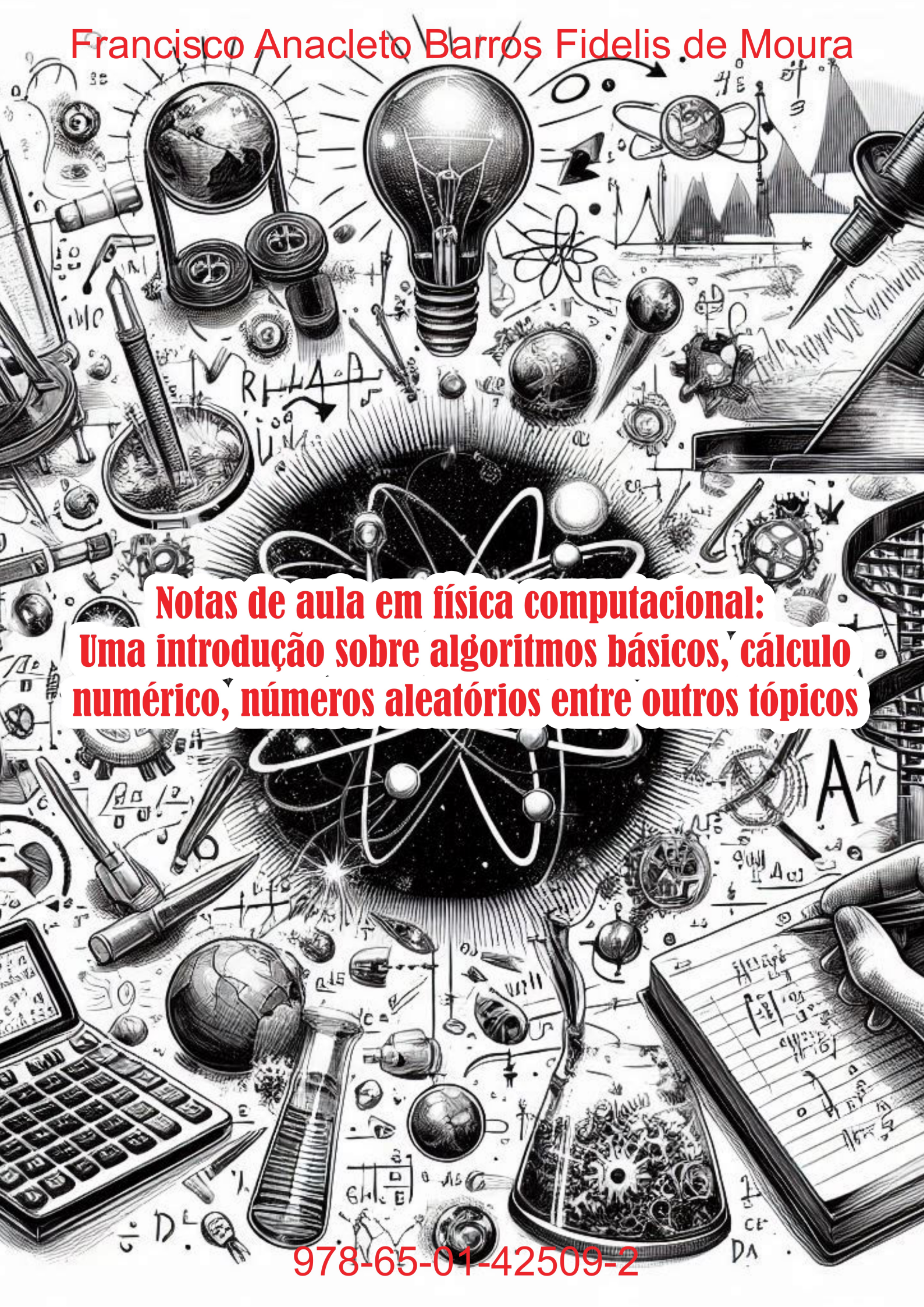
Francisco Anacleto Barros Fidelis de Moura



Notas de aula em física computacional: Uma introdução sobre algoritmos básicos, cálculo numérico, números aleatórios entre outros tópicos

978-65-01-42509-2

Notas de Aula em Física Computacional: Uma introdução sobre Algoritmos Básicos, Cálculo Numérico, Números Aleatórios entre outros tópicos

Francisco Anacleto Barros Fidelis de Moura

Prefácio

Neste eBook, convido você a embarcar em uma jornada de descobertas pelo fascinante universo dos algoritmos e do cálculo numérico. Em um mundo cada vez mais movido pela computação, compreender esses conceitos é essencial para qualquer estudante, pesquisador ou profissional das áreas de ciência, tecnologia, engenharia e matemática.

A primeira parte deste livro é dedicada aos algoritmos fundamentais. Começaremos com os princípios básicos da lógica computacional e da programação estruturada, avançando gradualmente para técnicas mais elaboradas de resolução de problemas. Os algoritmos são as ferramentas que usamos para dar vida às ideias — soluções eficientes e organizadas que tornam possível resolver desde simples operações matemáticas até problemas complexos do mundo real.

Na sequência, exploraremos o campo do cálculo numérico, uma área vital da matemática aplicada que lida com métodos para encontrar soluções aproximadas para problemas que não podem ser resolvidos de forma analítica. Abordaremos estratégias para calcular derivadas e integrais, bem como resolver equações e sistemas lineares com precisão numérica controlada. Esses métodos são amplamente utilizados na modelagem de fenômenos físicos, econômicos, biológicos e em diversas outras aplicações práticas.

Dedicaremos também uma seção especial aos números aleatórios, elementos indispensáveis em simulações computacionais e métodos estatísticos modernos. Vamos discutir como geradores de números aleatórios funcionam, como avaliá-los, e como aplicá-los em técnicas como as simulações de Monte Carlo — uma poderosa abordagem usada para resolver problemas que envolvem incerteza ou variabilidade.

Mais do que transmitir conteúdo, este eBook busca despertar a curiosidade e inspirar a criatividade. Através de uma abordagem prática, com exemplos comentados e explicações acessíveis, você será convidado a compreender os conceitos em profundidade e aplicá-los de forma confiante em suas próprias experiências acadêmicas e profissionais. Ao final desta leitura, espero que você esteja não apenas mais preparado, mas também mais motivado a explorar os fundamentos da computação e do cálculo numérico, levando consigo uma base sólida e o entusiasmo necessário para continuar aprendendo.

Agradecimentos

Expresso minha mais profunda gratidão a todos que contribuíram para a realização deste trabalho. Em especial, agradeço à minha esposa Martha, ao meu filho Miguel e aos meus pais, Sr. Fidelis e Dona Daia, pelo apoio e incentivo constantes. Meus sinceros agradecimentos também aos meus alunos e alunas, cuja participação foi essencial na construção deste curso. Sou igualmente grato ao Instituto de Física da UFAL, bem como aos meus colegas, amigos e colaboradores, que sempre me inspiram a aprimorar a qualidade do nosso trabalho diário.

Contents

0.1 Introdução à Computação Científica e ao Cálculo Numérico . . . 7
0.1.1 Algoritmos: Fundamentos e Aplicações . . . 7
0.1.2 Cálculo Numérico: Conceitos e Ferramentas . . . 7
0.1.3 Interpolação: conectando os pontos . . . 8
0.1.4 Regressão Linear: tendências nos dados . . . 9
0.1.5 Mapas caóticos, sistemas dinâmicos e o caminhante aleatório . . . 9
0.1.6 Geração de Números Aleatórios e Simulações . . . 9
0.1.7 Modelagem com Desordem Correlacionada . . . 9
0.1.8 Soluções Numéricas de Equações e Sistemas . . . 10
0.2 Conceitos Básicos de Programação em C . . . 12
0.2.1 Variáveis e Tipos de Dados . . . 12
0.2.2 Entrada e Saída de Dados . . . 15
0.2.3 Escrita de dados em Arquivo de Saída em C . . . 17
0.2.4 Operações Matemáticas Básicas . . . 19
0.2.5 Estruturas Condicionais (if, else, else if) . . . 20
0.2.6 Laços de Repetição (for e while) . . . 21
0.2.7 Funções . . . 21
0.2.8 Arrays e Manipulação de Coleções . . . 21
0.2.9 Manipulação de Strings . . . 22
0.2.10 Estruturas de Dados Simples (Structs) . . . 23
0.2.11 Tratamento de Erros em C (validação de entrada) . . . 23
0.2.12 Números Pseudo-Aleatórios em C . . . 24
0.3 Aprofundando os Conceitos de C com Novos Exemplos . . . 28
0.3.1 Mais Operações Básicas com Dois Números . . . 28
0.3.2 Explicando a importância dos `Loops` . . . 30
0.3.3 Mais exemplos sobre manipulação de Vetores . . . 32
0.3.4 Mais informações sobre o uso de Condicionais em C . . . 33
0.3.5 Ordenando três valores usando comandos `if` em C . . . 34
0.3.6 Exemplo: Ordenando quatro valores usando comandos `if` em C . . . 35
0.3.7 Ordenando um vetor e mantendo a correspondência com outro vetor em C . . . 36
0.3.8 Exemplo de Programa em C para Cálculo de Funções Trigonométricas . . . 37
0.3.9 Cálculo do Histograma Normalizado em Linguagem C . . . 40
0.3.10 Geração de Números Aleatórios com Distribuição Gaussiana e Análise de Histograma . . . 41
0.3.11 Geração de Números Aleatórios com Distribuição em Lei de Potência . . . 43
0.4 Derivada Numérica . . . 46
0.4.1 Definição Matemática . . . 46
0.4.2 Derivada Numérica por Diferenças Finitas . . . 46
0.4.3 Exemplo 1: Derivada de $f(x) = x^2$ com Diferença Central . . . 46
0.4.4 Exemplo 2: Derivada Numérica de Vetores $x[i], y[i]$ . . . 47

