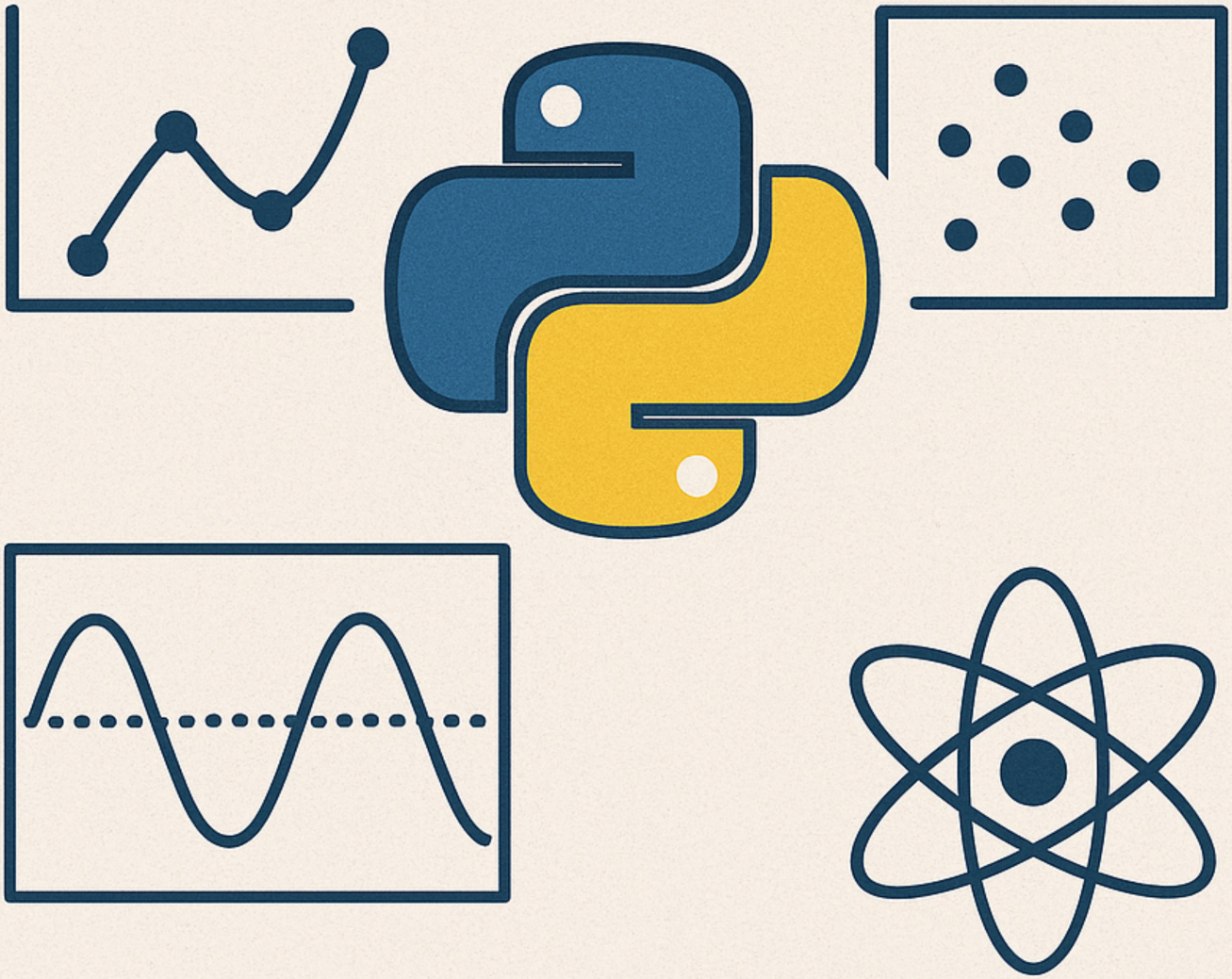
Francisco Anacleto Barros Fidelis de Moura

FÍSICA COMPUTACIONAL EM PYTHON

NOTAS ESSENCIAIS



ISBN 978-65-01-81621-0

Física Computacional em Python : Notas Essenciais

Francisco Anacleto Barros Fidelis de Moura

Prefácio

Estas notas de aula foram elaboradas como material de apoio ao curso de Física Computacional, com foco na implementação de algoritmos numéricos em linguagem Python . O objetivo central é oferecer aos estudantes uma introdução clara, prática e progressiva às ferramentas computacionais mais empregadas atualmente na física e em áreas correlatas. Ao longo do texto, conceitos essenciais de programação e de métodos numéricos são apresentados de maneira gradual, sempre acompanhados de exemplos e códigos completos, explorando bibliotecas amplamente utilizadas na comunidade científica.

A **Parte 1** aborda os fundamentos de programação em Python, incluindo estruturas básicas da linguagem, manipulação de arrays com `NumPy`, visualização de dados com `Matplotlib` e rotinas iniciais de cálculo numérico. Na **Parte 2**, o material avança para tópicos como integração de equações diferenciais ordinárias, simulações estocásticas, transformadas de Fourier, técnicas de diagonalização de matrizes e outros métodos lineares e espectrais fundamentais para problemas de autovalores e autovetores em física.

A proposta não é apenas disponibilizar algoritmos prontos, mas também discutir seus fundamentos, limitações, interpretações físicas e aplicações. Assim, o estudante é incentivado a desenvolver tanto o domínio técnico do uso de Python no contexto científico quanto a compreensão conceitual dos métodos numéricos empregados.

Agradecimentos

Manifesto minha apreciação aos estudantes que participaram ativamente da construção deste curso, oferecendo contribuições valiosas por meio de perguntas, sugestões e desafios práticos. Sou igualmente grato ao Instituto de Física da universidade à qual pertenço, assim como aos colegas, amigos e colaboradores, cujo diálogo contínuo tem sido essencial para o aprimoramento do nosso trabalho acadêmico e formativo. Por fim, agradeço à minha família pelo apoio constante e pela presença cotidiana, fundamentais ao longo de toda esta jornada.

Sumário

Parte 1

A **Parte 1** do curso de Física Computacional tem como objetivo introduzir o estudante às ferramentas fundamentais que servirão de base para todo o desenvolvimento posterior da disciplina. O ponto de partida é a *programação em linguagem Python*, que será utilizada ao longo do curso como principal instrumento para a implementação dos algoritmos numéricos.

Nesta etapa inicial, o foco estará no aprendizado da sintaxe da linguagem, do uso de estruturas de dados básicas e do desenvolvimento de programas simples, que ajudarão a construir familiaridade com a lógica computacional. Em seguida, exploraremos aplicações diretas, como a manipulação de **números complexos em Python**, recurso já nativo da linguagem e amplamente empregado em problemas de física.

Passaremos então à discussão de **técnicas elementares de cálculo numérico**, incluindo derivadas e integrais aproximadas, com apoio das bibliotecas `NumPy` e `SciPy`. Esses métodos permitem compreender, em um nível conceitual e prático, as limitações e a precisão das aproximações computacionais.

Também será apresentada a geração de **números aleatórios**, implementada de forma simples em Python, ferramenta essencial para métodos de Monte Carlo e para modelagens estocásticas que aparecerão em partes posteriores do curso. Por fim, introduziremos a **regressão linear**, utilizando rotinas já disponíveis em bibliotecas científicas para análise de dados experimentais e para avaliar a eficiência dos algoritmos desenvolvidos.

Assim, a Parte 1 constitui uma preparação sólida, reunindo fundamentos de programação e conceitos numéricos iniciais, fornecendo ao estudante as bases necessárias para compreender e aplicar os métodos mais avançados que serão tratados nas próximas etapas do curso.

1 Introdução à Programação em Python

Antes de elaborar programas mais sofisticados, é essencial dominar os comandos fundamentais, as funções nativas e as estruturas básicas da linguagem Python. Esse conhecimento constitui a base para a criação de códigos corretos , eficientes e fáceis de manter , além de tornar o processo de depuração significativamente mais simples. Compreender bem esses elementos também ajuda a desenvolver uma intuição sólida sobre como o Python "pensa": como ele organiza dados, como executa instruções e como diferentes partes de um programa interagem entre si. Isso não apenas acelera o aprendizado, mas também evita erros comuns e permite que o programador se concentre nos aspectos conceituais dos problemas que deseja resolver, e não nos detalhes técnicos da sintaxe. A tabela a seguir reúne alguns dos componentes mais importantes da linguagem — desde comandos básicos e tipos de dados essenciais até estruturas de controle e funções amplamente utilizadas. Cada item é acompanhado de uma breve descrição ou exemplo prático, funcionando como uma referência rápida tanto para iniciantes quanto para usuários intermediários que desejam reforçar ou revisar seus conhecimentos fundamentais. Trata-se, portanto, de um panorama conciso, porém abrangente, de ferramentas indispensáveis para quem está dando os primeiros passos em Python ou consolidando sua base para aplicações científicas, computacionais ou de uso geral.

Comando / Função	Descrição / Exemplo
`import math`	Biblioteca matemática: `math.sin(x), math.exp(x), math.sqrt(x).`
`import numpy as np`	Biblioteca para arrays e cálculo numérico: `np.array([1,2,3]).`
`import random`	Números aleatórios: `random.random(), random.randint(1,10).`
`def f(x): ...`	Define função: `def f(x): return x 2.`
`print()`	Imprime valores: `print("x =", x).`
`input()`	Lê entrada do usuário: `n = int(input("Digite: ")).`
`for i in range(n):`	Loop com contador: `for i in range(10): print(i).`
`while cond:`	Loop condicional: `while x<10: x+=1.`
`if ... elif ... else`	Condicional: `if x>0: ... elif x==0: ... else: ....`
`break`	Encerra loop imediatamente.
`continue`	Pula para a próxima iteração do loop.
`return`	Retorna valor de uma função: `return x 2.`
`len()`	Tamanho de lista/array/string: `len([1,2,3]).`
`type()`	Retorna tipo: `type(3.14).`
`int(), float(), str()`	Conversão de tipos: `float("3.14").`
`list, tuple, dict, set`	Estruturas de dados: `[1,2], (1,2), {"a":1}, {1,2,3}.`
`with open("arq.txt","r") as f:`	Manipulação de arquivos: leitura/escrita segura.
`try ... except ...`	Tratamento de erros: `try: x=1/0 except ZeroDivisionError: ...`
`class Objeto: ...`	Define classe: `class Ponto: def __init__(self,x,y): ....`
`__init__`	Construtor em classes: inicializa atributos.
`lambda x: x 2`	Função anônima (curta).
`map(), filter(), sum(), max(), min()`	Funções úteis para coleções.
`enumerate()`	Itera com índice: `for i,v in enumerate(lista): ....`
`zip()`	Itera sobre múltiplas listas: `for a,b in zip(L1,L2): ....`
`f-strings`	Formatação de strings: `f"x = {x:.2f}".`
`help()`	Ajuda interativa: `help(math.sin).`
`dir()`	Lista atributos/funções de um objeto.
`__name__ == "__main__"`	Identifica execução direta de um script.

Esta tabela fornece um resumo das funções e comandos mais usados em Python, com exemplos curtos e explicativos. Ela serve como referência rápida para iniciantes e para consultas durante a programação de exercícios ou projetos. Vamos discutir, com mais profundidade, alguns dos comandos e funções fundamentais de Python listados na tabela anterior. Eles formam a base de praticamente todo programa em Python — desde simulações numéricas até scripts simples de manipulação de dados.

Importação de Bibliotecas (`import numpy as np`, `import math`)

Python possui um ecossistema vasto de bibliotecas. Para realizar cálculos científicos de forma eficiente, utilizamos principalmente:

- `math`: contém funções matemáticas básicas (seno, exponencial, raiz).
- `NumPy`: fornece arrays de alta performance e operações vetorizadas.

Importar bibliotecas evita reinventar a roda e torna o código mais simples e rápido.

Definição de Funções (`def f(x): ...`)

Funções permitem encapsular cálculos e reutilizar código. São blocos que recebem parâmetros, realizam operações e retornam resultados. Por exemplo:

$$f(x) = x^2 \quad \Rightarrow \quad \texttt{def f(x): return x**2}$$

Usar funções torna o código modular, organizado e mais fácil de depurar.

Impressão de Resultados (`print()`)

A função `print()` é essencial para depuração e comunicação. Em simulações, muitas vezes imprimimos valores intermediários para verificar se o algoritmo está funcionando corretamente. Com *f-strings*, podemos formatar saídas com clareza: `print(f"x = {x:.3f}")`.

Estruturas Condicionais (`if / elif / else`)

São fundamentais para tomar decisões no código. Elas permitem executar diferentes blocos de acordo com condições lógicas.

$$\text{Se } x > 0, \text{ faça A; se } x = 0, \text{ faça B; caso contrário faça C.}$$

Quase todos os algoritmos fazem uso extenso dessas decisões condicionais.

Loops `for` e `while`

Loops permitem repetir operações várias vezes:

- `for`: usado quando há um número conhecido de iterações.
- `while`: usado quando repetimos até que uma condição seja satisfeita.

Simulações físicas iterativas (como métodos numéricos ou updates de animação) dependem fortemente desses construtos.

Estruturas de Dados (`list`, `tuple`, `dict`, `set`)

Estas são coleções fundamentais para organizar informações:

- **listas**: mutáveis, usadas para armazenar sequências de valores.
- **tuplas**: imutáveis, usadas para dados fixos.
- **dicionários**: pares chave–valor, úteis para dados rotulados.
- **sets**: coleções sem elementos repetidos.

Todas aparecem frequentemente em códigos científicos.

Conversão de Tipos (`int()`, `float()`, `str()`)

Conversões são comuns quando lemos dados externos, trabalhamos com entradas do usuário, ou precisamos garantir o tipo correto para cálculos numéricos (por exemplo, divisões precisas em ponto flutuante).

Tratamento de Erros (`try ... except ...`)

Permite capturar erros e impedir que o programa quebre abruptamente. Por exemplo: `try: x = 1/0 except ZeroDivisionError: ...` Útil quando há risco de erros numéricos ou operações inválidas.

Definição de Classes (`class Nome: ...`)

Classes permitem criar estruturas mais complexas — como partículas, campos, corpos rígidos, etc. Dentro de uma classe, o método `__init__` funciona como um construtor, inicializando atributos. Simulações orientadas a objetos frequentemente usam classes para representar entidades físicas.

Execução Direta do Script (`if __name__ == "__main__":`)

Essa construção permite que o arquivo seja usado tanto como:

- **script executável**, para rodar simulações;
- **módulo importável**, para aproveitar funções em outros códigos.

É padrão em projetos bem organizados.

2 Exemplos gerais

Antes de avançarmos para simulações mais elaboradas em Física, é importante construir uma base sólida sobre como comandos e estruturas básicas da linguagem Python funcionam na prática. Esta seção reúne pequenos programas exemplificando operações essenciais — desde cálculos matemáticos simples até estruturas de controle, criação de funções e manipulação de dados. O objetivo não é ensinar programação de forma exaustiva, mas fornecer um repositório de exemplos curtos, diretos e funcionais que sirvam como referência rápida durante o estudo. Cada exemplo é pensado para destacar um conceito crucial, mostrando como ele aparece no código real e como pode ser aplicado em situações típicas de resolução de problemas ou construção de simulações físicas. A seguir, começamos com operações matemáticas elementares, mostrando como utilizar a biblioteca `math` e como realizar algumas das operações mais frequentes em scripts Python.

Operações Matemáticas Simples

A seguir temos um pequeno programa em Python que ilustra como realizar operações matemáticas básicas, como soma, multiplicação e potenciação.

```
# Programa simples em Python que faz operações matemáticas básicas
import math  # biblioteca matemática

a = 2.0
```

```
b = 3.0

# soma
print("Soma:", a + b)

# produto
print("Produto:", a * b)

# potenciação
print("Potência:", math.pow(a, b))
```

Esse exemplo mostra como usar a biblioteca `math` e funções básicas de saída com `print` em Python.

Estruturas de Repetição (Loops)

Laços permitem repetir instruções várias vezes sem reescrevê-las. Um exemplo clássico é imprimir números em sequência usando o comando `for`.

```
# Programa que imprime os números de 1 a 10

# Loop que começa em 1 e vai até 10 (inclusive)
for i in range(1, 11):
    # Em cada passo, o valor de i é mostrado na tela
    print(i)

# Indica que o programa terminou corretamente
print("Fim do programa")
```

Esse programa mostra o uso do `for` em Python para repetir instruções de forma eficiente, sem precisar escrever 10 comandos de impressão.

Máximo e Mínimo de um Conjunto de Dados

Em Python, os dados são frequentemente armazenados em **listas**, que são estruturas de dados flexíveis e muito utilizadas. Uma lista é capaz de guardar diferentes valores (números, textos, etc.) em sequência, permitindo acesso direto a cada elemento através de índices (o primeiro índice é o zero). No exemplo abaixo, usamos uma lista simples de inteiros para ilustrar o cálculo dos valores máximo e mínimo.

```
# Programa que encontra o máximo e o mínimo de uma lista de dados
v = [3, 7, -2, 10, 5]   # definição de uma lista em Python

# Inicializa max e min com o primeiro elemento
max_val = v[0]
min_val = v[0]

# Percorre a lista comparando cada valor
for num in v[1:]:
    if num > max_val:
        max_val = num  # atualiza máximo
    if num < min_val:
        min_val = num  # atualiza mínimo

# Exibe os resultados finais
print("Max =", max_val, ", Min =", min_val)
```

Nesse código, criamos uma lista `v` com cinco elementos e usamos um laço `for` para percorrer seus valores. A cada iteração, comparamos o número atual com os valores já armazenados em `max\_val` e `min\_val`. Esse método é simples e eficiente: cada elemento da lista é verificado apenas uma vez. Além disso, em Python também é possível usar funções prontas, como `max(v)` e `min(v)`, que retornam diretamente o maior e o menor valor da lista, respectivamente. Isso mostra a flexibilidade da linguagem ao lidar com operações comuns de análise de dados.
Embora possamos encontrar o valor máximo e mínimo percorrendo manualmente a lista, como mostrado anteriormente, o Python oferece funções internas (`built-ins`) que tornam essa tarefa mais simples e direta. As funções `max()` e `min()` realizam exatamente esse trabalho de forma eficiente e com código muito mais conciso.

```
# Encontrando máximo e mínimo usando funções internas do Python
v = [3, 7, -2, 10, 5]

max_val = max(v)
min_val = min(v)

print("Max =", max_val, ", Min =", min_val)
```

Esse estilo de escrita aproveita os recursos da linguagem, reduz a chance de erros e deixa o programa mais legível, especialmente em operações simples e muito comuns como esta.

Séries de Taylor: Aproximação da Exponencial

A função exponencial e^x pode ser aproximada pela série de Taylor:

$$e^x \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

```
# Aproximação de e^x usando a série de Taylor
x = 1.0        # ponto de avaliação
soma = 1.0     # começa com o termo inicial (1)
termo = 1.0    # termo atual da série
N = 10         # número de termos usados

# Calcula a soma dos termos da série
for n in range(1, N+1):
    termo *= x / n      # atualiza termo: x^n / n!
    soma += termo       # adiciona à soma

# Mostra a aproximação obtida
print("Aproximacao de e^x em x =", x, ":", soma)
```

Esse código mostra como uma série infinita pode ser truncada para obter aproximações práticas de funções matemáticas.

Média e Desvio Padrão

Calcular estatísticas básicas, como a média aritmética e o desvio padrão, é essencial em diversas aplicações científicas. O exemplo a seguir mostra uma implementação direta desses cálculos usando as funções `sum()` e `len()`, combinadas com a biblioteca `math`.

```
# Calcula média e desvio padrão de uma lista
import math

v = [2.0, 4.0, 4.0, 4.0, 5.0]
n = len(v)

# Calcula média
media = sum(v) / n

# Calcula variância
var = 0.0
for valor in v:
    var += (valor - media)**2
var /= n

desvio = math.sqrt(var)

print("Media =", media, ", Desvio =", desvio)
```

Embora o Python ofereça funções internas que facilitam o cálculo da média (como `sum()`), também é instrutivo compreender como realizar o mesmo processo manualmente, acumulando valores com um laço. O código abaixo mostra o cálculo da média usando apenas operações elementares e um loop explícito:

```
# Cálculo da média "na força bruta", sem usar sum()
v = [2.0, 4.0, 4.0, 4.0, 5.0]

soma = 0.0
contador = 0

for valor in v:
    soma += valor
    contador += 1

media_bruta = soma / contador
print("Media (forca bruta) =", media_bruta)
```

A diferença entre os dois métodos está principalmente no nível de abstração. O primeiro utiliza ferramentas internas do Python, tornando o código mais curto, mais legível e menos propenso a erros. Já o método "na força bruta" ilustra como o cálculo efetivamente ocorre internamente: acumulando valores passo a passo. Essa abordagem é útil para fins didáticos e para compreender a lógica fundamental dos algoritmos estatísticos.

Tabuada com Loops Aninhados

Um exemplo simples do uso de loops duplos é a impressão da tabuada de multiplicação.

```
# Imprime a tabuada de 1 a 10
for i in range(1, 11):
    for j in range(1, 11):
        print(f"{i:2d} x {j:2d} = {i*j:2d}", end="   ")
    print()  # quebra de linha
```

Esse programa mostra como loops aninhados podem ser usados para gerar tabelas de valores.

Números Primos

Verificar se um número é primo é um problema clássico em programação. O código a seguir gera todos os primos até 50.

```
# Programa em Python que lista todos os números primos até 50
limite = 50

for n in range(2, limite+1):
    primo = True  # assume que o número é primo

    # Testa divisores de 2 até sqrt(n)
    for d in range(2, int(n 0.5)+1):
        if n % d == 0:   # se houver divisor exato
            primo = False
            break        # interrompe o laço

    if primo:
        print(n, end=" ")

print()  # quebra de linha
```

Esse código mostra como implementar um teste de primalidade simples.

Sequência de Fibonacci

A sequência de Fibonacci é definida por $F_0 = 0$, $F_1 = 1$ e

$$F_{n+1} = F_n + F_{n-1}.$$

```
# Gera os primeiros termos da sequência de Fibonacci
N = 10
f0, f1 = 0, 1

print(f0, f1, end=" ")

for i in range(2, N):
    fn = f0 + f1
    print(fn, end=" ")
    f0, f1 = f1, fn

print()
```

Esse código mostra como construir sequências recursivas de forma iterativa.

Leitura e Escrita em Arquivos

Manipular arquivos é uma tarefa fundamental em Python, especialmente quando lidamos com conjuntos de dados armazenados externamente. O exemplo abaixo mostra como salvar uma sequência de números em um arquivo texto e depois lê-los de volta para realizar uma operação simples:

```
# Escreve e lê números de um arquivo (exemplo básico)

# Escreve os números em um arquivo
with open("dados.txt", "w") as f:
    for i in range(1, 6):
```

```
        f.write(str(i) + "\n")

# Lê os números do arquivo e soma
soma = 0
with open("dados.txt", "r") as f:
    for linha in f:
        soma += int(linha.strip())

print("Soma dos numeros =", soma)
```

Esse exemplo introduz a estrutura básica para leitura e escrita usando o comando `with open()`, que garante o fechamento correto do arquivo. A seguir, mostramos como trabalhar com tabelas contendo várias colunas, um formato comum em aplicações científicas.

Exemplo: lendo um arquivo com 3 colunas
Suponha um arquivo chamado `tabela3.txt` com o seguinte conteúdo:

```
1.0  2.0  3.0
4.0  5.0  6.0
7.0  8.0  9.0
```

O Python pode ler cada linha, separar os valores e armazená-los em listas:

```
# Leitura de tabela com 3 colunas

col1 = []
col2 = []
col3 = []

with open("tabela3.txt", "r") as f:
    for linha in f:
        a, b, c = linha.split()
        col1.append(float(a))
        col2.append(float(b))
        col3.append(float(c))

print(col1)
print(col2)
print(col3)
```

Exemplo: arquivo com 4 ou 5 colunas (método genérico)
Para tabelas maiores, podemos usar um método mais flexível, que funciona para qualquer número de colunas:

```
# Leitura de tabela com número arbitrário de colunas

tabela = []

with open("tabela5.txt", "r") as f:
    for linha in f:
        valores = linha.split()
        linha_float = [float(v) for v in valores]
        tabela.append(linha_float)

print("Tabela lida:")
for linha in tabela:
    print(linha)
```

Nesse formato, cada linha do arquivo é convertida para uma lista de números, e todas as linhas são armazenadas em uma grande lista (uma "lista de listas"), formando uma estrutura semelhante a uma matriz. Essa abordagem é simples, poderosa e aparece com frequência em código científico. Assim, com poucos comandos, Python permite ler dados tabulares, manipular listas de valores e realizar cálculos subsequentes de forma clara e direta.

