos intervalos (bins),
# ou seja, os pontos que delimitam as classes.

# Salvar histogramas nos arquivos
np.savetxt("GCv2.dat", np.column_stack((bin_edges[:-1], hist_lcg)))
np.savetxt("RUv2.dat", np.column_stack((bin_edges[:-1], hist_random)))

# A função np.savetxt salva os dados em arquivos de texto no formato de colunas.
# - O primeiro argumento é o nome do arquivo onde os dados serão armazenados.
# - A função np.column_stack() combina dois arrays colunares em um
# único array bidimensional,
#   garantindo que cada linha do arquivo contenha os valores
# correspondentes dos dois arrays.
# - bin_edges[:-1] representa os limites inferiores de
#  cada intervalo (bin) do histograma,
#   pois o array bin_edges tem um elemento a mais do que
#   hist_lcg e hist_random (delimitando os bins).
# - hist_lcg e hist_random contêm as densidades normalizadas do histograma.
# - Assim, cada linha do arquivo salvo conterá o limite inferior
# do bin e a densidade correspondente.
```

Este código anterior primeiro define o gerador congruente linear e sua função de geração. Em seguida, gera 10^6 números aleatórios com ambos os métodos e calcula os histogramas normalizados usando a função `numpy.histogram()` com a opção `density=True`, garantindo que a soma das áreas das barras seja igual a 1. Finalmente, os histogramas são salvos nos arquivos `GCv2.dat` e `RUv2.dat`, com cada linha contendo o limite inferior do *bin* e sua respectiva densidade. Os resultados dos dois programas são os mesmos (mantendo obviamente a mesma fonte de dados).

7 Solução numérica de alguns problemas de mecânica clássica

Em diversos problemas da física clássica, nos deparamos com equações diferenciais ordinárias (EDOs) que descrevem a evolução temporal de sistemas físicos. Um exemplo clássico é o movimento de uma partícula sob a ação de uma força: a segunda lei de Newton, $F = ma$, pode ser reescrita como uma equação diferencial de segunda ordem envolvendo a posição da partícula em função do tempo. Em muitos casos, essas equações não admitem uma solução analítica simples, o que nos leva a buscar soluções numéricas.

O método de Euler é uma das abordagens mais simples e intuitivas para resolver numericamente EDOs. Trata-se de um método explícito e de primeira ordem, o que significa que a precisão do resultado depende diretamente do tamanho do passo de tempo escolhido para a simulação. Mesmo sendo um método básico, ele é extremamente útil para introduzir o conceito de integração numérica e fornecer uma visão clara de como os computadores tratam a evolução de sistemas físicos em pequenos intervalos de tempo.

A ideia central do método de Euler é aproximar a derivada de uma função pela razão de incrementos. Dado um ponto inicial conhecido, o próximo valor da função é estimado utilizando a inclinação da curva naquele ponto. Essa abordagem transforma uma equação diferencial contínua em uma sequência de iterações discretas que podem ser facilmente implementadas em Python.

Ao longo desta seção, exploraremos como aplicar o método de Euler para resolver problemas clássicos como o sistema massa-mola, movimento uniformemente acelerado, o pêndulo simples e a queda livre com resistência do ar. Além de apresentar os algoritmos em Python, discutiremos as limitações do método e como ele pode ser aprimorado. Essa base fornecerá ao leitor as ferramentas necessárias para lidar com sistemas dinâmicos simples e entender os fundamentos da simulação computacional em física.

7.1 Exemplo de Aplicação do Método de Euler na Equação $\frac{d^2x}{dt^2} = -kx$

Nesta seção, resolveremos numericamente a equação diferencial

$$\frac{d^2x}{dt^2} = -kx \tag{5}$$

utilizando o método de Euler. Esta equação descreve um oscilador harmônico simples, como uma massa $m = 1$ presa a uma mola com constante elástica k. Vamos considerar que no tempo $t = 0$ a posição e velocidade da massa são dados por : $x(t = 0) = 1$ e $v(t = 0) = 0$. Para resolver essa equação numericamente, devemos reescrevê-la como um sistema de duas equações de primeira ordem. Definimos a velocidade v como:

$$v = \frac{dx}{dt}. \tag{6}$$

Substituindo essa relação na equação do movimento, obtemos:

$$\frac{dv}{dt} = -kx. \tag{7}$$

Agora, aplicamos o método de Euler para aproximar as derivadas. A definição de derivada de uma função $f(t)$ é dada por:

$$\frac{df}{dt} \approx \frac{f(t+\Delta t) - f(t)}{\Delta t}. \tag{8}$$

Aplicando essa aproximação à equação da velocidade:

$$\frac{dx}{dt} \approx \frac{x_{n+1} - x_n}{\Delta t}, \tag{9}$$

e rearranjando, obtemos a equação de atualização para x:

$$x_{n+1} = x_n + v_n \Delta t. \tag{10}$$

De forma análoga, aplicamos a definição de derivada à equação da aceleração:

$$\frac{dv}{dt} \approx \frac{v_{n+1} - v_n}{\Delta t}. \tag{11}$$

Reorganizando a equação, temos:

$$v_{n+1} = v_n - k x_n \Delta t. \tag{12}$$

Dessa forma, as equações do método de Euler permitem calcular iterativamente os valores de x e v a partir das condições iniciais, fornecendo uma aproximação numérica para a solução da equação do oscilador harmônico. O método de Euler é de primeira ordem, ou seja, introduz um erro proporcional a Δt. Para obter uma boa precisão, Δt deve ser pequeno, mas isso aumenta o número de passos necessários para atingir um tempo final T. A escolha adequada de Δt é essencial para equilibrar precisão e eficiência computacional. Agora, implementamos este método em Python. Primeiro, definimos os parâmetros do problema:

```
k = 1.0        # Constante da mola
m = 1.0        # Massa (assumida como 1 para simplificar)
dt = 0.1       # Passo de tempo
T = 10.0       # Tempo total
N = int(T/dt)  # Número de passos
x0 = 1.0       # Condição inicial de x
v0 = 0.0       # Condição inicial de v

t_values = []  # Lista para armazenar os valores de t
x_values = []  # Lista para armazenar os valores de x
v_values = []  # Lista para armazenar os valores de v
```

Criamos um loop para calcular a evolução de x e v ao longo do tempo e armazenamos os valores:

```
x = x0
v = v0
for i in range(N):
    t = i * dt
    t_values.append(t)
    x_values.append(x)
    v_values.append(v)

    # Atualiza x e v pelo método de Euler
    x = x + v * dt
    v = v - k * x * dt
```

Agora, salvamos os resultados em um arquivo chamado ‘massamola.dat‘, onde cada linha contém o valor de t, x e v:

```
with open("massamola.dat", "w") as file:
    for t, x, v in zip(t_values, x_values, v_values):
        file.write(f"{t} {x} {v}\n")
```

Aqui está o código completo para a simulação e salvamento dos dados:

```
k = 1.0        # Constante da mola
m = 1.0        # Massa

dt = 0.1       # Passo de tempo
T = 10.0       # Tempo total
N = int(T/dt)  # Número de passos

x0 = 1.0       # Condição inicial de x
v0 = 0.0       # Condição inicial de v

t_values = []  # Lista para armazenar os valores de t
x_values = []  # Lista para armazenar os valores de x
v_values = []  # Lista para armazenar os valores de v

x = x0
v = v0
for i in range(N):
    t = i * dt
    t_values.append(t)  # Armazena o instante de tempo atual na lista t_values
    x_values.append(x)  # Armazena a posição x do oscilador no instante atual
    v_values.append(v)  # Armazena a velocidade v do oscilador no instante atual


    # Atualiza x e v pelo método de Euler
    x = x + v * dt
    v = v - k * x * dt

# Abre um arquivo chamado "massamola.dat" no modo de escrita ("w")
with open("massamola.dat", "w") as file:
    for t, x, v in zip(t_values, x_values, v_values):
# Percorre simultaneamente as listas de tempo (t_values),
# posição (x_values) e velocidade (v_values)
        file.write(f"{t} {x} {v}\n")
# Escreve no arquivo os valores de tempo, posição e
#velocidade, separados por um espaço, em cada linha
```

Portanto, apresentamos como implementar o método de Euler para resolver numericamente a equação diferencial $\frac{d^2x}{dt^2} = -kx$. O código fornecido imprime os valores de x e v ao longo do tempo e os armazena no arquivo ‘massamola.dat‘, permitindo visualizar a solução aproximada e usá-la para análises posteriores. A visualização deste arquivo de dados pode ser feita usando aplicativos do tipo ”xmgrace” (ou equivalente). Entretanto, se faz importante salientar que o Python possui bibliotecas especializadas para a geração de gráficos, sendo a ‘matplotlib‘ uma das mais populares. Dentro dela, a subbiblioteca ‘pyplot‘ fornece funções simples e intuitivas para criar visualizações. A função ‘plt.figure(figsize=(largura, altura))‘ define o tamanho do gráfico, enquanto ‘plt.plot(x, y, label=”Nome”, color=”cor”)‘ traça uma curva baseada nos dados fornecidos. Para melhorar a interpretação, ‘plt.xlabel(”Nome”)‘ e ‘plt.ylabel(”Nome”)‘ adicionam rótulos aos eixos, e ‘plt.title(”Título”)‘ insere um título descritivo. A função ‘plt.legend()‘ exibe a legenda das curvas, e

‘plt.grid()‘ adiciona uma grade ao fundo do gráfico. Finalmente, ‘plt.show()‘ exibe o gráfico gerado. Com essas funções, é possível criar representações gráficas eficientes e personalizadas de maneira prática. O código a seguir use boa parte destas funções para gerar o gráfico na tela do computador.