0.4.5 Exemplo 3: Derivada Numérica de Dados em Arquivo 47
0.5 Integração Numérica: Aproximação de Integrais com Somatórios em C 49
0.5.1 Exemplo com função analítica: $f(x) = e^x$ no intervalo $[0, 1]$ 50
0.5.2 Exemplo com dados tabulados salvos em arquivo de dados 51
0.6 Interpolação 53
0.6.1 Interpolação Simples 54
0.6.2 Interpolação Polinomial Simples (Fórmula de Lagrange) 56
0.6.3 Interpolação por Splines 57
0.6.4 Interpolação por Spline Cúbico Natural com Número Arbitrário de Pontos . 60
0.7 Regressão Linear 63
0.8 Mapas Caóticos e Dinâmica Não Linear 67
0.8.1 O Mapa Logístico 67
0.8.2 O Mapa de Verhulst 69
0.9 Caminhante Aleatório em Uma Dimensão 70
0.9.1 Descrição do Modelo 71
0.9.2 Importância do Modelo 71
0.10 Estatística Básica: Cálculo de Médias, Variância, Skewness e Curtose 73
0.10.1 Medidas estatísticas em uma série aleatória 75
0.10.2 Normalização de uma série de dados a partir de um arquivo 76
0.10.3 Cálculo da Autocorrelação 78
0.11 Desordem Correlacionada 80
0.11.1 Definição e Função de Correlação 80
0.11.2 Modelo AR(1) (Auto-Regressivo de Primeira Ordem) 80
0.11.3 Modelo AR(2) (Auto-Regressivo de Segunda Ordem) 81
0.11.4 Desordem com Correlação Exponencial: $C(r) \sim e^{-r/L}$ 82
0.11.5 Desordem com Correlação Exponencial via Decomposição de Cholesky . . . 84
0.11.6 Comparação entre Séries AR(1) e Desordem Correlacionada via Decomposição de Cholesky 86
0.11.7 Desordem correlacionada com densidade espectral do tipo lei de potência . . 89
0.12 Integração de Monte Carlo 92
0.12.1 Exemplo: Integração de $f(x) = x^2$ em $[0, 2]$ 92
0.12.2 Método da Amostragem por Rejeição 93
0.13 Transformada de Fourier: Fundamentos Teóricos e Aplicações 95
0.13.1 Definição Matemática 95
0.13.2 Transformada de Fourier em Dados Discretos 95
0.14 Introdução à Solução Numérica de Sistemas Lineares 98
0.14.1 Método de Eliminação de Gauss 98
0.14.2 Método de Gauss-Seidel 100
0.15 Método de Newton-Raphson para Sistemas de Equações Não Lineares 104
0.16 Resolvendo Numericamente uma Equação Integral 107
0.16.1 Equações Integrais: Um Exemplo com Transferência Radiativa 108
0.17 Resolução Numérica de Equações Transcendentais: Um Exemplo com o Método da Bisseção 110
0.17.1 O Método da Bisseção 110
0.17.2 Resolvendo a equação $xe^x = 2$ com o método de Newton-Raphson 112
0.18 Considerações Finais 115

0.1 Introdução à Computação Científica e ao Cálculo Numérico

A computação moderna é alicerçada na aplicação de algoritmos, que consistem em sequências finitas de instruções bem definidas, projetadas para resolver problemas específicos. Esses algoritmos estão presentes em praticamente todas as áreas do conhecimento, desde tarefas simples como a ordenação de listas até simulações complexas de sistemas físicos, desempenhando um papel essencial na automação e eficiência do tratamento de dados. A capacidade de traduzir problemas do mundo real em modelos computacionais resolvíveis tornou-se um diferencial nas ciências exatas, engenharias, finanças e em diversas áreas interdisciplinares.

Nesta seção, exploramos os fundamentos da computação científica, incluindo algoritmos, cálculo numérico, geração de números aleatórios e simulações baseadas em métodos estatísticos. Além disso, abordamos técnicas para a solução de sistemas lineares e não lineares, equações transcendentais e integrais, bem como modelos com desordem correlacionada — todos fundamentais para a modelagem precisa de fenômenos complexos.

0.1.1 Algoritmos: Fundamentos e Aplicações

Algoritmos são conjuntos organizados de instruções que definem passo a passo como um problema deve ser resolvido. A clareza, correção e eficiência são características cruciais de um bom algoritmo. A análise de complexidade algorítmica, que avalia o consumo de tempo e espaço, é fundamental na escolha de algoritmos para aplicações práticas.

Entre os algoritmos mais utilizados, destacam-se:

- **Ordenação**: Algoritmos como *Bubble Sort*, *Merge Sort* e *Quick Sort* organizam dados com diferentes graus de eficiência. O *Quick Sort*, por exemplo, é amplamente preferido devido à sua complexidade média $O(n \log n)$.
- **Busca**: Métodos como busca linear e busca binária são aplicados para localizar elementos em conjuntos de dados. A busca binária, com complexidade $O(\log n)$, é particularmente eficaz em listas ordenadas.

0.1.2 Cálculo Numérico: Conceitos e Ferramentas

O cálculo numérico é uma ferramenta essencial para a resolução de problemas matemáticos cujas soluções analíticas são desconhecidas ou inviáveis. Ele fornece métodos para obter aproximações numéricas com precisão controlada.

Derivação Numérica

A derivada representa a taxa de variação de uma função, sendo fundamental para análises de crescimento, estabilidade e otimização. Quando a função não possui expressão analítica, utilizamos aproximações:

- **Diferença Progressiva**: $f'(x) \approx \dfrac{f(x+h) - f(x)}{h}$
- **Diferença Retrospectiva**: $f'(x) \approx \dfrac{f(x) - f(x-h)}{h}$
- **Diferença Central**: $f'(x) \approx \dfrac{f(x+h) - f(x-h)}{2h}$, método com erro de ordem superior.

Integração Numérica

A integral de uma função representa, geometricamente, a área sob sua curva, e desempenha um papel fundamental em diversas áreas da ciência, como física, estatística e engenharia. No entanto, nem sempre é possível calcular uma integral de forma exata, especialmente quando lidamos com funções complexas, sem forma analítica conhecida, ou com dados experimentais. Nesses casos, recorremos à integração numérica — um conjunto de técnicas que aproximam o valor da integral com base em avaliações da função em pontos discretos.

Entre os métodos mais utilizados, destacam-se:

- **Regra do Trapézio**: Este método aproxima a área sob a curva dividindo o intervalo de integração em subintervalos e estimando a área em cada um como a de um trapézio. É uma técnica simples, eficiente e bastante útil para funções suaves, onde pequenas subdivisões já garantem bons resultados.
- **Regra de Simpson**: Utiliza parábolas, em vez de segmentos de reta, para aproximar a função em cada subintervalo. Como resultado, alcança maior precisão, especialmente quando a função tem curvatura significativa. A aplicação da Regra de Simpson exige que o número de subdivisões seja par, mas compensa essa limitação com maior eficiência e acurácia em muitos casos práticos.

Esses métodos não apenas facilitam o cálculo de integrais difíceis, como também servem de base para técnicas mais avançadas de simulação e modelagem, como os métodos de Monte Carlo e as equações diferenciais resolvidas numericamente. Ao longo deste material, exploraremos como essas ferramentas se encaixam em problemas do mundo real e como podem ser implementadas computacionalmente de forma robusta.

0.1.3 Interpolação: conectando os pontos

Imagine que você tenha uma série de pontos obtidos experimentalmente — por exemplo, valores de temperatura ao longo do dia ou medições de uma função difícil de descrever analiticamente. A interpolação surge como uma ferramenta poderosa para preencher as lacunas entre esses pontos. Ela permite que a gente "costure" os dados, criando uma função suave que passa exatamente por eles, como se estivéssemos desenhando uma curva à mão ligando cada valor observado.

Mais do que apenas desenhar linhas, a interpolação tem aplicações práticas muito sérias: desde o controle de trajetórias em sistemas robóticos até a reconstrução de imagens em gráficos computacionais. A ideia central é simples: se você conhece o comportamento de uma função em alguns pontos, pode estimar como ela se comporta entre esses pontos com certo grau de confiança.

Existem diferentes formas de se fazer interpolação, e cada uma tem suas vantagens. Algumas são mais simples e diretas, como a interpolação linear. Outras, como a interpolação polinomial ou por splines, oferecem curvas mais suaves, ideais para quando os dados mudam de forma gradual ou exigem uma aproximação mais refinada.

Vale lembrar que interpolar não é o mesmo que extrapolar — enquanto a interpolação trabalha dentro da faixa dos dados conhecidos, a extrapolação tenta prever o que vem depois (ou antes), o que pode ser bem mais arriscado e sujeito a erros maiores.

Neste "ebook", vamos ver como aplicar técnicas de interpolação na prática e entender como escolher o método mais adequado para cada situação.

0.1.4 Regressão Linear: tendências nos dados

Enquanto a interpolação busca construir uma função que passa exatamente pelos pontos de dados, a regressão linear tem uma missão diferente: encontrar uma tendência geral em meio aos dados,

mesmo que eles não se alinhem perfeitamente. Ela é, essencialmente, uma técnica de ajuste — uma forma de resumir o comportamento de um conjunto de dados com uma equação simples, geralmente uma reta.

Pense, por exemplo, na relação entre a altura e o peso de um grupo de pessoas. Os dados reais podem estar dispersos, mas é possível que exista uma tendência geral: pessoas mais altas tendem a pesar mais. A regressão linear tenta capturar essa tendência, ajustando uma reta que "melhor representa" todos os pontos, mesmo que nenhum deles esteja exatamente sobre essa linha.