Mapa Logístico

O mapa logístico é um dos modelos mais estudados em dinâmica não-linear e sistemas caóticos. Ele descreve a evolução de uma população normalizada $x_n \in [0,1]$ ao longo de iterações discretas, sujeita a uma taxa de crescimento r. Sua forma é dada pela equação recursiva:

$$x_{n+1} = r\, x_n(1 - x_n).$$

Apesar de sua simplicidade, o mapa logístico apresenta uma rica variedade de comportamentos: desde convergência para um ponto fixo, passando por oscilações periódicas, até o aparecimento de caos determinístico para valores suficientemente altos de r. Ele se tornou um exemplo clássico da transição universal da ordem para o caos e é frequentemente utilizado para introduzir conceitos como bifurcações, sensibilidade às condições iniciais e expoentes de Lyapunov.
Para $0 < r < 1$, o sistema decai para 0. Já para $1 < r < 3$, ele converge para um ponto fixo diferente de zero. Entre $3 < r < 3.57$, o sistema passa por sucessivas duplicações de período até entrar em regime caótico. Essa progressão gera o famoso *diagrama de bifurcação*, um dos ícones da teoria do caos. Assim, o mapa logístico fornece um laboratório matemático perfeito para explorar ideias fundamentais da dinâmica não-linear utilizando apenas uma equação iterativa simples. O código para simular o mapa pode ser encontrado a seguir:

```
# Mapa logístico: gera dados para diagrama de bifurcação
r_min, r_max, r_step = 2.5, 4.0, 0.01
n_trans = 100   # iterações de transiente (não salvas)
n_iter = 50     # iterações salvas

with open("bifurcacao.txt", "w") as f:
    r = r_min
    while r <= r_max:
        x = 0.5  # condição inicial

        # Descarta transientes
        for _ in range(n_trans):
            x = r * x * (1 - x)

        # Salva os próximos valores de x
        for _ in range(n_iter):
            x = r * x * (1 - x)
            f.write(f"{r} {x}\n")

        r += r_step
```

Após executar este programa, o arquivo `bifurcacao.txt` conterá pares r e x, que podem ser utilizados para plotar o diagrama de bifurcação usando ferramentas gráficas.

Uma das grandes vantagens do uso do Python em Física Computacional é a sua capacidade de integrar, em um único código, tanto o cálculo numérico quanto a visualização gráfica dos resultados. Isso elimina a necessidade de exportar dados para outros programas e torna o fluxo de trabalho mais direto e eficiente. No exemplo a seguir, vamos implementar novamente o mapa logístico e incluir uma destas potencialidades usando a famosa biblioteca `matplotlib`.

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

# -----------------------
# Parâmetros do problema
# -----------------------
r_min, r_max, r_step = 2.5, 4.0, 0.001  # intervalo do parâmetro r
n_trans = 1000   # iterações descartadas (transiente)
n_iter = 200     # iterações que serão plotadas
x0 = 0.5         # condição inicial

# Vetor de valores de r
r_values = np.arange(r_min, r_max, r_step)

# Listas para armazenar os resultados
R, X = [], []

# -----------------------
# Loop principal
# -----------------------
for r in r_values:
    x = x0
    # Descartar o transiente
    for _ in range(n_trans):
        x = r * x * (1 - x)
    # Guardar os valores estáveis
    for _ in range(n_iter):
        x = r * x * (1 - x)
        R.append(r)
        X.append(x)

# -----------------------
# Gráfico do diagrama
# -----------------------
plt.figure(figsize=(8,6))
plt.plot(R, X, ',k', alpha=0.25)  # vírgula = marcador mínimo
plt.title("Diagrama de Bifurcação do Mapa Logístico")
plt.xlabel("r")
plt.ylabel("x")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig("bifurcacao_python.png", dpi=300)
plt.show()
```

Ao executar este código em Python, o programa não apenas realiza os cálculos, mas também gera diretamente a figura `bifurcacao_python.png`. Essa integração entre simulação e visualização é uma das maiores forças do Python no contexto da Física Computacional, tornando-o uma ferramenta poderosa tanto para ensino quanto para pesquisa.
A seguir detalhamos cada parte do código que gera o diagrama de bifurcação do mapa logístico. O programa é dividido em blocos funcionais, e cada um deles cumpre um papel

específico na construção da figura.

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

Esse bloco importa as bibliotecas fundamentais: `numpy`, utilizada para lidar com vetores e operações numéricas eficientes, e `matplotlib`, responsável pela criação de gráficos.

```
r_min, r_max, r_step = 2.5, 4.0, 0.001  # intervalo do parâmetro r
n_trans = 1000   # iterações descartadas (transiente)
n_iter = 200     # iterações que serão plotadas
x0 = 0.5         # condição inicial
```

Aqui definimos os parâmetros do mapa logístico. O valor de `r` varia em um intervalo contínuo, com passo muito pequeno para produzir um gráfico detalhado. Também definimos quantas iterações serão descartadas (`n_trans`) e quantas serão armazenadas para o diagrama (`n_iter`). A condição inicial `x0` fixa o ponto de partida da dinâmica.

```
r_values = np.arange(r_min, r_max, r_step)
```

Criamos um vetor com todos os valores de `r` que serão testados. O uso de `np.arange` garante alta performance e precisão adequada para o estudo do sistema dinâmico.

```
R, X = [], []
```

As listas `R` e `X` armazenarão, respectivamente, os valores do parâmetro `r` e os valores de `x` após o transiente. Esses pares formam o conjunto de pontos que compõem o diagrama de bifurcação.

```
for r in r_values:
    x = x0
    # Descartar o transiente
    for _ in range(n_trans):
        x = r * x * (1 - x)
    # Guardar os valores estáveis
    for _ in range(n_iter):
        x = r * x * (1 - x)
        R.append(r)
        X.append(x)
```

Este bloco é o núcleo do código. Para cada valor de r, o mapa logístico é iterado muitas vezes. As primeiras iterações são descartadas, pois representam o comportamento transiente. Em seguida, as iterações estáveis são armazenadas, fornecendo os pontos que refletem a dinâmica assintótica do sistema.

```
plt.figure(figsize=(8,6))
plt.plot(R, X, ',k', alpha=0.25)
plt.title("Diagrama de Bifurcação do Mapa Logístico")
plt.xlabel("r")
plt.ylabel("x")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig("bifurcacao_python.png", dpi=300)
plt.show()
```

Por fim, o gráfico é criado. O marcador `','` produz pontos extremamente pequenos, ideal para diagramas densos. O arquivo é salvo em alta resolução e exibido na tela. Esse acoplamento entre simulação e visualização ilustra bem a utilidade do Python em estudos numéricos e dinâmicos.

Manipulação básica de matrizes

A seguir mostramos como realizar operações fundamentais de Álgebra Linear em Python utilizando a biblioteca `NumPy`. Criamos duas matrizes A e B, calculamos o produto matricial $C = AB$ e, em seguida, o traço da matriz resultante.

```
# Exemplo de multiplicação de matrizes e cálculo do traço

import numpy as np

# Definição das matrizes A e B

A = np.array([[1, 2, 3],
[0, 1, 4],
[5, 6, 0]])

B = np.array([[1, 0, 2],
[3, 1, 1],
[0, 4, 2]])

# Produto matricial: C = A x B

C = np.dot(A, B)

print("Matriz A:\n", A)
print("\nMatriz B:\n", B)
print("\nProduto C = A x B:\n", C)

# Cálculo do traço de C (soma dos elementos da diagonal)

trace_C = np.trace(C)
print("\nTraço da matriz C:", trace_C)
```

Esse exemplo ilustra o uso das funções `np.dot()` para multiplicação de matrizes e `np.trace()` para obter o traço, duas operações fundamentais em vários problemas de Física e Matemática Aplicada.

3 Números Complexos em Python

Em Python, os números complexos são suportados nativamente pela linguagem, sem a necessidade de bibliotecas adicionais. Basta usar a letra `j` (em vez de `i`) para indicar a unidade imaginária. Por exemplo, o número $z = 2 + 3i$ pode ser escrito em Python como `z = 2 + 3j`. Além disso, funções matemáticas úteis para números complexos estão disponíveis no módulo `cmath`, que é uma extensão da biblioteca matemática para o domínio complexo.

```
# Operações básicas com números complexos em Python

import cmath   # biblioteca para funções matemáticas complexas

# Definição de dois números complexos
z1 = 2 + 3j
z2 = 1 - 4j
```

```
# Soma
soma = z1 + z2

# Produto
produto = z1 * z2

# Conjugado de z1
conj_z1 = z1.conjugate()

# Módulo ao quadrado de z1
modulo2 = abs(z1)   2

# Exponencial de z1
exp_z1 = cmath.exp(z1)

# Exibe os resultados
print("z1 =", z1)
print("z2 =", z2, "\n")

print("Soma =", soma)
print("Produto =", produto)
print("Conjugado de z1 =", conj_z1)
print("|z1|^2 =", modulo2)
print("Exp(z1) =", exp_z1)
```

Esse código ilustra as operações básicas com números complexos em Python: soma, produto, cálculo do conjugado, módulo e exponencial.

Como Python trata os complexos como *tipos nativos*, podemos usá-los de forma direta em expressões matemáticas. Por exemplo:

```
# Exemplo de operações diretas
z = 1 + 1j
print(z 2)       # potência
print(abs(z))     # módulo
print(cmath.phase(z))  # argumento (fase em radianos)
```

Isso mostra uma vantagem de Python em relação a outras linguagens: não é necessário definir estruturas especiais nem usar bibliotecas adicionais para trabalhar com números complexos. Eles já fazem parte da linguagem e podem ser combinados diretamente em qualquer expressão matemática.

4 Derivada Numérica em Python

A derivada de uma função $f(x)$ pode ser aproximada numericamente usando técnicas de diferenças finitas. Esses métodos são especialmente úteis quando a expressão analítica da derivada é difícil de obter ou quando a função só está disponível de forma discreta ou computacional.

Método da Diferença Progressiva

A fórmula da diferença progressiva surge a partir da expansão de Taylor de $f(x+h)$ em torno de x:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \cdots$$

Isolando $f'(x)$, obtemos:

$$f'(x) = \frac{f(x+h) - f(x)}{h} - \frac{h}{2}f''(x) + O(h^2)$$

Assim, a aproximação numérica:

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

tem erro de ordem $O(h)$. Isso significa que diminuindo h o erro diminui linearmente, embora valores muito pequenos de h possam amplificar erros de arredondamento.

```
# Derivada numérica: diferença progressiva
import math

# Função a derivar
def f(x):
    return math.sin(x)

x = 1.0
h = 0.001

# Fórmula da diferença progressiva
deriv = (f(x + h) - f(x)) / h
print(f"Derivada aproximada em x={x}: {deriv}")
```

O método é simples e computacionalmente barato, mas a precisão é limitada pela ordem do erro.

Diferença Central

Uma abordagem mais precisa utiliza valores simetricamente distribuídos em torno de x. Usando as expansões de Taylor:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4)$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4)$$

Subtraindo as duas expressões:

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{2h^3}{6}f'''(x) + O(h^5)$$

Dividindo por $2h$:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6}f'''(x) + O(h^4)$$

Assim, a fórmula da diferença central é:

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

com erro de ordem $O(h^2)$, consideravelmente menor do que na diferença progressiva.

```
# Derivada numérica: diferença central
import math

def f(x):
    return math.cos(x)

x = 1.0
h = 0.001

# Fórmula da diferença central
deriv = (f(x + h) - f(x - h)) / (2 * h)
print(f"Derivada (central) em x={x}: {deriv}")
```

A vantagem principal desse método é que, ao usar pontos de ambos os lados, os termos pares da expansão de Taylor se cancelam, reduzindo o erro de truncamento. Consequentemente, a diferença central é, para o mesmo valor de h, muito mais precisa do que a diferença progressiva ou regressiva.

Derivadas a partir de um vetor de dados

Se dispomos de um vetor de dados representando valores de x e $y = f(x)$, podemos calcular aproximações da derivada usando diferenças finitas entre elementos consecutivos ou diferenças centrais. Este método é útil quando só temos dados discretos, seja de experimentos ou de simulações numéricas.

```
# Derivada de dados tabelados

x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]  # exemplo: y = x^2

# diferenças centrais
for i in range(1, len(x)-1):
    deriv = (y[i+1] - y[i-1]) / (x[i+1] - x[i-1])
    print(f"x={x[i]} -> derivada ~ {deriv}")
```

Este método permite estimar a derivada mesmo sem conhecer a função analiticamente. É especialmente útil para dados experimentais ou resultados de simulações discretas. A abordagem com diferenças centrais melhora a precisão em relação à diferença progressiva.

Derivadas a partir de Arquivo de Dados

Quando temos dados experimentais armazenados em um arquivo (por exemplo `tabela.dat`), podemos calcular a derivada aproximada usando diferenças finitas entre pontos consecutivos. Abaixo mostramos um exemplo simples.
Exemplo de arquivo `tabela.dat` (duas colunas: x e y):

```
0.0 0.0
0.5 0.25
1.0 1.0
1.5 2.25
2.0 4.0
2.5 6.25
3.0 9.0
```

```
# Derivada de dados lidos de arquivo

# Lê os dados do arquivo
x, y = [], []
with open("tabela.dat", "r") as f:
    for linha in f:
        xi, yi = map(float, linha.split())
        x.append(xi)
        y.append(yi)

# Calcula derivadas usando diferenças centrais
for i in range(1, len(x)-1):
    deriv = (y[i+1] - y[i-1]) / (x[i+1] - x[i-1])
    print(f"x={x[i]:.2f} -> derivada ~ {deriv:.2f}")
```

Após executar este programa, serão exibidas no terminal as derivadas aproximadas para cada ponto (exceto nos extremos). Este método é útil para analisar dados experimentais ou simulações, permitindo obter derivadas mesmo sem conhecer a função analiticamente.

5 Integração Numérica

A integral definida pode ser aproximada numericamente por métodos simples. Podemos implementar a **regra dos retângulos** em Python. O método consiste em dividir o intervalo $[a, b]$ em N partes iguais e aproximar a área sob a curva pela soma das áreas de retângulos de base $h = (b - a)/N$. Assim,

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

onde $x_i = a + ih$. Python permite expressar esse algoritmo de forma clara e compacta.

```
# Integração numérica usando a regra dos retângulos

import math

# Função que desejamos integrar
def f(x):
    return math.sin(x)

# Limites de integração
a = 0.0
b = math.pi

# Número de subdivisões
N = 100
h = (b - a) / N

soma = 0.0

# Soma dos valores da função nos pontos da esquerda
for i in range(N):
    x_i = a + i * h
    soma += f(x_i)

integral = h * soma

print("Integral aproximada (retangulos):", integral)
```

O código reproduz exatamente o método numérico apresentado anteriormente: cada retângulo tem base h e altura dada pelo valor da função no ponto inicial de cada subintervalo. A estrutura em Python torna o código mais conciso e legível, sem alterar o algoritmo.

Regra do Trapézio

A regra do trapézio fornece uma aproximação mais precisa da integral definida ao substituir cada retângulo por um trapézio. Em cada subintervalo, utiliza-se a média entre os valores da função nos extremos. A fórmula é

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

onde $h = (b-a)/N$ e $x_i = a + ih$. Esse método é simples, eficiente e apresenta erro significativamente menor do que a regra dos retângulos.

```
# Integração numérica usando a regra do trapézio

import math

# função a integrar
def f(x):
    return math.sin(x)

a = 0.0       # limite inferior
b = math.pi   # limite superior
N = 100       # número de subdivisões
h = (b - a) / N

# soma dos pontos internos
soma = 0.0
for i in range(1, N):
    x_i = a + i * h
    soma += f(x_i)

# cálculo final da integral
integral = h * ( (f(a) + f(b)) / 2 + soma )

print("Integral aproximada (trapezio) =", integral)
```

O algoritmo consiste em acumular os valores da função nos pontos internos e somar a contribuição dos extremos, ponderados por 1/2. Esse método é amplamente usado em aplicações práticas graças à sua precisão e simplicidade de implementação.

Integração a partir de Arquivo de Dados

Em muitas situações, não dispomos de uma expressão analítica para a função, mas apenas de valores tabulados obtidos experimentalmente ou por simulações. Nestes casos, ainda podemos aproximar a integral usando a regra do trapézio. A ideia é aplicar o método diretamente aos pares (x_i, y_i) lidos de um arquivo.

Exemplo de arquivo `tabela2.dat`:

```
0.0 0.0
0.5 0.25
```

```
1.0 1.0
1.5 2.25
2.0 4.0
2.5 6.25
3.0 9.0
```

```
# Integração usando dados tabulados e regra do trapézio

# leitura dos dados
x = []
y = []

with open("tabela2.dat", "r") as fp:
    for linha in fp:
        xi, yi = linha.split()
        x.append(float(xi))
        y.append(float(yi))

# cálculo da integral aproximada
soma = 0.0
for i in range(len(x) - 1):
    dx = x[i+1] - x[i]
    soma += (y[i] + y[i+1]) * dx / 2.0

print("Integral aproximada =", soma)
```

A integral é construída somando a área de vários trapézios, cada um definido pelos pares consecutivos (x_i, y_i). Esse procedimento é extremamente útil para dados discretos e aparece com frequência em análises científicas.

Integral Dupla

Para funções de duas variáveis, podemos aproximar integrais duplas dividindo a região de integração em uma grade retangular. Cada ponto contribui com $f(x_i, y_j)\,\Delta x\,\Delta y$, de modo que a soma dupla aproxima a integral:

$$\iint f(x,y)\,dx\,dy \approx \sum_{i=0}^{N_x-1} \sum_{j=0}^{N_y-1} f(x_i, y_j)\,\Delta x\,\Delta y.$$

Essa técnica é direta e funciona bem quando a função é suave na região considerada.

```
# Integral dupla simples: f(x,y) = x*y

def f(x, y):
    return x * y

Nx, Ny = 50, 50          # pontos na direção x e y
ax, bx = 0.0, 1.0        # limites em x
ay, by = 0.0, 1.0        # limites em y

hx = (bx - ax) / Nx
hy = (by - ay) / Ny

soma = 0.0
```

```
for i in range(Nx):
    for j in range(Ny):
        x = ax + i * hx
        y = ay + j * hy
        soma += f(x, y)

integral = soma * hx * hy

print("Integral dupla aproximada =", integral)
```

O método consiste em percorrer toda a grade, avaliando a função em cada ponto e multiplicando pelo elemento de área $\Delta x\, \Delta y$. Essa técnica é a base de muitos esquemas numéricos para resolver problemas em duas dimensões, especialmente em física computacional, métodos de volumes finitos e simulação de campos contínuos.

6 Introdução aos Números Aleatórios

Em Python, números aleatórios são gerados pelo módulo padrão `random`, que implementa um gerador **pseudoaleatório**. Assim como em C, a sequência depende de uma semente inicial, mas Python fornece funções de mais alto nível e mais convenientes.

Gerando Números Inteiros Aleatórios

Para gerar inteiros pseudoaleatórios no intervalo $[a, b]$, usa-se `random.randint(a,b)`:

```
# Geração de números inteiros pseudoaleatórios
import random

for i in range(5):
    r = random.randint(0, 100)  # inteiro entre 0 e 100
    print(r)
```

A cada execução, a sequência será a mesma, a menos que se defina explicitamente a semente.

Definindo a Semente do Gerador

Para variar a sequência de números pseudoaleatórios, usamos `random.seed()`:

```
# Definindo uma semente variável
import random
import time

random.seed(time.time())  # usa o tempo atual como semente
for i in range(5):
    print(random.randint(0, 100))
```

Também é possível apenas chamar `random.seed()` sem argumentos, e Python usa uma semente baseada no sistema operacional.

Distribuição Uniforme no Intervalo [0,1]

Para gerar números reais uniformemente distribuídos no intervalo $[0, 1]$, usamos:

$$x = \texttt{random.random()}$$

```
# Números uniformes em [0,1]
import random

for i in range(5):
    x = random.random()
    print(x)
```

Estes valores são utilizados em simulações, métodos de Monte Carlo e vários algoritmos estatísticos.

Histograma de Números Aleatórios Uniformes

Um histograma pode ser construído dividindo-se o intervalo $[0, 1]$ em subintervalos ("bins") e contando quantos números caem em cada região.

```
# Histograma de números aleatórios uniformes
import random

N = 1000
bins = 10
hist = [0]*bins   # vetor para contagem

for i in range(N):
    u = random.random()     # número em [0,1]
    k = int(u * bins)       # identifica o bin
    if k >= bins:
        k = bins - 1
    hist[k] += 1

# exibe o histograma
for i in range(bins):
    print(f"Bin {i}: {hist[i]}")
```

Se N for suficientemente grande, os valores devem se distribuir aproximadamente de modo uniforme.

Distribuição $P(x) = Cx^n$

Para gerar números distribuídos segundo

$$P(x) \sim x^n \quad \text{em} \quad [0, 1],$$

usamos a transformação:

$$x = u^{1/(n+1)}, \qquad u \in [0, 1].$$

O código abaixo gera tais números, salva em arquivo e calcula um histograma simples.

```
# Geração de números com P(x) ~ x^n e histograma
import random
import math

n = 2           # expoente da distribuição
N = 1000        # quantidade de números
bins = 10
hist = [0]*bins

# salva valores em arquivo
with open("numeros.txt", "w") as f:
    for i in range(N):
        u = random.random()
        x = u**(1/(n+1))     # transformação
        f.write(f"{x}\n")

        k = int(x * bins)
        if k >= bins:
            k = bins - 1
        hist[k] += 1

# salva histograma em arquivo
with open("histograma.txt", "w") as f:
    for i in range(bins):
        f.write(f"{i} {hist[i]}\n")

# mostra na tela
for i in range(bins):
    print(f"Bin {i}: {hist[i]}")
```

O arquivo `numeros.txt` conterá os números distribuídos como x^n, e o arquivo `histograma.txt` registrará o histograma correspondente.

Geradores Lineares Congruentes

Embora Python possua um gerador próprio, também podemos implementar nosso próprio **gerador linear congruente** (LCG):

$$X_{k+1} = (aX_k + c) \bmod m$$

A seguir geramos uma sequência de números normalizados em $[0, 1]$ usando um LCG simples e gravamos em arquivo:

```
# Gerador linear congruente (LCG) e escrita em arquivo
a = 1664525
c = 1013904223
m = 2**31
x = 12345        # semente inicial
N = 1000

with open("lcg_python.dat", "w") as f:
    for _ in range(N):
        x = (a*x + c) % m
        u = x / m      # normaliza para [0,1]
        f.write(f"{u}\n")
```

O arquivo `lcg_python.dat` conterá 1000 números pseudoaleatórios gerados manualmente pelo LCG.

Distribuição Gaussiana (Box–Muller)

A distribuição normal pode ser gerada usando o método de Box–Muller, que transforma dois números uniformes $u_1, u_2 \in [0,1]$ em números gaussianos:

$$z = \sqrt{-2\ln u_1}\cos(2\pi u_2)$$

O código abaixo gera números gaussianos, grava o resultado e produz um histograma simples:

```
# Geração de números gaussianos via Box-Muller
import random
import math

N = 1000
bins = 20
hist = [0]*bins

with open("gauss_python.dat","w") as f:
    for _ in range(N):
        u1 = random.random()
        u2 = random.random()
        z = math.sqrt(-2*math.log(u1)) * math.cos(2*math.pi*u2)
        f.write(f"{z}\n")

        k = int((z + 4) / 8 * bins)   # intervalo aproximado [-4,4]
        if 0 <= k < bins:
            hist[k] += 1

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

O arquivo `gauss_python.dat` inclui os valores gaussianos, enquanto `hist_gauss_python.dat` contém a contagem em cada classe do histograma.

Distribuição Lorentziana

A distribuição de Lorentz é dada por:

$$x = \tan\left[\pi\left(u - 0.5\right)\right], \quad u \in (0,1)$$

Ela possui "caudas largas", resultando em maior probabilidade de valores extremos. O código abaixo gera valores Lorentzianos e um histograma simples:

```
# Geração de números Lorentzianos e histograma
import random
import math

N = 1000
bins = 20
x_min, x_max = -10, 10
dx = (x_max - x_min)/bins
hist = [0]*bins

with open("lorentz_python.dat","w") as f:
    for _ in range(N):
        u = random.random()
```

```
        x = math.tan(math.pi*(u - 0.5))
        f.write(f"{x}\n")

        if x_min <= x < x_max:
            k = int((x - x_min)/dx)
            hist[k] += 1

with open("hist_lorentz_python.dat","w") as f:
    for i in range(bins):
        center = x_min + (i+0.5)*dx
        f.write(f"{center} {hist[i]}\n")
```

O histograma registrado em `hist_lorentz_python.dat` permite visualizar as características de cauda pesada.