```
import numpy as np


# Importa a biblioteca necessária para gráficos
import matplotlib.pyplot as plt



# Parâmetros do sistema
k = 1.0        # Constante da mola
m = 1.0        # Massa
dt = 0.1       # Passo de tempo
T = 10.0       # Tempo total
N = int(T/dt)  # Número de passos

# Condições iniciais
x0 = 1.0
v0 = 0.0

# Listas para armazenar os valores
t_values = []
x_values = []
v_values = []

x = x0
v = v0

# Método de Euler para resolver as equações diferenciais
for i in range(N):
    t = i * dt
    t_values.append(t)
    x_values.append(x)
    v_values.append(v)

    # Atualização pelo método de Euler
    x = x + v * dt
    v = v - k * x * dt

# Salva os dados em um arquivo
with open("massamola.dat", "w") as file:
    for t, x, v in zip(t_values, x_values, v_values):
        file.write(f"{t} {x} {v}\n")


# Geração do gráfico


# Cria uma nova figura com tamanho 8x5 polegadas
plt.figure(figsize=(8,5))

# Plota x(t) versus t, com cor azul e rótulo para a legenda
plt.plot(t_values, x_values, label="x(t)", color="blue")

# Plota v(t) versus t, com cor vermelha e rótulo para a legenda
plt.plot(t_values, v_values, label="v(t)", color="red")
```

```
# Adiciona rótulo ao eixo x indicando que representa o tempo
plt.xlabel("Tempo (t)")

# Adiciona rótulo ao eixo y indicando que representa os valores de x e v
plt.ylabel("Valores de x e v")

# Define o título do gráfico para indicar que representa
#o oscilador harmônico resolvido pelo método de Euler
plt.title("Oscilador Harmônico - Método de Euler")

# Adiciona uma legenda ao gráfico, identificando
# as curvas de x(t) e v(t)
plt.legend()

# Adiciona uma grade ao fundo do gráfico para facilitar
# a leitura dos valores
plt.grid()

# Exibe o gráfico na tela
plt.show()
```

7.2 Resolução Numérica do Oscilador Harmônico Amortecido $\frac{d^2x}{dt^2} = -kx - bv$

O oscilador harmônico simples com amortecimento proporcional à velocidade é descrito pela equação diferencial:

$$\frac{d^2x}{dt^2} = -kx - bv, \tag{13}$$

onde b é o coeficiente de amortecimento e $v = \frac{dx}{dt}$ é a velocidade. A introdução do termo $-bv$ adiciona um efeito dissipativo ao sistema, reduzindo a amplitude das oscilações ao longo do tempo. Para resolver essa equação numericamente, utilizamos o método de Euler, que aproxima as derivadas por diferenças finitas:

$$x_{n+1} = x_n + v_n \Delta t, \tag{14}$$
$$v_{n+1} = v_n + (-kx_n - bv_n)\Delta t. \tag{15}$$

A seguir, apresentamos um código em Python que implementa essa solução numérica.

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

# Parâmetros do sistema
k = 1.0        # Constante da mola
b = 0.2        # Coeficiente de amortecimento
m = 1.0        # Massa
dt = 0.1       # Passo de tempo
T = 10.0       # Tempo total
N = int(T/dt)  # Número de passos
```

```
# Condições iniciais
x0 = 1.0
v0 = 0.0

# Listas para armazenar os valores
t_values = []
x_values = []
v_values = []

x = x0
v = v0

# Método de Euler para resolver as equações diferenciais
for i in range(N):
    t = i * dt
    t_values.append(t)
    x_values.append(x)
    v_values.append(v)

    # Atualização pelo método de Euler
    x = x + v * dt
    v = v - (k * x + b * v) * dt

# Salva os dados em um arquivo
with open("oscilador_amortecido.dat", "w") as file:
    for t, x, v in zip(t_values, x_values, v_values):
        file.write(f"{t} {x} {v}\n")

# Geração do gráfico
plt.figure(figsize=(8,5))
plt.plot(t_values, x_values, label="x(t)", color="blue")
plt.plot(t_values, v_values, label="v(t)", color="red")
plt.xlabel("Tempo (t)")
plt.ylabel("Valores de x e v")
plt.title("Oscilador Harm\^onico Amortecido - M\'etodo de Euler")
plt.legend()
plt.grid()
plt.show()
```

A introdução do termo de amortecimento resulta na redução progressiva da amplitude das oscilações, como esperado para um oscilador amortecido. O método de Euler fornece uma solução aproximada, adequada para valores pequenos de Δt.

7.3 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.

Equação de Movimento

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

$$m\frac{dv}{dt} = mg - kv, \tag{16}$$

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.

Solução Analítica

Separando as variáveis:

$$\frac{dv}{mg - kv} = \frac{dt}{m}. \tag{17}$$

Integramos ambos os lados:

$$\int \frac{dv}{mg - kv} = \int \frac{dt}{m}. \tag{18}$$

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

$$-\frac{1}{k}\int \frac{du}{u} = \frac{t}{m} + C. \tag{19}$$

A solução da integral é:

$$-\frac{1}{k}\ln|mg - kv| = \frac{t}{m} + C. \tag{20}$$

Isolando v:

$$v(t) = \frac{mg}{k}\left(1 - e^{-\frac{k}{m}t}\right). \tag{21}$$

A velocidade terminal é dada por:

$$v_T = \frac{mg}{k}. \tag{22}$$

Resolução Numérica

Para resolver a equação 16 numericamente, vamos aplicar o método de Euler. Este é um método simples e direto, baseado na aproximação das derivadas por diferenças finitas.

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

$$\frac{dv}{dt} = g - \frac{k}{m}v. \tag{23}$$

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

$$\frac{dv}{dt} = g - \beta v. \tag{24}$$

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.

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

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
```

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. O programa em python que efetua os passos descritos anteriormente está incluído abaixo.

```
# Importação das bibliotecas necessárias

# Importa o numpy para trabalhar com arrays e realizar cálculos numéricos
import numpy as np

# Importa o matplotlib.pyplot para criar gráficos
import matplotlib.pyplot as plt


# Importa o matplotlib.animation para criar animações
import matplotlib.animation as animation

# Parâmetros físicos do problema

m = 1.0  # Define a massa do objeto em quilogramas (kg)

# Define a aceleração gravitacional em metros por segundo ao quadrado (m/s²)
g = 9.81

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


# Cria um array de tempos, que vai de 0 até t_max, com intervalos de dt
t_values = np.arange(0, t_max, dt)

# Cria um array para armazenar as velocidades em cada instante de tempo
v_values = np.zeros_like(t_values)

# Define a velocidade inicial como 0 metros por segundo (m/s)
v = 0.0


# 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.
# Calcula a variação de velocidade para o intervalo de tempo d
    dv = (m * g - k * v) / m * dt  t

# Atualiza a velocidade somando a variação calculada à velocidade anterior
    v += dv

# Armazena a nova velocidade no array v_values para o tempo correspondente
    v_values[i] = v

# 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

# Define o limite do eixo Y (velocidade), ajustado
# para incluir a velocidade terminal
ax.set_ylim(0, 1.1 * (m * g / k))


ax.set_xlabel("Tempo (s)")  # Define o rótulo do eixo X como "Tempo (s)"

# Define o rótulo do eixo Y como "Velocidade (m/s)"
ax.set_ylabel("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
# Cria a linha de pontos azuis conectados por uma linha
# (bo-), com espessura de linha 2
line, = ax.plot([], [], "bo-", lw=2)

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

# 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
# ‘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
```

7.4 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.

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:

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

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(thet

# 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.

7.5 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 $y = 20\,\text{m}$ e uma posição inicial $x = 0\,\text{m}$, com velocidade inicial de translação zero, velocidade de rotação inicial diferente de zero , aceleração da gravidade g e a força de Magnus.

Equações de Movimento

O movimento da bola é governado pelas seguintes equações diferenciais de segunda ordem, considerando as forças envolvidas:

- **Força gravitacional**: A aceleração devido à gravidade é constante e atua na direção y, com valor $g = 9.81\,\text{m/s}^2$.
- **Força de Magnus**: A força de Magnus é uma força perpendicular à direção da velocidade da bola e é proporcional à sua velocidade e ao vetor de rotação. Sua fórmula é dada por:

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

  onde:
  - ω é o vetor de rotação da bola, que neste caso está na direção z, com valor constante $\omega = 1\,\text{rad/s}$,
  - $\mathbf{v} = (v_x, v_y)$ é a velocidade da bola,
  - ρ é a densidade do ar ($\rho = 1.225\,\text{kg/m}^3$),
  - S é a área da seção transversal da bola, dada por $S = \pi r^2$ com $r = 0.1\,\text{m}$,
  - $\|\mathbf{v}\|$ é o módulo da velocidade.
- **Força de arrasto**: 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,
  - $A = \pi r^2$ é a área da seção transversal.

As equações de movimento podem ser expressas em termos de acelerações nas direções x e y:

$$\frac{d^2x}{dt^2} = \frac{F_{Dx}}{m} + \frac{F_{Mx}}{m}$$
$$\frac{d^2y}{dt^2} = \frac{F_{Dy}}{m} + \frac{F_{My}}{m} - g$$

onde: - m é a massa da bola, - F_{Dx} e F_{Dy} são as componentes da força de arrasto, - F_{Mx} e F_{My} são as componentes da força de Magnus.

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.

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.

Aqui está a implementação numérica do método de Euler, utilizando o código a seguir:

```
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()
```

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.

7.6 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.

Implementação Computacional

A implementação numérica do modelo é realizada no Python, utilizando a biblioteca `matplotlib` para a visualização das trajetórias e a animação da simulação. O código a seguir mostra como a simulação é realizada para diferentes valores do coeficiente de resistência c, e como as trajetórias são animadas.