A ideia central da regressão linear é minimizar os erros — mais precisamente, as diferenças entre os valores observados e os valores previstos pela reta ajustada. Esse ajuste é feito de forma a minimizar a soma dos quadrados desses erros, num processo conhecido como "mínimos quadrados".

Esse tipo de análise é amplamente utilizado em diversas áreas. Na física, pode servir para modelar comportamentos aproximadamente lineares, como a expansão térmica de materiais. Em economia, ajuda a entender correlações entre variáveis, como o impacto do consumo sobre o crescimento. E na ciência de dados, é um dos modelos mais usados para previsões simples.

Apesar da simplicidade, a regressão linear pode ser muito poderosa, principalmente como ponto de partida. Ela também abre as portas para modelos mais complexos, como a regressão múltipla (com várias variáveis) e modelos não lineares, que serão abordados mais adiante neste material.

0.1.5 Mapas caóticos, sistemas dinâmicos e o caminhante aleatório

Sistemas dinâmicos descrevem como estados evoluem no tempo segundo regras bem definidas. Mesmo regras simples podem gerar comportamentos extremamente complexos, como ocorre nos mapas caóticos. O famoso mapa logístico, por exemplo, mostra como a ordem pode dar lugar ao caos com pequenas mudanças em um parâmetro. Já o caminhante aleatório representa um processo estocástico, em que cada passo é decidido por sorte — como uma partícula se movendo ao acaso. Enquanto sistemas caóticos são determinísticos, mas imprevisíveis a longo prazo, os caminhantes aleatórios são imprevisíveis por natureza. Ambos os conceitos são fundamentais para entender fenômenos como difusão, turbulência e flutuações em sistemas físicos e biológicos.

0.1.6 Geração de Números Aleatórios e Simulações

A geração de números aleatórios é fundamental em diversas simulações, incluindo métodos estatísticos, como os de Monte Carlo, usados para estimar quantidades físicas ou probabilísticas.

- **Geradores Pseudo-Aleatórios (PRNGs)**: Como o *Linear Congruential Generator*, geram sequências determinísticas que simulam aleatoriedade.

- **Método de Monte Carlo**: Utiliza amostragem estatística para resolver problemas complexos, como o cálculo de integrais multidimensionais, simulações térmicas e modelagem de incertezas.

0.1.7 Modelagem com Desordem Correlacionada

A introdução de desordem em modelos físicos é essencial para a descrição de materiais reais, onde imperfeições e flutuações ocorrem naturalmente. A desordem correlacionada difere da desordem puramente aleatória (ruído branco) por conter dependências espaciais ou temporais.

- **Desordem com Correlação de Curto Alcance**: Pode ser modelada por processos autoregressivos (AR), onde cada ponto depende de seus vizinhos imediatos.

- **Desordem com Correlação de Longo Alcance**: Gerada, por exemplo, via filtragem espectral (método de Fourier inversa), permitindo simular espectros com leis de potência ou distribuições Lorentzianas.
- **Aplicações**: A desordem correlacionada é relevante em transportes eletrônicos (modelo de Anderson modificado), dinâmica de fluidos, propagação de ondas em meios complexos, e materiais desordenados.

0.1.8 Soluções Numéricas de Equações e Sistemas

Sistemas Lineares

Sistemas lineares da forma $A\vec{x} = \vec{b}$ surgem em diversas aplicações. Métodos diretos e iterativos são empregados para resolvê-los:

- **Método de Eliminação de Gauss**: Estratégia clássica para resolver sistemas lineares, eliminando variáveis sequencialmente.
- **Métodos Iterativos**: Como Jacobi, Gauss-Seidel e Gradiente Conjugado, são úteis para matrizes esparsas e de grande porte.

Sistemas Não Lineares

Equações do tipo $f(x) = 0$ aparecem frequentemente em problemas de engenharia e física. Alguns métodos clássicos incluem:

- **Método da Bisseção**: Requer intervalo com mudança de sinal e converge lentamente, mas de forma garantida.
- **Método de Newton-Raphson**: Utiliza a derivada da função para obter convergência quadrática, mas requer uma boa estimativa inicial.

Equações Integrais e Transcendentais

Muitos problemas físicos, como os que surgem em difusão, eletrostática e mecânica estatística, conduzem naturalmente à formulação de equações que não admitem solução analítica direta. Entre essas, destacam-se as **equações integrais**, que envolvem uma função desconhecida que aparece dentro de uma integral. Tais equações são comuns em problemas físicos com interação não local, como a equação de Fredholm. Para resolvê-las numericamente, utilizamos métodos como a quadratura, que substituem a integral por uma soma ponderada e reduzem o problema a um sistema linear. Também encontramos as **equações transcendentais**, que contêm funções não algébricas, como exponenciais, logaritmos, senos, cossenos, entre outras. Como essas equações não podem ser resolvidas por métodos algébricos tradicionais, recorremos a métodos numéricos iterativos para encontrar suas raízes. Neste capítulo, veremos como essas equações podem ser tratadas numericamente de forma eficiente, com foco em algoritmos simples e aplicáveis a uma ampla gama de problemas reais.

A compreensão aprofundada dos algoritmos, métodos numéricos e técnicas de simulação é fundamental para o avanço da computação científica moderna. Esses conhecimentos não apenas capacitam pesquisadores e engenheiros a resolverem problemas complexos, mas também fornecem as ferramentas necessárias para realizar aproximações precisas e eficientes em uma variedade de áreas científicas e tecnológicas. No contexto da física computacional, engenharia e ciências aplicadas, os métodos numéricos permitem que soluções aproximadas sejam encontradas para problemas que, de outra

forma, seriam intratáveis analiticamente. Ao longo deste eBook, exploraremos detalhadamente os conceitos, abordagens e algoritmos mais importantes nesses campos, oferecendo uma análise prática e exemplos aplicados que ilustram sua relevância no mundo real. Através de uma abordagem teórica e prática equilibrada, este material visa fornecer uma base sólida que possibilite a aplicação desses métodos em contextos tão diversos quanto modelagem física, análise de sistemas dinâmicos, engenharia de processos e simulações computacionais. Assim, os leitores estarão mais preparados para aplicar essas técnicas de forma eficaz em seus próprios projetos e pesquisas, aprimorando suas habilidades e contribuindo para o avanço do conhecimento nas áreas de computação científica e suas aplicações multidisciplinares.

0.2 Conceitos Básicos de Programação em C

Nesta seção, daremos os primeiros passos no universo da programação utilizando a linguagem C, uma das linguagens mais influentes e amplamente utilizadas na história da computação. Desenvolvida nos anos 1970, C continua sendo uma escolha sólida para quem deseja compreender os fundamentos da programação e da estrutura interna dos computadores. Sua sintaxe direta e seu desempenho eficiente fazem dela uma excelente linguagem para iniciantes, além de ser muito empregada em sistemas embarcados, jogos, simulações científicas e softwares de alto desempenho.

Nosso objetivo aqui é apresentar os conceitos fundamentais de forma clara, didática e progressiva, oferecendo uma base sólida para que você possa escrever seus próprios programas com segurança e autonomia. A linguagem C é estruturada, o que significa que ela permite ao programador organizar o código em blocos lógicos e reutilizáveis, facilitando o entendimento e a manutenção dos programas.

Você aprenderá a declarar e manipular variáveis, trabalhar com diferentes tipos de dados (inteiros, reais, caracteres, etc.), realizar operações matemáticas e lógicas, e interagir com o usuário por meio de entrada e saída de dados. Também exploraremos estruturas de controle de fluxo, como as instruções condicionais (`if`, `else`) e os laços de repetição (`while`, `for`), que são essenciais para implementar tomadas de decisão e repetições automáticas em programas.

Cada novo conceito será acompanhado de exemplos práticos e bem comentados, para que você possa observar como a teoria se traduz em código real. Além disso, ao final de cada seção, sugestões de exercícios serão propostas para reforçar o aprendizado por meio da prática.

Independentemente da sua familiaridade prévia com programação, esta seção foi elaborada para guiá-lo passo a passo, ajudando a construir uma compreensão sólida e intuitiva da linguagem C. Com dedicação e curiosidade, você logo será capaz de criar programas úteis, eficientes e elegantes, abrindo caminho para estudos mais avançados em algoritmos, estruturas de dados e métodos numéricos.

Vamos começar!

0.2.1 Variáveis e Tipos de Dados

Em C, variáveis são espaços reservados na memória do computador usados para armazenar valores que podem variar ao longo da execução do programa. Cada variável deve ser declarada com um tipo, que determina o tamanho e o formato dos dados que ela pode armazenar.

Principais tipos de dados em C:

- `int` — armazena números inteiros, como `10`, `-3`, `0`.
- `float` — armazena números reais com ponto flutuante, como `3.14`, `-2.5`.
- `double` — semelhante ao `float`, porém com maior precisão.
- `char` — armazena um único caractere, como `'a'`, `'Z'`.
- `char[]` — vetor de caracteres usado para armazenar strings (palavras ou frases).

Exemplo 1: Definição de variáveis com diferentes tipos

```
/* Definindo variáveis de tipos diferentes */
#include <stdio.h>

int main() {
    int idade = 25;              // Número inteiro
    float altura = 1.75;         // Número real com precisão simples
```

```
    double peso = 70.45;          // Número real com maior precisão
    char inicial = 'J';           // Um único caractere
    char nome[] = "João";         // Cadeia de caracteres (string)

    printf("Nome: %s\n", nome);
    printf("Inicial: %c\n", inicial);
    printf("Idade: %d anos\n", idade);
    printf("Altura: %.2f metros\n", altura);
    printf("Peso: %.2lf kg\n", peso);

    return 0;
}
```

Neste exemplo, usamos diferentes tipos de variáveis. Repare que:

- `%d` é usado para inteiros.
- `%f` para `float`.
- `%lf` para `double`.
- `%c` para caracteres.
- `%s` para strings.