Autocorrelação Simples

A autocorrelação é uma ferramenta fundamental para analisar dependências internas de uma série temporal. Para uma sequência de valores x_i (com $i = 1, \ldots, N$), a autocorrelação no lag k quantifica o quanto valores separados por k passos estão linearmente relacionados. Em outras palavras, ela mede se o sistema "lembra" do seu estado após k iterações.
A definição padrão não normalizada da autocorrelação é:

$$C(k) = \frac{1}{N-k} \sum_{i=1}^{N-k} (x_i - \bar{x})(x_{i+k} - \bar{x}),$$

onde $\bar{x}$ é a média amostral:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i.$$

Para $k = 0$, obtemos:

$$C(0) = \frac{1}{N} \sum_{i=1}^{N} (x_i - \bar{x})^2,$$

que é simplesmente a variância da série. Para $k > 0$, valores positivos de $C(k)$ indicam que termos grandes tendem a ser seguidos por termos grandes, enquanto valores negativos sugerem alternância (comportamento oscilatório). Já valores próximos de zero indicam ausência de dependência linear, típico de ruído branco.
O comportamento do decaimento de $C(k)$ com o lag é especialmente relevante em simulações estocásticas. Um decaimento lento indica forte correlação temporal e aumenta o chamado *tempo de correlação*, reduzindo a eficiência estatística. Em muitos processos de Monte Carlo, espera-se um decaimento aproximadamente exponencial,

$$C(k) \sim e^{-k/\tau},$$

onde τ é o tempo de autocorrelação.
O exemplo abaixo gera uma série uniforme, calcula a média e em seguida a autocorrelação até um lag máximo definido pelo usuário. Como a série é aleatória e não correlacionada, esperamos que $C(k) \approx 0$ para $k > 0$, com flutuações de ordem $1/\sqrt{N}$.

```
# Autocorrelação simples de uma série uniforme
import random

N = 2000
```

```
MAX_LAG = 50

x = [random.random() for _ in range(N)]
media = sum(x)/N
C = []

for k in range(MAX_LAG+1):
    soma = 0.0
    for i in range(N-k):
        soma += (x[i] - media)*(x[i+k] - media)
    C.append(soma/(N-k))
    print(f"Lag {k}: C = {C[k]}")
```

A saída exibe a autocorrelação para cada lag. Como esperado para ruído branco, os valores para $k > 0$ flutuam em torno de zero.

Exemplo: cálculo da autocorrelacao em Série com Correlação Gaussiana

Para ilustrar como a autocorrelação pode ser calculada em um conjunto de dados, vamos construir uma série temporal que *não* seja ruído branco, mas sim um processo com correlação de curto alcance. Uma forma simples e bastante eficiente de produzir tal série é suavizar um conjunto de números aleatórios através de um núcleo (kernel) Gaussiano.

Considere uma sequência de valores aleatórios A_m distribuídos uniformemente no intervalo $[-1, 1]$. A série correlacionada x_n é construída como

$$x_n = \sum_{m=1}^{N} A_m \exp\left[-\frac{(n-m)^2}{L^2}\right],$$

onde L controla o **comprimento de correlação**. Quanto maior o valor de L, mais suave será a série resultante, pois cada ponto x_n será influenciado por um número maior de termos vizinhos.
A função Gaussiana atua como um filtro de suavização: ela atribui maior peso aos termos A_m próximos de n e peso exponencialmente menor aos mais distantes. Assim, a série final apresenta correlação temporal cuja largura é controlada justamente por L.

O código abaixo implementa esse procedimento e, em seguida, calcula a autocorrelação $C(k)$ para diferentes valores de L. A expectativa é que:

- Para valores pequenos de L, a série é apenas levemente suavizada e a autocorrelação decai muito rapidamente.
- Para valores moderados, o decaimento de $C(k)$ torna-se mais lento.
- Para valores grandes de L, a série apresenta flutuações amplamente suavizadas e a autocorrelação se estende por muitos passos.

```
# Série temporal com correlação Gaussiana e gráfico das autocorrelações
import math
import random
import matplotlib.pyplot as plt

def gera_serie_correlacionada(N, L):
    # A_m: números aleatórios entre -1 e 1
    A = [2*random.random() - 1 for _ in range(N)]
    x = []
```

```
    for n in range(N):
        soma = 0.0
        for m in range(N):
            soma += A[m] * math.exp(-((n-m)**2)/(L*L))
        x.append(soma)
    return x

def autocorrelacao(x, MAX_LAG):
    N = len(x)
    media = sum(x)/N
    C = []
    for k in range(MAX_LAG+1):
        soma = 0.0
        for i in range(N-k):
            soma += (x[i] - media)*(x[i+k] - media)
        C.append(soma/(N-k))
    return C

# Parâmetros gerais
N = 2000
MAX_LAG = 50
Ls = [2, 4, 6]

plt.figure(figsize=(8,5))

for L in Ls:
    x = gera_serie_correlacionada(N, L)
    C = autocorrelacao(x, MAX_LAG)
    plt.plot(range(MAX_LAG+1), C, label=f"L={L}")

plt.xlabel("Lag k")
plt.ylabel("C(k)")
plt.title("Autocorrelação para diferentes comprimentos de correlação L")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

O gráfico gerado pelo código ilustra visualmente essa dependência do comprimento de correlação L. As três curvas se separam de forma nítida: quanto maior o valor de L, mais lenta é a queda de $C(k)$, indicando que a série "carrega memória" por um número maior de passos. Essa separação não é apenas qualitativa; ela mostra como a convolução Gaussiana aplicada na construção da série introduz uma suavização cujo alcance é proporcional a L. Assim, ao observarmos o decaimento das curvas no gráfico — rápido para $L = 2$, moderado para $L = 4$ e lento para $L = 6$ — percebemos diretamente a relação entre o filtro imposto e o comportamento estatístico resultante. Este exemplo evidencia como a autocorrelação funciona como uma ferramenta sensível para diagnosticar estruturas internas de dependência temporal e reforça a importância de controlar L ao projetar ou analisar simulações numéricas, especialmente em problemas onde a eficiência estatística depende do tempo de correlação da série.

Produtos Aleatórios de Matrizes e o Teorema de Furstenberg

Um dos resultados centrais da teoria ergódica aplicada a sistemas lineares aleatórios é o **Teorema de Furstenberg**, que estabelece condições sob as quais produtos de matrizes aleatórias em $SL(2, \mathbb{R})$ exibem um **expoente de Lyapunov positivo**. Esse comportamento está

na base de diversos fenômenos físicos, como a localização de Anderson em 1D, a estabilidade de soluções diferenciais sujeitas a ruído multiplicativo e a dinâmica de sistemas lineares com coeficientes aleatórios. Nesta subseção apresentamos um pequeno experimento numérico que ilustra o fenômeno essencial previsto pelo teorema: ao multiplicarmos matrizes aleatórias independentes, a norma de um vetor inicial cresce tipicamente de forma exponencial, de modo que o expoente de Lyapunov associado ao produto é positivo.

Exemplo Numérico com Matrizes Aleatórias

Considere uma sequência de matrizes aleatórias 2×2 da forma

$$M_n = \begin{pmatrix} \cos\theta_n & -\sin\theta_n \\ \sin\theta_n & \cos\theta_n \end{pmatrix} \begin{pmatrix} \lambda_n & 0 \\ 0 & 1/\lambda_n \end{pmatrix},$$

onde θ_n é um ângulo aleatório uniforme em $[0, 2\pi]$ e λ_n é um fator de estiramento aleatório positivo. O produto dessas matrizes,

$$M_N M_{N-1} \cdots M_1,$$

gera um comportamento multiplicativo típico, e pode ser usado para estimar numericamente o expoente de Lyapunov associado. O método computacional consiste em aplicar sucessivamente as matrizes aleatórias sobre um vetor inicial v_0, produzindo a sequência

$$v_n = M_n M_{n-1} \cdots M_1 \, v_0.$$

Entretanto, a norma de v_n cresce (ou decai) exponencialmente com n, o que rapidamente leva a *overflow* ou *underflow* numérico. Para contornar esse problema, utiliza-se uma técnica padrão: após cada iteração, normalizamos o vetor e registramos separadamente o fator de normalização.
Mais precisamente, escrevemos

$$v_n = \frac{w_n}{\|w_n\|}, \qquad w_n = M_n v_{n-1},$$

e acumulamos o valor $\log \|w_n\|$ ao longo do processo. Dessa forma, mesmo que v_n permaneça sempre com norma unitária no computador, toda a informação sobre o crescimento do produto é preservada na soma logarítmica

$$S_N = \sum_{n=1}^{N} \log \|w_n\|.$$

O **expoente de Lyapunov** é então estimado pela média logarítmica

$$\gamma_N = \frac{1}{N} S_N = \frac{1}{N} \sum_{n=1}^{N} \log \|M_n v_{n-1}\|,$$

que converge para o expoente verdadeiro γ quando N é suficientemente grande. Em outras palavras,

$$\gamma = \lim_{N \to \infty} \frac{1}{N} \log \|M_N M_{N-1} \cdots M_1\|.$$

Essa estratégia de normalização periódica é essencial para evitar erros numéricos sem alterar o valor do expoente de Lyapunov, pois apenas decomponha o crescimento global em contribuições locais registradas passo a passo.

```
import numpy as np

def lyapunov_random_matrices(N=200000):
    """
    Calcula numericamente o expoente de Lyapunov associado a um
    produto de matrizes aleatórias 2x2 usando normalização periódica.
    """

    v = np.array([1.0, 0.0])  # vetor inicial
    log_norm = 0.0            # acumula crescimento logarítmico

    for n in range(N):
        # ângulo aleatório uniforme
        theta = np.random.uniform(0, 2*np.pi)

        # estiramento aleatório positivo
        lambda_n = np.random.uniform(1.0, 2.0)

        # matriz de rotação
        R = np.array([[np.cos(theta), -np.sin(theta)],
                      [np.sin(theta),  np.cos(theta)]])

        # matriz de estiramento
        S = np.array([[lambda_n,       0.0],
                      [0.0,       1.0/lambda_n]])

        # matriz aleatória = R S
        M = R @ S

        # aplica transformação no vetor
        v = M @ v

        # normalização periódica
        norm = np.linalg.norm(v)
        v = v / norm

        # acumula crescimento log do produto
        log_norm += np.log(norm)

    # expoente de Lyapunov estimado
    gamma = log_norm / N
    return gamma

# Exemplo de execução:
gamma_est = lyapunov_random_matrices(100000)
print("Expoente de Lyapunov estimado:", gamma_est)
```

A execução do algoritmo acima produz, tipicamente, valores positivos para o expoente de Lyapunov,

$$\gamma \approx 0.2 - 0.5,$$

dependendo dos intervalos escolhidos para os parâmetros aleatórios. O fato essencial é que $\gamma > 0$, uma manifestação numérica direta do Teorema de Furstenberg: sob condições de não-comutatividade e distribuição não-degenerada, o produto de matrizes aleatórias gera um crescimento exponencial típico. Esse comportamento emerge de maneira robusta e universal, e serve como ponto de partida para análises mais profundas, como a teoria de localização de Anderson via matrizes de transferência.

Caminhante Aleatório Clássico

O modelo de caminhante aleatório unidimensional é uma das construções mais simples e fundamentais da física estatística. Ele descreve um processo de difusão no qual uma partícula executa passos elementares iguais, porém aleatórios, ao longo de uma linha. A posição x_i após i passos é atualizada segundo

$$x_{i+1} = x_i \pm 1,$$

com probabilidade igual para os dois sentidos. Esse processo constitui uma realização de uma cadeia de Markov simples, na qual cada passo depende apenas do estado atual e não da história anterior.
Do ponto de vista estatístico, o caminhante aleatório é um exemplo paradigmático de processo difusivo, pois o deslocamento quadrático médio,

$$\langle x^2 \rangle (n) = \frac{1}{n} \sum_{i=1}^{n} x_i^2,$$

cresce linearmente com o número de passos n. Esse comportamento,

$$\langle x^2 \rangle \propto n,$$

é uma marca da difusão clássica e contrasta com processos superdifusivos ou subdifusivos, que apresentam expoentes diferentes.

Simulação Computacional

O código abaixo implementa o caminhante aleatório em 1D e calcula a evolução temporal de $\langle x^2 \rangle$. A cada passo o programa atualiza a posição, acumula o valor de x^2 e grava no arquivo `x2_python.dat` o número do passo e o valor médio correspondente.

```
# Caminhante aleatório clássico em 1D
import random

Npassos = 1000
pos = 0
x2_acum = 0.0

with open("x2_python.dat", "w") as f:
    for i in range(1, Npassos+1):
        passo = random.choice([-1, 1])
        pos += passo
        x2_acum += pos*pos
        f.write(f"{i} {x2_acum/i}\n")
```

Ao plotar os dados gerados, observa-se claramente a tendência linear esperada para $\langle x^2 \rangle$, refletindo a natureza difusiva do processo.

Animação Simples do Caminhante Aleatório

Para tornar a caminhada aleatória mais visual e intuitiva, podemos usar a biblioteca `matplotlib` para criar uma animação simples com um ponto representando o caminhante e um rastro mostrando seu caminho. Essa abordagem mantém o código acessível a iniciantes e produz uma representação visualmente mais bonita do movimento errático.
O código abaixo cria a animação:

```
# Animação simples de caminhante aleatório em 1D com matplotlib
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation

Npassos = 200
x = [0]  # posição inicial
rastro = [0]  # histórico para o rastro

fig, ax = plt.subplots()
ax.set_xlim(-30, 30)
ax.set_ylim(-0.2, 0.2)
point, = ax.plot([], [], 'ro', markersize=10)   # caminhante
line, = ax.plot([], [], 'b-', alpha=0.25)      # rastro

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

def update(frame):
    passo = random.choice([-1, 1])
    x.append(x[-1] + passo)
    rastro.append(x[-1])
    point.set_data(x[-1], 0)
    line.set_data(rastro, [0]*len(rastro))
    return point, line

ani = animation.FuncAnimation(fig, update, frames=Npassos, init_func=init,
                              blit=True, interval=50)
plt.show()
```

Neste código:

- O caminhante é representado por um ponto vermelho (`ro`) que se move ao longo do eixo horizontal.
- O histórico do movimento é mostrado por uma linha azul semitransparente, criando um efeito de rastro.
- A animação é executada em tempo real, com cada passo aparecendo a cada 50 ms.

Vamos apresentar agora um detalhamento de cada parte do código.

```
# Importa bibliotecas
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Aqui importamos:

- `random` para gerar os passos aleatórios do caminhante;
- `matplotlib.pyplot` para criar gráficos;
- `matplotlib.animation` para animar o gráfico em tempo real.

```
Npassos = 200
x = [0]        # posição inicial
rastro = [0]   # histórico para o rastro
```

Define-se:

- `Npassos`: número total de passos do caminhante;
- `x`: lista que armazena a posição atual e histórica do caminhante;
- `rastro`: lista que mantém o histórico para desenhar a linha do rastro.

```
fig, ax = plt.subplots()
ax.set_xlim(-30, 30)
ax.set_ylim(-0.2, 0.2)
point, = ax.plot([], [], 'ro', markersize=10)   # caminhante
line, = ax.plot([], [], 'b-', alpha=0.25)      # rastro
```

Aqui configuramos o gráfico:

- `fig, ax = plt.subplots()`: cria a figura e os eixos;
- `ax.set_xlim` e `ax.set_ylim`: definem os limites visíveis do gráfico;
- `point`: ponto vermelho representando o caminhante (`'ro'` indica red circle);
- `line`: linha azul semitransparente representando o rastro (`alpha=0.25` ajusta a transparência).

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

Função de inicialização da animação:

- Limpa os dados do ponto e do rastro antes de iniciar;
- Retorna os objetos que serão animados (`point, line`).

```
def update(frame):
    passo = random.choice([-1, 1])
    x.append(x[-1] + passo)
    rastro.append(x[-1])
    point.set_data(x[-1], 0)
    line.set_data(rastro, [0]*len(rastro))
    return point, line
```

Função chamada a cada frame da animação:

- Gera um passo aleatório de -1 ou $+1$;
- Atualiza a posição atual e adiciona ao histórico `rastro`;
- Atualiza a posição do ponto (`point`) e do rastro (`line`);
- Retorna os objetos animados.

```
ani = animation.FuncAnimation(fig, update, frames=Npassos, init_func=init,
                              blit=True, interval=50)
plt.show()
```

Por fim:

- `FuncAnimation` cria a animação, chamando `update` em cada frame;
- `frames=Npassos` define o número total de frames;
- `init_func=init` usa a função de inicialização;
- `blit=True` melhora a eficiência redesenhando apenas os elementos que mudam;
- `interval=50` define o intervalo de tempo entre frames em milissegundos;
- `plt.show()` exibe a animação na tela.

O resultado é uma animação simples, clara e visualmente agradável do caminhante aleatório, com o ponto vermelho representando a posição atual e a linha azul semitransparente mostrando o rastro histórico.

6.1 Animação do Caminhante Aleatório em 2D

O caminhante aleatório em duas dimensões é uma generalização natural do caso unidimensional: uma partícula (ou agente) realiza uma sequência de passos aleatórios sobre uma malha (lattice) ou no plano contínuo. No modelo mais simples (lattice 2D), a cada passo a partícula move-se uma unidade para cima, baixo, direita ou esquerda com igual probabilidade. Esse modelo é um elemento didático central em física estatística, teoria das probabilidades e processos estocásticos. Fisicamente, ele exemplifica difusão e a relação $\langle r^2 \rangle \propto t$ (movimento browniano simplificado).

A subseção a seguir apresenta um código simples em Python que cria uma animação usando `matplotlib`. A animação mostra um ponto representando o caminhante e um rastro (trajetória) dos passos anteriores. O código é escrito de forma clara e comentada para uso em aulas e notas.

```
# Animação de um caminhante aleatório em 2D (lattice) com matplotlib
# - O caminhante começa na origem (0,0)
# - Em cada passo move-se uma unidade: direita, esquerda, cima ou baixo (prob. igual)
# - A trajetória (trail) é desenhada e o ponto atual é destacado
# - É possível ajustar N_steps, interval e limites de plot

import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation

# ----------------------
# Parâmetros da simulação
# ----------------------
N_steps = 800           # número de passos da animação
step_size = 1           # tamanho do passo na grade (1 = lattice unit)
x = [0]                 # histórico de coordenada x (inicia na origem)
y = [0]                 # histórico de coordenada y
trail_length = 200      # número máximo de pontos exibidos no rastro

# ----------------------
```

```
# Configuração da figura
# ----------------------
fig, ax = plt.subplots(figsize=(6,6))
ax.set_xlim(-40, 40)
ax.set_ylim(-40, 40)
ax.set_aspect('equal')

# ponto atual do caminhante (vermelho)
walker_point, = ax.plot([], [], 'ro', markersize=8, label='Walker')

# rastro/trajectory (linha azul semi-transparente)
trail_line, = ax.plot([], [], 'b-', alpha=0.4, linewidth=1)

# nuvem de visitas (pontos pretos transparentes) -- opcional
visits_scatter = ax.scatter([], [], s=8, c='black', alpha=0.02)

ax.set_title('2D Random Walk (Lattice)', color='black')
ax.legend(loc='upper right')

# ----------------------
# Funções de animação
# ----------------------
def init():
    walker_point.set_data([], [])
    trail_line.set_data([], [])
    visits_scatter.set_offsets([])   # limpa scatter
    return walker_point, trail_line, visits_scatter

def update(frame):
    # escolhe um passo aleatório entre os quatro vizinhos
    move = random.choice([(1,0), (-1,0), (0,1), (0,-1)])
    new_x = x[-1] + move[0] * step_size
    new_y = y[-1] + move[1] * step_size

    x.append(new_x)
    y.append(new_y)

    # atualiza ponto do caminhante
    walker_point.set_data(new_x, new_y)

    # atualiza rastro: mantemos apenas os últimos 'trail_length' pontos
    xs = x[-trail_length:]
    ys = y[-trail_length:]
    trail_line.set_data(xs, ys)

    # atualiza scatter de visitas (opcional: mostra densidade de pontos visitados)
    # aqui convertemos histórico em array de coordenadas; para eficiência em simulações
    # maiores pode-se usar estruturas mais eficientes (histogram2d / acumuladores).
    coords = list(zip(x, y))
    visits_scatter.set_offsets(coords)

    return walker_point, trail_line, visits_scatter

# ----------------------
# Execução da animação
# ----------------------
ani = animation.FuncAnimation(fig, update, frames=N_steps, init_func=init,
                              blit=True, interval=40, repeat=False)

plt.show()
```

Explicação detalhada do código:

- **Cabeçalho e imports**

```
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Importamos o módulo `random` para gerar passos aleatórios, `matplotlib.pyplot` para plotagem e `matplotlib.animation` para criar a animação em tempo real.

- **Parâmetros da simulação**

```
N_steps = 800
step_size = 1
x = [0]
y = [0]
trail_length = 200
```

`N_steps` controla quantos passos serão gerados (e portanto quantos frames). `step_size` define a distância de cada passo no lattice. `x` e `y` são listas que guardam o histórico das posições; iniciamos na origem. `trail_length` limita quantos pontos do histórico desenharemos no rastro (economiza desenho).

- **Configuração da figura**

```
fig, ax = plt.subplots(figsize=(6,6))
ax.set_xlim(-40, 40)
ax.set_ylim(-40, 40)
ax.set_aspect('equal')

walker_point, = ax.plot([], [], 'ro', markersize=8, label='Walker')
trail_line, = ax.plot([], [], 'b-', alpha=0.4, linewidth=1)
visits_scatter = ax.scatter([], [], s=8, c='black', alpha=0.02)
```

Definimos a janela de visualização, a proporção dos eixos e os objetos gráficos: `walker_point` é o marcador atual; `trail_line` é a linha do rastro; `visits_scatter` é um scatter mostrando todos os pontos visitados (opcional, com baixa opacidade para evidenciar densidade).

- **Função init()**

```
def init():
    walker_point.set_data([], [])
    trail_line.set_data([], [])
    visits_scatter.set_offsets([])
    return walker_point, trail_line, visits_scatter
```

A função de inicialização limpa os artistas animados. É chamada pelo `FuncAnimation` antes da primeira frame. Retorna a tupla dos objetos que serão atualizados.

- **Função update(frame)**

```
def update(frame):
    move = random.choice([(1,0), (-1,0), (0,1), (0,-1)])
    new_x = x[-1] + move[0] * step_size
    new_y = y[-1] + move[1] * step_size

    x.append(new_x)
```

```
        y.append(new_y)
        walker_point.set_data(new_x, new_y)

        xs = x[-trail_length:]
        ys = y[-trail_length:]
        trail_line.set_data(xs, ys)

        coords = list(zip(x, y))
        visits_scatter.set_offsets(coords)

        return walker_point, trail_line, visits_scatter
```

A cada chamada escolhemos aleatoriamente um dos quatro movimentos do lattice. Atualizamos as listas `x` e `y`, redesenhamos o ponto atual e o rastro. O `visits_scatter` é atualizado com todas as posições visitadas — isso fornece uma impressão visual da distribuição de visitas (útil para ver aglomerações). Retornamos os artistas que mudaram para o `blit` funcionar eficientemente.

- **Criação e execução da animação**

```
ani = animation.FuncAnimation(fig, update, frames=N_steps, init_func=init,
                              blit=True, interval=40, repeat=False)

plt.show()
```

`FuncAnimation` conecta a função de atualização à figura; cada frame chama `update`. `interval=40` define 25 fps. O argumento `blit=True` melhora a performance ao redesenhar apenas o necessário. `repeat=False` evita que a animação reinicie automaticamente ao terminar.

Observações e extensões sugeridas

- Para obter resultados reprodutíveis (útil em aulas), defina a semente: `random.seed(12345)` antes do loop.
- Pode-se calcular a *mean squared displacement* (MSD) facilmente:

$$\mathrm{MSD}(n) = \langle x(n)^2 + y(n)^2 \rangle,$$

  armazenando $x[i]^2 + y[i]^2$ a cada passo e fazendo a média por múltiplas trajetórias.
- Em vez de passos de lattice, experimente passos gaussianos: use `dx = np.random.normal(0, sigma)` e `dy = np.random.normal(0, sigma)`.
- Para gravação em arquivo (MP4/GIF) use `matplotlib.animation.FFMpegWriter` ou `PillowWriter` — lembre-se de gerar a animação a partir de uma figura separada se quiser salvar sem a interface interativa.

O exemplo acima é intencionalmente simples, claro e fácil de modificar — ideal para iniciar esta caminhada no aprendizado de animações usando python; sem falar que também é um bom ponto de partida para experimentos computacionais em física estatística e processos estocásticos.

7 Regressão Linear

A regressão linear é uma das ferramentas estatísticas fundamentais para identificar relações entre variáveis. Dado um conjunto de pontos (x_i, y_i), queremos ajustar uma reta que melhor descreva a tendência dos dados:

$$y = a + bx.$$

Os coeficientes são obtidos minimizando a soma dos quadrados dos desvios entre os valores observados e os previstos pela reta (*método dos mínimos quadrados*). As expressões analíticas são:

$$b = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}, \qquad a = \bar{y} - b\,\bar{x}.$$

O exemplo abaixo implementa esse cálculo em Python de forma direta e didática, sem usar bibliotecas externas como `numpy`, para ilustrar claramente cada etapa do processo.