```
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

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.

7.7 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. 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.

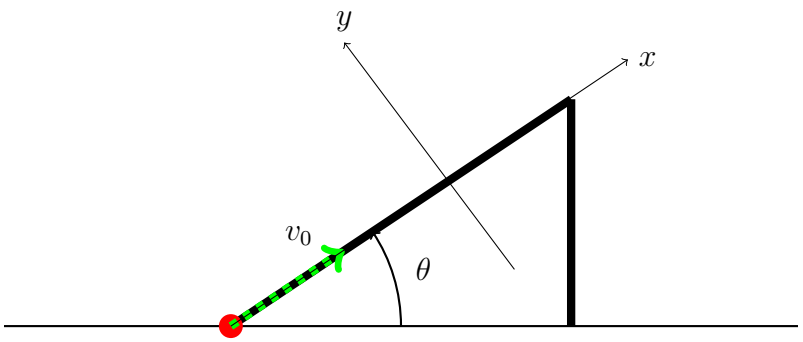


Figure 1: Esquema do movimento do corpo em um plano inclinado com resistência do ar. O corpo parte com velocidade inicial v_0 na base do plano. O eixo x está paralelo ao plano inclinado e o eixo y é perpendicular ao plano. O ângulo θ é o ângulo de inclinação do plano.

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.

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.

8 Derivação e integração numérica

A derivada e a integração numérica são ferramentas fundamentais na análise de dados e resolução de problemas científicos e de engenharia. Quando a função que descreve um fenômeno não é conhecida de forma analítica, mas seus valores estão disponíveis em pontos discretos (como em experimentos ou simulações), métodos numéricos tornam-se essenciais. A derivada numérica permite estimar a taxa de variação de uma grandeza com base nesses valores discretos, enquanto a integração numérica fornece uma maneira de calcular áreas, volumes ou quantidades acumuladas mesmo sem uma expressão exata da função. Esses métodos aproximam o comportamento da função usando operações simples com os dados disponíveis. Embora envolvam aproximações, são extremamente eficazes e amplamente utilizados na prática. Técnicas como diferenças finitas e regras de integração, como os métodos dos trapézios ou de Simpson, são exemplos comuns. A precisão dessas abordagens depende da densidade dos pontos e da suavidade da função. Assim, derivar e integrar numericamente são processos centrais na modelagem e análise de sistemas reais, especialmente quando apenas dados discretos estão disponíveis.

8.1 Derivação Numérica: Aproximação de Derivadas com Diferenças Finitas

A derivação numérica é uma técnica utilizada para aproximar a derivada de uma função quando sua forma analítica não está disponível, ou quando lidamos com dados experimentais discretos. Em vez de usar o conceito de limite como no cálculo diferencial, usamos aproximações baseadas em diferenças finitas.

Fórmulas Básicas de Diferença Finita

Seja $f(x)$ uma função suave e h um pequeno incremento, temos as seguintes fórmulas de diferenças finitas:

- Derivada para frente:
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$
- Derivada para trás:
$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$
- Derivada centrada:
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

A derivada centrada é geralmente mais precisa, pois cancela erros de ordem $O(h)$, levando a um erro de ordem $O(h^2)$.

Exemplo com função analítica: derivada de $\sin(x)$

Considere a função $f(x) = \sin(x)$. Vamos calcular numericamente sua derivada usando a fórmula centrada:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Vamos aplicar essa fórmula em todo o intervalo $x \in [0, 2\pi]$, e comparar com a derivada exata $f'(x) = \cos(x)$. Abaixo está um programa em Python que faz isso, plota o gráfico da derivada numérica e da derivada exata na mesma figura:

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

# Parâmetros
h = 0.01
x = np.arange(0, 2*np.pi, h)

# Função original e derivada exata
f = np.sin(x)
df_exact = np.cos(x)

# Derivada numérica centrada
df_numeric = (np.sin(x + h) - np.sin(x - h)) / (2 * h)

# Plotagem
plt.plot(x, df_numeric, label='Derivada Numérica (centrada)', linestyle='--')
plt.plot(x, df_exact, label='Derivada Exata (cos(x))', color='black')
plt.xlabel('x')
plt.ylabel('f\'(x)')
plt.title('Derivada de sin(x): numérica vs. exata')
plt.legend()
plt.grid(True)
plt.show()
```

Esse programa mostra visualmente a boa concordância entre a derivada numérica e a derivada analítica da função $\sin(x)$. A precisão da aproximação depende do valor de h, sendo que valores muito pequenos podem introduzir erro numérico devido à precisão finita dos cálculos.

Exemplo com tabela de dados

Agora vamos aplicar a derivada centrada a uma tabela com dados experimentais:

x	$f(x)$
0.0	1.0
0.2	1.221
0.4	1.491
0.6	1.822
0.8	2.225
1.0	2.718
1.2	3.320

Vamos calcular a derivada centrada para todos os pontos intermediários (exceto as bordas), e usar a derivada para frente e para trás nas extremidades.

Código em Python (com os dados fixos):

```
# Derivação numérica de uma tabela de dados

# Lista dos valores de x e f(x)
x_vals = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2]
f_vals = [1.0, 1.221, 1.491, 1.822, 2.225, 2.718, 3.320]
```

```
# Tamanho do passo h (supõe igualmente espaçado)
h = x_vals[1] - x_vals[0]

# Lista para armazenar as derivadas
derivadas = []

# Derivada para frente no primeiro ponto
df_forward = (f_vals[1] - f_vals[0]) / h
derivadas.append(df_forward)

# Derivada centrada nos pontos internos
for i in range(1, len(x_vals) - 1):
    df_central = (f_vals[i+1] - f_vals[i-1]) / (2 * h)
    derivadas.append(df_central)

# Derivada para trás no último ponto
df_backward = (f_vals[-1] - f_vals[-2]) / h
derivadas.append(df_backward)

# Mostrar os resultados
print("x\tf(x)\tf'(x) (aprox)")
for i in range(len(x_vals)):
    print(f"{x_vals[i]:.1f}\t{f_vals[i]:.3f}\t{derivadas[i]:.3f}")
```

Código em Python (lendo os dados de um arquivo):

Suponha que temos um arquivo chamado `tabela.dat` com o seguinte conteúdo:

```
0.0  1.0
0.2  1.221
0.4  1.491
0.6  1.822
0.8  2.225
1.0  2.718
1.2  3.320
```

O programa abaixo lê esse arquivo, calcula as derivadas numéricas, e exibe os resultados:

```
# Derivação numérica de dados lidos de um arquivo

# Abrir o arquivo e ler os dados
x_vals = []
f_vals = []

with open("tabela.dat", "r") as f:
    for linha in f:
        if linha.strip() == "":
            continue  # Ignorar linhas vazias
        x_str, fx_str = linha.split()
        x_vals.append(float(x_str))
        f_vals.append(float(fx_str))

# Verificar se há pelo menos 3 pontos
if len(x_vals) < 3:
    print("Erro: são necessários pelo menos 3 pontos ")
```

```
    exit()

# Supondo que o espaçamento seja constante
h = x_vals[1] - x_vals[0]

# Lista para armazenar as derivadas
derivadas = []

# Derivada para frente no primeiro ponto
df_forward = (f_vals[1] - f_vals[0]) / h
derivadas.append(df_forward)

# Derivada centrada nos pontos internos
for i in range(1, len(x_vals) - 1):
    df_central = (f_vals[i+1] - f_vals[i-1]) / (2 * h)
    derivadas.append(df_central)

# Derivada para trás no último ponto
df_backward = (f_vals[-1] - f_vals[-2]) / h
derivadas.append(df_backward)

# Mostrar os resultados
print("x\tf(x)\tf'(x) (aprox)")
for i in range(len(x_vals)):
    print(f"{x_vals[i]:.1f}\t{f_vals[i]:.3f}\t{derivadas[i]:.3f}")
```

Esse programa é bastante útil em análises experimentais, onde os dados são obtidos por medições e armazenados em arquivos. Ele facilita a obtenção de derivadas aproximadas sem precisar conhecer a função analítica $f(x)$. A derivação numérica, embora sujeita a erros associados à discretização e ruído experimental, é uma ferramenta poderosa em análise computacional e científica, sendo amplamente utilizada em simulações numéricas, análise de dados experimentais e métodos de elementos finitos.

8.2 Integração Numérica: Aproximação de Integrais com Somatórios

A integração numérica é uma técnica fundamental para estimar a área sob uma curva quando a função não pode ser integrada analiticamente ou quando lidamos com dados discretos. Ao invés de usar integrais definidas analíticas, utilizamos somatórios que aproximam a área.

Métodos Clássicos de Integração Numérica

Seja $f(x)$ uma função contínua definida no intervalo $[a, b]$, que queremos integrar numericamente. Os principais métodos são:

- **Regra do Retângulo:**

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{n-1} f(x_i)$$

  onde $h = \frac{b-a}{n}$ e $x_i = a + ih$. É uma aproximação simples com erro de ordem $O(h)$.

- **Regra do Trapézio:**

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)\right]$$

  Este método é mais preciso do que o retângulo, com erro de ordem $O(h^2)$.

- **Regra de Simpson:**

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4\sum_{\text{ímpares}} f(x_i) + 2\sum_{\text{pares}} f(x_i) + f(x_n)\right]$$

Requer que n seja par. É um método de ordem superior, com erro de ordem $O(h^4)$, baseado em aproximações quadráticas.

Exemplo com função analítica: comparação de métodos de integração

Vamos calcular numericamente a integral da função $f(x) = e^x$ no intervalo $[0, 1]$, utilizando três métodos básicos de integração numérica: o retângulo (ponto médio), o trapézio e a regra de Simpson. Para isso, dividimos o intervalo em $n = 5$ subintervalos igualmente espaçados.

O valor exato da integral é $\int_0^1 e^x dx = e - 1 \approx 1{,}71828$, que usaremos como referência para comparar as aproximações.

Código em Python :