Modificadores de tipo:

Você também pode modificar os tipos básicos com palavras-chave como:

- `short`, `long` — modificam o tamanho da variável (número de bytes).
- `unsigned` — permite apenas valores positivos (aumenta o limite superior).

Exemplo 2: Modificadores de tipo

```
#include <stdio.h>

int main() {
    unsigned int idade = 30;     // Apenas valores positivos
    long int populacao = 7800000000; // Número grande

    printf("Idade: %u anos\n", idade);
    printf("População mundial: %ld pessoas\n", populacao);

    return 0;
}
```

Boas práticas:

- Sempre inicialize suas variáveis ao declará-las.
- Use nomes descritivos (ex: `altura`, `notaFinal`) para facilitar a leitura do código.
- Evite usar variáveis sem necessidade, e declare-as no menor escopo possível.

Em resumo, escolher corretamente o tipo de dado e declarar variáveis de forma clara e segura é fundamental para o funcionamento e manutenção eficiente dos programas em C.

0.2.2 Entrada e Saída de Dados

Em linguagem C, a entrada e saída de dados são realizadas principalmente com as funções `scanf()` e `printf()`, ambas definidas no cabeçalho `<stdio.h>`. Essas funções permitem interagir com o usuário por meio do terminal, sendo fundamentais em programas interativos.

- `scanf()` é usada para ler dados inseridos pelo usuário (entrada).
- `printf()` é usada para exibir mensagens ou resultados na tela (saída).

Exemplo 1: Leitura e escrita com um número real

```
/* Recebe um número do usuário e exibe o dobro desse valor */
#include <stdio.h>

int main() {
    float numero;

    printf("Digite um número: ");
    scanf("%f", &numero);  // Lê um número float do usuário

    printf("O dobro do número é: %.2f\n", numero * 2);

    return 0;
}
```

Neste exemplo, usamos `%f` para ler e imprimir um número real (do tipo `float`). O especificador `.2f` indica que o número será exibido com duas casas decimais.

Exemplo 2: Leitura de dois inteiros e exibição da soma

```
#include <stdio.h>

int main() {
    int a, b;

    printf("Digite dois números inteiros separados por espaço: ");
    scanf("%d %d", &a, &b);  // Lê dois inteiros

    printf("Soma = %d\n", a + b);

    return 0;
}
```

Observe que usamos `%d` para trabalhar com inteiros. O uso de `&a` e `&b` é necessário porque `scanf()` precisa do endereço das variáveis para armazenar os dados lidos.

Exemplo 3: Leitura de uma string (palavra)

```
#include <stdio.h>

int main() {
    char nome[30];  // Vetor para armazenar até 29 caracteres + '\0'
```

```
    printf("Digite seu nome (sem espaços): ");
    scanf("%s", nome);  // Lê uma palavra (sem espaços)

    printf("Olá, %s!\n", nome);

    return 0;
}
```

Neste exemplo, usamos `%s` para ler uma cadeia de caracteres. Atenção: `scanf("%s", nome)` não lê nomes compostos (com espaços). Para ler uma linha completa, pode-se usar `fgets()`, como mostrado a seguir.

Exemplo 4: Leitura de uma linha inteira com `fgets()`

```
#include <stdio.h>

int main() {
    char frase[100];

    printf("Digite uma frase: ");
    fgets(frase, sizeof(frase), stdin);  // Lê uma linha inteira

    printf("Você digitou: %s", frase);

    return 0;
}
```

A função `fgets()` permite ler até um caractere de nova linha ou até o limite de caracteres especificado. Ela é mais segura do que `gets()` (obsoleta e perigosa) e mais adequada para ler textos com espaços.

Dicas úteis:

- Sempre inclua o cabeçalho `#include <stdio.h>` ao usar `printf()` e `scanf()`.
- Use especificadores corretos: `%d` (int), `%f` (float), `%lf` (double), `%c` (char), `%s` (string).
- Cuidado com `scanf()` seguido de `fgets()` — pode ser necessário limpar o buffer usando `getchar()`.

0.2.3 Escrita de dados em Arquivo de Saída em C

Nesta subseção, mostramos como abrir um arquivo para escrita em linguagem C e registrar dados simples como inteiros, números reais e palavras. Usaremos a função `fopen()` para criar e abrir o arquivo, e a função `fprintf()` para escrever os dados. O exemplo abaixo cria um arquivo chamado `saida.dat` e escreve nele três tipos de informações:

- Três números inteiros: 10, 20, 30
- Três números reais: 3.14, 2.71, 1.41
- Três palavras: casa, sol, livro

Segue o código completo:

```
/* Programa em C para gravar dados simples em um arquivo */

#include <stdio.h>

int main() {
    FILE *fp;

    // Abrir arquivo para escrita ("w" sobrescreve se já existir)
    fp = fopen("saida.dat", "w");

    if (fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    // Escrevendo três inteiros
    fprintf(fp, "Inteiros: %d %d %d\n", 10, 20, 30);

    // Escrevendo três números reais
    fprintf(fp, "Reais: %.2f %.2f %.2f\n", 3.14, 2.71, 1.41);

    // Escrevendo três palavras
    fprintf(fp, "Palavras: casa sol livro\n");

    // Fechando o arquivo
    fclose(fp);

    return 0;
}
```

Observações Importantes

- O arquivo será criado no mesmo diretório onde o programa foi executado.

- O modo `"w"` de abertura cria um novo arquivo ou sobrescreve um existente.
- Após a execução, o conteúdo de `saida.dat` será algo assim:

```
Inteiros: 10 20 30
Reais: 3.14 2.71 1.41
Palavras: casa sol livro
```

- Você pode visualizar o conteúdo do arquivo com editores de texto ou comandos no terminal, como:

```
cat saida.dat
```

Visualização com o xmgrace

Se o arquivo contiver apenas números (por exemplo, colunas de dados), ele poderá ser visualizado graficamente com o `xmgrace`. Suponha que você tenha um arquivo com duas colunas, `dados.dat`. Para visualizar:

```
xmgrace dados.dat
```

Isso abrirá a interface gráfica do `xmgrace`, onde é possível ajustar títulos, eixos, escalas e salvar gráficos.

0.2.4 Operações Matemáticas Básicas

A linguagem C permite realizar operações matemáticas simples utilizando os operadores aritméticos padrão:

- `+` : soma
- `-` : subtração
- `*` : multiplicação
- `/` : divisão
- `%` : módulo (resto da divisão inteira)

Além disso, C possui operadores de **atribuição combinada**, que são formas mais compactas de atualizar o valor de uma variável com base nela mesma:

- `a += b` é equivalente a `a = a + b`
- `a -= b` é equivalente a `a = a - b`
- `a *= b` é equivalente a `a = a * b`
- `a /= b` é equivalente a `a = a / b`
- `a %= b` é equivalente a `a = a % b`

Exemplo 1: Operações Aritméticas Simples

```
/* Realizando operações matemáticas simples */
#include <stdio.h>

int main() {
    int a = 5;
    int b = 3;

    int soma = a + b;
    int multiplicacao = a * b;
    float divisao = (float)a / b;
    int modulo = a % b; // Resto da divisão inteira

    printf("Soma: %d\n", soma);
    printf("Multiplicação: %d\n", multiplicacao);
    printf("Divisão: %.2f\n", divisao);
    printf("Resto da divisão: %d\n", modulo);

    return 0;
}
```

Exemplo 2: Operadores de Atribuição Combinada

```
/* Usando operadores de atribuição combinada */
#include <stdio.h>

int main() {
    int x = 10;

    printf("Valor inicial de x: %d\n", x);

    x += 5;  // x = x + 5
    printf("Após x += 5: %d\n", x);

    x -= 2;  // x = x - 2
    printf("Após x -= 2: %d\n", x);

    x *= 3;  // x = x * 3
    printf("Após x *= 3: %d\n", x);

    x /= 4;  // x = x / 4
    printf("Após x /= 4: %d\n", x);

    x %= 3;  // x = x % 3
    printf("Após x %%= 3: %d\n", x);

    return 0;
}
```

Esses operadores tornam o código mais conciso e facilitam atualizações de variáveis dentro de laços e outras estruturas de controle.

0.2.5 Estruturas Condicionais (if, else, else if)

As estruturas condicionais permitem que o programa tome decisões com base em condições. Usamos `if`, `else if` e `else` para verificar essas condições.

Exemplo:

```
/* Verifica se o número é positivo, negativo ou zero */
#include <stdio.h>

int main() {
    float numero;

    printf("Digite um número: ");
    scanf("%f", &numero);

    if (numero > 0) {
        printf("O número é positivo.\n");
    } else if (numero < 0) {
        printf("O número é negativo.\n");
    } else {
        printf("O número é zero.\n");
    }

    return 0;
}
```

Neste exemplo, o programa verifica se o número inserido é positivo, negativo ou zero e exibe a mensagem correspondente.

0.2.6 Laços de Repetição (for e while)

Os laços de repetição permitem que o código execute um conjunto de instruções várias vezes. O laço `for` é usado para repetir um bloco de código um número conhecido de vezes, enquanto o `while` continua executando enquanto uma condição for verdadeira.