```
# Regressão linear simples em Python (mínimos quadrados)

# Dados de exemplo
x = [1, 2, 3, 4, 5]
y = [2, 4, 5, 4, 5]
n = len(x)

# Cálculo das médias
media_x = sum(x) / n
media_y = sum(y) / n

# Cálculo do coeficiente angular b e do intercepto a
num = 0.0   # numerador da expressão de b
den = 0.0   # denominador da expressão de b
for i in range(n):
    num += (x[i] - media_x) * (y[i] - media_y)
    den += (x[i] - media_x)**2

b = num / den            # inclinação da reta
a = media_y - b*media_x  # intercepto

print(f"Ajuste linear: y = {a} + {b} x")
```

Esse procedimento funciona bem quando os dados têm tendência aproximadamente linear. A inspeção dos resíduos, isto é, das diferenças $y_i - (a + bx_i)$, permite avaliar a qualidade do ajuste.

Ajuste de Lei de Potência

Muitos fenômenos físicos seguem relações do tipo:

$$y = Cx^n,$$

onde C é um coeficiente e n é o expoente da lei de potência. Essa função não é linear em x, mas podemos torná-la linear aplicando logaritmo nos dois lados:

$$\ln y = \ln C + n\,\ln x.$$

Definindo

$$Y = \ln y, \qquad X = \ln x,$$

obtemos uma relação linear

$$Y = A + nX, \qquad A = \ln C.$$

Assim, basta aplicar regressão linear clássica sobre os pares (X_i, Y_i). Depois do ajuste, recuperamos as constantes físicas por:

$$C = e^A.$$

A seguir mostramos uma implementação em Python utilizando esse procedimento com um conjunto de dados exemplo ($y = x^2$):

```
# Regressão linear aplicada a uma lei de potência: y = C * x^n
import math

# Dados exemplo (segue aproximadamente y = x^2)
x = [1, 2, 3, 4, 5, 6]
y = [1, 4, 9, 16, 25, 36]
npts = len(x)

# Transformação logarítmica
X = [math.log(xi) for xi in x]
Y = [math.log(yi) for yi in y]

# Cálculo das médias
media_X = sum(X) / npts
media_Y = sum(Y) / npts

# Ajuste linear para identificar expoente n e coeficiente C
num = 0.0
den = 0.0
for i in range(npts):
    num += (X[i] - media_X) * (Y[i] - media_Y)
    den += (X[i] - media_X)**2

n_exp = num / den        # expoente n
A = media_Y - n_exp*media_X
C = math.exp(A)          # constante C

print(f"Lei de potência ajustada: y = {C} * x^{n_exp}")
```

Esse método é amplamente utilizado em física estatística, sistemas críticos, dinâmica não linear e diversas áreas onde leis de escala e comportamentos em potência emergem naturalmente. A linearização logarítmica permite utilizar ferramentas simples de regressão linear para extrair parâmetros fundamentais do modelo.

8 Interpolação

A interpolação é uma ferramenta essencial em física computacional e análise numérica. Seu objetivo é estimar o valor de uma função em pontos onde ela não foi medida ou calculada, utilizando apenas valores conhecidos em pontos discretos. Métodos simples, como a **interpolação linear**, conectam pontos por linhas retas, enquanto métodos mais sofisticados — como **interpolação polinomial** e **splines cúbicas** — permitem reconstruir curvas suaves e estáveis. A escolha do método depende da natureza dos dados, do grau de suavidade desejado e da complexidade computacional aceitável.

Interpolação Linear

A interpolação linear é o método mais simples e direto. Dados dois pontos (x_0, y_0) e (x_1, y_1), o valor interpolado em um ponto intermediário x é:

$$y(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0).$$

O código abaixo implementa esse cálculo em Python de forma clara e direta:

```
# Interpolação linear em Python

def interp_linear(x, x0, y0, x1, y1):
    """Retorna o valor interpolado linearmente entre dois pontos."""
    return y0 + ( (y1 - y0)/(x1 - x0) ) * (x - x0)

# Exemplo simples
x0, y0 = 1.0, 2.0
x1, y1 = 3.0, 6.0
x  = 2.0  # ponto onde queremos interpolar

y = interp_linear(x, x0, y0, x1, y1)
print(f"Interpolação linear em x={x:.2f} -> y={y:.2f}")
```

Esse método funciona muito bem quando os pontos estão próximos ou quando a função é quase linear entre eles. Em funções curvas, entretanto, pode introduzir erros significativos — sendo então preferível recorrer a métodos polinomiais ou splines.

Interpolação Polinomial (Lagrange)

A interpolação polinomial busca um polinômio único que passe *exatamente* pelos pontos dados. Para três pontos (x_0, y_0), (x_1, y_1) e (x_2, y_2), o polinômio de Lagrange de grau 2 é:

$$P(x) = y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x),$$
$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

Segue uma implementação simples do método em Python:

```
# Interpolação polinomial de Lagrange (3 pontos)

def lagrange3(x, x0, y0, x1, y1, x2, y2):
    """Interpolação de Lagrange para 3 pontos."""
    L0 = ((x - x1)*(x - x2))/((x0 - x1)*(x0 - x2))
    L1 = ((x - x0)*(x - x2))/((x1 - x0)*(x1 - x2))
    L2 = ((x - x0)*(x - x1))/((x2 - x0)*(x2 - x1))
    return y0*L0 + y1*L1 + y2*L2

# Exemplo: y = x^2
x0, y0 = 1, 1
x1, y1 = 2, 4
x2, y2 = 3, 9
x = 2.5

y = lagrange3(x, x0, y0, x1, y1, x2, y2)
print(f"Lagrange (3 pontos) em x={x:.2f} -> y={y:.2f}")
```

O método captura curvaturas e é exato para polinômios de grau até 2 (nesse caso), mas pode gerar oscilações para muitos pontos — o famoso problema de *Runge*. Apesar disso, é extremamente útil para pequenos conjuntos de dados e como construção teórica.

Interpolação por Splines Cúbicas

Splines cúbicas produzem curvas extremamente suaves ao usar polinômios cúbicos em cada intervalo entre pontos, garantindo:

- continuidade da função $S(x)$;
- continuidade da primeira derivada $S'(x)$;
- continuidade da segunda derivada $S''(x)$.

Isso evita oscilações indesejadas e faz das splines a escolha padrão em computação científica. Para pontos $(x_0, y_0), \ldots, (x_n, y_n)$, cada trecho é:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

Resolver os coeficientes envolve um sistema linear tridiagonal, mas podemos ilustrar o conceito com uma *spline aproximada* para três pontos, suficientemente didática:

```
# Uma "mini-spline" cúbica aproximada para 3 pontos usando Lagrange

def spline3(x, x0, y0, x1, y1, x2, y2):
    """
    Aproximação simples de uma spline usando
    o polinômio de Lagrange (3 pontos).
    Útil para fins didáticos.
    """
    L0 = ((x - x1)*(x - x2))/((x0 - x1)*(x0 - x2))
    L1 = ((x - x0)*(x - x2))/((x1 - x0)*(x1 - x2))
    L2 = ((x - x0)*(x - x1))/((x2 - x0)*(x2 - x1))
    return y0*L0 + y1*L1 + y2*L2

# Exemplo com y = x^2
x0,y0 = 0,0
x1,y1 = 1,1
x2,y2 = 2,4
x = 1.5

y = spline3(x, x0, y0, x1, y1, x2, y2)
print(f"Spline cúbica aproximada em x={x:.2f} -> y={y:.2f}")
```

Embora a spline cúbica real envolva mais condições e resulte em uma curva mais suave, este exemplo ilustra a ideia de interpolar com polinômios suaves entre pontos. Para aplicações sérias (dados experimentais, curvas suaves, simulações), recomenda-se usar bibliotecas numéricas como `scipy.interpolate`, que implementam splines cúbicas completas com controle de condições de fronteira.

9 Propagação de Erros e Precisão Numérica

Toda a Física Computacional repousa sobre duas ideias centrais: (i) a tradução de um problema físico contínuo para um conjunto finito de operações aritméticas e (ii) o reconhecimento de que cada uma dessas operações introduz uma **aproximação**. A combinação dessas aproximações — por discretização, arredondamento, truncamento ou limites de representação — determina a confiabilidade final de qualquer resultado numérico. Assim, compreender a origem, a natureza e o acúmulo dos erros é tão essencial quanto dominar os próprios métodos numéricos.

Tipos Fundamentais de Erros

No estudo de algoritmos computacionais aplicados à Física, é tradicional classificarmos os erros em duas categorias principais. Embora ambos afetem o resultado final, eles têm origens conceituais distintas e exigem estratégias diferentes de controle e mitigação. A seguir, apresentamos essas duas classes fundamentais e discutimos como elas surgem no fluxo de um cálculo numérico real.

Erro de Truncamento

O erro de truncamento é aquele que surge quando **substituímos um processo matemático contínuo ou infinito por uma aproximação finita**. Ele é inerente ao próprio método numérico adotado: toda fórmula aproximada, todo esquema de integração, toda expansão finita implica um desvio sistemático em relação ao valor exato.

- **Exemplo:** Utilizar apenas alguns termos de uma série de Taylor para aproximar uma função suave; substituir a derivada exata por uma aproximação de diferenças finitas, como
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$
- **Controle:** O erro de truncamento depende da ordem do método e do tamanho do **passo de discretização** h (ou Δx, Δt). Métodos de ordem mais alta e passos menores tendem a reduzir esse erro, embora à custa de maior custo computacional.

Erro de Arredondamento (Round-off Error)

O erro de arredondamento é consequência da **natureza finita da aritmética em computadores**. Como máquinas digitais armazenam números com um número limitado de bits, apenas um subconjunto discreto dos números reais pode ser representado exatamente. Todos os demais são armazenados como aproximações — e cada operação aritmética pode introduzir um erro adicional.

- **Exemplo:** O número π não pode ser representado exatamente; computadores guardam algo como 3.141592653589793, descartando infinitos dígitos. Operações como subtrações entre números quase iguais (*catastrophic cancellation*) amplificam esse erro.
- **Controle:** Determinado pela **precisão da máquina** (simples, dupla ou estendida) e pela estrutura algorítmica empregada. Reordenar operações, evitar cancelamentos e usar métodos numericamente estáveis são estratégias fundamentais.

O erro total que aparece em um cálculo real é a combinação dessas duas contribuições:

$$E_{\text{Total}} = E_{\text{Truncamento}} + E_{\text{Arredondamento}}.$$

Em algoritmos bem projetados, tende-se a escolher métodos em que o erro de truncamento domina, mantendo o erro de arredondamento pequeno e controlado. Entretanto, em simulações longas, problemas mal condicionados ou operações instáveis, o acúmulo de erros de arredondamento pode tornar-se significativo — às vezes até maior que o próprio erro de truncamento — exigindo especial cuidado na análise numérica.

Representação de Ponto Flutuante e Máquina Epsilon

Para compreender a origem dos erros de arredondamento, é essencial entender como os números reais são armazenados em um computador. A maioria das linguagens modernas — incluindo Python e NumPy — utiliza o padrão **IEEE 754** de *ponto flutuante*, que fornece uma forma padronizada e eficiente de representar números reais com precisão limitada.
Um número em ponto flutuante x é codificado na forma

$$x = \pm(1 + f)\, 2^e,$$

onde:

- $\pm$ é o **bit de sinal**, que indica se o número é positivo ou negativo;
- f é a **mantissa** (ou fração normalizada), responsável pelos dígitos significativos armazenados;
- e é o **expoente**, que controla a escala do número e determina o alcance dos valores representáveis.

O Python (via NumPy) adota, por padrão, a **precisão dupla** (`float64`), que utiliza 64 bits distribuídos em:

- 1 bit para o sinal,
- 11 bits para o expoente,
- 52 bits para a mantissa.

Isso garante aproximadamente **15 a 17 dígitos decimais** de precisão — valor que determina a exatidão máxima de qualquer cálculo numérico feito nesse formato.

O Conceito de Máquina Epsilon (ϵ_m)

O **Máquina Epsilon** (ϵ_m) é um dos parâmetros mais fundamentais da aritmética de ponto flutuante. Ele mede a menor variação relativa detectável pelo sistema: é o menor número positivo tal que

$$1.0 + \epsilon_m > 1.0$$

em aritmética de máquina.
Se um valor δ satisfaz $\delta < \epsilon_m$, então a soma

$$1.0 + \delta = 1.0$$

é indistinguível do próprio 1.0 — o incremento é tão pequeno que se perde na representação.
O ϵ_m controla a precisão relativa e determina o nível máximo de erro de arredondamento que se pode esperar em cada operação elementar. Para números `float64`, temos:

$$\epsilon_m \approx 2.22 \times 10^{-16}.$$

Abaixo segue um pequeno trecho de código que demonstra o cálculo de ϵ_m e sua interpretação prática:

```
import numpy as np

# Máquina Epsilon para precisão dupla (float64)
epsilon = np.finfo(float).eps
print(f"Máquina Epsilon: {epsilon:.2e}")

# Testes clássicos:
if (1.0 + epsilon) > 1.0:
    print("1.0 + epsilon > 1.0 é verdadeiro")
else:
    print("1.0 + epsilon > 1.0 é falso")

if (1.0 + (epsilon / 2)) == 1.0:
    print("1.0 + (epsilon/2) == 1.0 é verdadeiro")
else:
    print("1.0 + (epsilon/2) == 1.0 é falso")
```

Esse pequeno experimento ilustra de maneira concreta uma das limitações fundamentais da aritmética de ponto flutuante: números muito pequenos simplesmente não são resolvidos pela máquina. O fato de que $(1.0 + \epsilon_m)$ é distinguível de 1.0, enquanto $(1.0 + \epsilon_m/2)$ não é, evidencia o *tamanho mínimo do degrau* entre números representáveis na vizinhança de 1.0. Esse degrau não é constante em todo o eixo real — ele cresce proporcionalmente ao próprio valor de x, pois a distribuição dos números representáveis é **logaritmicamente espaçada**. Assim, compreender o papel de ϵ_m e seus efeitos práticos é essencial para interpretar corretamente erros de arredondamento, avaliar a estabilidade numérica de algoritmos e antecipar limitações inevitáveis em cálculos científicos.

O Erro de Truncamento: O Papel da Série de Taylor

Enquanto o erro de arredondamento é um efeito imposto pela arquitetura da máquina, o **erro de truncamento** decorre exclusivamente das aproximações matemáticas que fazemos ao substituir um processo contínuo (derivadas, integrais, equações diferenciais) por um procedimento algébrico discreto. A ferramenta central para analisar e quantificar esse erro é a **Série de Taylor**, que fornece uma expansão sistemática de uma função suave em torno de um ponto. Para uma função suficientemente diferenciável, a expansão de Taylor em torno de a é:

$$f(x) = f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!}f''(a) + \cdots + \frac{(x-a)^n}{n!}f^{(n)}(a) + R_n,$$

onde R_n é o **termo resto**, que contém exatamente o erro introduzido ao truncarmos a série após o termo de ordem n. Em geral, esse resto é dominado pelo primeiro termo desprezado, isto é,

$$R_n = \mathcal{O}((x-a)^{n+1}).$$

Exemplo: Erro na Derivada Numérica

Um dos exemplos mais ilustrativos ocorre no cálculo numérico de derivadas. Considere a aproximação de **diferença finita progressiva**:

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

Expandindo $f(x+h)$ em série de Taylor:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \mathcal{O}(h^3).$$

Substituindo essa expansão na aproximação numérica e reorganizando:

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(x) + \mathcal{O}(h^2).$$

O termo adicional

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

é precisamente o **erro de truncamento** dominante da fórmula de diferença progressiva. Como ele é proporcional a h, dizemos que o método é de **primeira ordem**:

$$\text{Erro de Truncamento} \propto \mathcal{O}(h).$$

Isso tem uma interpretação prática simples: se reduzirmos o passo de discretização h em um fator 10, o erro de truncamento também é reduzido por um fator 10. Essa relação direta entre o passo e a precisão é essencial para o controle numérico — passos menores aumentam a exatidão, mas também o custo computacional, exigindo sempre um equilíbrio entre eficiência e precisão.

A Propagação de Erros Inerentes

Quando realizamos um cálculo numérico ou uma medida experimental, as quantidades manipuladas não são conhecidas com exatidão perfeita: cada variável de entrada possui um erro associado. A **propagação de erros** estuda precisamente como essas incertezas são transmitidas — e possivelmente amplificadas — pelo cálculo, afetando a precisão do resultado final. Considere uma grandeza calculada a partir de outras variáveis,

$$y = f(x_1, x_2, \ldots, x_n),$$

onde cada entrada x_i carrega uma incerteza absoluta Δx_i. Nosso objetivo é quantificar a incerteza resultante Δy.

O Caso Geral: A Diferencial Total

Se as perturbações Δx_i forem suficientemente pequenas, podemos aproximar a variação em y pela diferencial total:

$$\Delta y \approx \frac{\partial f}{\partial x_1}\Delta x_1 + \frac{\partial f}{\partial x_2}\Delta x_2 + \cdots + \frac{\partial f}{\partial x_n}\Delta x_n.$$

Essa expressão mostra algo profundo: **a sensibilidade de y ao erro de cada variável é controlada pela derivada parcial correspondente**. Variáveis de grande impacto (derivadas grandes) amplificam seus erros; já variáveis pouco influentes transmitem apenas uma fração deles. Para estimativas conservadoras — quando queremos o *erro máximo possível* — tomamos valores absolutos:

$$|\Delta y| \approx \left|\frac{\partial f}{\partial x_1}\right| |\Delta x_1| + \left|\frac{\partial f}{\partial x_2}\right| |\Delta x_2| + \cdots + \left|\frac{\partial f}{\partial x_n}\right| |\Delta x_n|.$$

Essa versão é frequentemente usada em engenharia, onde se deseja garantir limites superiores de segurança.

Propagação de Incertezas Aleatórias: Soma de Variâncias

Em Física e Estatística, os erros são na maior parte das vezes tratados como grandezas **aleatórias**, não determinísticas. Supondo que:

- os erros das variáveis x_i são independentes;
- cada erro é descrito por seu desvio padrão σ_{x_i};
- o comportamento da função pode ser linearizado localmente (pequenas variações),

então a incerteza em y é dada pela soma ponderada das **variâncias**:

$$\sigma_y^2 \approx \left(\frac{\partial f}{\partial x_1}\right)^2 \sigma_{x_1}^2 + \left(\frac{\partial f}{\partial x_2}\right)^2 \sigma_{x_2}^2 + \cdots + \left(\frac{\partial f}{\partial x_n}\right)^2 \sigma_{x_n}^2.$$

Essa fórmula é extraordinariamente importante: ela mostra que cada incerteza contribui para σ_y de forma **quadrática**, sendo modulada pela sensibilidade da função. Erros de entrada podem ser amplificados, suprimidos ou até dominados por um único termo — um fenômeno chamado *erro dominante*. Em resumo, a propagação de erros fornece um mapa claro de como a estrutura matemática de uma função governa a precisão final: toda incerteza tem um caminho bem definido até o resultado, e compreender esse caminho é essencial para qualquer análise numérica confiável.

Regras Práticas para Propagação de Incertezas

Ao manipular variáveis medidas com incerteza, algumas regras simples — porém fundamentais — ajudam a estimar corretamente o erro propagado.

- **Adição e Subtração** ($y = x_1 \pm x_2$):
  Os erros *absolutos* se combinam quadraticamente:
  $$\sigma_y^2 = \sigma_{x_1}^2 + \sigma_{x_2}^2.$$
- **Multiplicação e Divisão** ($y = x_1 x_2$ ou $y = x_1/x_2$):
  Os erros *relativos* se combinam quadraticamente:
  $$\left(\frac{\sigma_y}{y}\right)^2 = \left(\frac{\sigma_{x_1}}{x_1}\right)^2 + \left(\frac{\sigma_{x_2}}{x_2}\right)^2.$$

O Perigo do Cancelamento Subtrativo

O **cancelamento subtrativo** é uma das armadilhas mais graves da Física Computacional. Ele surge quando dois números quase iguais são subtraídos, provocando a perda de dígitos significativos e a *explosão do erro relativo*.

Como Ocorre a Catástrofe Numérica

Considere dois números armazenados em precisão dupla,

$$a \approx 123.456789, \qquad b \approx 123.456788.$$

Embora ambos tenham sido representados com um erro de arredondamento típico de $\mathcal{O}(10^{-16})$, sua diferença é da ordem de 10^{-6}. Assim:

1. **Arredondamento Inicial:** a e b já possuem incertezas mínimas devido à representação binária.

2. **Subtração:** Ao computar $y = a - b$, obtemos um valor muito pequeno.

3. **Amplificação do Erro Relativo:** O erro absoluto permanece praticamente o mesmo, mas o denominador diminui drasticamente — o erro relativo explode.

Formalmente,

$$y = a - b, \qquad \sigma_y^2 = \sigma_a^2 + \sigma_b^2.$$

Se $\sigma_a \approx \sigma_b$, então $\sigma_y \approx \sqrt{2}\,\sigma_a$, mas

$$\frac{\sigma_y}{|y|} \gg 1,$$

o que torna y essencialmente inútil do ponto de vista numérico.

Exemplo Clássico: Fórmula de Bhaskara

A equação quadrática

$$ax^2 + bx + c = 0$$

leva às soluções:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Quando $|b| \gg |4ac|$, o termo

$$-b + \sqrt{b^2 - 4ac}$$

envolve a subtração de dois números quase iguais, tornando uma das raízes numericamente instável.

Remédio Numérico: A abordagem estável evita subtrações perigosas:

1. Calcule a raiz **estável**:

$$x_{\text{estável}} = \frac{-b - \operatorname{sgn}(b)\sqrt{b^2 - 4ac}}{2a}.$$

   O uso de $\operatorname{sgn}(b)$ garante que a soma dos termos tenha magnitude maior, evitando cancelamento.

2. Recupere a raiz **instável** via a relação exata:

$$x_{\text{instável}} = \frac{c}{a\, x_{\text{estável}}}.$$

Esse procedimento produz valores numericamente robustos, mesmo quando a fórmula tradicional falha.

Condicionamento e Estabilidade Numérica

A qualidade de uma computação não depende apenas da precisão do algoritmo, mas também da *sensibilidade* do problema aos erros de entrada.

Condicionamento de um Problema

Um problema é **bem-condicionado** se pequenos erros na entrada produzem pequenos erros na saída; caso contrário, é **mal-condicionado**. Essa sensibilidade é quantificada pelo **número de condicionamento** κ:

$$\frac{\text{Erro Relativo de Saída}}{\text{Erro Relativo de Entrada}} \approx \kappa.$$

Valores $\kappa \gg 1$ indicam forte amplificação dos erros.

Estabilidade de um Algoritmo

Um algoritmo é **numericamente estável** quando os erros inerentes (principalmente os de arredondamento) não são amplificados durante o processo computacional.

- **Algoritmos Estáveis:** o erro final é comparável ao erro intrínseco do próprio problema.
- **Algoritmos Instáveis:** amplificam consideravelmente erros intermediários; o cancelamento subtrativo é um exemplo típico de instabilidade.

Ao desenvolver código em Física Computacional, o ideal é empregar um **algoritmo estável** para resolver um **problema bem-condicionado**, minimizando assim a propagação e amplificação de erros numéricos.
Em última análise, compreender a interação entre condicionamento, estabilidade numérica e cancelamento subtrativo não é apenas um detalhe técnico, mas uma parte essencial da formação de qualquer físico computacional. A precisão de uma simulação raramente é determinada apenas pela potência do método utilizado ou pela resolução numérica empregada, mas sim pela atenção cuidadosa aos caminhos pelos quais o erro se propaga, se amplifica ou é atenuado ao longo do cálculo. Muitos dos fenômenos que estudamos — desde integrais oscilatórias até dinâmicas caóticas e problemas fortemente estocásticos — são inerentemente sensíveis a pequenas perturbações, o que torna ainda mais crucial reconhecer quando um resultado numérico é confiável ou quando pode ser apenas um artefato da máquina. O domínio dessas ferramentas conceituais nos permite projetar algoritmos robustos, escolher representações matemáticas adequadas, evitar armadilhas clássicas como o cancelamento catastrófico e, sobretudo, interpretar nossos resultados com maturidade científica. Em última instância, a computação numérica não é apenas uma extensão da matemática teórica, mas um campo em que decisões aparentemente pequenas — como reescrever uma expressão, trocar uma variável, ou alterar a ordem de operações — podem determinar a diferença entre um cálculo instável e uma simulação capaz de revelar estruturas profundas da física. Assim, cultivar sensibilidade aos aspectos numéricos é, ao mesmo tempo, um exercício de rigor e uma forma de respeito ao próprio objeto estudado: a natureza expressa por meio de números, mas que exige cuidado para ser traduzida de forma fiel dentro das limitações inevitáveis do computador.

Parte 2

Estas próximas notas de aula introduzem os temas que serão desenvolvidos na **Parte 2** do curso de Física Computacional. O foco recai sobre algumas das principais técnicas numéricas amplamente utilizadas em física, apresentadas de forma conceitual e acompanhadas de suas formulações discretas, exemplos aplicados e implementações em **Python**. Iniciaremos com métodos para a solução de **equações diferenciais ordinárias** (EDOs), tomando como exemplo central o sistema massa–mola sem atrito. Esse modelo simples e clássico servirá como fio condutor para discutir aspectos fundamentais de algoritmos numéricos, incluindo *estabilidade*, *conservação de energia*, *precisão* e o comportamento de diferentes esquemas de integração (como Euler, Euler-Cromer e Verlet). Em seguida, estudaremos métodos de **diagonalização numérica de matrizes**, essenciais para vários problemas em física computacional, como análise de modos normais, solução de Hamiltonianos discretizados, dinâmica quântica e transformações lineares gerais. Apresentaremos abordagens diretas e iterativas, explorando suas vantagens, limitações e aplicações. O objetivo geral desta parte é fornecer um material claro, progressivo e útil para estudantes, enfatizando não apenas a implementação prática dos algoritmos em Python, mas também suas interpretações físicas e o raciocínio que fundamenta cada método. Assim, espera-se que o aluno desenvolva não apenas habilidade técnica, mas também uma compreensão crítica sobre como e por que os métodos funcionam.

10 Método de Euler (Explícito)

O método de Euler *explícito* é a abordagem mais simples e direta para resolver equações diferenciais ordinárias numericamente. Sua ideia central consiste em aproximar a derivada por uma diferença finita, assumindo que a inclinação da solução é aproximadamente constante dentro de um intervalo de tempo pequeno Δt. Assim, dado um estado conhecido no instante t_n, estimamos o próximo estado em $t_{n+1} = t_n + \Delta t$ pela fórmula

$$y_{n+1} = y_n + \Delta t \cdot f(y_n, t_n), \tag{1}$$

onde $f(y,t)$ representa a derivada $\frac{dy}{dt}$. Por ser um método de primeira ordem, apresenta erro local de $\mathcal{O}(\Delta t^2)$ e erro global acumulado de $\mathcal{O}(\Delta t)$. Apesar de sua simplicidade e facilidade de implementação, o método pode se tornar instável quando aplicado a integrações longas, passos de tempo grandes ou sistemas fisicamente rígidos.

Equações para o Sistema Massa–Mola

Considere o sistema massa–mola ideal (sem atrito). A dinâmica é descrita pela equação diferencial de segunda ordem:

$$\frac{d^2x}{dt^2} = -\omega^2 x,$$

que pode ser reescrita como um sistema equivalente de duas equações de primeira ordem:

$$\begin{cases} \frac{dx}{dt} = v, \\ \frac{dv}{dt} = -\omega^2 x. \end{cases}$$

Essa reescrita é fundamental, pois a maioria dos métodos numéricos para EDOs (como Euler, Euler-Cromer e Runge–Kutta) foi desenvolvida para sistemas de primeira ordem. Além disso, tal abordagem permite interpretar o sistema como um ponto movendo-se no espaço de fases (x, v), facilitando análises de estabilidade e energia.

Aplicando o método de Euler explícito obtemos as atualizações:

$$\begin{aligned} x_{n+1} &= x_n + \Delta t\, v_n, \\ v_{n+1} &= v_n - \Delta t\, \omega^2 x_n. \end{aligned}$$

Essas equações são simples e rápidas de implementar, mas produzem um crescimento artificial de energia ao longo do tempo — um comportamento não físico que ilustra as limitações do método de Euler.

Código em Python: Massa–Mola com Euler Explícito

O código abaixo implementa o método de Euler explícito para o sistema massa–mola, salvando os dados de posição e velocidade ao longo do tempo em um arquivo. Incluímos comentários detalhados para facilitar o entendimento linha a linha.