```
# Integração numérica de f(x) = exp(x) no intervalo [0,1]
# Usando os métodos: Retângulo (ponto médio), Trapézio e Simpson

import math

# Número de subdivisões
n = 5
a = 0
b = 1
h = (b - a) / n

# Método do retângulo (ponto médio)
soma_ret = 0
for i in range(n):
    xi = a + i*h
    x_meio = xi + h/2
    soma_ret += math.exp(x_meio)
integral_ret = h * soma_ret

# Método do trapézio
soma_trap = 0.5 * (math.exp(a) + math.exp(b))
for i in range(1, n):
    xi = a + i*h
    soma_trap += math.exp(xi)
integral_trap = h * soma_trap

# Método de Simpson
soma_simp = math.exp(a) + math.exp(b)
for i in range(1, n):
    xi = a + i*h
    if i % 2 == 0:
        soma_simp += 2 * math.exp(xi)
    else:
        soma_simp += 4 * math.exp(xi)
integral_simp = (h / 3) * soma_simp
```

```
# Resultados
print(f"Integral (retângulo) = {integral_ret:.5f}")
print(f"Integral (trapézio)  = {integral_trap:.5f}")
print(f"Integral (Simpson)   = {integral_simp:.5f}")
```

Esse programa mostra como aplicar os três métodos de forma simples, apenas usando laços `for` e funções básicas. Os resultados obtidos se aproximam bem do valor exato da integral. A escolha do número de subdivisões n influencia diretamente a precisão da aproximação. Métodos mais elaborados como Simpson geralmente oferecem maior precisão com menos subdivisões.

Exemplo com tabela de dados

Considere a seguinte tabela de dados obtida experimentalmente (ou por avaliação de uma função conhecida):

x	$f(x)$
0.0	1.0
0.2	1.221
0.4	1.491
0.6	1.822
0.8	2.225
1.0	2.718
1.2	3.320

Vamos aplicar os métodos de integração numérica diretamente a esses dados. Para isso, o conteúdo da tabela foi salvo em um arquivo chamado `tabela.dat`, com duas colunas separadas por espaço.

Código em Python para integrar os dados com três métodos:

```
# Leitura dos dados do arquivo e integração numérica usando 3 métodos

# Lê os dados do arquivo
x_vals = []
f_vals = []
with open("tabela.dat", "r") as arq:
    for linha in arq:
        x, fx = map(float, linha.strip().split())
        x_vals.append(x)
        f_vals.append(fx)

n = len(x_vals) - 1
h = x_vals[1] - x_vals[0]

# Método do Retângulo (ponto médio)
soma_ret = 0
for i in range(n):
    x_meio = (x_vals[i] + x_vals[i+1]) / 2
    f_meio = (f_vals[i] + f_vals[i+1]) / 2
    soma_ret += f_meio
integral_ret = h * soma_ret

# Método do Trapézio
soma_meio = 0
for i in range(1, n):
```

```
    soma_meio += 2 * f_vals[i]
soma_total = f_vals[0] + soma_meio + f_vals[-1]
integral_trap = (h / 2) * soma_total

# Método de Simpson (exige número par de subintervalos)
if n % 2 == 0:
    soma_simp = f_vals[0] + f_vals[-1]
    for i in range(1, n):
        if i % 2 == 0:
            soma_simp += 2 * f_vals[i]
        else:
            soma_simp += 4 * f_vals[i]
    integral_simp = (h / 3) * soma_simp
else:
    integral_simp = None

# Impressão dos resultados
print(f"Integral (Retângulo) = {integral_ret:.5f}")
print(f"Integral (Trapézio)  = {integral_trap:.5f}")
if integral_simp is not None:
    print(f"Integral (Simpson)   = {integral_simp:.5f}")
else:
    print("Método de Simpson requer número par de subintervalos.")
```

Esse programa mostra como usar integração numérica com dados tabulados lidos de um arquivo, algo comum em análises experimentais ou simulações. O método do retângulo (ponto médio) é simples e rápido, mas pode apresentar maior erro. A regra do trapézio já melhora consideravelmente a precisão, especialmente quando os dados são suaves. O método de Simpson, quando aplicável (número par de subintervalos), geralmente fornece a melhor aproximação entre os três. Em casos com poucos pontos ou dados ruidosos, métodos mais simples podem ser preferíveis por sua robustez.

9 Problemas gerais de Física contemporânea

A física contemporânea lida com uma ampla gama de fenômenos, desde sistemas clássicos relativamente simples até estruturas extremamente complexas envolvendo muitos corpos, interações não lineares e comportamentos emergentes. A crescente demanda por métodos eficazes de análise e modelagem levou ao uso intensivo de ferramentas computacionais, que hoje se tornaram essenciais tanto na pesquisa quanto no ensino da física. A possibilidade de explorar numericamente equações e sistemas que não admitem solução analítica abre portas para a investigação de problemas reais com maior profundidade e flexibilidade.

Nesta seção, abordaremos temas fundamentais da física por meio de uma abordagem computacional intuitiva e didática. O objetivo é apresentar exemplos simples, mas cientificamente relevantes, que podem ser utilizados para ilustrar conceitos importantes e desenvolver uma compreensão mais profunda da dinâmica dos sistemas físicos. A resolução numérica de equações diferenciais, por exemplo, permite simular a evolução temporal de sistemas clássicos como o pêndulo, a partícula sujeita a forças variadas ou mesmo sistemas não lineares com comportamento caótico.

Além dos sistemas mecânicos, será explorado o estudo da propagação de temperatura em meios contínuos, um problema clássico da física do estado sólido e da termodinâmica, frequentemente modelado por equações diferenciais parciais. Ao simular a difusão de calor, pode-se visualizar como condições iniciais e de contorno afetam a evolução térmica do sistema, proporcionando uma excelente oportunidade de conexão entre teoria e fenômeno físico observado.

Também incluiremos o modelo SIR, utilizado para descrever a dinâmica de epidemias. Apesar de sua simplicidade, esse modelo captura de forma eficaz a interação entre indivíduos suscetíveis, infectados e recuperados, revelando como parâmetros biológicos e sociais afetam a propagação de doenças. Esse tipo de abordagem é especialmente útil para conectar a física com outras áreas do conhecimento, como a biologia e a epidemiologia.

Outro tema de destaque é o uso de simulações estocásticas baseadas em métodos como a integração de Monte Carlo. Tais métodos são indispensáveis para o estudo de sistemas complexos onde o comportamento coletivo surge da interação entre elementos simples. Essa abordagem é amplamente aplicada em física estatística, termodinâmica de sistemas de muitos corpos, e na análise de transições de fase.

Se faz importante salientar que o estudo de sistemas dinâmicos discretos, como o mapa logístico e o mapa de Lozi, é não apenas relevante, mas fundamental dentro do contexto da física contemporânea. Esses modelos, embora matematicamente simples, capturam com notável fidelidade aspectos essenciais de sistemas reais que apresentam comportamento não linear e sensibilidade às condições iniciais. O mapa logístico, por exemplo, tem aplicações diretas em modelos populacionais e em sistemas ecológicos, onde o crescimento depende de recursos limitados. Sua estrutura revela uma transição gradual da ordem ao caos por meio de bifurcações sucessivas, um comportamento observado em diversos fenômenos físicos, como osciladores forçados e circuitos eletrônicos não lineares. O mapa de Lozi, por sua vez, por ser bidimensional e possuir atratores estranhos, permite estudar a geometria e a dinâmica de sistemas caóticos de forma mais visual e concreta, com aplicações que vão desde a modelagem de redes neurais até o comportamento de plasmas confinados. O estudo desses mapas fornece, assim, uma janela conceitual e computacional para compreender como sistemas determinísticos podem exibir dinâmicas imprevisíveis e complexas, fenômeno esse que está presente em áreas tão distintas quanto a climatologia, a física de partículas e a neurociência. Explorar tais modelos é, portanto, um passo importante para desenvolver a intuição sobre o caos e suas implicações em sistemas naturais e tecnológicos do mundo real.

Adicionalmente, serão apresentados exemplos envolvendo autômatos celulares, ferramentas computacionais simples que, apesar de sua definição discreta e regras locais, são capazes de reproduzir dinâmicas surpreendentemente ricas. Tais modelos são valiosos tanto para a física quanto para áreas como computação, biologia e teoria da complexidade, e ajudam a ilustrar como padrões e estruturas emergem de interações elementares.

Ao integrar esses temas em uma abordagem unificada, buscamos evidenciar como a simulação numérica não é apenas uma ferramenta auxiliar, mas uma extensão natural do pensamento físico. Com ela, podemos explorar sistemas além das limitações da análise matemática tradicional, realizar experimentos computacionais e, sobretudo, desenvolver uma intuição mais refinada sobre os mecanismos que governam os fenômenos naturais. Por meio desses exemplos, o leitor será gradualmente introduzido ao raciocínio computacional aplicado à física, ao mesmo tempo em que consolida conceitos teóricos fundamentais.

9.1 Vibração de uma corda com extremidades fixas: solução numérica da equação de onda

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).

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:

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

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.

9.2 Equação de Langevin, Movimento Browniano e Método Numérico de Heun

A equação de Langevin descreve a dinâmica de uma partícula sujeita a forças determinísticas e estocásticas. Originalmente, foi formulada para modelar o *movimento browniano*, fenômeno observado por Robert Brown ao estudar o movimento aleatório de partículas suspensas em um fluido. A

equação de Langevin para a velocidade $v(t)$ de uma partícula de massa m sujeita a um termo de dissipação γ e a um ruído aleatório $\eta(t)$ é dada por:

$$m\frac{dv}{dt} = -\gamma v + \eta(t), \tag{25}$$

onde:

- γ representa o coeficiente de amortecimento,
- $\eta(t)$ é um termo estocástico com média zero e correlação

  $$\langle \eta(t)\eta(t')\rangle = 2D\delta(t-t')$$

  , onde D é a intensidade do ruído.

A posição da partícula $x(t)$ obedece a:

$$\frac{dx}{dt} = v(t). \tag{26}$$

Integrando a equação de Langevin, obtemos a evolução da posição da partícula. Estatisticamente, isso leva à relação:

$$\langle x^2(t)\rangle = 2Dt, \tag{27}$$

mostrando que o deslocamento quadrático médio cresce linearmente com o tempo, característica fundamental do movimento browniano. O método de Heun melhora a precisão do método de Euler. O procedimento iterativo para resolver as equações diferenciais é o seguinte:

(a) Geração do ruído gaussiano:

$$\eta_n = \sqrt{2D}\frac{\xi_n}{\sqrt{\Delta t}}, \quad \text{com} \quad \xi_n \sim N(0,1). \tag{28}$$

(b) Predição de v^* e x^*:

$$v^* = v_n + \Delta t\left(-\frac{\gamma}{m}v_n + \frac{\eta_n}{m}\right), \tag{29}$$

$$x^* = x_n + \Delta t v_n. \tag{30}$$

(c) Correção:

$$v_{n+1} = v_n + \frac{\Delta t}{2}\left(f_n + f^*\right), \tag{31}$$

$$x_{n+1} = x_n + \frac{\Delta t}{2}(v_n + v^*), \tag{32}$$

onde f_n e f^* são as acelerações calculadas nos instantes t_n e t^*, respectivamente. Essas acelerações são definidas como:

$$f_n = -\frac{\gamma}{m}v_n + \frac{\eta_n}{m}, \tag{33}$$

$$f^* = -\frac{\gamma}{m}v^* + \frac{\eta^*}{m}, \tag{34}$$

onde η_n e η^* são realizações diferentes do termo estocástico em tempos consecutivos.

O seguinte código implementa a solução numérica da equação de Langevin usando o método de Heun e calcula o desvio quadrático médio (MSD):