Exemplo:

```
/* Laço for: imprime os números de 1 a 5 */
#include <stdio.h>

int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Neste caso, o laço `for` imprime os números de 1 a 5.

0.2.7 Funções

Funções são blocos de código que realizam uma tarefa específica. Elas ajudam a organizar o código e tornam a reutilização possível.

Exemplo:

```
/* Função que soma dois números */
#include <stdio.h>

int soma(int a, int b) {
    return a + b;
}

int main() {
    int resultado = soma(3, 5);
    printf("A soma é: %d\n", resultado);
    return 0;
}
```

Neste exemplo, a função `soma` recebe dois parâmetros, realiza a soma e retorna o resultado. Em seguida, o resultado é impresso.

0.2.8 Arrays e Manipulação de Coleções

Em C, podemos utilizar arrays (vetores) para armazenar coleções de dados do mesmo tipo. Você pode acessar, modificar e percorrer os elementos de um array.

Exemplo:

```
#include <stdio.h>

int main() {
    // Array de números inteiros
    int numeros[5] = {1, 2, 3, 4};

    // Adicionando um valor na última posição
```

```
    numeros[4] = 5;

    // Acessando e imprimindo o primeiro número
    printf("Primeiro número: %d\n", numeros[0]);

    // Imprimindo todos os números
    printf("Lista atualizada: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", numeros[i]);
    }
    printf("\n");

    return 0;
}
```

O programa define um array de inteiros, atribui um novo valor à última posição, imprime o primeiro número e depois imprime todos os elementos do array.

0.2.9 Manipulação de Strings

Em C, strings são representadas como arrays de caracteres terminados pelo caractere nulo `\0`. Podemos manipulá-las usando funções da biblioteca `<string.h>`.

Exemplo:

```
#include <stdio.h>
#include <string.h>

int main() {
    char nome[20] = "João";
    char sobrenome[20] = "Silva";
    char nome_completo[40];

    // Concatenando strings
    strcpy(nome_completo, nome);       // Copia o nome para nome_completo
    strcat(nome_completo, " ");        // Adiciona um espaço
    strcat(nome_completo, sobrenome);  // Adiciona o sobrenome

    // Exibindo o nome completo
    printf("Nome completo: %s\n", nome_completo);

    return 0;
}
```

Neste exemplo, usamos as funções `strcpy` e `strcat` da biblioteca `<string.h>` para concatenar as strings e formar o nome completo.

0.2.10 Estruturas de Dados Simples (Structs)

Diferente do Python, em C não temos dicionários nativos, mas podemos criar estruturas de dados usando `structs`. Isso nos permite armazenar diferentes tipos de dados agrupados em uma única variável.

Exemplo (Struct):

```
#include <stdio.h>
```

```
// Definindo uma struct para armazenar contatos
struct Contato {
    char nome[30];
    char telefone[15];
};

int main() {
    // Criando e inicializando um contato
    struct Contato contato1 = {"João", "1234-5678"};

    // Exibindo o telefone do contato
    printf("Telefone de %s: %s\n", contato1.nome, contato1.telefone);

    return 0;
}
```

Neste exemplo, criamos uma estrutura `Contato` com dois campos: nome e telefone. Em seguida, inicializamos a struct e imprimimos os dados do contato.

0.2.11 Tratamento de Erros em C (validação de entrada)

Na linguagem C, não existe um mecanismo automático como `try/except` do Python. Porém, é possível tratar erros de entrada manualmente, validando o retorno das funções. Por exemplo, ao ler um número inteiro do usuário, devemos verificar se a leitura foi bem-sucedida.
Exemplo:

```
// Tratamento simples de erro de entrada em C
#include <stdio.h>

int main() {
    int numero;
    printf("Digite um número inteiro: ");
    if (scanf("%d", &numero) == 1) {
        printf("Você digitou: %d\n", numero);
    } else {
        printf("Erro: Você não digitou um número inteiro!\n");
    }
    return 0;
}
```

Aqui, o programa tenta ler um número inteiro usando `scanf`. Se o usuário digitar algo que não seja um número inteiro, `scanf` falha e o programa exibe uma mensagem de erro. Esse tipo de verificação é essencial em programas que exigem robustez.

Estes são os conceitos fundamentais que todo iniciante em C deve compreender. Eles formam a base para escrever programas simples e começar a explorar as capacidades da linguagem.

0.2.12 Números Pseudo-Aleatórios em C

Vamos iniciar agora o estudo dos números aleatórios na linguagem C. Começaremos com a implementação de um gerador de números pseudo-aleatórios clássico na literatura: o **Gerador Congruente Linear** (LCG). Este é um dos métodos mais simples e conhecidos para gerar sequências de números aleatórios. Vamos explicar seu funcionamento e como implementá-lo em C de forma simples e direta.

Método dos Geradores Congruentes Lineares (LCG)

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

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

Onde:

- X_n é o número pseudo-aleatório na n-ésima iteração;
- a é o multiplicador;
- c é o incremento;
- m é o módulo;
- X_0 é a semente inicial (valor inicial).

O algoritmo começa com um valor inicial X_0 e aplica a fórmula recursivamente para gerar a sequência de números. Abaixo temos uma implementação simples deste gerador em C.
Exemplo:

```
// Implementação simples de LCG em C
#include <stdio.h>

unsigned long a = 1664525;
unsigned long c = 1013904223;
unsigned long m = 4294967296; // 2^32
unsigned long X = 123456;     // Semente inicial

// Função que gera o próximo número da sequência
unsigned long lcg() {
    X = (a * X + c) % m;
    return X;
}

int main() {
    int i;
    printf("Gerando 10 números pseudo-aleatórios com LCG:\n");
    for (i = 0; i < 10; i++) {
        printf("%lu\n", lcg());
    }
    return 0;
}
```

Neste exemplo, usamos parâmetros comuns para a, c e m — valores frequentemente utilizados em LCGs clássicos. A semente X_0 é definida com um valor inicial arbitrário. A função `lcg()` gera um novo número pseudo-aleatório a cada chamada, e imprimimos os primeiros 10 números da sequência. Esse tipo de gerador é útil para simulações simples ou testes, mas não é recomendado para aplicações que exigem segurança criptográfica ou alta aleatoriedade. O sucesso de um Gerador Congruente Linear depende da escolha adequada dos parâmetros a, c e m. Vamos discutir a escolha de cada um deles:

- **Módulo** m: Em C, geralmente escolhemos m como uma potência de 2 (por exemplo, 2^{32}) para facilitar os cálculos com operações de módulo em sistemas digitais. A operação `%` (mod) retorna o resto da divisão, garantindo que os valores permaneçam no intervalo de 0 a $m - 1$.

- **Multiplicador** a: Deve ser escolhido cuidadosamente para garantir boa distribuição dos números gerados. Alguns valores são bem conhecidos por produzirem sequências de alta qualidade.
- **Incremento** c: Normalmente é um número positivo. Sua escolha afeta diretamente o período da sequência. Um bom valor ajuda a aumentar a aleatoriedade.
- **Semente** X_0: É o valor inicial da sequência. Modificando a semente, obtemos diferentes sequências pseudo-aleatórias.

O objetivo principal é maximizar o **período** do gerador, ou seja, o número de iterações antes que a sequência comece a se repetir. Com os parâmetros corretos, o LCG pode atingir um período máximo de m. Se faz importante salientar que, embora o Gerador Congruente Linear seja simples e fácil de implementar, ele possui algumas limitações em termos de qualidade dos números aleatórios gerados, como a possibilidade de repetição e a falta de aleatoriedade verdadeira. Esse tipo de gerador é útil em situações em que a simplicidade e a eficiência são mais importantes do que a qualidade dos números aleatórios, como em algumas simulações simples ou jogos. No entanto, para aplicações que exigem uma aleatoriedade de alta qualidade, como simulações científicas avançadas ou criptografia, métodos mais sofisticados, como o Mersenne Twister, são frequentemente utilizados. Este exemplo, embora simples, demonstra a importância de entender os fundamentos dos geradores de números aleatórios e como esses algoritmos podem ser implementados de maneira direta. Portanto, em linhas gerais, o código anterior implementa um gerador de números pseudo-aleatórios utilizando o método Linear Congruential Generator (LCG). A sequência gerada é determinística, baseada na semente inicial X_0 e nos parâmetros m, a e c. Os números gerados são inteiros no intervalo de 0 até $m-1$, o que resulta em uma sequência de números pseudo-aleatórios. Para que os números gerados fiquem no intervalo de 0 a 1, basta dividir os valores gerados por m, o que transformará os números inteiros em números reais no intervalo $[0, 1)$. O código seguinte faz esta implementação:

```
#include <stdio.h>

int m = 2147483647;    // Módulo: maior valor positivo de um int (2^31 - 1)
int a = 1664525;       // Multiplicador
int c = 1013904223;    // Incremento
int X0 = 42;           // Semente inicial