```
# Simulação massa-mola com método de Euler explícito

import math
```

```
# Parâmetros físicos e numéricos
omega = 1.0      # frequência natural
dt = 0.01        # passo de tempo
N = 1000         # número total de passos

# Condições iniciais
x = 1.0          # posição inicial
v = 0.0          # velocidade inicial

# Abre arquivo de saída
with open("massa_mola_euler.dat", "w") as f:

    for n in range(N):
        t = n * dt

        # Grava tempo, posição e velocidade
        f.write(f"{t} {x} {v}\n")

        # Atualização via Euler explícito
        x_novo = x + dt * v
        v_novo = v - dt * omega**2 * x

        # Atualiza variáveis
        x = x_novo
        v = v_novo
```

A execução deste programa gera o arquivo `massa_mola_euler.dat`, que pode ser utilizado para analisar a trajetória no espaço de fases ou o comportamento temporal da posição e da velocidade. Em particular, ao plotar $x(t)$ ou $v(t)$, observa-se que a amplitude cresce gradualmente — um sintoma clássico da instabilidade do método de Euler para sistemas oscilatórios. Esse exemplo é, portanto, ideal para introduzir comparações com métodos mais estáveis, como Euler-Cromer ou Verlet, estudados nas seções seguintes.

11 Diferença Finita Centrada no Tempo

Um método simples, estável e eficiente para resolver EDOs de segunda ordem, como o sistema massa–mola, é o método das **diferenças finitas centradas no tempo**. Esse esquema utiliza três pontos consecutivos da malha temporal para aproximar a derivada de segunda ordem, permitindo uma integração explícita com boa conservação de energia.

Equação Geral

Para uma equação diferencial do tipo:

$$\frac{d^2x}{dt^2} = f(x,t),$$

a derivada de segunda ordem pode ser aproximada por diferenças finitas centradas:

$$\frac{x_{n+1} - 2x_n + x_{n-1}}{\Delta t^2} \approx \frac{d^2x}{dt^2}.$$

Aplicando ao sistema massa–mola sem atrito, para o qual $f(x_n,t) = -\omega^2 x_n$:

$$\frac{x_{n+1} - 2x_n + x_{n-1}}{\Delta t^2} = -\omega^2 x_n.$$

Isolando x_{n+1}, obtemos a fórmula de recorrência explícita:

$$x_{n+1} = 2x_n - x_{n-1} - \Delta t^2\, \omega^2 x_n. \tag{2}$$

Esse método é de segunda ordem no tempo, apresentando erro global de $\mathcal{O}(\Delta t^2)$, superior ao método de Euler explícito, que é apenas de primeira ordem.

Inicialização

Como a fórmula requer conhecer x_{n-1}, precisamos de dois valores iniciais:

- $x_0 = x(t = 0)$,
- x_1, obtido via expansão de Taylor:

$$x_1 = x_0 + \Delta t\, v_0 - \frac{1}{2}\Delta t^2\, \omega^2 x_0.$$

Desse modo, podemos iniciar a evolução com precisão consistente com o método de segunda ordem.

Vantagens do Método

O esquema centrado é conhecido por:

- apresentar **boa estabilidade** em sistemas oscilatórios,
- preservar a energia total de forma mais realista do que o Euler explícito,
- gerar uma trajetória temporal suave e sem crescimento artificial de energia,
- ser simples de implementar e computacionalmente barato.

A velocidade não é calculada diretamente, mas pode ser estimada de forma consistente por:

$$v_n \approx \frac{x_{n+1} - x_{n-1}}{2\Delta t},$$

o que é suficiente para análises energéticas ou reconstrução do movimento.

Código em Python: Massa–Mola com Diferença Finita

A seguir, apresentamos uma implementação completamente funcional e comentada em Python puro, utilizando apenas listas e operações básicas. O objetivo é manter o código o mais pedagógico possível.

```
import math

# ----------------------------
# Parâmetros da simulação
# ----------------------------
N = 1000          # número de passos de tempo
dt = 0.01         # passo de tempo
omega = 1.0       # frequência natural do oscilador

# ----------------------------
```

```
# Condições iniciais
# ----------------------------
x0 = 1.0          # posição inicial
v0 = 0.0          # velocidade inicial

# Lista de posições ao longo do tempo
x = [0.0] * N

# Condição inicial
x[0] = x0

# Cálculo de x[1] usando expansão de Taylor
x[1] = x0 + dt * v0 - 0.5 * dt * dt * omega**2 * x0

# ----------------------------
# Evolução temporal via diferenças finitas centradas
# ----------------------------
for n in range(1, N-1):
    x[n+1] = 2*x[n] - x[n-1] - (dt*dt) * omega**2 * x[n]

# ----------------------------
# Impressão dos resultados
# Cada linha: tempo    posição
# ----------------------------
for n in range(N):
    t = n * dt
    print(f"{t:.5f} {x[n]:.7f}")
```

Esse programa imprime na tela duas colunas: o tempo t e a posição $x(t)$. Caso desejado, as saídas podem ser redirecionadas para um arquivo com:

```
python simulacao.py > massa_mola_dif_finitas.dat
```

Com isso, o método de diferenças finitas centradas apresenta-se como uma ferramenta poderosa, simples e precisa para o estudo de sistemas oscilatórios, sendo amplamente utilizado em física computacional e métodos numéricos.

12 Método de Runge–Kutta de 4ª Ordem (RK4)

O método de Runge–Kutta de 4ª ordem (RK4) é um dos métodos explícitos mais precisos e amplamente utilizados para resolver equações diferenciais ordinárias (EDOs). Em cada passo temporal, ele calcula quatro estimativas da derivada, ponderando-as de forma a obter uma aproximação com erro global de ordem $\mathcal{O}(\Delta t^4)$:

$$
\begin{aligned}
k_1 &= f(y_n, t_n), \\
k_2 &= f\left(y_n + \frac{\Delta t}{2} k_1,\ t_n + \frac{\Delta t}{2}\right), \\
k_3 &= f\left(y_n + \frac{\Delta t}{2} k_2,\ t_n + \frac{\Delta t}{2}\right), \\
k_4 &= f\left(y_n + \Delta t\, k_3,\ t_n + \Delta t\right), \\
y_{n+1} &= y_n + \frac{\Delta t}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right).
\end{aligned}
$$

Aplicação ao sistema massa–mola

O sistema massa–mola sem atrito pode ser expresso como um sistema de duas EDOs de primeira ordem:

$$\frac{dx}{dt} = v, \qquad \frac{dv}{dt} = -\omega^2 x.$$

Representamos o vetor de variáveis como:

$$\mathbf{y}(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}, \qquad \frac{d\mathbf{y}}{dt} = \begin{pmatrix} v \\ -\omega^2 x \end{pmatrix} \equiv \mathbf{f}(\mathbf{y}, t).$$

Assim, a atualização de RK4 toma a forma:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{\Delta t}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right),$$

onde cada vetor $\mathbf{k}_i$ possui componentes de posição e velocidade:

$$\mathbf{k}_i = \begin{pmatrix} k_{i,x} \\ k_{i,v} \end{pmatrix}.$$

A atualização final para cada variável é:

$$x_{n+1} = x_n + \frac{\Delta t}{6}(k_{1,x} + 2k_{2,x} + 2k_{3,x} + k_{4,x}),$$

$$v_{n+1} = v_n + \frac{\Delta t}{6}(k_{1,v} + 2k_{2,v} + 2k_{3,v} + k_{4,v}).$$

O RK4 fornece excelente precisão e é amplamente usado em sistemas oscilatórios por manter bem a forma da solução e apresentar baixa dissipação numérica.

Código em Python: Massa–Mola com RK4

Abaixo apresentamos uma implementação simples, clara e comentada do método RK4 em Python puro. O código não utiliza bibliotecas externas, visando fins pedagógicos.

```
import math

# ------------------------------------
# Parâmetros da simulação
# ------------------------------------
N = 1000        # número total de passos
dt = 0.01       # passo de tempo
omega = 1.0     # frequência natural da mola

# ------------------------------------
# Condições iniciais
# ------------------------------------
x = 1.0         # posição inicial
v = 0.0         # velocidade inicial

# ------------------------------------
# Função que define o sistema:
# retorna dx/dt e dv/dt
# ------------------------------------
def f(x, v):
    dxdt = v
```

```
    dvdt = -omega**2 * x
    return dxdt, dvdt

# -----------------------------------
# Integração usando RK4
# Cada linha impressa: t, x(t), v(t)
# -----------------------------------
for n in range(N):
    t = n * dt

    # Impressão dos valores atuais
    print(f"{t:.5f} {x:.7f} {v:.7f}")

    # Cálculo dos coeficientes do método RK4
    k1x, k1v = f(x, v)
    k2x, k2v = f(x + 0.5*dt*k1x, v + 0.5*dt*k1v)
    k3x, k3v = f(x + 0.5*dt*k2x, v + 0.5*dt*k2v)
    k4x, k4v = f(x + dt*k3x,    v + dt*k3v)

    # Atualização final
    x += (dt/6) * (k1x + 2*k2x + 2*k3x + k4x)
    v += (dt/6) * (k1v + 2*k2v + 2*k3v + k4v)
```

Esse programa imprime o tempo, a posição e a velocidade a cada passo. Para salvar a saída em arquivo, basta utilizar:

```
python3 rk4_massa_mola.py > massa_mola_rk4.dat
```

Com o método RK4, a trajetória do oscilador harmônico é reproduzida com grande precisão e excelente estabilidade, sendo ideal para análises numéricas onde a conservação da energia desempenha um papel fundamental.

13 Método de Adams–Bashforth de 2ª Ordem

O método de Adams–Bashforth é um esquema explícito **multi-passo**, ou seja, utiliza valores anteriores da função derivada $f(t, y)$ para prever o próximo valor da solução. A forma geral vem da aproximação da integral da EDO:

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t)\, dt,$$

substituindo a função $f(t)$ por um polinômio interpolador de Lagrange de primeiro grau construído com os pontos t_{n-1} e t_n. A integração desse polinômio leva a aproximação:

$$\int_{t_n}^{t_{n+1}} f(t)\, dt \approx \frac{\Delta t}{2}\,(3f_n - f_{n-1}),$$

que resulta na fórmula explícita:

$$y_{n+1} = y_n + \frac{\Delta t}{2}(3f_n - f_{n-1}).$$

Aplicação ao sistema massa–mola

Para o oscilador harmônico simples:

$$\frac{dx}{dt} = v, \qquad \frac{dv}{dt} = -\omega^2 x,$$

podemos aplicar a fórmula de Adams–Bashforth separadamente a cada equação. Assim, para dois passos consecutivos:

$$x_{n+1} = x_n + \frac{\Delta t}{2}\,(3v_n - v_{n-1}),$$
$$v_{n+1} = v_n + \frac{\Delta t}{2}\,\big(-3\omega^2 x_n + \omega^2 x_{n-1}\big).$$

Inicialização

O método precisa de dois valores de x e v: x_0, x_1 e v_0, v_1. Eles podem ser obtidos com um passo do método de Euler:

$$x_1 = x_0 + \Delta t\, v_0, \qquad v_1 = v_0 - \Delta t\, \omega^2 x_0.$$

Código em Python: Adams–Bashforth (2ª ordem)

Abaixo implementamos o método em Python, de modo didático e totalmente comentado:

```
import math

# ----------------------------
# Parâmetros do problema
# ----------------------------
N = 1000
dt = 0.01
omega = 1.0

# Arrays para armazenar x e v
x = [0.0] * N
v = [0.0] * N

# ----------------------------
# Condições iniciais
# ----------------------------
x[0] = 1.0   # posição inicial
v[0] = 0.0   # velocidade inicial

# Passo de inicialização com Euler
x[1] = x[0] + dt * v[0]
v[1] = v[0] - dt * omega**2 * x[0]

# ----------------------------
# Integração Adams-Bashforth 2ª ordem
# ----------------------------
for n in range(1, N-1):
    # Registra valores no console (ou redirecione para arquivo)
    t = n * dt
    print(f"{t:.5f} {x[n]:.7f} {v[n]:.7f}")

    # Fórmulas do método
```

```
x[n+1] = x[n] + 0.5 * dt * (3*v[n] - v[n-1])
v[n+1] = v[n] + 0.5 * dt * (-3*omega**2*x[n] + omega**2*x[n-1])
```

Esse método é mais preciso do que Euler explícito e frequentemente mais rápido que RK4 quando se deseja reutilizar derivadas já calculadas. No entanto, sua estabilidade depende da escolha do passo Δt, especialmente em sistemas oscilatórios.

14 Método de Taylor de 2ª Ordem

O método de Taylor consiste em expandir a solução em série até uma ordem desejada. Para ordem 2:

$$y_{n+1} = y_n + \Delta t\, y'_n + \frac{\Delta t^2}{2}\, y''_n,$$

onde é necessário conhecer não apenas a derivada de primeira ordem, mas também a de segunda ordem.

Aplicação ao sistema massa–mola

$$\frac{dx}{dt} = v, \qquad \frac{dv}{dt} = -\omega^2 x.$$

As derivadas de segunda ordem são:

$$\frac{d^2x}{dt^2} = -\omega^2 x, \qquad \frac{d^2v}{dt^2} = -\omega^2 v.$$

Substituindo na expansão de Taylor:

$$x_{n+1} = x_n + \Delta t\, v_n - \frac{\Delta t^2}{2}\, \omega^2 x_n,$$
$$v_{n+1} = v_n - \Delta t\, \omega^2 x_n - \frac{\Delta t^2}{2}\, \omega^2 v_n.$$

Código em Python: Taylor de 2^a ordem

Coloco agora uma Implementação clara e totalmente comentada do método de Taylor de 2^a ordem:

```
import math

# ----------------------------
# Parâmetros
# ----------------------------
N = 1000
dt = 0.01
omega = 1.0

# Condições iniciais
x = 1.0
v = 0.0

# ----------------------------
```

```
# Método de Taylor de 2a ordem
# ----------------------------
for n in range(N):
    t = n * dt
    print(f"{t:.5f} {x:.7f} {v:.7f}")

    # Aplicação direta das fórmulas de Taylor
    x_new = x + dt * v - 0.5 * dt**2 * omega**2 * x
    v_new = v - dt * omega**2 * x - 0.5 * dt**2 * omega**2 * v

    # Atualiza para o próximo passo
    x, v = x_new, v_new
```

O método de Taylor de 2ª ordem é extremamente simples e fornece boa precisão para sistemas cujas derivadas são fáceis de computar analiticamente — como o oscilador harmônico. Entretanto, para sistemas mais complexos, calcular derivadas de ordem superior pode ser difícil, e por isso métodos como RK4 se tornam mais práticos.

15 Método de Verlet com Velocidade (Velocity-Verlet)

O método *Velocity-Verlet* é amplamente utilizado em dinâmica molecular, simulação de sistemas mecânicos e integrações hamiltonianas devido à sua excelente conservação de energia e boa estabilidade. Ele é um integrador de segunda ordem que calcula novas posições e velocidades utilizando informações de aceleração no passo atual e no passo seguinte. Para o sistema massa–mola sem atrito, cuja aceleração é

$$a(t) = -\omega^2 x(t),$$

o método atualiza posição, aceleração e velocidade da seguinte forma:

$$\begin{aligned} x_{n+1} &= x_n + v_n\,\Delta t + \frac{1}{2} a_n\,\Delta t^2 \\ a_{n+1} &= -\omega^2 x_{n+1} \\ v_{n+1} &= v_n + \frac{1}{2}(a_n + a_{n+1})\,\Delta t \end{aligned}$$

A interpretação é simples: primeiro atualizamos a posição assumindo aceleração constante no intervalo; depois usamos a nova posição para calcular uma aceleração mais precisa; por fim, atualizamos a velocidade com a média das acelerações inicial e final. Esse esquema reduz erros globais e conserva energia muito melhor que o método de Euler.

Código em Python: Sistema Massa-Mola com Velocity-Verlet

O código abaixo implementa o método de forma simples, com variáveis escalares e muitos comentários explicativos:

```
import numpy as np

# Parâmetros do problema
N = 1000              # número de passos
dt = 0.01             # passo de tempo
omega = 1.0           # frequência natural
```

```
# Condições iniciais
x = 1.0              # posição inicial
v = 0.0              # velocidade inicial
a = -omega**2 * x    # aceleração inicial

# Listas para armazenar os resultados
ts, xs, vs = [], [], []

for i in range(N):
    t = i * dt
    ts.append(t)
    xs.append(x)
    vs.append(v)

    # Atualização da posição (usando aceleração atual)
    x_new = x + v * dt + 0.5 * a * dt**2

    # Nova aceleração baseada na nova posição
    a_new = -omega**2 * x_new

    # Atualização da velocidade (média das acelerações)
    v_new = v + 0.5 * (a + a_new) * dt

    # Prepara para o próximo passo
    x, v, a = x_new, v_new, a_new
```

Os vetores `ts`, `xs` e `vs` podem ser salvos ou usados diretamente para gráficos. O método Velocity-Verlet é um dos mais recomendados quando se deseja simulação precisa por longos intervalos de tempo.

16 Método Leap-Frog

O método *Leap-Frog* é outro integrador explícito de segunda ordem muito usado em física computacional. Seu nome vem do fato de que velocidade e posição "saltam uma sobre a outra" em passos alternados: as velocidades são calculadas em tempos semi-inteiros $t_{n+1/2}$, enquanto as posições são calculadas em tempos inteiros t_n. Esse desfasamento produz um integrador simpético que conserva energia muito bem.

$$a(t) = -\omega^2 x(t)$$

O algoritmo é:

$$\begin{aligned} v_{n+\frac{1}{2}} &= v_n + \frac{1}{2}\Delta t\, a_n \\ x_{n+1} &= x_n + \Delta t\, v_{n+\frac{1}{2}} \\ a_{n+1} &= -\omega^2 x_{n+1} \\ v_{n+1} &= v_{n+\frac{1}{2}} + \frac{1}{2}\Delta t\, a_{n+1} \end{aligned}$$

A vantagem principal é que o método Leap-Frog preserva quase perfeitamente a energia em sistemas oscilatórios, superando métodos como Euler ou mesmo Runge-Kutta em simulações muito longas.

Código em Python: Sistema Massa-Mola com Leap-Frog

A implementação em Python é direta. Como o método mantém uma velocidade em meio passo, usamos uma variável auxiliar `v_half`:

```
import numpy as np

# Parâmetros
N = 1000
dt = 0.01
omega = 1.0

# Condições iniciais
x = 1.0
v = 0.0
a = -omega**2 * x

# Primeiro meio passo da velocidade
v_half = v + 0.5 * dt * a

# Armazenamento
ts, xs, vs = [], [], []

for i in range(N):
    t = i * dt

    # A velocidade física no instante t é v_half - (dt/2)*a
    v_real = v_half - 0.5 * dt * a

    ts.append(t)
    xs.append(x)
    vs.append(v_real)

    # Atualiza posição usando a velocidade em meio passo
    x = x + dt * v_half

    # Nova aceleração
    a = -omega**2 * x

    # Atualiza velocidade para o próximo meio passo
    v_half = v_half + 0.5 * dt * a
```

Esse método é simples, rápido e muito estável, sendo um dos mais usados em simulações de sistemas hamiltonianos, como partículas em potenciais, oscilações acopladas e dinâmica molecular. Em resumo, tanto o método *Velocity-Verlet* quanto o *Leap-Frog* são integradores de segunda ordem que conservam energia com excelente precisão, sendo ideais para o estudo do oscilador harmônico. A escolha entre eles depende principalmente da forma como

se deseja armazenar ou interpretar as velocidades: em passos completos (Verlet) ou em meio passo (Leap-Frog).

17 Resolução de Equações Diferenciais Estocásticas: Método de Euler-Maruyama

Equações diferenciais estocásticas (EDEs) modelam sistemas onde há influência de flutuações aleatórias, como ruído térmico, interações moleculares imprevisíveis ou efeitos quânticos. Uma EDE geral pode ser escrita como

$$\frac{dy}{dt} = f(y,t) + g(y,t)\,\xi(t),$$

onde $f(y,t)$ representa a parte determinística, $g(y,t)$ controla a intensidade do ruído, e $\xi(t)$ é um ruído branco gaussiano, formalmente interpretado como a derivada do Movimento Browniano. Como esse termo aleatório impede a existência de soluções analíticas em grande parte dos casos, empregamos métodos numéricos apropriados.

Método de Euler-Maruyama

O método **Euler-Maruyama** é a versão estocástica do método de Euler explícito. Ele aproxima a solução discreta por

$$y_{n+1} = y_n + f(y_n,t_n)\,\Delta t + g(y_n,t_n)\,\Delta W_n,$$

onde o incremento estocástico ΔW_n é simulado como

$$\Delta W_n = \sqrt{\Delta t}\,\eta_n, \quad \text{com } \eta_n \sim \mathcal{N}(0,1).$$

Esse termo imita o comportamento do Movimento Browniano, cuja variância cresce linearmente com o tempo.

Aplicação: Equação de Langevin com Ruído Térmico

A equação de Langevin descreve uma partícula sujeita a atrito viscoso e flutuações térmicas, sendo amplamente utilizada em física estatística e simulações de dinâmica molecular:

$$\begin{cases} \dfrac{dx}{dt} = v(t), \\ \dfrac{dv}{dt} = -\dfrac{\gamma}{m}v(t) + \dfrac{1}{m}\,\xi(t). \end{cases}$$

O ruído térmico é relacionado à temperatura do meio por meio da intensidade $\sqrt{2\gamma k_B T}$. A discretização via Euler-Maruyama resulta em

$$v_{n+1} = v_n - \frac{\gamma}{m}v_n\,\Delta t + \frac{1}{m}\sqrt{2\gamma k_B T\,\Delta t}\,\eta_n,$$
$$x_{n+1} = x_n + v_{n+1}\Delta t.$$

Esse método gera uma trajetória *estocástica*, e não uma solução única. Assim, repetimos a simulação várias vezes e calculamos médias estatísticas.

Deslocamento quadrático médio

Um observable importante no estudo de processos difusivos é o deslocamento quadrático médio:

$$\langle x^2(t) \rangle = \frac{1}{R} \sum_{r=1}^{R} x_r^2(t).$$

Para o movimento browniano clássico, Einstein mostrou que $\langle x^2(t) \rangle \propto t$, o que caracteriza difusão normal.

Código em Python: Langevin via Euler-Maruyama

A seguir apresentamos um código Python *extremamente simples*, com variáveis escalares, loops diretos e comentários detalhados. O objetivo é ilustrar a implementação da equação de Langevin com o método de Euler-Maruyama sem complicações adicionais.