```
import numpy as np
import random

# Parâmetros
N_PART = 1000
N_STEPS = 10000
DT = 0.01
GAMMA = 1.0
D = 1.0
M = 1.0

# Função para gerar número gaussiano
def gaussiano():
    u1 = random.random()
    u2 = random.random()
    return np.sqrt(-2.0 * np.log(u1)) * np.cos(2.0 * np.pi * u2)

# Inicialização
x = np.zeros(N_PART)
v = np.zeros(N_PART)
x0 = np.zeros(N_PART)
MSD = np.zeros(N_STEPS)

# Loop principal de simulação
for n in range(N_STEPS):
    for i in range(N_PART):
        eta_n = np.sqrt(2.0 * D) * gaussiano() / np.sqrt(DT)
        v_star = v[i] + DT * (-GAMMA / M * v[i] + eta_n / M)
        x_star = x[i] + DT * v[i]
        eta_star = np.sqrt(2.0 * D) * gaussiano() / np.sqrt(DT)
        f_n = -GAMMA / M * v[i] + eta_n / M
        f_star = -GAMMA / M * v_star + eta_star / M
        v[i] = v[i] + 0.5 * DT * (f_n + f_star)
        x[i] = x[i] + 0.5 * DT * (v[i] + v_star)

    # Calcular o MSD
    sum_dx = np.sum((x - x0)**2)
    MSD[n] = sum_dx / N_PART

# Salvar resultados em arquivo
np.savetxt("msd.dat", np.column_stack((np.arange(N_STEPS) * DT, MSD)))
```

A equação de Langevin fornece um modelo robusto para o estudo do movimento browniano. A implementação numérica via método de Heun melhora a precisão na solução da equação, permitindo a análise detalhada do desvio quadrático médio da trajetória das partículas.

9.3 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.

Equações do Modelo SIR

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

$$\frac{dS}{dt} = -\beta \cdot S \cdot I \tag{35}$$

$$\frac{dI}{dt} = \beta \cdot S \cdot I - \gamma \cdot I \tag{36}$$

$$\frac{dR}{dt} = \gamma \cdot I \tag{37}$$

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. 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. Abaixo segue um código em Python que simula a dinâmica do modelo SIR utilizando o método de Euler e gera uma animação para ilustrar a evolução da epidemia.

```
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.

9.4 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.

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. 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.

9.5 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:

$$x_{n+1} = 1 - a|x_n| + y_n, \tag{38}$$

$$y_{n+1} = bx_n. \tag{39}$$

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

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.

9.6 Modelo de Neurônio Integrate-and-Fire: Formulação e Simulação com o Método de Euler

Do ponto de vista fisiológico, o funcionamento de um neurônio pode ser descrito como um processo dinâmico de integração e resposta a estímulos. Neurônios recebem sinais elétricos ou químicos de outras células, os quais se manifestam como correntes de entrada que modificam o potencial elétrico ao longo da membrana celular. Esse potencial, chamado *potencial de membrana*, evolui continuamente no tempo em resposta aos estímulos recebidos. Quando essa evolução atinge um valor limiar bem definido — denominado *potencial de disparo* ou *threshold* — o neurônio gera um *potencial de ação* ou *spike*, que consiste em uma resposta elétrica rápida e transitória. Este sinal é então transmitido a outros neurônios por meio das sinapses. Logo após o disparo, o neurônio entra em um estado de repolarização, em que o potencial de membrana é resetado para um valor de repouso, impedindo disparos imediatos subsequentes. Esse ciclo de integração e disparo é a base da comunicação neuronal. O modelo **Integrate-and-Fire** abstrai esse comportamento de forma elegante: ele ignora os detalhes moleculares e biofísicos do spike real e foca na dinâmica da integração do potencial e no mecanismo de disparo/reset, permitindo simulações computacionais simples, eficientes e biologicamente plausíveis.

Equação Diferencial do Modelo

O potencial de membrana $V(t)$ evolui segundo a equação:

$$\tau \frac{dV}{dt} = -V(t) + RI(t)$$

ou, equivalentemente:

$$\frac{dV}{dt} = -\frac{V(t)}{\tau} + \frac{RI(t)}{\tau}$$

Onde:

- $V(t)$ é o potencial de membrana em função do tempo.
- τ é a constante de tempo do neurônio (produto da resistência e da capacitância).
- R é a resistência da membrana.
- $I(t)$ é a corrente de entrada aplicada ao neurônio.

Regra de Disparo

Quando o potencial $V(t)$ atinge ou ultrapassa um limiar V_{th}, consideramos que o neurônio gerou um disparo (*spike*). Imediatamente após o disparo, o potencial é resetado para um valor mais baixo V_{reset}, simulando o comportamento biológico observado.

Solução Numérica via Método de Euler

A equação diferencial pode ser integrada numericamente utilizando o método de Euler explícito. Discretizando o tempo em passos de tamanho Δt, temos:

$$V_{n+1} = V_n + \Delta t \left(-\frac{V_n}{\tau} + \frac{RI_n}{\tau} \right)$$

Após a atualização, verificamos se $V_{n+1} \geq V_{\text{th}}$. Se isso ocorrer, o valor de V_{n+1} é resetado para V_{reset} e registramos o tempo de disparo. Abaixo apresentamos um programa em Python que implementa o modelo Integrate-and-Fire usando o método de Euler.

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

# Parâmetros do modelo
tau = 10.0           # constante de tempo (ms)
R = 1.0              # resistência da membrana (MOhm)
V_th = 1.0           # limiar de disparo (V)
V_reset = 0.0        # valor de reset após o spike (V)
I_const = 1.5        # corrente de entrada constante (uA)

# Parâmetros da simulação
T_total = 100        # tempo total da simulação (ms)
dt = 0.1             # passo de tempo (ms)
N = int(T_total / dt)  # número total de passos

# Vetores para tempo e potencial
t = np.linspace(0, T_total, N)
V = np.zeros(N)

# Lista para registrar os tempos de disparo
spike_times = []

# Loop de simulação usando Euler
for i in range(1, N):
    # Cálculo da derivada dV/dt
    dV = dt * (-V[i-1]/tau + R * I_const / tau)

    # Atualização do potencial
    V[i] = V[i-1] + dV

    # Verificação do limiar de disparo
    if V[i] >= V_th:
        spike_times.append(t[i])  # registramos o tempo do spike
        V[i] = V_reset            # resetamos o potencial

# Plot do potencial de membrana ao longo do tempo
plt.figure(figsize=(10, 4))
plt.plot(t, V, label='Potencial de membrana V(t)')
plt.xlabel('Tempo (ms)')
plt.ylabel('Potencial (V)')
plt.title('Modelo Integrate-and-Fire com entrada constante')
plt.grid(True)