// Função que gera o próximo número da sequência
int gerar_aleatorio(int X) {
    return (a * X + c) % m;
}

int main() {
    int X = X0;
    for (int i = 0; i < 10; i++) {
        X = gerar_aleatorio(X);
        // Divide por m para normalizar o valor no intervalo [0, 1)
        printf("%f\n", (float)X / m);
    }

    return 0;
}
```

Gostaria de explicar que oo utilizar apenas o tipo ‘int‘ na implementação de um gerador congruente linear, enfrentamos algumas limitações importantes. O tipo ‘int‘ em C geralmente representa números inteiros com sinal e possui um intervalo limitado, variando de aproximadamente -2^{31} até

$2^{31}-1$. Isso significa que operações como a multiplicação $a \times X$ podem facilmente exceder esse intervalo, provocando estouro (overflow) e gerando resultados incorretos ou negativos. Como o método do gerador congruente linear depende de aritmética modular precisa para manter a qualidade da sequência pseudoaleatória, qualquer erro causado por overflow pode comprometer a uniformidade e o período da sequência gerada. Além disso, o fato de 'int' permitir valores negativos pode fazer com que os números gerados estejam fora do intervalo esperado, principalmente se o módulo não for cuidadosamente escolhido. Por isso, embora o uso de 'int' torne o código mais simples e adequado para fins didáticos, ele reduz a confiabilidade do gerador, sendo mais indicado para aplicações básicas ou para fins de aprendizado.

Gerador Uniforme em C

Na linguagem C, a biblioteca padrão `stdlib.h` fornece a função `rand()`, que gera números pseudo-aleatórios inteiros no intervalo de 0 até `RAND_MAX`, onde `RAND_MAX` é uma constante definida no sistema. Para gerar números reais uniformemente distribuídos no intervalo $[a, b]$, pode-se usar a seguinte fórmula:

$$x = a + (b - a) \times \frac{\texttt{rand()}}{\texttt{RAND\_MAX}}$$

O código a seguir mostra como gerar 10 números pseudo-aleatórios entre 0 e 1 utilizando apenas recursos básicos da linguagem C:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    int i;
    float numero;
    int seed = 42;

    // Definindo a semente para reprodução da sequência
    srand(seed);  // ou: srand(time(NULL)) para aleatoriedade real

    // Gerando 10 números aleatórios entre 0 e 1
    for (i = 0; i < 10; i++) {
        numero = (float) rand() / RAND_MAX;
        printf("%f\n", numero);
    }

    return 0;
}
```

Este código usa a função `srand()` para definir a semente do gerador com um valor fixo (no caso, 42), permitindo a reprodução da mesma sequência de números. A função `rand()` retorna um número inteiro pseudo-aleatório entre 0 e `RAND_MAX`. Ao dividir esse valor por `RAND_MAX`, obtemos um número real (float) no intervalo aproximado de $[0, 1]$. Essa abordagem é simples e adequada para aplicações introdutórias, embora o gerador `rand()` tenha limitações em termos de qualidade e período da sequência gerada. Para aplicações mais exigentes, recomenda-se o uso de bibliotecas mais modernas ou algoritmos avançados.

0.3 Aprofundando os Conceitos de C com Novos Exemplos

Na seção anterior, exploramos os conceitos fundamentais da programação em C, compreendendo desde a criação de variáveis até estruturas básicas de controle de fluxo. Agora, daremos um passo adiante, aplicando esses conceitos em novos exemplos que ilustram sua utilidade em diferentes contextos.

Nos próximos tópicos, apresentaremos programas que utilizam e combinam os conceitos previamente aprendidos, permitindo uma compreensão mais sólida da lógica de programação. Inicialmente, os exemplos serão simples, reforçando os fundamentos já discutidos. À medida que avançamos, os desafios aumentam gradativamente, introduzindo novas possibilidades e exigindo uma aplicação mais criativa das ferramentas que C oferece.

O objetivo desta seção é ajudar você a consolidar seus conhecimentos, desenvolvendo uma maior familiaridade com a linguagem e sua sintaxe. Além disso, exploraremos situações comuns na programação, incentivando o pensamento lógico e a capacidade de resolver problemas de forma estruturada. Com a prática contínua, será possível não apenas compreender a teoria, mas também aplicá-la de maneira eficiente na construção de programas mais complexos.

Vamos, então, aprofundar nossos estudos e expandir nossas habilidades em C com exemplos práticos um pouco mais desafiadores.

0.3.1 Mais Operações Básicas com Dois Números

Este programa pede ao usuário para inserir dois números e, em seguida, realiza algumas operações matemáticas básicas: soma, subtração, multiplicação e divisão.

```
/* Este programa pede ao usuário dois
 * números e realiza operações básicas */

#include <stdio.h>

int main() {
    float A, B;  // Declaração das variáveis

    // Solicita ao usuário que insira dois valores
    printf("Digite o valor de A: ");
    scanf("%f", &A);
    printf("Digite o valor de B: ");
    scanf("%f", &B);

    // Realiza operações matemáticas
    float soma = A + B;
    float multiplicacao = A * B;
    float subtracao = A - B;

    // Verifica se B é diferente de zero antes de dividir
    if (B != 0) {
        float divisao = A / B;
        printf("Divisao: %.2f\n", divisao);
    } else {
        printf("Erro: divisao por zero.\n");
    }

    // Exibe os resultados
    printf("Soma: %.2f\n", soma);
    printf("Multiplicacao: %.2f\n", multiplicacao);
    printf("Subtracao: %.2f\n", subtracao);
```

```
    return 0;
}
```

- A função `scanf()` lê os valores digitados pelo usuário e armazena nas variáveis `A` e `B`.
- Os operadores `+`, `-`, `*` e `/` realizam as operações matemáticas básicas.
- A verificação `if (B != 0)` evita a divisão por zero.
- A função `printf()` é usada para exibir os resultados.

Cálculo de Raiz Quadrada

Neste exemplo, vamos criar um programa em C que solicita dois valores ao usuário e, a partir desses valores, calcula e exibe as seguintes raízes quadradas:

- Raiz quadrada de A,
- Raiz quadrada de B,
- Raiz quadrada de $|A - B|$,
- Raiz quadrada de $A \times B$.

Passo 1: Solicitação dos Valores

O programa começa pedindo ao usuário para inserir dois valores, A e B, que podem ser quaisquer números reais. O código em C para essa parte é o seguinte:

```
#include <stdio.h>

int main() {
    float A, B;

    // Solicita os valores ao usuário
    printf("Digite o valor de A: ");
    scanf("%f", &A);
    printf("Digite o valor de B: ");
    scanf("%f", &B);

    return 0;
}
```

Utilizamos `scanf()` para ler os valores fornecidos pelo usuário e armazená-los nas variáveis `A` e `B` do tipo `float`.

Passo 2: Cálculo das Raízes Quadradas

Após a entrada dos valores, o programa calcula as raízes quadradas de A, B, $|A-B|$ e $A \times B$. Para isso, usamos a função `sqrt()` da biblioteca `math.h`.

```
#include <stdio.h>
#include <math.h>  // Necessária para sqrt() e fabs()

int main() {
    float A, B;
```

```
    printf("Digite o valor de A: ");
    scanf("%f", &A);
    printf("Digite o valor de B: ");
    scanf("%f", &B);

    // Cálculo das raízes quadradas
    float raiz_A = sqrt(A);
    float raiz_B = sqrt(B);
    float raiz_AB = sqrt(fabs(A - B));       // usa valor absoluto
    float raiz_produto = sqrt(A * B);        // raiz de A vezes B

    // Exibe os resultados
    printf("Raiz de A: %.2f\n", raiz_A);
    printf("Raiz de B: %.2f\n", raiz_B);
    printf("Raiz de |A - B|: %.2f\n", raiz_AB);
    printf("Raiz de A * B: %.2f\n", raiz_produto);

    return 0;
}
```

Note que utilizamos a função `fabs()` (função de valor absoluto para números `float`) para garantir que o argumento da raiz seja positivo. Assim como em Python, a raiz quadrada de um número negativo não está definida nos números reais.

0.3.2 Explicando a importância dos `Loops`

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

Sem utilizar loop (`for`)

No exemplo abaixo, somamos os números de 1 até 10 um a um, linha por linha, sem usar nenhuma estrutura de repetição.

```
#include <stdio.h>

int main() {
    int soma = 0;

    // Soma manual dos números de 1 até 10
    soma = 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10;

    printf("Soma dos números de 1 a 10 (sem loop): %d\n", soma);
    return 0;
}
```

Embora funcione para um número pequeno de valores, esse método não é prático para intervalos maiores.

Com estrutura de repetição (`for`)

Agora, vejamos o mesmo problema resolvido de forma muito mais eficiente com a estrutura de repetição `for`.

```
#include <stdio.h>

int main() {
    int soma = 0;
    int i;

    // Loop que soma os números de 1 até 10
    for (i = 1; i <= 10; i++) {
        soma += i;
    }

    printf("Soma dos números de 1 a 10 (com loop): %d\n", soma);
    return 0;
}
```

Este código é muito mais compacto e pode ser facilmente adaptado para somar até 100, 1000 ou qualquer outro número.

Soma dos quadrados dos números ímpares de 1 até 100

Este exemplo mostra como utilizar um loop para somar os quadrados apenas dos números ímpares de 1 até 100.

```
#include <stdio.h>

int main() {
    int soma_simples = 0;
    int soma_quadrados = 0;
    int i;

    // Loop que percorre apenas números ímpares de 1 a 100
    for (i = 1; i <= 100; i += 2) {
        soma_simples += i;
        soma_quadrados += i * i;
    }

    printf("Soma simples dos ímpares: %d\n", soma_simples);
    printf("Soma dos quadrados dos ímpares: %d\n", soma_quadrados);
    return 0;
}
```

Soma dos números de 1 a 1000 com passo 3

Neste exemplo, somamos todos os números naturais entre 1 e 1000 que estão espaçados por 3, ou seja: 1, 4, 7, 10, ..., até 1000.

```
#include <stdio.h>

int main() {
    int soma = 0;
    int i;

    // Loop com incremento de 3 (números espaçados por 3)
    for (i = 1; i <= 1000; i += 3) {
        soma += i;
    }

    printf("Soma dos números de 1 a 1000 com passo 3: %d\n", soma);
```

```
    return 0;
}
```

Esses exemplos mostram como os loops tornam o código mais flexível, reutilizável e eficiente, sendo uma ferramenta fundamental em qualquer linguagem de programação, inclusive C.

0.3.3 Mais exemplos sobre manipulação de Vetores

Nesta subseção, vamos resolver computacionalmente o problema de calcular a soma e o produto dos números de 1 a 20. Para isso, utilizaremos um vetor que armazena esses valores e um loop que percorre esse vetor, realizando as operações desejadas.