```
import numpy as np

# Parâmetros físicos
gamma = 1.0      # atrito viscoso
kB = 1.0         # constante de Boltzmann
T = 1.0          # temperatura
m = 1.0          # massa

# Parâmetros numéricos
dt = 0.01        # passo de tempo
N = 10000        # número de passos
R = 1000         # número de trajetórias para média

# Vetor para acumular <x^2> ao longo do tempo
mean_x2 = np.zeros(N)

for r in range(R):

    # Estado inicial da partícula em cada trajetória
    x = 0.0
    v = 0.0

    for i in range(N):

        # Gera um número gaussiano ~ N(0,1)
        eta = np.random.normal()

        # Termo estocástico proporcional à temperatura
        noise = np.sqrt(2.0 * gamma * kB * T * dt) * eta

        # Atualização da velocidade (Euler-Maruyama)
        v = v - (gamma/m) * v * dt + noise / m

        # Atualização da posição
```

```
        x = x + v * dt

        # Acumula x^2 para média estatística
        mean_x2[i] += x * x

# Finaliza dividindo pelo número de trajetórias
mean_x2 = mean_x2 / R

# mean_x2 agora contém <x^2(t)> para cada tempo
```

Esse código é ideal para fins didáticos: oferece clareza, simplicidade e mostra explicitamente como o termo de ruído entra na equação via $\sqrt{\Delta t}$. Para aplicações mais avançadas, podem ser incluídos termos multiplicativos $g(y,t)$, ruídos correlacionados, e análise de convergência forte/fraca.

18 Método de Ordem Superior para EDEs: Integrador Estocástico de Milstein

Embora o método de Euler-Maruyama seja simples e amplamente utilizado, sua ordem de convergência forte é apenas $\mathcal{O}(\Delta t^{1/2})$. Para certos problemas, especialmente aqueles com ruído multiplicativo, é desejável empregar métodos de ordem superior. Um dos esquemas mais conhecidos é o **método de Milstein**, que aumenta a ordem de convergência forte para $\mathcal{O}(\Delta t)$.

Método de Milstein

Para uma EDE do tipo

$$dy = f(y,t)\,dt + g(y,t)\,dW_t,$$

o método de Milstein produz a aproximação

$$y_{n+1} = y_n + f(y_n,t_n)\,\Delta t + g(y_n,t_n)\,\Delta W_n + \frac{1}{2}g(y_n,t_n)\,g'(y_n,t_n)\left((\Delta W_n)^2 - \Delta t\right).$$

A correção adicional depende da derivada $g'(y,t) = \partial g/\partial y$. Em casos onde o ruído é aditivo (g = constante), o termo de correção desaparece e Milstein se reduz ao Euler-Maruyama. Já para ruído multiplicativo, esse termo melhora significativamente a estabilidade e a precisão numérica.

Aplicação: Versão Multiplicativa da Equação de Langevin

Para exemplificar o método de Milstein sem complicações físicas desnecessárias, consideraremos uma versão modificada da equação de Langevin, onde a intensidade do ruído depende da própria velocidade:

$$\frac{dv}{dt} = -\frac{\gamma}{m}v(t) + \sigma\, v(t)\,\xi(t), \qquad \frac{dx}{dt} = v(t).$$

Aqui, o ruído multiplicativo $g(v) = \sigma v$ exige o uso de um integrador de ordem superior. O termo de correção passa a envolver

$$g(v)g'(v) = (\sigma v)(\sigma) = \sigma^2 v.$$

Discretizando com Milstein, temos:

$$v_{n+1} = v_n - \frac{\gamma}{m} v_n \, \Delta t + \sigma v_n \, \Delta W_n + \frac{1}{2}\sigma^2 v_n \left[(\Delta W_n)^2 - \Delta t \right],$$

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

Assim como antes, o processo é estocástico e exige múltiplas trajetórias para estimar quantidades médias como o deslocamento quadrático médio.

Deslocamento quadrático médio

A definição permanece idêntica:

$$\langle x^2(t) \rangle = \frac{1}{R} \sum_{r=1}^{R} x_r^2(t).$$

No caso multiplicativo, entretanto, o comportamento difusivo pode desviar do regime linear simples, dependendo dos parâmetros σ e γ.

Código em Python: Langevin Multiplicativo via Milstein

A seguir apresentamos um código Python igualmente simples, mantendo a filosofia de clareza e explicitação completa dos passos. O ruído multiplicativo é implementado diretamente e a correção do método de Milstein aparece de forma transparente.

```
import numpy as np

# Parâmetros físicos
gamma = 1.0     # atrito viscoso
m = 1.0         # massa
sigma = 0.2     # intensidade multiplicativa do ruído

# Parâmetros numéricos
dt = 0.01       # passo de tempo
N = 10000       # número de passos
R = 1000        # número de trajetórias

# Armazena <x^2(t)>
mean_x2 = np.zeros(N)

for r in range(R):

    # Condições iniciais
    x = 0.0
    v = 1.0   # velocidade inicial não nula (importante no caso multiplicativo)

    for i in range(N):

        # Incremento browniano ~ N(0, dt)
        eta = np.random.normal()
        dW = np.sqrt(dt) * eta

        # Termos da EDE
```

```
        drift = -(gamma/m) * v
        diffusion = sigma * v

        # Correção de Milstein: 0.5 * g * g'
        correction = 0.5 * sigma**2 * v * (dW*dW - dt)

        # Atualização da velocidade (Milstein)
        v = v + drift * dt + diffusion * dW + correction

        # Atualização da posição
        x = x + v * dt

        # Acumula x^2
        mean_x2[i] += x * x

# Média final
mean_x2 = mean_x2 / R

# mean_x2 contém <x^2(t)>
```

Esse exemplo mostra como métodos de ordem superior capturam corretamente efeitos associados ao ruído multiplicativo, que são ignorados por integradores mais simples. Em problemas reais de física estatística e simulação estocástica, esse tipo de integrador é essencial para precisão e estabilidade.

19 Integração Numérica via Monte Carlo

O método de **Monte Carlo** é uma técnica estocástica de integração numérica que se baseia na geração de números aleatórios para estimar valores de integrais. Em vez de subdividir o domínio em intervalos como nos métodos determinísticos (trapézio, Simpson etc.), o método Monte Carlo aproxima a integral pela média das avaliações da função em pontos sorteados aleatoriamente. Essa abordagem é especialmente poderosa em integrais de alta dimensão, onde métodos determinísticos tornam-se inviáveis devido ao chamado *"problema da explosão combinatória"*.

Formulação Geral

Considere a integral definida:

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

Sorteamos N pontos aleatórios x_i uniformemente distribuídos em $[a, b]$. A estimativa Monte Carlo é:

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

A incerteza estatística típica diminui como:

$$\text{erro} \sim \frac{1}{\sqrt{N}},$$

independentemente da dimensão do problema.

Exemplo: Estimativa de π via Monte Carlo

Um clássico exemplo consiste em estimar o valor de π usando a área de um quarto de círculo inscrito no quadrado de lado 1. Geramos pontos aleatórios $(x, y) \in [0, 1]^2$ e contamos quantos caem dentro do círculo unitário:

$$x^2 + y^2 \leq 1.$$

Como

$$\frac{\text{área do quarto de círculo}}{\text{área do quadrado}} = \frac{\pi}{4},$$

então

$$\pi \approx 4 \, \frac{N_{\text{dentro}}}{N}.$$

Código em Python: Estimativa de π

A seguir um código simples, didático, usando apenas NumPy e loops explícitos.

```
import numpy as np

# Número total de pontos do método Monte Carlo
N = 200000

# Contador de pontos dentro do círculo
dentro = 0

for i in range(N):
    # Sorteia x, y em [0,1]
    x = np.random.rand()
    y = np.random.rand()

    # Verifica se (x,y) pertence ao quarto de círculo
    if x*x + y*y <= 1.0:
        dentro += 1

# Estimativa final de pi
pi_est = 4.0 * dentro / N
print("Estimativa de pi =", pi_est)
```

O código ilustra o espírito estatístico do método Monte Carlo: cada ponto atua como um ensaio independente, e a estimativa melhora conforme N aumenta.

Aplicações e Observações

- Extremamente útil em **alta dimensão**, onde quadraturas determinísticas se tornam proibitivamente caras.
- Pode ser aplicado a integrais múltiplas, volumes, estimativas de valores esperados e problemas em física estatística (ensemble canônico, caminhos aleatórios, dinâmica molecular etc.).
- A convergência é lenta, mas pode ser acelerada com técnicas avançadas como *importance sampling*, *stratified sampling* e *Markov Chain Monte Carlo* (MCMC).

20 Integração Monte Carlo em Alta Dimensão

Considere agora a integral multidimensional:

$$I_d = \int_0^1 \cdots \int_0^1 \exp\left(-\sum_{i=1}^{d} x_i^2\right) dx_1 \cdots dx_d,$$

no hiper-cubo unitário $[0,1]^d$. Para dimensões elevadas, métodos determinísticos se tornam inviáveis, pois o número de pontos necessários cresce exponencialmente com d. Já o erro do método Monte Carlo depende apenas de N, não de d.

Estimativa via Monte Carlo

Geramos N vetores:

$$\mathbf{x}^{(j)} = (x_1^{(j)}, \cdots, x_d^{(j)}),$$

com componentes distribuídas uniformemente em $[0,1]$. A estimativa é simplesmente:

$$I_d \approx \frac{1}{N} \sum_{j=1}^{N} \exp\left(-\sum_{i=1}^{d} (x_i^{(j)})^2\right),$$

pois o volume do domínio é 1.

Comentário sobre a convergência

Mesmo para dimensões grandes (por exemplo, $d = 20, 50, 100$), o método continua viável, uma vez que o custo computacional cresce linearmente em Nd e o erro depende apenas de $1/\sqrt{N}$.

Código em Python: Integração Monte Carlo Multidimensional

Abaixo temos uma implementação mínima, completamente transparente:

```
import numpy as np

# Parâmetros
N = 200000   # número de amostras Monte Carlo
d = 10       # dimensão da integral

soma = 0.0

for j in range(N):
    # Gera vetor d-dimensional com amostragem uniforme
    x = np.random.rand(d)

    # Soma dos quadrados
    sq = np.sum(x * x)

    # Acumula a função avaliada no ponto
    soma += np.exp(-sq)

# Estimativa final
Id_est = soma / N
print(f"Estimativa da integral em {d} dimensões =", Id_est)
```

Mesmo com $d = 10$, o método fornece uma estimativa útil com custo computacional baixo. O exemplo ilustra bem a grande vantagem do método Monte Carlo: sua eficiência permanece prática em dimensões muito altas, nas quais métodos clássicos se tornam inoperantes. Esta seção fornece uma visão clara e didática das principais ideias por trás da integração Monte Carlo — tanto em baixa quanto em alta dimensão — e apresenta códigos básicos em Python que deixam explícitos todos os passos envolvidos na amostragem e na construção das estimativas estatísticas.

21 Transformada Discreta de Fourier (DFT)

A Transformada Discreta de Fourier (DFT) é uma das ferramentas centrais em física computacional e análise de sinais. Ela permite decompor um conjunto discreto de dados — normalmente amostrados no tempo — em suas componentes de frequência. Essa decomposição é essencial para estudar oscilações, detectar periodicidades e analisar espectros de energia, tanto em sistemas clássicos quanto quânticos.

Definição Matemática

Para uma série real ou complexa composta por N pontos x_n, a DFT é definida como:

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi kn/N}, \qquad k = 0, 1, \ldots, N-1. \tag{3}$$

O coeficiente complexo X_k contém duas informações fundamentais:

- **Amplitude**: dada por $|X_k|$, indica a "força" da componente de frequência.
- **Fase**: dada por $\arg(X_k)$, informa o deslocamento relativo dessa componente.

No caso de sinais reais, vale a simetria:

$$X_k = X_{N-k}^*,$$

o que reduz o custo computacional. No entanto, aqui usaremos sempre a forma completa para destacar a estrutura matemática da DFT.

Interpretação Física

A DFT pode ser interpretada como o cálculo do acoplamento entre o sinal e modos senoidais de diferentes frequências. Cada modo funciona como uma "base vibracional" do sistema. Assim, a DFT fornece um espectro que responde à pergunta:

> "Quais frequências contribuem para formar o sinal observado?"

Esse ponto de vista é fundamental em mecânica estatística, por exemplo, na análise de autocorrelações temporais, ruído espectral, simulações de Monte Carlo e dinâmica molecular.

A implementação em Python abaixo utiliza apenas bibliotecas básicas (`math` e `cmath`), isto é, **sem FFT** e sem NumPy, para fins pedagógicos:

```
import math
import cmath

N = 256
PI = math.pi

# Sinal: seno simples de frequência 5
x = [math.sin(2.0 * PI * 5 * n / N) for n in range(N)]

# Vetor para armazenar a DFT
X = [0] * N

# Cálculo direto da DFT
for k in range(N):
    soma = 0+0j
    for n in range(N):
        angle = -2.0 * PI * k * n / N
        soma += x[n] * cmath.exp(1j * angle)
    X[k] = soma

# Impressão do módulo das frequências
for k in range(N):
    print(k, abs(X[k]))
```

Observações Finais

- A implementação direta é útil para estudo, mas computacionalmente custosa.
- Em aplicações reais, usa-se a FFT (*Fast Fourier Transform*), que reduz o custo para $\mathcal{O}(N \log N)$.
- A DFT é essencial em análises espectrais de autocorrelação, simulações estocásticas e estudos de modos vibracionais.

22 Solução Numérica de Sistemas Lineares

Sistemas de equações lineares do tipo

$$A\mathbf{x} = \mathbf{b}$$

são onipresentes em física computacional. Eles aparecem, por exemplo, na discretização de equações diferenciais, em métodos variacionais, em problemas de otimização e em modelos de transporte de calor e difusão. Resolver eficientemente tais sistemas é essencial para qualquer simulação numérica.

Nesta seção apresentamos dois métodos fundamentais:

- **Eliminação de Gauss**: método direto geral para qualquer matriz quadrada não singular.
- **Método de Thomas**: versão especializada e muito eficiente para matrizes tridiagonais.

Os códigos a seguir são todos apresentados em **Python**, de forma didática, usando apenas listas nativas (sem NumPy), de modo a explicitar a lógica dos algoritmos numéricos.

22.1 Eliminação de Gauss (sem pivotamento)

Dado um sistema linear

$$A\mathbf{x} = \mathbf{b},$$

a eliminação de Gauss transforma a matriz A em uma matriz triangular superior. Isso é feito através da eliminação sucessiva dos termos abaixo da diagonal principal. Em seguida, o sistema triangular é resolvido via *substituição regressiva*.

Embora versões mais robustas incluam pivotamento parcial ou total, apresentamos aqui a versão mais simples para fins didáticos.

Algoritmo básico

1. Para cada coluna $k = 0, 1, \ldots, N-2$:

   (a) Para cada linha $i = k+1, \ldots, N-1$:

$$m_{ik} = \frac{A_{ik}}{A_{kk}}$$

$$A_{ij} \leftarrow A_{ij} - m_{ik} A_{kj}, \quad b_i \leftarrow b_i - m_{ik} b_k$$

2. Após a eliminação, o sistema é triangular:

$$A_{ij} x_j = b_i$$

   e resolvemos por substituição regressiva:

$$x_i = \frac{1}{A_{ii}} \left(b_i - \sum_{j=i+1}^{N-1} A_{ij} x_j \right).$$

Código em Python: Eliminação de Gauss para sistema 3×3

O código abaixo implementa exatamente este algoritmo usando listas comuns de Python, de forma clara e objetiva.

```
# Eliminação de Gauss simples (sem pivotamento) para sistema 3x3

A = [
    [2, -1,  1],
    [3,  3,  9],
    [3,  3,  5]
]

b = [8, 0, -6]
N = 3

# Eliminação
for k in range(N-1):
    for i in range(k+1, N):
        fator = A[i][k] / A[k][k]
        for j in range(k, N):
            A[i][j] -= fator * A[k][j]
        b[i] -= fator * b[k]
```

```
# Substituição regressiva
x = [0] * N
for i in range(N-1, -1, -1):
    soma = sum(A[i][j] * x[j] for j in range(i+1, N))
    x[i] = (b[i] - soma) / A[i][i]

print("Solução:", x)
```

Esse exemplo ilustra o fluxo completo do algoritmo: redução para triangular superior e resolução por substituição regressiva.

22.2 Sistemas Lineares Tridiagonais

Muitas aplicações em física computacional geram sistemas lineares com matriz tridiagonal, como problemas de difusão, discretização de EDOs e métodos de diferenças finitas. Uma matriz tridiagonal possui a forma:

$$A = \begin{bmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{bmatrix},$$

onde apenas três diagonais possuem elementos não nulos.

Sistemas tridiagonais podem ser resolvidos pelo método geral de Gauss, mas isso desperdiça a estrutura especial da matriz. Em vez disso, existe um algoritmo específico com complexidade linear $\mathcal{O}(N)$: o **método de Thomas**.

Método de Thomas

Esse método é uma forma especializada da eliminação de Gauss, aproveitando que cada linha depende apenas dos vizinhos imediatos. Ele consiste em duas etapas:

1. **Forward sweep**: elimina a subdiagonal e modifica os coeficientes.
2. **Back substitution**: resolve o sistema triangular resultante.

Dado o sistema

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i,$$

com $a_1 = 0$ e $c_N = 0$, o algoritmo é extremamente estável quando a matriz é diagonal dominante.

Código em Python: Método de Thomas

O código a seguir resolve o sistema tridiagonal do exemplo:

$$\begin{bmatrix} 4 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 15 \\ 15 \\ 15 \\ 10 \end{bmatrix}$$

```
# Método de Thomas para sistemas tridiagonais

a = [0, 1, 1, 1]  # subdiagonal (a[0] ignorado)
b = [4, 4, 4, 3]  # diagonal principal
c = [1, 1, 1, 0]  # superdiagonal (c[-1] ignorado)
d = [15, 15, 15, 10]
N = 4

# Vetores modificados
c_prime = [0] * N
d_prime = [0] * N

# Forward sweep
c_prime[0] = c[0] / b[0]
d_prime[0] = d[0] / b[0]

for i in range(1, N):
    m = b[i] - a[i] * c_prime[i-1]
    c_prime[i] = c[i] / m if i < N-1 else 0
    d_prime[i] = (d[i] - a[i] * d_prime[i-1]) / m

# Back substitution
x = [0] * N
x[-1] = d_prime[-1]

for i in range(N-2, -1, -1):
    x[i] = d_prime[i] - c_prime[i] * x[i+1]

print("Solução (Thomas):", x)
```

O método de Thomas é preferido sempre que a matriz é tridiagonal, pois reduz drasticamente o custo computacional para $O(N)$, além de ser numericamente estável para matrizes diagonais dominantes.

Resumo

- A eliminação de Gauss é um método geral aplicável a qualquer matriz quadrada.
- Métodos especializados, como o método de Thomas, aproveitam estruturas particulares (tridiagonalidade) para aumentar eficiência.
- Em problemas reais, bibliotecas como NumPy usam algoritmos otimizados e versões com pivotamento, mas conhecer os métodos diretos é essencial para entender a base dos métodos numéricos.

23 Método da Bisseção

O método da bisseção é uma das técnicas numéricas mais simples e robustas para encontrar raízes reais de uma equação do tipo

$$f(x) = 0,$$

desde que a função seja contínua e apresente uma troca de sinal em um intervalo inicial $[a, b]$. Essa condição, expressa como

$$f(a)\, f(b) < 0,$$

garante, pelo Teorema do Valor Intermediário, que existe pelo menos uma raiz dentro do intervalo.

A ideia central do método consiste em **dividir repetidamente o intervalo** ao meio, identificando em qual subintervalo ocorre a troca de sinal. Assim, a cada iteração o intervalo contendo a raiz é reduzido pela metade, o que garante convergência, embora de forma relativamente lenta (convergência linear).

Formulação do Método

Em cada iteração, calcula-se o ponto médio do intervalo:

$$c = \frac{a+b}{2}.$$

A seguir, avalia-se o sinal de $f(c)$:

- Se $f(c) = 0$, então c é a raiz exata.
- Se $f(a)\,f(c) < 0$, então a raiz está em $[a, c]$.
- Se $f(c)\,f(b) < 0$, então a raiz está em $[c, b]$.

O processo se repete até que o tamanho do intervalo seja menor que uma tolerância prescrita ϵ, ou até que $f(c)$ seja suficientemente pequeno.

Exemplo: Raiz de um Polinômio de 4º Grau

Considere agora o polinômio

$$g(x) = 2x^4 - 5x^2 + x - 1.$$

Uma análise simples mostra que:

$$g(0) = -1, \qquad g(1) = 2 - 5 + 1 - 1 = -3,$$

e que em um ponto ligeiramente acima de 1, por exemplo

$$g(1.5) = 2(1.5)^4 - 5(1.5)^2 + 1.5 - 1 = 10.125 - 11.25 + 1.5 - 1 = -0.625,$$

enquanto em

$$g(2) = 2(16) - 5(4) + 2 - 1 = 32 - 20 + 2 - 1 = 13 > 0.$$

Portanto, existe ao menos uma raiz no intervalo:

$$[1.5,\, 2],$$

que contém uma clara mudança de sinal:

$$g(1.5)\,g(2) < 0.$$

Implementação em Python

A seguir apresentamos uma implementação detalhada do método da bisseção em Python, com comentários extensivos para fins didáticos. O programa:

- define a função polinomial;
- verifica automaticamente se há mudança de sinal no intervalo inicial;
- executa o processo iterativo da bisseção;
- armazena todos os valores das iterações para possível análise posterior;
- salva os dados em um arquivo texto.

```
import numpy as np

# ---------------------------------------------------------
# Definição da função polinomial
# g(x) = 2x^4 - 5x^2 + x - 1
# ---------------------------------------------------------
def g(x):
    return 2*x**4 - 5*x**2 + x - 1

# ---------------------------------------------------------
# Parâmetros do método
# ---------------------------------------------------------
a = 1.5          # limite inferior do intervalo inicial
b = 2.0          # limite superior do intervalo inicial
eps = 1e-6       # tolerância de convergência
max_it = 100     # número máximo de iterações permitidas

# ---------------------------------------------------------
# Verificação da mudança de sinal
# O método da bisseção só é válido se:
# g(a) * g(b) < 0
# ---------------------------------------------------------
if g(a) * g(b) >= 0:
    raise ValueError("Intervalo inválido: g(a) * g(b) >= 0.\
O método da bisseção não pode ser aplicado.")