# Marcar os momentos dos spikes
for spike in spike_times:
```

```
    plt.axvline(x=spike, color='red', linestyle='--', alpha=0.5)

plt.legend()
plt.tight_layout()
plt.show()
```

Este modelo, embora simples, captura a essência do comportamento neuronal. Ele é eficiente para simulações de grandes redes neurais, especialmente em modelos computacionais de neurociência e aprendizado de máquina. O próximo passo seria considerar correntes variáveis no tempo $I(t)$, adicionar ruído, ou construir uma rede de múltiplos neurônios interconectados.

9.7 Modelo de Hodgkin-Huxley

O modelo de **Hodgkin-Huxley** é um dos pilares da neurociência computacional e da biofísica moderna. Desenvolvido em 1952 por Alan Hodgkin e Andrew Huxley, esse modelo descreve com notável precisão a geração e propagação de impulsos elétricos (potenciais de ação) ao longo do axônio de um neurônio, a partir de experimentos realizados no axônio gigante da lula *Loligo*. Trata-se de um modelo biofísico detalhado, formulado a partir de equações diferenciais ordinárias não lineares, que representam os mecanismos de transporte iônico através da membrana celular. Ele considera explicitamente os canais de sódio (Na^+) e potássio (K^+), além de uma corrente de fuga (leak), cada uma com suas próprias dinâmicas de ativação e inativação dependentes do potencial de membrana. Diferentemente de modelos simplificados, como o Integrate-and-Fire, o modelo de Hodgkin-Huxley reproduz com fidelidade os eventos eletrofisiológicos fundamentais, incluindo a despolarização, repolarização e o período refratário, fornecendo uma descrição quantitativa rica dos processos neuronais.

Motivação Física e Fisiológica

Do ponto de vista biofísico, a membrana do neurônio pode ser modelada como um circuito elétrico que regula a condução de sinais elétricos ao longo da célula. Esse modelo permite uma compreensão quantitativa do comportamento eletrofisiológico do neurônio. A bicamada lipídica da membrana atua como um capacitor, representado pela capacitância C_m, que armazena cargas elétricas separadas em seus dois lados. Além disso, a membrana possui canais específicos para diferentes íons, como sódio (Na^+) e potássio (k^+), cuja abertura ou fechamento modifica a condutância da membrana de maneira dinâmica, permitindo ou restringindo o fluxo desses íons. Essas condutâncias variáveis são essenciais para a geração e propagação do potencial de ação. Cada tipo de íon tem um potencial de reversão associado, que pode ser interpretado como uma fonte eletromotriz no circuito equivalente, impulsionando o movimento dos íons conforme seus gradientes eletroquímicos. A interação entre essas componentes — capacitância, condutâncias variáveis e fontes eletromotrizes — descreve de forma eficaz a atividade elétrica da membrana, permitindo a propagação de sinais ao longo do axônio e entre neurônios. Esse modelo biofísico fundamenta-se nos princípios da eletrodinâmica e é a base para a formulação de equações como as de Hodgkin-Huxley, que descrevem com precisão a excitabilidade neuronal.

O modelo descreve o fluxo de corrente total como:

$$C_m \frac{dV}{dt} = I_{\text{ext}} - I_{\text{Na}} - I_{\text{K}} - I_{\text{L}}$$

onde:

- V é o potencial de membrana.
- $I_{\text{Na}} = \bar{g}_{\text{Na}} m^3 h (V - E_{\text{Na}})$: corrente de sódio.
- $I_{\text{K}} = \bar{g}_{\text{K}} n^4 (V - E_{\text{K}})$: corrente de potássio.

- $I_{\mathrm{L}} = \bar{g}_{\mathrm{L}}(V - E_{\mathrm{L}})$: corrente de fuga (leak).

As variáveis m, h, n representam as probabilidades de abertura dos canais e obedecem equações diferenciais do tipo:

$$\frac{dx}{dt} = \alpha_x(V)(1-x) - \beta_x(V)x, \quad \text{com } x \in \{m, h, n\}$$

Equações completas do modelo

$$\begin{aligned}
C_m \frac{dV}{dt} &= I_{\mathrm{ext}} - \bar{g}_{\mathrm{Na}} m^3 h (V - E_{\mathrm{Na}}) - \bar{g}_{\mathrm{K}} n^4 (V - E_{\mathrm{K}}) - \bar{g}_{\mathrm{L}}(V - E_{\mathrm{L}}) \\
\frac{dm}{dt} &= \alpha_m(V)(1-m) - \beta_m(V)m \\
\frac{dh}{dt} &= \alpha_h(V)(1-h) - \beta_h(V)h \\
\frac{dn}{dt} &= \alpha_n(V)(1-n) - \beta_n(V)n
\end{aligned}$$

As funções α e β dependem do potencial V e são dadas por:

$$\begin{aligned}
\alpha_n(V) &= \frac{0.01(10-V)}{\exp\left(\frac{10-V}{10}\right) - 1}, & \beta_n(V) &= 0.125 \exp\left(\frac{-V}{80}\right) \\
\alpha_m(V) &= \frac{0.1(25-V)}{\exp\left(\frac{25-V}{10}\right) - 1}, & \beta_m(V) &= 4 \exp\left(\frac{-V}{18}\right) \\
\alpha_h(V) &= 0.07 \exp\left(\frac{-V}{20}\right), & \beta_h(V) &= \frac{1}{\exp\left(\frac{30-V}{10}\right) + 1}
\end{aligned}$$

Simulação Numérica com o Método de Euler

Vamos implementar esse sistema em Python utilizando o método de Euler explícito para resolver as equações diferenciais.

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

# Parâmetros do modelo
Cm = 1.0      # Capacitância da membrana (uF/cm^2)
gNa = 120.0   # Condutância máxima de Na (mS/cm^2)
gK = 36.0     # Condutância máxima de K (mS/cm^2)
gL = 0.3      # Condutância de fuga (mS/cm^2)
ENa = 115.0   # Potencial de reversão de Na (mV)
EK = -12.0    # Potencial de reversão de K (mV)
EL = 10.6     # Potencial de reversão da corrente de fuga (mV)
Vrest = 0.0   # Potencial de repouso (mV)

# Funções auxiliares
def alpha_n(V): return 0.01*(10-V)/(np.exp((10-V)/10)-1)
def beta_n(V): return 0.125*np.exp(-V/80)
def alpha_m(V): return 0.1*(25-V)/(np.exp((25-V)/10)-1)
def beta_m(V): return 4*np.exp(-V/18)
def alpha_h(V): return 0.07*np.exp(-V/20)
def beta_h(V): return 1/(np.exp((30-V)/10)+1)
```

```
# Estímulo
def I_inj(t):
    return 10.0 if 10 < t < 40 else 0.0

# Inicialização
dt = 0.01
T = 50
time = np.arange(0, T, dt)
V = np.zeros_like(time)
m = np.zeros_like(time)
h = np.zeros_like(time)
n = np.zeros_like(time)

# Condições iniciais
V[0] = Vrest
m[0] = alpha_m(Vrest)/(alpha_m(Vrest) + beta_m(Vrest))
h[0] = alpha_h(Vrest)/(alpha_h(Vrest) + beta_h(Vrest))
n[0] = alpha_n(Vrest)/(alpha_n(Vrest) + beta_n(Vrest))

# Integração com Euler
for i in range(1, len(time)):
    INa = gNa * m[i-1]**3 * h[i-1] * (V[i-1] - ENa)
    IK = gK * n[i-1]**4 * (V[i-1] - EK)
    IL = gL * (V[i-1] - EL)
    I_ext = I_inj(time[i-1])

    dV = (I_ext - INa - IK - IL) / Cm
    dm = alpha_m(V[i-1]) * (1 - m[i-1]) - beta_m(V[i-1]) * m[i-1]
    dh = alpha_h(V[i-1]) * (1 - h[i-1]) - beta_h(V[i-1]) * h[i-1]
    dn = alpha_n(V[i-1]) * (1 - n[i-1]) - beta_n(V[i-1]) * n[i-1]

    V[i] = V[i-1] + dt * dV
    m[i] = m[i-1] + dt * dm
    h[i] = h[i-1] + dt * dh
    n[i] = n[i-1] + dt * dn

# Plotagem do potencial de membrana
plt.figure(figsize=(10, 4))
plt.plot(time, V)
plt.title("Modelo de Hodgkin-Huxley: Potencial de Membrana")
plt.xlabel("Tempo (ms)")
plt.ylabel("Potencial (mV)")
plt.grid()
plt.show()
```

Este modelo captura com realismo o comportamento de um neurônio em resposta a estímulos externos. Apesar da sua complexidade maior comparado a modelos como o Integrate-and-Fire, ele fornece informações detalhadas sobre os mecanismos de disparo, incluindo fases de despolarização, repolarização e período refratário. Por isso, o modelo de Hodgkin-Huxley é considerado uma referência na modelagem matemática de neurônios.

9.8 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:

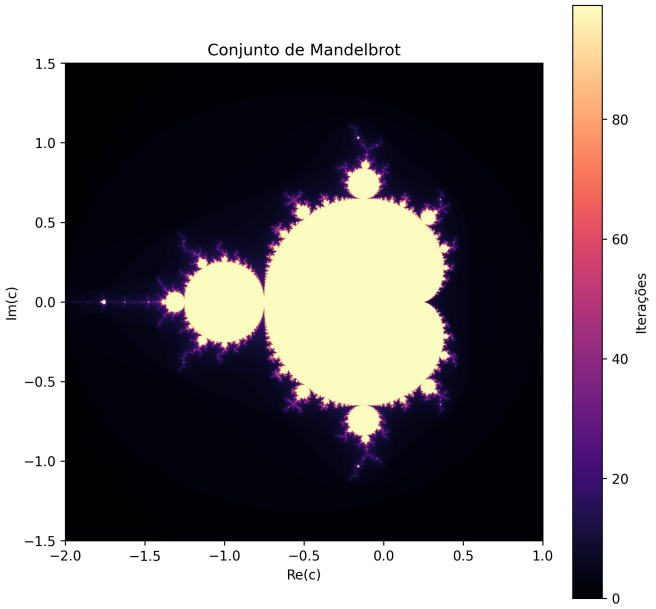


Figure 2: Conjunto de Mandelbrot gerado pelo programa.

$$z_{n+1} = z_n^2 + c, \tag{40}$$

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.