O procedimento consiste em três etapas principais:

- Criar um vetor com os números de 1 a 20.
- Inicializar duas variáveis: uma para armazenar a soma e outra para o produto dos elementos.
- Usar um loop para acessar cada elemento do vetor, somando e multiplicando os valores de forma acumulativa.

Esse tipo de manipulação de vetor é muito comum em programação e é fundamental para o processamento eficiente de conjuntos de dados. No exemplo a seguir, apresentamos a implementação desse algoritmo em linguagem C.

```
#include <stdio.h>

int main() {
    int vetor[20];
    int i;
    int soma = 0;
    long long int produto = 1;

    // Preenche o vetor com os números de 1 a 20
    for (i = 0; i < 20; i++) {
        vetor[i] = i + 1;
    }

    // Calcula a soma e o produto dos elementos do vetor
    for (i = 0; i < 20; i++) {
        soma += vetor[i];        // Acumula a soma
        produto *= vetor[i];     // Acumula o produto
    }

    // Exibe os resultados
    printf("Soma dos elementos do vetor: %d\n", soma);
    printf("Produto dos elementos do vetor: %lld\n", produto);

    return 0;
}
```

Note que utilizamos o tipo `long long int` para a variável `produto`, pois o resultado da multiplicação de vários números inteiros pode ser muito grande e exceder o limite do tipo `int`. Além disso, a indexação em C começa do 0 (zero), então o loop percorre os índices de 0 até 19, correspondendo aos 20 elementos do vetor. Este exemplo ilustra bem como manipular vetores em C e como loops são úteis para realizar cálculos sobre coleções de dados.

0.3.4 Mais informações sobre o uso de Condicionais em C

Em C, utilizamos estruturas condicionais para executar diferentes blocos de código dependendo do resultado de uma condição lógica. O comando `if` permite avaliar expressões e decidir qual código será executado com base em seu valor (verdadeiro ou falso).

Estrutura do Comando `if`

A estrutura básica de um condicional em C é:

```
if (condicao) {
    // Código executado se a condição for verdadeira
} else if (outra_condicao) {
    // Código executado se a primeira condição for falsa, mas esta for verdadeira
} else {
    // Código executado se nenhuma das condições anteriores for verdadeira
}
```

Agora, vejamos um exemplo prático utilizando esse conceito.

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

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

```
#include <stdio.h>

int main() {
    float num1, num2, num3, num4;
    float media;
    float maior, menor;

    // Solicita quatro números ao usuário
    printf("Digite o primeiro número: ");
    scanf("%f", &num1);
    printf("Digite o segundo número: ");
    scanf("%f", &num2);
    printf("Digite o terceiro número: ");
    scanf("%f", &num3);
    printf("Digite o quarto número: ");
    scanf("%f", &num4);

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

    // Determina o maior número
    maior = num1;  // Assume que num1 é o maior inicialmente
    if (num2 > maior) {
        maior = num2;
    }
    if (num3 > maior) {
        maior = num3;
    }
    if (num4 > maior) {
        maior = num4;
    }

    // Determina o menor número
```

```
    menor = num1;  // Assume que num1 é o menor inicialmente
    if (num2 < menor) {
        menor = num2;
    }
    if (num3 < menor) {
        menor = num3;
    }
    if (num4 < menor) {
        menor = num4;
    }

    // Exibe os resultados
    printf("Maior número: %.2f\n", maior);
    printf("Menor número: %.2f\n", menor);
    printf("Média dos números: %.2f\n", media);

    return 0;
}
```

Vamos analisar os detalhes do código:

- Os valores são lidos com `scanf()` e armazenados como números reais do tipo `float`.
- A média é calculada somando os quatro números e dividindo o resultado por 4.
- Para encontrar o maior número, assumimos que o primeiro número é o maior e comparamos com os demais usando comandos `if`.
- O mesmo procedimento é aplicado para encontrar o menor número.
- O programa imprime o maior e o menor número, além da média calculada.

O uso de `if`, `else if` e `else` é essencial para controle de fluxo em C. No exemplo acima usamos apenas `if` encadeados, mas é possível escrever funções auxiliares para tornar o código mais modular, ou usar arrays e laços para automatizar o processo com mais números.

0.3.5 Ordenando três valores usando comandos `if` em C

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

```
#include <stdio.h>

int main() {
    float x, y, z, temp;

    // Solicita os três números ao usuário
    printf("Digite o primeiro número: ");
    scanf("%f", &x);
    printf("Digite o segundo número: ");
```

```
    scanf("%f", &y);
    printf("Digite o terceiro número: ");
    scanf("%f", &z);

    // Algoritmo de ordenação manual usando if
    if (x > y) {
        temp = x;
        x = y;
        y = temp;
    }
    if (y > z) {
        temp = y;
        y = z;
        z = temp;
    }
    if (x > y) {
        temp = x;
        x = y;
        y = temp;
    }

    // Exibe os números ordenados
    printf("Números em ordem crescente: %.2f %.2f %.2f\n", x, y, z);

    return 0;
}
```

0.3.6 Exemplo: Ordenando quatro valores usando comandos `if` em C

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

```
#include <stdio.h>

int main() {
    float a, b, c, d, temp;

    // Solicita os quatro números ao usuário
    printf("Digite o primeiro número: ");
    scanf("%f", &a);
    printf("Digite o segundo número: ");
    scanf("%f", &b);
    printf("Digite o terceiro número: ");
    scanf("%f", &c);
    printf("Digite o quarto número: ");
    scanf("%f", &d);

    // Algoritmo de ordenação manual usando if
```

```
    if (a > b) {
        temp = a; a = b; b = temp;
    }
    if (b > c) {
        temp = b; b = c; c = temp;
    }
    if (c > d) {
        temp = c; c = d; d = temp;
    }
    if (a > b) {
        temp = a; a = b; b = temp;
    }
    if (b > c) {
        temp = b; b = c; c = temp;
    }
    if (a > b) {
        temp = a; a = b; b = temp;
    }

    // Exibe os números ordenados
    printf("Números em ordem crescente: %.2f %.2f %.2f %.2f\n", a, b, c, d);

    return 0;
}
```

0.3.7 Ordenando um vetor e mantendo a correspondência com outro vetor em C

Suponha que temos dois vetores $x[i = 1, \ldots, 5]$ e $y[i = 1, \ldots, 5]$, onde cada par (x_i, y_i) representa uma relação entre os valores de x e y. Nosso objetivo é ordenar x do menor para o maior e reorganizar y de modo a manter a relação inicial entre os pares. Isso pode ser feito em C utilizando comparações e trocas manuais entre os elementos dos vetores. O programa a seguir implementa esse processo de forma simples, utilizando apenas comandos básicos da linguagem C:

```
#include <stdio.h>

int main() {
    int i, j;
    float x[5] = {3.2, 1.5, 4.8, 2.0, 0.9};  // Vetor x com valores de exemplo
    float y[5] = {10, 20, 30, 40, 50};       // Vetor y com valores associados a x
    float temp;

    // Exibindo os vetores originais
    printf("Vetor x original: ");
    for (i = 0; i < 5; i++) {
        printf("%.2f ", x[i]);
    }
    printf("\nVetor y original: ");
    for (i = 0; i < 5; i++) {
        printf("%.2f ", y[i]);
    }

    // Ordenando x e reorganizando y usando um algoritmo simples (Bubble Sort)
    for (i = 0; i < 5 - 1; i++) {
        for (j = 0; j < 5 - 1 - i; j++) {
            if (x[j] > x[j + 1]) {
                // Troca os valores em x
                temp = x[j];
```

```
                x[j] = x[j + 1];
                x[j + 1] = temp;

                // Troca os valores correspondentes em y
                temp = y[j];
                y[j] = y[j + 1];
                y[j + 1] = temp;
            }
        }
    }

    // Exibindo os vetores após a ordenação
    printf("\n\nVetor x ordenado: ");
    for (i = 0; i < 5; i++) {
        printf("%.2f ", x[i]);
    }
    printf("\nVetor y reorganizado: ");
    for (i = 0; i < 5; i++) {
        printf("%.2f ", y[i]);
    }
    printf("\n");

    return 0;
}
```

Neste programa, utilizamos dois laços aninhados para comparar e trocar os valores de x sempre que necessário. Quando uma troca é feita em x, fazemos também uma troca correspondente em y, garantindo que os pares (x_i, y_i) permaneçam associados corretamente. Esse método simples e direto é eficiente para pequenos conjuntos de dados e ajuda a compreender o processo de ordenação mantendo estruturas associadas.