# ---------------------------------------------------------
# Lista para armazenar os dados das iterações
# Cada elemento será uma tupla: (iteração, ponto_médio, g(ponto_médio))
# ---------------------------------------------------------
dados = []

# ---------------------------------------------------------
# Loop principal do método da bisseção
# ---------------------------------------------------------
for it in range(max_it):

    # Cálculo do ponto médio do intervalo atual
    c = (a + b) / 2.0

    # Armazenando a iteração
    dados.append((it, c, g(c)))

    # Critério de parada baseado no valor da função
    if abs(g(c)) < eps:
```

```
        break

    # Seleção do subintervalo que contém a raiz
    # Mantemos sempre o intervalo onde há mudança de sinal
    if g(a) * g(c) < 0:
        b = c     # a raiz continua no subintervalo [a,c]
    else:
        a = c     # a raiz está no subintervalo [c,b]

# ------------------------------------------------------
# Impressão do resultado final
# ------------------------------------------------------
print("Raiz aproximada:", c)
print("g(c) =", g(c))

# ------------------------------------------------------
# Salvando dados das iterações para análise gráfica posterior
# ------------------------------------------------------
with open("bissecao.dat", "w") as fp:
    for it, c, gc in dados:
        fp.write(f"{it} {c:.10f} {gc:.10f}\n")
```

O método da bisseção é especialmente útil em situações em que se busca confiabilidade sobre velocidade. Sempre que se conhece um intervalo com troca de sinal, o método garante convergência, independentemente da forma da função. Além disso, seu comportamento previsível o torna ideal para introdução ao estudo de métodos numéricos e para análises exploratórias em física computacional.

24 Método de Newton-Raphson

O método de Newton-Raphson é uma técnica iterativa extremamente eficiente para encontrar raízes de equações não lineares do tipo

$$f(x) = 0.$$

A ideia central é usar não apenas o valor da função, mas também a sua derivada. A cada iteração, constrói-se uma aproximação para a raiz a partir da tangente à curva de $f(x)$ no ponto atual. Esse uso da informação da derivada torna o método significativamente mais rápido do que técnicas puramente intervalares, como a bisseção.

A fórmula de iteração é:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Quando a aproximação inicial x_0 está suficientemente próxima da raiz, o método apresenta **convergência quadrática**, isto é, o erro diminui muito rapidamente, dobrando essencialmente o número de casas decimais corretas a cada iteração.

Aplicação: Raiz de um Polinômio de 4º Grau

Considere o polinômio

$$f(x) = x^4 - 3x^3 - 7x^2 + 27x - 18,$$

cuja derivada é

$$f'(x) = 4x^3 - 9x^2 - 14x + 27.$$

Escolhendo um valor inicial razoável, como $x_0 = 0.5$, podemos aplicar o método iterativamente até atingir a precisão desejada.

Código em Python: Método de Newton-Raphson

A seguir, apresentamos uma implementação simples do método em Python. O código registra as iterações para permitir análises posteriores.

```
import numpy as np

# Função polinomial f(x)
def f(x):
    return x**4 - 3*x**3 - 7*x**2 + 27*x - 18

# Derivada f'(x)
def df(x):
    return 4*x**3 - 9*x**2 - 14*x + 27

# Parâmetros do método
x = 0.5        # chute inicial
eps = 1e-6     # tolerância
max_it = 100   # número máximo de iterações

dados = []     # lista para armazenar iterações

for it in range(max_it):
    x_new = x - f(x)/df(x)    # fórmula de Newton
    dados.append((it, x_new, f(x_new)))

    if abs(x_new - x) < eps:
        break

    x = x_new

print("Raiz aproximada:", x)
print("f(x) =", f(x))

# Salvando os dados
with open("newton.dat", "w") as fp:
    for it, xn, fx in dados:
        fp.write(f"{it} {xn:.10f} {fx:.10f}\n")
```

Exemplo Adicional: Determinação do Potencial Químico

O método de Newton-Raphson é amplamente utilizado em mecânica estatística, especialmente quando precisamos **determinar o potencial químico** μ de um sistema a partir de uma equação de normalização. Por exemplo, no caso de um gás clássico ideal no ensemble grande canônico, a densidade é dada por:

$$n = \frac{1}{\lambda^3} e^{\beta\mu},$$

onde λ é o comprimento térmico de de Broglie e $\beta = 1/(k_B T)$. Se conhecemos o valor desejado de n, podemos reorganizar a equação para resolver μ, mas em vários modelos mais realistas a equação assume formas não invertíveis analiticamente. Assim, precisamos resolver numericamente a condição

$$F(\mu) = n_{\text{alvo}} - n(\mu) = 0.$$

Abaixo mostramos um exemplo simples em que determinamos o valor de μ que produz uma densidade dada n_{alvo}. Consideramos um gás clássico para fins ilustrativos.

```
import numpy as np

# Parâmetros físicos (exemplo)
beta = 1.0
lambda_th = 1.0          # comprimento térmico
n_target = 0.1           # densidade desejada

# Função densidade do gás clássico ideal
def n(mu):
    return np.exp(beta * mu) / lambda_th**3

# Função F(mu) = n_target - n(mu)
def F(mu):
    return n_target - n(mu)

# Derivada F'(mu)
def dF(mu):
    return -beta * n(mu)

# Método de Newton
mu = -5.0          # chute inicial
eps = 1e-6
max_it = 50

for it in range(max_it):
    mu_new = mu - F(mu)/dF(mu)

    if abs(mu_new - mu) < eps:
        break

    mu = mu_new

print("Potencial químico:", mu)
print("Checagem da densidade:", n(mu))
```

Este exemplo ilustra como métodos iterativos são essenciais em sistemas estatísticos mais realistas, onde funções de partição e equações de estado levam a expressões não triviais para grandezas termodinâmicas como o potencial químico. O método de Newton-Raphson, pela sua rapidez e simplicidade, torna-se uma ferramenta indispensável em cálculos computacionais de mecânica estatística.

25 Autovalor e Autovetor: Combinação dos Métodos de Bisseção e Thomas

Antes de iniciar a aplicação dos métodos de Bisseção e Thomas para o cálculo de autovalor e autovetor vamos fazer uma breve revisão sobre multiplicação de matrizes em python. Em linhas gerais, a multiplicação de matrizes é uma operação fundamental em computação científica, álgebra linear e simulações numéricas. Em Python, é possível realizá-la de duas formas principais: (i) utilizando loops explícitos, no estilo "força bruta", o que ajuda a

entender o mecanismo básico da operação; e (ii) utilizando as funcionalidades internas eficientes do Python moderno (particularmente `NumPy`), que oferecem desempenho muito superior.

Multiplicação de uma Matriz por um Escalar

Dado um escalar c e uma matriz A, a multiplicação por escalar consiste em multiplicar cada elemento da matriz pelo valor constante:

$$(cA)_{ij} = c\, A_{ij}.$$

Multiplicação Matriz–Matriz

Para duas matrizes compatíveis A (de tamanho $m \times n$) e B (de tamanho $n \times p$), o produto $C = AB$ é definido como:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

A seguir mostramos as duas abordagens: uma implementação manual e outra usando `NumPy`.

Código "força bruta" com loops

```
# Multiplicação por escalar
def scalar_mult(c, A):
    m = len(A)
    n = len(A[0])
    C = [[0.0 for _ in range(n)] for _ in range(m)]
    for i in range(m):
        for j in range(n):
            C[i][j] = c * A[i][j]
    return C

# Multiplicação matriz-matriz (A de m x n, B de n x p)
def mat_mult(A, B):
    m = len(A)
    n = len(A[0])
    p = len(B[0])
    # matriz resultado m x p
    C = [[0.0 for _ in range(p)] for _ in range(m)]
    for i in range(m):
        for j in range(p):
            s = 0.0
            for k in range(n):
                s += A[i][k] * B[k][j]
            C[i][j] = s
    return C

# Exemplo simples
A = [[1, 2], [3, 4]]
B = [[2, 0], [1, 2]]

print("A*B (força bruta):", mat_mult(A, B))
print("3*A:", scalar_mult(3, A))
```

Código usando NumPy

A biblioteca `NumPy` fornece operações vetorizadas altamente otimizadas, incluindo:

$$AB \rightarrow A@B$$

$$cA \rightarrow c * A$$

```
import numpy as np

A = np.array([[1, 2],
              [3, 4]])

B = np.array([[2, 0],
              [1, 2]])

# Multiplicação por escalar
C1 = 3 * A

# Multiplicação matriz-matriz
C2 = A @ B

print("3*A:\n", C1)
print("A@B:\n", C2)
```

O uso de `@` ou da função `np.matmul` é recomendado para aplicações em física computacional, álgebra linear numérica e simulações, pois aproveita otimizações de baixo nível (BLAS/LAPACK), tornando o código significativamente mais rápido e conciso.

Agora que revisamos este tópico importante, vamos aplicar dois métodos numéricos em sequência — bisseção e Thomas — para determinar um autovalor e o autovetor correspondente de uma matriz tridiagonal simétrica H. Essa abordagem é inspirada no estudo do **Modelo de Anderson 1D**, em que o Hamiltoniano é dado por:

$$H_{ij} = \varepsilon_i \delta_{ij} - t(\delta_{i,j+1} + \delta_{i,j-1}),$$

com ε_i representando desordem e t o termo de hopping. Este tipo de matriz é tridiagonal, o que permite o uso eficiente do método de Thomas na resolução do sistema linear.

O problema de autovalores consiste em encontrar λ e $\vec{v}$ tais que:

$$H\vec{v} = \lambda\vec{v} \quad \Leftrightarrow \quad (H - \lambda I)\vec{v} = 0.$$

A estratégia é:

1. Aplicar o **método da bisseção** para encontrar um autovalor, procurando o zero de $\det(H - \lambda I)$.
2. Substituir esse valor em $H - \lambda I$ e usar o **método de Thomas** para resolver o sistema homogêneo e extrair o autovetor associado.

Como o sistema é homogêneo, precisamos fixar uma condição para evitar a solução trivial; aqui impomos $v_0 = 1$. Após a solução, normalizamos o autovetor.

Aplicação no Modelo de Anderson 1D com $N = 6$

Vamos considerar por exemplo a matriz do modelo de Anderson 1d com $N = 6$ sítios. A matriz é dada por:

$$H = \begin{bmatrix} \varepsilon_1 & -t & 0 & 0 & 0 & 0 \\ -t & \varepsilon_2 & -t & 0 & 0 & 0 \\ 0 & -t & \varepsilon_3 & -t & 0 & 0 \\ 0 & 0 & -t & \varepsilon_4 & -t & 0 \\ 0 & 0 & 0 & -t & \varepsilon_5 & -t \\ 0 & 0 & 0 & 0 & -t & \varepsilon_6 \end{bmatrix}$$

Usaremos termo de hopping $t = 1$ e valores típicos de desordem moderada:

$$\varepsilon = (0.2,\ -1.1,\ 0.7,\ -0.4,\ 0.5,\ -0.9).$$

Código em Python — Bisseção + Thomas

O código a seguir implementa:

- cálculo do determinante tridiagonal via eliminação (sem construir a matriz completa);
- método da bisseção para encontrar um autovalor;
- método de Thomas para extrair o autovetor associado;
- normalização final.

```
import numpy as np

# Parâmetros do modelo de Anderson 1D
t = 1.0
eps = np.array([0.2, -1.1, 0.7, -0.4, 0.5, -0.9])
N = len(eps)

# ------------------------------------------------------------
# Determinante tridiagonal de (H - lambda I)
# ------------------------------------------------------------
def det(lambda_):
    # diagonal principal modificada
    d = eps - lambda_
    # cópia que sofrerá eliminação
    b = d.copy()

    # eliminação tridiagonal para determinante
    for i in range(1, N):
        m = (-t) / b[i-1]
        b[i] = b[i] - m * (-t)

    return b[-1]

# ------------------------------------------------------------
# Método da bisseção para buscar autovalor
# ------------------------------------------------------------
def bissecao(a, b, tol=1e-6, max_it=100):
    fa = det(a)
    fb = det(b)

    for _ in range(max_it):
        c = 0.5 * (a + b)
        fc = det(c)

        if fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc

        if abs(b - a) < tol:
            break
```

```
    return 0.5 * (a + b)

# Intervalo inicial (pode ser ajustado conforme o espectro)
lambda_found = bissecao(-3, 3)
print("Autovalor encontrado:", lambda_found)

# ------------------------------------------------------------
# Método de Thomas para resolver (H - lambda I)v = 0
# com v[0] = 1 fixado
# ------------------------------------------------------------

# Construindo diagonais
d = eps - lambda_found               # diagonal principal
a_sub = -t * np.ones(N-1)            # subdiagonal
a_sup = -t * np.ones(N-1)            # superdiagonal

# RHS para sistema homogêneo com v[0]=1
rhs = np.zeros(N)
rhs[1] = t  # deslocamento devido a v[0] = 1

# Vetores auxiliares para Thomas
cp = np.zeros(N-1)
dp = np.zeros(N)

# Forward elimination
cp[0] = a_sup[0] / d[0]
dp[0] = 0  # v[0] fixado, não entra no sistema

dp[1] = (rhs[1]) / d[1]

for i in range(1, N-1):
    denom = d[i] - a_sub[i-1] * cp[i-1]
    cp[i] = a_sup[i] / denom
    dp[i+1] = (-a_sub[i-1] * dp[i]) / denom

# Backward substitution
v = np.zeros(N)
v[0] = 1.0
v[-1] = dp[-1]

for i in range(N-2, 0, -1):
    v[i] = dp[i] - cp[i] * v[i+1]

# Normalização
norm = np.linalg.norm(v)
v /= norm

print("\nAutovetor normalizado:")
for i in range(N):
    print(f"v[{i}] = {v[i]:.6f}")

# ------------------------------------------------------------
# Verificação: H v ~ lambda v
# ------------------------------------------------------------
H = np.zeros((N,N))
for i in range(N):
    H[i,i] = eps[i]
    if i < N-1:
        H[i,i+1] = -t
```

```
        H[i+1,i] = -t

check = H @ v
print("\nVerificação H v:")
print(check)
print("\nlambda v:")
print(lambda_found * v)
```

O método da bisseção fornece um autovalor com robustez garantida, desde que o intervalo inicial contenha uma mudança de sinal no determinante tridiagonal. Em seguida, o método de Thomas permite resolver de forma altamente eficiente o sistema linear associado, já que a matriz é tridiagonal. O resultado é a obtenção de um autovalor do Hamiltoniano do modelo de Anderson e seu autovetor associado, ambos verificados numericamente pela comparação entre $H\vec{v}$ e $\lambda\vec{v}$. Pequenas diferenças numéricas são esperadas devido às aproximações inerentes aos métodos utilizados.

26 Autovetor e Autovalor : Método da Potência

O **método da potência** é uma técnica simples e eficiente para estimar o autovalor dominante (aquele de maior módulo) e seu autovetor associado. Seja $H \in \mathbb{R}^{N\times N}$ uma matriz simétrica (ou Hermitiana), com autovalores ordenados por módulo

$$|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_N|.$$

Dado um vetor inicial $\mathbf{v}^{(0)}$ que tenha componente não nula ao longo do autovetor dominante $\mathbf{u}_1$, a iteração

$$\mathbf{w}^{(n+1)} = H\,\mathbf{v}^{(n)}, \qquad \mathbf{v}^{(n+1)} = \frac{\mathbf{w}^{(n+1)}}{\|\mathbf{w}^{(n+1)}\|}$$

conduz a $\mathbf{v}^{(n)} \to \mathbf{u}_1$. Uma estimativa eficiente para o autovalor é dada pelo quociente de Rayleigh,

$$\lambda^{(n)} = (\mathbf{v}^{(n)})^T H\,\mathbf{v}^{(n)}.$$

Critério de parada. A iteração pode ser interrompida quando

$$\frac{|\lambda^{(n)} - \lambda^{(n-1)}|}{|\lambda^{(n)}|} < \tau \qquad \text{ou} \qquad \|\mathbf{v}^{(n)} - \mathbf{v}^{(n-1)}\| < \tau.$$

Algoritmo (passo a passo)

1. Escolha $\mathbf{v}^{(0)} \neq 0$ e normalize.
2. Para $n = 0, 1, 2, \ldots$:
   (a) Compute $\mathbf{w}^{(n+1)} = H\,\mathbf{v}^{(n)}$.
   (b) Normalize: $\mathbf{v}^{(n+1)} = \mathbf{w}^{(n+1)}/\|\mathbf{w}^{(n+1)}\|$.
   (c) Estime $\lambda^{(n+1)} = (\mathbf{v}^{(n+1)})^T H\,\mathbf{v}^{(n+1)}$.
   (d) Teste o critério de parada.

Exemplo com uma matriz simétrica 3×3

Consideremos agora a matriz

$$H = \begin{pmatrix} 4 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 3 \end{pmatrix}.$$

Essa matriz é simétrica, possui espectro real e tem autovalor dominante $\lambda_{\max} \approx 4.56155$. Aplicaremos o método da potência usando o vetor inicial

$$\mathbf{v}^{(0)} = \frac{1}{\sqrt{3}}(1,1,1)^T.$$

Implementação em Python

A seguir apresentamos um código simples e totalmente comentado, adequado para fins didáticos. Ele:

- define a matriz H e o vetor inicial;
- executa o método da potência passo a passo;
- normaliza a cada iteração;
- calcula o quociente de Rayleigh;
- interrompe quando atinge a tolerância.

```
import numpy as np

# ------------------------------------------------------------
# Matriz simétrica 3x3
# ------------------------------------------------------------
H = np.array([
    [4.0, 1.0, 0.0],
    [1.0, 2.0, 1.0],
    [0.0, 1.0, 3.0]
])

# ------------------------------------------------------------
# Vetor inicial normalizado
# ------------------------------------------------------------
v = np.array([1.0, 1.0, 1.0], dtype=float)
v = v / np.linalg.norm(v)   # normalização inicial

# Parâmetros do método
tol = 1e-10
max_it = 1000

lambda_old = 0.0

for it in range(max_it):

    # Multiplicação matriz-vetor: w = H v
    w = H @ v

    # Cálculo do quociente de Rayleigh: autovalor estimado
```

```
    lambda_new = v @ w

    # Normalização para próxima iteração
    v = w / np.linalg.norm(w)

    # Teste de convergência
    if abs(lambda_new - lambda_old) < tol:
        break

    lambda_old = lambda_new

# ------------------------------------------------------------
# Impressão dos resultados
# ------------------------------------------------------------
print("Autovalor dominante aproximado:", lambda_new)
print("Autovetor correspondente (normalizado):", v)

# Teste: verificar se Hv ~ lambda v
Hv = H @ v
print("\nTeste Hv versus lambda*v:")
for i in range(3):
    print(f"Hv[{i}] = {Hv[i]:.10f}   lambda*v[{i}] = {(lambda_new*v[i]):.10f}")
```

O código acima implementa de forma direta o método da potência. Em cada iteração o produto Hv faz com que a componente ao longo do autovetor dominante seja amplificada, e a normalização mantém o vetor em escala controlada. O quociente de Rayleigh fornece uma estimativa do autovalor dominante, que converge rapidamente. Ao final, o teste direto $H\mathbf{v} \approx \lambda\mathbf{v}$ confirma numericamente que o vetor obtido é um autovetor associado ao maior autovalor de H.

Exemplo de uma matriz simétrica 4×4

Consideremos agora a matriz simétrica tridiagonal

$$H_4 = \begin{pmatrix} 3 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 3 \end{pmatrix}.$$

Ela representa um sistema linear com acoplamento local entre vizinhos imediatos, comum em modelos de cadeias unidimensionais. Seus autovalores são aproximadamente

$$\{2.1716,\ 3,\ 4,\ 5.8284\},$$

de modo que o autovalor dominante é

$$\lambda_{\max} \approx 5.82842712.$$

Usando o vetor inicial normalizado

$$\mathbf{v}^{(0)} = \tfrac{1}{2}(1,1,1,1)^T,$$

podemos aplicar o método da potência para obter iterativamente o autovalor dominante e seu autovetor associado.

Código em Python

A seguir apresentamos uma implementação simples, comentada e adequada para fins didáticos. Ela executa:

- multiplicação matriz–vetor,
- estimativa do autovalor pelo quociente de Rayleigh,
- normalização do vetor,
- critério de parada baseado na variação do autovalor.

```
import numpy as np

# ------------------------------------------------------------
# Matriz simétrica 4x4 do exemplo
# ------------------------------------------------------------
H = np.array([
    [3.0, 1.0, 0.0, 0.0],
    [1.0, 4.0, 1.0, 0.0],
    [0.0, 1.0, 4.0, 1.0],
    [0.0, 0.0, 1.0, 3.0]
])

# ------------------------------------------------------------
# Vetor inicial: (1,1,1,1)^T normalizado
# ------------------------------------------------------------
v = np.array([1.0, 1.0, 1.0, 1.0], dtype=float)
v = v / np.linalg.norm(v)

tol = 1e-12     # tolerância para convergência
max_it = 1000   # máximo de iterações
lambda_old = 0.0

for it in range(max_it):

    # Multiplicação matriz-vetor: w = H v
    w = H @ v

    # Cálculo do autovalor via quociente de Rayleigh
    lambda_new = v @ w

    # Normalização do vetor para próxima iteração
    v = w / np.linalg.norm(w)

    # Teste da convergência
    if abs(lambda_new - lambda_old) < tol:
        break

    lambda_old = lambda_new

# ------------------------------------------------------------
# Resultados finais
# ------------------------------------------------------------
print("Autovalor dominante aproximado:", lambda_new)
print("Autovetor correspondente (normalizado):", v)

# Teste da relação Hv ~ lambda v
Hv = H @ v
```

```
print("\nComparação Hv versus lambda*v:")
for i in range(4):
    print(f"Hv[{i}] = {Hv[i]:.10f}   lambda*v[{i}] = {(lambda_new*v[i]):.10f}")
```

O código acima demonstra a eficiência do método da potência: a componente do vetor ao longo do autovetor dominante é amplificada a cada iteração, e a normalização evita crescimento numérico descontrolado. O quociente de Rayleigh fornece uma excelente aproximação para o autovalor dominante já nas primeiras iterações. Ao final, a comparação direta entre $H\mathbf{v}$ e $\lambda\mathbf{v}$ confirma a consistência da solução e a convergência para o autovetor associado ao maior autovalor de H.

26.1 Método da Potência Inversa com Deslocamento para uma Matriz Simétrica 4×4

O **método da potência inversa com deslocamento** é uma técnica poderosa para isolar um autovalor específico de uma matriz, especialmente quando sabemos aproximadamente sua posição no espectro. Diferentemente do método da potência comum — que sempre converge para o maior autovalor em magnitude — o método inverso com deslocamento permite aproximar *qualquer* autovalor que esteja próximo de um chute inicial μ.

Ideia Geral do Método

Dada uma matriz simétrica H, queremos encontrar o autovalor mais próximo de um valor arbitrário μ. Para isso, definimos a matriz deslocada

$$A_{\text{shifted}} = H - \mu I,$$

onde I é a identidade. Se λ é um autovalor de H, então $\lambda - \mu$ é um autovalor de A_{shifted}. Logo, os autovalores de sua inversa são

$$\frac{1}{\lambda - \mu}.$$

Portanto, o autovalor de H mais próximo de μ corresponde ao *maior autovalor em magnitude* de $(H - \mu I)^{-1}$. Assim, basta aplicar a potência inversa:

$$(H - \mu I)\, w = v, \quad \Rightarrow \quad v \leftarrow \frac{w}{\|w\|}.$$

Cada iteração resolve um sistema linear. Ao final, o autovalor aproximado de H é obtido com o quociente de Rayleigh:

$$\lambda \approx \frac{v^T H v}{v^T v}.$$

Método de Gauss Usado para Resolver o Sistema Linear

Em cada iteração precisamos resolver

$$(H - \mu I)w = v.$$

Para fins didáticos, usamos uma implementação **básica** do método de **eliminação de Gauss sem pivoteamento**, composta de:

1. formação da matriz aumentada $[A|b]$;

2. eliminação direta para obter uma matriz triangular superior;
3. substituição regressiva para recuperar o vetor solução.

Embora esse método não seja o mais robusto numericamente, ele é ideal para demonstrar o funcionamento interno do algoritmo.

Exemplo da Matriz

A matriz simétrica usada é

$$H = \begin{pmatrix} 4 & 1 & 2 & 0 \\ 1 & 3 & 0 & 1 \\ 2 & 0 & 5 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix},$$

e escolhemos como chute inicial $\mu = 6.5$, próximo do maior autovalor.

Código Didático Completo em Python

O código abaixo é o mais simples possível. Ele:

- implementa eliminação de Gauss manualmente;
- normaliza vetores "na mão";
- evita qualquer biblioteca além de `math`;
- usa apenas listas nativas do Python;
- comenta cada etapa com detalhes conceituais.