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.

9.9 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. 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.

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:

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

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 $[0, 2]$. A seguir, apresentamos um programa em Python que implementa esse método.

```
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.
```

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.

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 $[0, 2]$.

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:

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

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

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).

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 $g(x)$.

9.10 Decaimento Radioativo com Relaxação Temporal: Uma Abordagem Numérica Geral

O modelo clássico do decaimento radioativo assume que a taxa de decaimento λ é constante ao longo do tempo. Nesse caso, a equação diferencial que descreve o número $N(t)$ de núcleos radioativos remanescentes é dada por:

$$\frac{dN}{dt} = -\lambda N(t),$$

cuja solução é a conhecida expressão exponencial:

$$N(t) = N_0 e^{-\lambda t},$$

onde N_0 é o número inicial de núcleos no tempo $t = 0$. Entretanto, em muitos contextos experimentais e teóricos da física moderna, torna-se necessário considerar que a taxa de decaimento pode não ser constante. Fatores como variações no ambiente físico (temperatura, pressão, campos externos), acoplamentos com outros sistemas dinâmicos, ou até efeitos quânticos e cosmológicos, podem fazer com que λ varie com o tempo. Nesses casos, a equação diferencial que governa a dinâmica do sistema torna-se:

$$\frac{dN}{dt} = -\lambda(t) N(t),$$

onde agora $\lambda = \lambda(t)$ é uma função arbitrária do tempo. A solução formal dessa equação envolve uma integração temporal da taxa de decaimento:

$$N(t) = N_0 \exp\left(-\int_0^t \lambda(s)\, ds\right).$$

No entanto, para muitas escolhas realistas de $\lambda(t)$, a integral acima não pode ser obtida de forma analítica simples. Por isso, recorre-se a métodos numéricos para simular o comportamento do sistema. Como já explicamos anteriormente, o método de Euler é uma técnica numérica básica para resolver equações diferenciais ordinárias, especialmente útil para simulações rápidas e de fácil implementação. Adaptado ao caso de uma taxa de decaimento variável, a fórmula de atualização torna-se:

$$N(t + \Delta t) \approx N(t) - \lambda(t) \cdot N(t) \cdot \Delta t,$$

onde Δt é o passo de tempo da simulação. Essa aproximação é obtida a partir de uma discretização simples da derivada, válida para pequenos valores de Δt. Para ilustrar esse modelo generalizado, consideramos um caso em que a taxa de decaimento $\lambda(t)$ diminui exponencialmente com o tempo, modelando um sistema que se estabiliza gradualmente ou se desacopla de uma fonte externa. A função é definida como:

$$\lambda(t) = \lambda_0 e^{-t/\tau},$$

onde λ_0 é o valor inicial da taxa de decaimento, e τ é o tempo característico de relaxação. Esse tipo de comportamento aparece, por exemplo, em sistemas físicos onde a influência de um campo externo decai com o tempo. A seguir, apresentamos um código em Python que implementa essa ideia, utilizando o método de Euler para simular a evolução de $N(t)$, e também calcula a solução formal aproximada por integração numérica da taxa $\lambda(t)$.

```
# Simulação do decaimento com relaxação temporal usando método de Euler

import matplotlib.pyplot as plt
import numpy as np

# Parâmetros do sistema
N0 = 1000       # Número inicial de núcleos
dt = 0.1        # Passo de tempo
T = 50          # Tempo total de simulação

# Definição da função lambda(t): relaxação exponencial
def lambda_tempo(t):
    lambda0 = 0.2      # valor inicial da taxa
    tau = 20           # tempo de relaxação
    return lambda0 * np.exp(-t / tau)

# Inicialização das listas para armazenar os dados
tempos = [0]
valores_N = [N0]
N = N0
t = 0

# Loop de Euler com lambda variável
while t < T:
    lambda_now = lambda_tempo(t)
    N = N - lambda_now * N * dt
    t += dt
    tempos.append(t)
    valores_N.append(N)

# Cálculo da solução analítica aproximada (via integração numérica)
tempos_array = np.array(tempos)
lambdas = lambda_tempo(tempos_array)
integral_lambda = np.cumsum(lambdas * dt)
N_analitico = N0 * np.exp(-integral_lambda)

# Plotagem do resultado
plt.plot(tempos, valores_N, label='Euler (com (t))', color='blue')
plt.plot(tempos, N_analitico, label='Solução analítica aproximada',
         linestyle='--', color='red')

plt.xlabel('Tempo')
plt.ylabel('Número de núcleos')
plt.title('Decaimento radioativo com relaxação temporal')
plt.legend()
plt.grid(True)
plt.show()
```

O gráfico gerado pelo código compara a solução obtida pelo método de Euler com a solução formal aproximada baseada na integração da taxa de decaimento. A boa concordância entre ambas valida

o uso do método de Euler nesse contexto, ao menos para passos de tempo suficientemente pequenos. Esse exemplo mostra como adaptar modelos clássicos a situações mais realistas e complexas, mantendo a estrutura conceitual original, mas incorporando dependências temporais. Tais modificações são cruciais na modelagem de sistemas físicos reais, especialmente em contextos onde as condições do ambiente mudam ao longo do tempo — um cenário comum na física de sistemas abertos, astrofísica, e até em certas aplicações biomédicas.

9.11 Evolução temporal no modelo de Anderson 1D via método de Runge-Kutta 4ª ordem

Neste exemplo, estudamos numericamente a evolução temporal de um pacote de onda inicialmente localizado no centro de uma cadeia 1D com $N = 200$ sítios. O sistema é descrito pelo modelo de Anderson 1D, cuja Hamiltoniana (no limite tight-binding) é dada por:

$$H = \sum_{n=1}^{N} \varepsilon_n |n\rangle \langle n| - t \sum_{n=1}^{N-1} \left(|n\rangle \langle n+1| + |n+1\rangle \langle n| \right),$$

onde ε_n representa o potencial local em cada sítio. Para descrever desordem, assume-se que $\varepsilon_n \in [-W/2, W/2]$ são sorteados aleatoriamente em uma distribuição uniforme. O parâmetro t (aqui fixado como $t = 1$) é o coeficiente de hopping entre sítios vizinhos.

A evolução temporal do estado $|\psi(t)\rangle$ obedece à equação de Schrödinger dependente do tempo (com $\hbar = 1$):

$$i \frac{d}{dt} |\psi(t)\rangle = H |\psi(t)\rangle .$$

Para resolver numericamente essa equação, utilizamos o método de Runge-Kutta de quarta ordem (RK4) que é uma uma técnica amplamente utilizada na física computacional devido à sua precisão e estabilidade para a integração de equações diferenciais ordinárias. Em linhas gerais o formalismo pode ser aplicado da seguinte forma: Seja $\psi(t)$ a função de estado do sistema em um instante t, e queremos calcular $\psi(t+\Delta t)$ para um pequeno incremento temporal Δt. O método RK4 fornece uma aproximação para $\psi(t + \Delta t)$ a partir do valor conhecido de $\psi(t)$ por meio de uma média ponderada de quatro estimativas da derivada $d\psi/dt$. Especificamente, definimos:

$$\begin{aligned}
k_1 &= f(\psi(t), t) = -i\hat{H}\psi(t), \\
k_2 &= f\left(\psi(t) + \frac{\Delta t}{2} k_1, t + \frac{\Delta t}{2}\right) = -i\hat{H}\left(\psi(t) + \frac{\Delta t}{2} k_1\right), \\
k_3 &= f\left(\psi(t) + \frac{\Delta t}{2} k_2, t + \frac{\Delta t}{2}\right) = -i\hat{H}\left(\psi(t) + \frac{\Delta t}{2} k_2\right), \\
k_4 &= f\left(\psi(t) + \Delta t \cdot k_3, t + \Delta t\right) = -i\hat{H}\left(\psi(t) + \Delta t \cdot k_3\right).
\end{aligned}$$

A nova aproximação para o estado no tempo $t + \Delta t$ é então dada por:

$$\psi(t + \Delta t) \approx \psi(t) + \frac{\Delta t}{6} \left(k_1 + 2k_2 + 2k_3 + k_4\right).$$

Esse método possui **erro local de ordem** $\mathcal{O}(\Delta t^5)$ e **erro global acumulado de ordem** $\mathcal{O}(\Delta t^4)$, o que o torna significativamente mais preciso que métodos mais simples como Euler, especialmente em simulações de longa duração. No contexto do modelo de Anderson 1D para um elétron, essa equação é usada para descrever a evolução temporal da função de onda $\psi_n(t)$ sobre uma cadeia de sítios com desordem aleatória. A Hamiltoniana $\hat{H}$ é representada por uma matriz tridiagonal (com diagonal

aleatória no caso desordenado) e é usada diretamente no cálculo dos termos k_i acima por meio da multiplicação matricial $\hat{H}\psi$. O uso de RK4 para esse tipo de problema é vantajoso porque garante precisão suficiente mesmo com passos de tempo pequenos, evitando a acumulação de erros numéricos que poderiam afetar a interpretação física dos resultados — como a observação da localização de Anderson, espalhamento, ou propagação balística da função de onda.

Em nosso estudo, a condição inicial considerada é um pacote de onda localizado: $\psi_n(0) = \delta_{n,N/2}$. Abaixo, apresentamos um código em Python que realiza essa simulação, comparando dois casos: (i) sistema ordenado ($W = 0$) e (ii) sistema com desordem forte ($W = 4$). A cada passo de tempo da simulação, calculamos a **densidade de probabilidade** $|\psi_n(t)|^2$, que nos diz qual a chance de encontrar o elétron em cada sítio da cadeia naquele instante. Esses valores são armazenados como uma sequência de *quadros* — como se estivéssemos tirando uma fotografia do sistema a cada pequeno intervalo de tempo. Ao final da simulação, temos uma longa sequência de imagens que mostram como o pacote de onda vai se espalhando (ou não!) pela cadeia. Usando bibliotecas gráficas do Python, como `matplotlib`, conseguimos transformar esses quadros em um vídeo animado. Dessa forma, podemos literalmente **assistir** à evolução quântica do sistema. Mais do que apenas um gráfico estático, o vídeo permite ver claramente o contraste entre dois regimes muito diferentes: no caso sem desordem ($W = 0$), o pacote de onda se propaga de forma quase livre, espalhando-se simetricamente ao longo do tempo. Já no caso com desordem forte ($W = 4$), observamos que o elétron permanece praticamente confinado em torno da posição inicial, um comportamento conhecido como **localização de Anderson**. Essa visualização torna o fenômeno mais intuitivo e facilita a interpretação dos resultados, especialmente quando se está aprendendo os conceitos. Em vez de olhar apenas para números ou gráficos estáticos, o vídeo nos permite ver o tempo em ação, e acompanhar, passo a passo, como a desordem afeta a dinâmica do sistema. É uma forma poderosa — e até bonita — de conectar a física quântica com o mundo computacional.

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

# --- Parâmetros do sistema ---
N = 200             # Número de sítios na cadeia (tamanho do sistema)
t_hop = 1.0         # Parâmetro de hopping entre sítios vizinhos (energia)
dt = 0.01           # Passo de tempo para integração temporal
Tmax = 20           # Tempo total de simulação
steps = int(Tmax / dt)  # Número total de passos de tempo
center = N // 2     # Índice do sítio central (posição inicial do pacote)

# --- Função para criar o estado inicial ---
def inicial():
    # Cria vetor de zeros complexos (estado quântico)
    psi = np.zeros(N, dtype=complex)
    # Localiza o elétron inicialmente no centro da cadeia
    psi[center] = 1.0
    return psi

# --- Construção da Hamiltoniana para o modelo de Anderson 1D ---
def hamiltoniano(W):
    # Cria matriz NxN de zeros complexos
    H = np.zeros((N, N), dtype=complex)

    # Gera desordem aleatória: _n  [-W/2, W/2] para cada sítio
    eps = np.random.uniform(-W/2, W/2, N)

    for i in range(N):
        H[i, i] = eps[i]  # Potencial local (diagonal)
        if i < N - 1:
            # Acoplamento entre vizinhos: H[i, i+1] = H[i+1, i] = -t
            H[i, i+1] = H[i+1, i] = -t_hop
    return H

# --- Método de Runge-Kutta de 4ª ordem para resolver i d/dt = H ---
def rk4(psi, H, dt):
    # Calcula os quatro k's do RK4 (com fator -i devido à equação de Schrödinger)
    k1 = -1j * H @ psi
    k2 = -1j * H @ (psi + dt * k1 / 2)
    k3 = -1j * H @ (psi + dt * k2 / 2)
    k4 = -1j * H @ (psi + dt * k3)

    # Atualiza o estado psi segundo o esquema RK4
    return psi + dt * (k1 + 2*k2 + 2*k3 + k4) / 6

# --- Evolução temporal e armazenamento dos quadros para a animação ---
def evoluir(W):
    H = hamiltoniano(W)     # Monta a Hamiltoniana com desordem W
    psi = inicial()         # Inicializa o estado (t=0)
    frames = []             # Lista para armazenar |(n,t)|² ao longo do tempo

    for _ in range(steps):
```

```
        densidade = np.abs(psi)**2   # Densidade de probabilidade no espaço real
        frames.append(densidade.copy())  # Armazena o frame
        psi = rk4(psi, H, dt)        # Atualiza o estado usando RK4
    return np.array(frames)

# --- Executa a simulação para os dois casos ---
dados_W0 = evoluir(W=0)     # Sistema sem desordem (W = 0)
dados_W4 = evoluir(W=4)     # Sistema com desordem forte (W = 4)

# --- Preparação da figura para a animação lado a lado ---
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 6))

# Linhas (curvas) para serem atualizadas em cada quadro
linha1, = ax1.plot([], [], color='blue')  # W = 0
linha2, = ax2.plot([], [], color='red')   # W = 4

# Configurações dos eixos
ax1.set_ylim(0, 1)
ax2.set_ylim(0, 1)
ax1.set_xlim(0, N)
ax2.set_xlim(0, N)
ax1.set_title('Evolução sem desordem (W=0)')
ax2.set_title('Evolução com desordem (W=4)')

# Função de inicialização da animação (quadro vazio)
def init():
    linha1.set_data([], [])
    linha2.set_data([], [])
    return linha1, linha2

# Função que atualiza as curvas em cada quadro
def update(i):
    linha1.set_data(np.arange(N), dados_W0[i])
    linha2.set_data(np.arange(N), dados_W4[i])
    return linha1, linha2

# Criação da animação usando FuncAnimation
ani = animation.FuncAnimation(
    fig, update, frames=steps,
    init_func=init, blit=True
)

# Salva o vídeo final como arquivo MP4
ani.save("anderson_rk4_duplo.mp4", fps=30)

# Fecha a figura após salvar o vídeo
plt.close()
```

A animação gerada ilustra de forma clara o contraste entre os dois regimes físicos. No caso sem desordem ($W = 0$), observamos uma rápida propagação do pacote de onda ao longo da cadeia — um comportamento típico de **transporte balístico**, onde o elétron se move livremente através do sistema. Em contraste, na presença de desordem forte ($W = 4$), a densidade de probabilidade permanece concentrada ao redor do sítio inicial, mesmo após um longo tempo de evolução. Esse aprisionamento espacial da função de onda é a assinatura da chamada **localização de Anderson**, um fenômeno puramente quântico, causado pela interferência destrutiva gerada pela desordem aleatória. Esse experimento computacional simples fornece uma visualização direta de como pequenas mudanças nas condições do sistema — neste caso, a introdução da desordem — podem induzir transições qualitativas profundas no comportamento da dinâmica quântica. Ele mostra, na prática, como a combinação entre modelagem matemática, métodos numéricos e recursos visuais pode nos ajudar a explorar e compreender fenômenos complexos da física moderna. Mais do que resolver uma equação, aqui usamos o computador como uma lente para observar a beleza e a sutileza do mundo quântico em movimento.

10 Considerações Finais

Ao longo destas breves Notas de Aula em Física Computacional com Python, exploramos os fundamentos da programação aplicados ao estudo de sistemas físicos, partindo dos conceitos elementares da linguagem até chegar a métodos numéricos clássicos utilizados na resolução de problemas reais. A proposta foi apresentar uma introdução acessível, mas sólida, que permitisse mesmo aos leitores sem experiência prévia em programação acompanhar e aplicar, de forma progressiva, os conceitos abordados.

A física computacional desempenha hoje um papel central na ciência contemporânea. Muitos problemas relevantes da física não admitem soluções analíticas exatas e exigem abordagens numéricas para sua compreensão. Seja na integração de equações diferenciais, na simulação de sistemas dinâmicos, na análise estatística de dados experimentais ou na visualização de fenômenos complexos, o uso de ferramentas computacionais se tornou indispensável para físicos e cientistas em diversas áreas.

Neste material, procuramos oferecer uma introdução prática ao uso do Python como ferramenta de modelagem, análise e visualização de sistemas físicos. O enfoque foi sempre pedagógico: os códigos foram construídos com clareza e acompanhados de comentários explicativos que detalham passo a passo a lógica de implementação. O objetivo foi não apenas apresentar soluções prontas, mas também incentivar o leitor a entender o funcionamento de cada algoritmo e, com isso, desenvolver autonomia para adaptar e expandir os exemplos conforme suas próprias necessidades.

Além dos métodos numéricos tradicionais, como o método de Euler e integrações por diferenças finitas, também introduzimos o uso de gráficos e animações básicas. A visualização dinâmica de sistemas físicos é uma poderosa aliada no processo de aprendizado: ela revela comportamentos que muitas vezes passam despercebidos na análise puramente algébrica e convida à experimentação. Ao permitir que o leitor observe diretamente a evolução temporal de um sistema — seja uma partícula oscilando, uma onda se propagando ou um sistema decaindo — promovemos uma forma mais intuitiva e interativa de compreender a física.

Essas notas tiveram como foco a introdução à linguagem Python e sua aplicação em problemas iniciais de física geral, com incursões pontuais em temas ligeiramente mais avançados. A escolha por exemplos básicos foi intencional: acreditamos que a construção de uma base conceitual e computacional sólida é essencial antes de se avançar para aplicações mais sofisticadas. Embora o universo da física computacional seja vasto e inclua técnicas avançadas como métodos espectrais, simulações de Monte Carlo e dinâmica molecular, este material foi concebido como um primeiro passo — um convite à exploração.

Esperamos que estas pequenas Notas de Aula tenham cumprido seu papel de introduzir o leitor a esse campo fascinante, despertando a curiosidade e fornecendo as ferramentas iniciais para que ele possa seguir sua própria trajetória. A partir daqui, o caminho se abre para o aprofundamento em métodos mais elaborados, a investigação de problemas mais desafiadores e, acima de tudo, o desenvolvimento de um olhar mais investigativo e experimental sobre a física.

Bons estudos e boas simulações!