```
import math

# ------------------------------------------------------------
# Função para imprimir vetores de maneira amigável
# ------------------------------------------------------------
def print_vec(label, v):
    print(label, "(" + " ".join(f"{x:.10f}" for x in v) + ")")

# ------------------------------------------------------------
# Norma euclidiana de um vetor
# ------------------------------------------------------------
def norm(v):
    s = 0.0
    for x in v:
        s += x*x
    return math.sqrt(s)

# ------------------------------------------------------------
# Normalização de vetor
# ------------------------------------------------------------
def normalize(v):
    n = norm(v)
    return [x/n for x in v]

# ------------------------------------------------------------
# Produto interno
# ------------------------------------------------------------
```

```
def dot(u, v):
    s = 0.0
    for i in range(len(u)):
        s += u[i]*v[i]
    return s

# ------------------------------------------------------------
# Multiplicação matriz-vetor: H v
# ------------------------------------------------------------
def matvec(H, v):
    N = len(v)
    w = [0.0]*N
    for i in range(N):
        for j in range(N):
            w[i] += H[i][j]*v[j]
    return w

# ------------------------------------------------------------
# Eliminação de Gauss (método simples, sem pivoteamento)
# Resolve o sistema A x = b
# ------------------------------------------------------------
def gauss(A, b):
    N = len(b)

    # Criar matriz aumentada [A | b]
    M = []
    for i in range(N):
        linha = A[i][:] + [b[i]]
        M.append(linha)

    # Etapa de eliminação: transformamos M em triangular superior
    for k in range(N):
        pivot = M[k][k]

        # Dividir linha do pivô para deixar pivô = 1
        for j in range(k, N+1):
            M[k][j] /= pivot

        # Eliminar entradas abaixo do pivô
        for i in range(k+1, N):
            fator = M[i][k]
            for j in range(k, N+1):
                M[i][j] -= fator * M[k][j]

    # Substituição regressiva
    x = [0.0]*N
    for i in range(N-1, -1, -1):
        x[i] = M[i][N]
        for j in range(i+1, N):
            x[i] -= M[i][j] * x[j]

    return x

# ------------------------------------------------------------
# MATRIZ DO EXEMPLO
# ------------------------------------------------------------
H = [
    [4,1,2,0],
    [1,3,0,1],
    [2,0,5,1],
```

```
    [0,1,1,2]
]

N = 4
mu = 6.5  # chute inicial

# Construir matriz deslocada A = H - mu I
A = [[0.0]*N for _ in range(N)]
for i in range(N):
    for j in range(N):
        A[i][j] = H[i][j]
    A[i][i] -= mu  # subtrai mu da diagonal

# Vetor inicial
v = [1.0, 1.0, 1.0, 1.0]
v = normalize(v)

TOL = 1e-12
MAX_IT = 1000
lambda_old = 0.0

for it in range(MAX_IT):

    # Resolve (H - mu I) w = v pelo método de Gauss
    w = gauss(A, v)

    # Normaliza w para usar na próxima etapa
    w = normalize(w)

    # Quociente de Rayleigh: aproxima o autovalor de H
    Hw = matvec(H, w)
    lambda_new = dot(w, Hw) / dot(w, w)

    # Verifica convergência
    if abs(lambda_new - lambda_old) < TOL:
        break

    # Atualiza v e lambda_old
    v = w[:]
    lambda_old = lambda_new

# ------------------------------------------------------------
# RESULTADOS
# ------------------------------------------------------------
print(f"Autovalor aproximado próximo de mu={mu} : {lambda_new:.10f}")
print_vec("Autovetor aproximado =", v)

# Verificação: H v e lambda v devem ser quase iguais
Hv = matvec(H, v)
print("\nTeste Hv \approx  lambda v:")
for i in range(N):
    print(f"Hv[{i}] = {Hv[i]:.10f}   lambda*v[{i}] = {(lambda_new*v[i]):.10f}")
```

Discussão Final

O método da potência inversa com deslocamento é extremamente eficiente para localizar autovalores específicos, pois:

- o deslocamento μ "empurra" o problema para que o autovalor desejado se torne do-

minante na matriz inversa;

- cada iteração melhora a aproximação do autovetor associado;
- o quociente de Rayleigh fornece uma estimativa muito precisa do autovalor;
- mesmo usando um método simples para resolver sistemas lineares (Gauss básico), o algoritmo converge rapidamente.

Esse método é a base de algoritmos modernos como QR deslocado, métodos iterativos de Krylov e técnicas de diagonalização em mecânica quântica e métodos numéricos avançados.

26.2 Método da Potência Inversa com Deslocamento aplicado ao Modelo de Anderson 1D

O **método da potência inversa com deslocamento** é uma técnica iterativa que permite localizar autovalores interiores de uma matriz — isto é, autovalores que não são necessariamente os de maior módulo — ao escolher um *chute* μ próximo do autovalor desejado. Aplicado a matrizes tridiagonais (como as que surgem no Modelo de Anderson 1D), o custo de cada iteração pode ser mantido baixo usando um resolvedor dedicado a sistemas tridiagonais (o método de Thomas).

Ideia principal. Dada uma matriz simétrica $H \in \mathbb{R}^{N\times N}$ e um deslocamento μ, definimos

$$A_{\text{shifted}} = H - \mu I.$$

Se λ é um autovalor de H, então $\lambda - \mu$ é autovalor de A_{shifted} e $\frac{1}{\lambda-\mu}$ é autovalor de A_{shifted}^{-1}. Assim, o autovalor de H mais próximo de μ corresponde ao maior em módulo de A_{shifted}^{-1}. O esquema iterativo é então:

$$\text{(a)} \quad (H - \mu I)\, y_{k+1} = x_k \qquad \Rightarrow \qquad \text{(b)} \quad x_{k+1} = \frac{y_{k+1}}{\|y_{k+1}\|},$$

e a estimativa do autovalor na iteração k é obtida pelo quociente de Rayleigh

$$\lambda_k = x_k^\top H x_k.$$

Observações importantes sobre a implementação:

- Quando $H - \mu I$ é tridiagonal, resolvemos $(H - \mu I)y = x$ com o *algoritmo de Thomas* (eliminação direta adaptada a tridiagonais) — custo $O(N)$.
- Normalizamos y a cada iteração para evitar crescimento numérico e para obter o vetor unitário x.
- Critério de parada típico: $|\lambda_k - \lambda_{k-1}| < \text{tol}$ ou número máximo de iterações.
- Para robustez em produção é recomendável o pivoteamento parcial e o uso de bibliotecas numéricas estáveis (p.ex. LAPACK), mas aqui apresentamos uma versão educativa, simples e explícita.

Matriz do exemplo (Modelo de Anderson 1D):

$$H = \begin{bmatrix} 0.5 & -1 & 0 & 0 \\ -1 & -1.2 & -1 & 0 \\ 0 & -1 & 0.7 & -1 \\ 0 & 0 & -1 & -0.3 \end{bmatrix}, \qquad \mu = 1.7 \quad (\text{chute inicial}).$$

Implementação em Python (didática, sem dependências externas). Abaixo está o código mais básico possível — usa listas nativas, `math` para raiz quadrada, e funções explícitas para cada operação. Comentários detalham cada etapa.

```
# Potência Inversa com Deslocamento (exemplo tridiagonal - Modelo de Anderson 1D)
# Implementação didática em Python, sem dependências (lista + math).

import math

# ---------------------------
# Operações vetoriais básicas
# ---------------------------
def dot(u, v):
    """Produto interno entre vetores u e v."""
    s = 0.0
    for i in range(len(u)):
        s += u[i] * v[i]
    return s

def norm(v):
    """Norma euclidiana do vetor v."""
    return math.sqrt(dot(v, v))

def normalize(v):
    """Retorna uma cópia normalizada de v (norma 1)."""
    n = norm(v)
    if n == 0.0:
        raise ValueError("Tentativa de normalizar vetor nulo")
    return [x / n for x in v]

def matvec(H, v):
    """Multiplicação matriz-vetor H @ v (H é lista de listas)."""
    N = len(v)
    w = [0.0] * N
    for i in range(N):
        s = 0.0
        row = H[i]
        for j in range(N):
            s += row[j] * v[j]
        w[i] = s
    return w

# -----------------------------------------
# Método de Thomas (resolver Ax = d, tridiagonal)
# A é representada pelas três diagonais:
#   a: subdiagonal (length N-1)  -- A[i][i-1] for i=1..N-1
#   b: diagonal    (length N)    -- A[i][i]
#   c: superdiag   (length N-1)  -- A[i][i+1] for i=0..N-2
# d: rhs (length N)
#
# Retorna x solução (length N).
```

```
# Implementação assume que nenhum pivô será zero (sem pivoteamento).
# ------------------------------------------
def thomas(a, b, c, d):
    N = len(b)
    # cópias para não alterar entradas originais
    cp = [0.0] * (N-1)   # c'
    dp = [0.0] * N       # d'
    x = [0.0] * N

    # primeira linha (i=0)
    if b[0] == 0.0:
        raise ZeroDivisionError("Pivô zero na primeira linha do Thomas")
    cp[0] = c[0] / b[0]
    dp[0] = d[0] / b[0]

    # eliminação para cima (i = 1 .. N-2)
    for i in range(1, N-1):
        denom = b[i] - a[i-1] * cp[i-1]
        if denom == 0.0:
            raise ZeroDivisionError(f"Pivô zero na linha {i} do Thomas")
        cp[i] = c[i] / denom
        dp[i] = (d[i] - a[i-1] * dp[i-1]) / denom

    # última equação (i = N-1)
    denom = b[N-1] - a[N-2] * cp[N-2]
    if denom == 0.0:
        raise ZeroDivisionError("Pivô zero na última linha do Thomas")
    dp[N-1] = (d[N-1] - a[N-2] * dp[N-2]) / denom

    # substituição regressiva
    x[N-1] = dp[N-1]
    for i in range(N-2, -1, -1):
        x[i] = dp[i] - cp[i] * x[i+1]

    return x

# ------------------------------------------
# Dados do problema (matriz H tridiagonal armazenada como completa por clareza)
# ------------------------------------------
H = [
    [ 0.5, -1.0,  0.0,  0.0],
    [-1.0, -1.2, -1.0,  0.0],
    [ 0.0, -1.0,  0.7, -1.0],
    [ 0.0,  0.0, -1.0, -0.3]
]

N = 4
mu = 1.7   # chute
tol = 1e-8
max_it = 1000

# Construir as três diagonais de A = H - mu * I
a = [0.0] * (N-1)  # subdiagonal
b = [0.0] * N      # diagonal
c = [0.0] * (N-1)  # superdiagonal

for i in range(N):
    b[i] = H[i][i] - mu
    if i < N-1:
        c[i] = H[i][i+1]
```

```
    if i > 0:
        a[i-1] = H[i][i-1]

# vetor inicial (qualquer não-nulo; aqui escolhemos todos 1)
x = [1.0] * N
x = normalize(x)  # normaliza para estabilidade
lambda_old = 0.0
y = [0.0] * N

# Iterações da potência inversa com deslocamento
for it in range(1, max_it+1):
    # Resolve (H - mu I) y = x  (tridiagonal -> Thomas)
    y = thomas(a, b, c, x)

    # Normaliza y para obter novo x
    y_norm = norm(y)
    if y_norm == 0.0:
        raise ValueError("Vetor solução nulo -- verifique mu e a matriz")
    x = [val / y_norm for val in y]

    # Estimativa do autovalor via quociente de Rayleigh
    Hx = matvec(H, x)
    lambda_new = dot(x, Hx)  # como x é unitário, denom v^T v = 1

    # Critério de parada
    if abs(lambda_new - lambda_old) < tol:
        break

    lambda_old = lambda_new

# Impressão dos resultados
print("Resultado após", it, "iterações")
print(f"Autovalor aproximado próximo de mu={mu}: {lambda_new:.10f}")
print("Autovetor normalizado aproximado:")
for i, val in enumerate(x):
    print(f"  x[{i}] = {val:.10f}")

# Verificação: H x \approx lambda x
Hv = matvec(H, x)
print("\nVerificação Hv versus lambda*x:")
for i in range(N):
    print(f"  Hv[{i}] = {Hv[i]:.10f}   lambda*x[{i}] = {(lambda_new*x[i]):.10f}")
```

Explicação detalhada do código e do método (passo a passo):

1. **Preparação das diagonais.** A matriz $A = H - \mu I$ é tridiagonal; portanto guardamos apenas as três diagonais: a (subdiagonal), b (diagonal principal) e c (superdiagonal). Isso reduz armazenamento e operações.

2. **Vetor inicial.** Escolhemos $x^{(0)}$ não nulo (aqui todos 1) e o normalizamos para estabilidade numérica.

3. **Iteração principal.**

   - Resolva $Ay = x$ com Thomas (complexidade $O(N)$). A solução y corresponde a aplicar implicitamente A^{-1} em x.
   - Normalize y para obter o próximo x; dessa forma impedimos crescimento/decrescimento de norma.

- Calcule o quociente de Rayleigh $\lambda = x^\top Hx$. Se x estiver normalizado, não é necessário dividir por $x^\top x$.
- Verifique convergência: se $|\lambda - \lambda_{\text{ant}}| < \text{tol}$ pare.

4. **Resultado.** Ao convergir, x aproxima o autovetor associado ao autovalor de H mais próximo de μ; λ aproxima tal autovalor.

5. **Verificação final.** O teste $Hx \approx \lambda x$ confirma numericamente a qualidade da solução.

Em aplicações práticas, é importante lembrar que o método de Thomas somente funciona adequadamente quando o sistema tridiagonal deslocado permanece não singular — portanto, escolher μ exatamente igual a um autovalor torna $H - \mu I$ singular e leva à quebra do algoritmo. Além disso, em implementações reais recomenda-se utilizar bibliotecas robustas, como `numpy.linalg` ou `scipy.linalg.solve_banded`, que incluem estratégias de pivoteamento e tratam melhor problemas de estabilidade numérica. Finalmente, quando o espectro possui autovalores muito próximos entre si, a convergência do método pode se tornar lenta; nesses casos, é comum empregar variantes mais eficientes, como a iteração do quociente de Rayleigh, técnicas de aceleração ou métodos baseados em subespaços de Krylov, que melhoram significativamente o desempenho.

27 Cálculo Numérico de Autovalores e Autovetores em Python: Métodos Otimizados para Matrizes Simétricas, Tridiagonais e Esparsas

Como já debatemos anteriormente, em aplicações práticas de física computacional e mecânica estatística, é comum trabalhar com matrizes simétricas e, em muitos casos, com matrizes tridiagonais ou esparsas, como nos modelos tight-binding de Anderson, matrizes laplacianas ou hessianas discretizadas. O Python fornece um ecossistema altamente otimizado para o cálculo de autovalores e autovetores nesse contexto, principalmente através das bibliotecas `NumPy` e `SciPy`. Ou seja, existem funções já prontas dentro do python que fazem os cálculos de forma razoavelmente bem otimizados. Nesta seção apresentamos, de forma detalhada, as funções mais importantes para lidar com esses problemas, incluindo exemplos simples e diretos de uso. Todas as rotinas apresentadas são implementações eficientes de métodos clássicos, como decomposições espectrais especializadas e algoritmos iterativos do tipo ARPACK.

27.1 Autovalores e Autovetores de Matrizes Simétricas Densas com `numpy.linalg.eigh`

Quando a matriz é real e simétrica ($A = A^\top$), a forma mais recomendada de obter o espectro é via a função `numpy.linalg.eigh`. Essa função utiliza algoritmos específicos para matrizes Hermitianas/simétricas, garantindo maior estabilidade numérica, eficiência computacional e precisão em comparação ao método geral `eig`. Além disso, garante que os autovalores serão retornados em ordem crescente.

Exemplo básico:

```
import numpy as np
```

```
A = np.array([
    [ 2.0, -1.0,  0.0],
    [-1.0,  2.0, -1.0],
    [ 0.0, -1.0,  2.0]
])

# Autovalores e autovetores
w, v = np.linalg.eigh(A)

print("Autovalores:")
print(w)
print("\nAutovetores (colunas de v):")
print(v)
```

Comentários importantes:

- `eigh` é preferível a `eig` sempre que a matriz for simétrica.
- Os autovetores são retornados em colunas, alinhados com os autovalores correspondentes.
- A complexidade do método é aproximadamente $O(N^3)$, adequada para matrizes densas e tamanhos moderados ($N \sim 10^3$).

27.2 Matrizes Tridiagonais Simétricas: `scipy.linalg.eigh_tridiagonal`

O python também tem funções específicas para problemas na linha do modelo de Anderson 1D onde as matrizes aparecem naturalmente na forma tridiagonal :

$$T = \begin{pmatrix} d_1 & e_1 & 0 & \cdots & 0 \\ e_1 & d_2 & e_2 & \ddots & \vdots \\ 0 & e_2 & d_3 & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & e_{n-1} \\ 0 & \cdots & 0 & e_{n-1} & d_n \end{pmatrix}.$$

É desnecessário construir a matriz completa; basta fornecer a diagonal principal d e a diagonal secundária e. A rotina `scipy.linalg.eigh_tridiagonal` é especializada nesse caso e costuma ser extremamente rápida.

Exemplo básico:

```
import numpy as np
from scipy.linalg import eigh_tridiagonal

# Diagonal principal d e diagonal secundária e
d = np.array([2.0, 2.0, 2.0, 2.0])
e = np.array([-1.0, -1.0, -1.0])

# Autovalores e autovetores
w, v = eigh_tridiagonal(d, e)

print("Autovalores:")
print(w)
print("\nAutovetores (colunas):")
print(v)
```

Pontos relevantes:

- Não é necessário criar uma matriz $N \times N$, economizando memória.
- Métodos tridiagonais possuem complexidade reduzida ($O(N^2)$ ou melhor).
- É possível calcular apenas parte do espectro, usando o parâmetro `select`.

27.3 Matrizes Esparsas: Métodos Iterativos com `scipy.sparse.linalg.eigsh`

Para matrizes grandes e esparsas (tipicamente com dezenas ou centenas de milhares de graus de liberdade), algoritmos diretos tornam-se inviáveis. Para esse tipo de problema, utiliza-se métodos iterativos baseados em Krylov, como ARPACK, implementados em `scipy.sparse` Essa função é destinada a matrizes simétricas esparsas e permite extrair apenas alguns autovalores relevantes (maiores, menores ou próximos de um valor especificado via deslocamento).

Exemplo básico — maiores autovalores:

```
import numpy as np
from scipy.sparse import diags
from scipy.sparse.linalg import eigsh

# Construção de uma matriz laplaciana 1D esparsa
N = 100
d = 2 * np.ones(N)
e = -1 * np.ones(N-1)
A = diags([e, d, e], offsets=[-1, 0, 1], format='csr')

# 3 maiores autovalores
w, v = eigsh(A, k=3, which='LM')

print("Autovalores maiores:")
print(w)
```

Exemplo — cálculo de autovalores próximos de um deslocamento μ

Para matrizes grandes e esparsas, é comum que nos interessemos não pelos maiores autovalores, mas por aqueles situados *próximos de um certo valor μ*. Isso ocorre, por exemplo, em problemas de física do estado sólido, dinâmica de redes ou métodos iterativos de convergência acelerada. O método tradicional de potência não é eficiente para localizar autovalores em torno de μ, pois ele sempre converge para o autovalor dominante. Para contornar esse problema, utiliza-se o chamado **shift-and-invert**, implementado na função `eigsh` via o parâmetro `sigma=mu`. A ideia é simples: em vez de diagonalizar diretamente A, o método aplica a iteração ao operador

$$B = (A - \mu I)^{-1}.$$

Os autovalores de B são

$$\lambda_i(B) = \frac{1}{\lambda_i(A) - \mu},$$

ou seja, os autovalores de A mais próximos de μ tornam-se os de maior módulo em B. Isso transforma o problema em um caso tratável pelo método da potência inversa, mas

sem necessidade de calcular explicitamente B: o algoritmo apenas resolve sistemas lineares envolvendo $A - \mu I$. Esse truque é extraordinariamente eficiente quando queremos poucos autovalores de uma matriz de dimensão muito grande. A seguir, mostramos um exemplo completo utilizando uma matriz 10×10 aleatória apenas para fins didáticos.

```
import numpy as np
from scipy.sparse.linalg import eigsh
from scipy.sparse import csr_matrix

# Criamos uma matriz simétrica 10x10 (necessária para usar eigsh)
np.random.seed(1)
A = np.random.randn(10,10)
A = (A + A.T)/2   # torna simétrica

A_sparse = csr_matrix(A)

mu = 1.5  # deslocamento desejado

# Calcula k=3 autovalores mais próximos de mu usando shift-and-invert
w, v = eigsh(A_sparse, k=3, sigma=mu)

print("Autovalores de A próximos de mu = 1.5:")
print(w)

print("\nAutovetor correspondente ao autovalor mais próximo:")
print(v[:,0])
```

Comentários detalhados:

- O parâmetro `sigma=mu` ativa o modo *shift-and-invert*. Em vez de diagonalizar A, o método opera com $(A - \mu I)^{-1}$, de forma que autovalores próximos de μ são realçados.

- Não é necessário calcular a matriz inversa explicitamente. O algoritmo apenas resolve sistemas lineares do tipo $(A - \mu I)x = b$, o que é muito mais estável e eficiente.

- Para matrizes grandes, isso é crucial: métodos diretos custariam $O(N^3)$, enquanto resoluções iterativas de sistemas esparsos podem ser quase lineares.

- O argumento `k=3` retorna apenas três autovalores mais próximos de μ, o que é a essência dos métodos de autovalores modernos: obter *somente* o que é necessário.

- A função `eigsh` exige que a matriz seja simétrica (ou Hermitiana). O exemplo garante isso com `A = (A + A.T)/2`.

- O array `v` contém os autovetores coluna por coluna. Assim, `v[:,0]` é o autovetor associado ao autovalor `w[0]`.

Esse procedimento é um dos mais importantes em computação científica moderna, especialmente em problemas com matrizes esparsas gigantes — como métodos de Krylov, resolução de equações diferenciais, teoria quântica de bandas e análise espectral em redes complexas.

27.4 Recomendações Práticas

- Prefira sempre `eigh` para matrizes simétricas densas, pois é mais rápido e estável.
- Para matrizes tridiagonais, utilize `eigh_tridiagonal` — ela evita a montagem da matriz completa e explora a estrutura para obter grande desempenho.
- Para matrizes grandes e esparsas, `eigsh` é a ferramenta indicada, permitindo calcular somente os autovalores necessários.
- Em modelos físicos onde se deseja autovalores internos (perto de μ), use o argumento `sigma`.
- Em situações de instabilidade numérica ou matrizes mal-condicionadas, considere aumentar a tolerância dos métodos iterativos ou utilizar pré-condicionadores.

Em resumo, o cálculo de espectros no Python pode ser feito de forma altamente eficiente escolhendo a rotina adequada para cada estrutura matricial: `numpy.linalg.eigh` é ideal para matrizes simétricas densas, oferecendo simplicidade e robustez; `scipy.linalg.eigh_tridiagonal` explora a estrutura tridiagonal para alcançar desempenho superior; e `scipy.sparse.linalg.eigsh` permite lidar com matrizes esparsas de grande porte, extraindo apenas parte do espectro quando necessário. Com essas ferramentas, a análise espectral em modelos físicos — desde cadeias tight-binding até discretizações de operadores diferenciais — torna-se direta, eficiente e numericamente estável.

28 Conclusão: Fundamentos e Perspectivas em Física Computacional

O conteúdo destas notas estabelece uma **base metodológica sólida** e indispensável para a prática da Física Computacional. A jornada percorrida demonstrou como a linguagem Python e as bibliotecas científicas centrais (NumPy, Matplotlib e SciPy) constituem o ambiente essencial de modelagem, permitindo a transição rigorosa da formulação teórica para a quantificação e visualização. Abordamos desde o controle fundamental dos erros de arredondamento e truncamento, essenciais para a validade dos resultados, até a aplicação de técnicas avançadas. A rigorosa aplicação dos métodos de Runge-Kutta para a integração de Equações Diferenciais Ordinárias (EDOs) e a eficiência dos métodos espectrais para a análise de autovalores e autovetores equipam o leitor para a resolução de sistemas físicos dinâmicos e lineares. O domínio destas ferramentas permite abordar problemas de fronteira que, frequentemente, não são susceptíveis à solução analítica fechada, incluindo sistemas com alta não-linearidade, o comportamento de materiais em escala atômica ou simulações estatísticas complexas.

A principal virtude destes métodos reside na sua **flexibilidade e transferibilidade**. Os princípios de discretização espacial e temporal aqui desenvolvidos são escaláveis e diretamente aplicáveis a diversos campos, desde a adaptação de integradores a problemas de evolução quântica (como a equação de Schrödinger dependente do tempo) até o emprego dos Algoritmos de Monte Carlo, que servem de base para a termodinâmica de sistemas de muitos corpos. Encorajamos o leitor a reconhecer a **estrutura algorítmica universal** subjacente, utilizando o código de referência destas notas para iniciar investigações mais

aprofundadas. As próximas etapas podem incluir a implementação da Dinâmica Molecular para estudar transições de fase, a aplicação de Métodos de Diferenças Finitas em Equações Diferenciais Parciais (EDPs) ou o uso da Transformada Rápida de Fourier (FFT) na análise de dados e fenômenos ondulatórios. O código atua como uma **expressão formalizada da intuição física**, e o domínio da precisão numérica é crucial para a interpretabilidade dos resultados. A capacidade de modelar e simular é um pilar da pesquisa contemporânea. Esperamos que este material sustente o desenvolvimento contínuo do leitor como um profissional capaz de contribuir com rigor e discernimento para a compreensão quantitativa da natureza.