0.3.8 Exemplo de Programa em C para Cálculo de Funções Trigonométricas

Neste exemplo, apresentamos um programa em linguagem C que calcula as funções trigonométricas seno, cosseno e tangente para valores de θ variando de 0 até 2π. O programa realiza as seguintes tarefas:

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

Inclusão de Bibliotecas

Utilizamos as bibliotecas `stdio.h` para entrada e saída de dados, `math.h` para as funções trigonométricas e a constante π.

```
#include <stdio.h>
#include <math.h>
```

Definição do Intervalo de θ

A variável θ será variada de 0 até 2π, utilizando um laço `for`. Escolhemos 100 pontos igualmente espaçados.

```
#define N 100
#define PI 3.141592653589793
```

A seguir, mostramos o código completo. O programa cria três arquivos de saída contendo os valores de θ e das funções trigonométricas seno, cosseno e tangente.

```
#include <stdio.h>
#include <math.h>

#define N 100
#define PI 3.141592653589793

int main() {
    FILE *fsen, *fcos, *ftg;
    double theta, passo;
    int i;

    // Abrir arquivos para escrita
    fsen = fopen("seno.dat", "w");
    fcos = fopen("cos.dat", "w");
    ftg  = fopen("tg.dat", "w");

    passo = (2 * PI) / (N - 1);  // Calcula o passo entre os pontos

    for (i = 0; i < N; i++) {
        theta = i * passo;
        fprintf(fsen, "%.6f %.6f\n", theta, sin(theta));
        fprintf(fcos, "%.6f %.6f\n", theta, cos(theta));
        fprintf(ftg,  "%.6f %.6f\n", theta, tan(theta));
    }

    // Fechar os arquivos
    fclose(fsen);
    fclose(fcos);
    fclose(ftg);

    return 0;
}
```

Visualização dos Resultados com `xmgrace`

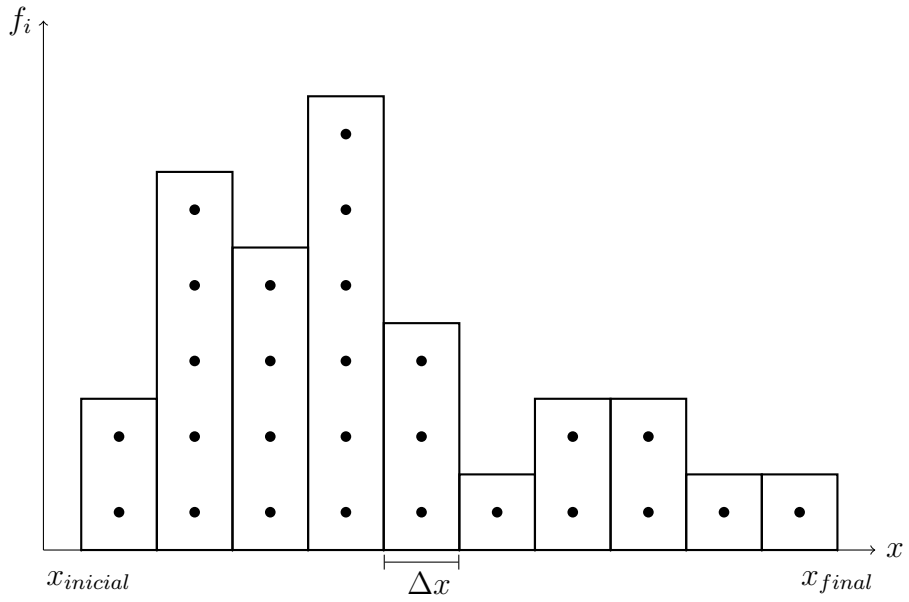
Embora o programa em C não produza gráficos diretamente, os dados salvos nos arquivos `seno.dat`, `cos.dat` e `tg.dat` podem ser visualizados com ferramentas como o `xmgrace`. Por exemplo:

- Para visualizar a função seno: `xmgrace seno.dat`
- Para visualizar a função cosseno: `xmgrace cos.dat`
- Para visualizar a função tangente: `xmgrace tg.dat`

Esses comandos devem ser executados no terminal Linux em um ambiente gráfico com o pacote `xmgrace` instalado. O programa abrirá uma janela gráfica com o gráfico correspondente aos dados do arquivo.

0.3.9 Cálculo do Histograma Normalizado em Linguagem C

O histograma é uma ferramenta estatística usada para representar a distribuição de uma sequência de números. Em termos simples, um **histograma** é uma representação gráfica da frequência de ocorrência de valores dentro de intervalos ("bins") definidos. O intervalo total dos dados é dividido em subintervalos de mesmo comprimento Δx, e contabiliza-se quantos elementos pertencem a cada um deles. Suponha um conjunto de dados $\{x_1, x_2, x_3, \ldots, x_N\}$ contido no intervalo $[x_{inicial}, x_{final}]$. Dividindo esse intervalo em M caixas (ou bins), a frequência absoluta f_i indica quantos valores de x caem dentro de cada bin. A seguir, ilustramos um histograma simples com 10 divisões:



A versão normalizada do histograma é obtida ao dividir cada frequência absoluta pelo número total de dados e pela largura do bin. O resultado representa uma estimativa da densidade de probabilidade. Abaixo, apresentamos um programa em linguagem C que gera 10^6 números pseudoaleatórios uniformemente distribuídos no intervalo $[0, 1]$ utilizando um gerador congruente linear (LCG) e calcula o histograma normalizado. O histograma é salvo no arquivo `histograma.dat`.

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000000       // Número de amostras
#define BINS 10         // Número de bins
#define A 1664525       // Parâmetros do LCG
#define C 1013904223
#define M 4294967296.0  // 2^32 (como double)
#define SEED 42         // Semente inicial

int main() {
    double x, bin_width = 1.0 / BINS;
    unsigned int X = SEED;
    int hist[BINS] = {0}; // Inicializa o histograma com zeros

    // Geração dos números aleatórios com LCG e contagem no histograma
    for (int i = 0; i < N; i++) {
        X = A * X + C;  // Atualiza valor com LCG
        x = (double)X / M;  // Normaliza para [0,1]

        int bin = (int)(x / bin_width);
```

```
        if (bin >= BINS) bin = BINS - 1; // Corrige borda superior

        hist[bin]++;  // Conta no bin correspondente
    }

    // Salvar histograma normalizado no arquivo
    FILE *fp = fopen("histograma.dat", "w");
    if (fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    for (int i = 0; i < BINS; i++) {
        double x_i = i * bin_width;
        double fi = hist[i] / (double)(N * bin_width); // Normalização
        fprintf(fp, "%lf %lf\n", x_i, fi);
    }

    fclose(fp);
    printf("Histograma salvo em 'histograma.dat'\n");
    return 0;
}
```

Este programa ilustra como um gerador linear congruente pode ser utilizado para gerar distribuições simples. Cada valor é mapeado para um bin com base na largura definida. A normalização final garante que a área sob o histograma (interpretação contínua) seja igual a 1. A análise gráfica pode ser feita externamente com ferramentas como Python, GNUPlot ou outros softwares de visualização.

0.3.10 Geração de Números Aleatórios com Distribuição Gaussiana e Análise de Histograma

Para gerar números aleatórios com distribuição **normal padrão** (média zero e variância 1), utilizamos o **método de Box-Muller**. Este método transforma dois números aleatórios uniformemente distribuídos $u_1, u_2 \in (0, 1)$ em dois números z_0, z_1 com distribuição normal padrão. A transformação é dada por:

$$z_0 = \sqrt{-2 \ln u_2} \cdot \cos(2\pi u_1), \quad z_1 = \sqrt{-2 \ln u_2} \cdot \sin(2\pi u_1)$$

Para avaliar se os números gerados seguem de fato uma distribuição gaussiana, construímos um histograma da amostra. Já debatemos o cálculo do histograma anteriormente entretanto vamos descrever novamente o procedimento agora adaptado para o caso da desordem Gaussiana:

1. Dividir o intervalo de interesse (neste caso, de -5 a 5) em subintervalos (bins) de largura $\Delta x = 0{,}1$.
2. Contar quantos valores da amostra caem dentro de cada bin (usando uma tolerância de 0.05 para capturar o centro do intervalo).
3. Normalizar os valores do histograma para que a área total sob a curva seja igual a 1, tornando-a uma aproximação da densidade de probabilidade.

Abaixo está o programa completo, revisado, comentado e simplificado para facilitar o entendimento:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define RAN() ((double)rand() / (double)(RAND_MAX))
#define pi M_PI

int main() {
    FILE *fil, *fil1;
    int i, bin;
    double ngau[500000];       // Vetor com números gaussiano
    double histo[100000];      // Histograma
    double xx[100000];         // Posições dos bins
    double r1, r2, r3, r4, r5;

    srand(time(NULL)); // Inicializa gerador de números aleatórios

    fil = fopen("numerosaleatoriosGAU.dat", "w");
    fil1 = fopen("histogramanumerosaleatoriosGAU.dat", "w");

    // Gera 200.000 números com distribuição normal (Box-Muller)
    for (i = 1; i <= 200000; i++) {
        r1 = RAN(); // u2
        r2 = RAN(); // u1
        r3 = sqrt(-2.0 * log(r1)); // sqrt(-2ln(u2))
        r4 = cos(2.0 * pi * r2);   // cos(2pi*u1)
        r5 = r3 * r4;              // número gaussiano
        ngau[i] = r5;
        fprintf(fil, "%g\n", r5);  // salva no arquivo
    }

    // Construção do histograma
    bin = 0;
    for (r1 = -5.0; r1 <= 5.0; r1 += 0.1) {
        bin++;
        r2 = 0.0;
        for (i = 1; i <= 200000; i++) {
            if (fabs(ngau[i] - r1) < 0.05) {
                r2 += 1.0;
            }
        }
        xx[bin] = r1;
        histo[bin] = r2;
    }

    // Normalização do histograma (área total = 1)
    double area = 0.0;
    for (i = 1; i < bin; i++) {
        area += (histo[i] + histo[i + 1]) * 0.5 * 0.1;
    }

    for (i = 1; i < bin; i++) {
```

```
        histo[i] = histo[i] / area;
        fprintf(fil1, "%g %g\n", xx[i], histo[i]);
    }

    fclose(fil);
    fclose(fil1);

    return 0;
}
```

Este programa gera uma grande amostra de números aleatórios com distribuição normal padrão utilizando o método de Box-Muller. A análise do histograma mostra que a distribuição dos dados se aproxima bem da forma de uma Gaussiana, centrada em zero, com maior densidade em torno da média e cauda decaindo rapidamente nos extremos, como esperado. A normalização do histograma garante que ele possa ser comparado com a função densidade de probabilidade teórica. Esse tipo de verificação é crucial ao trabalhar com simulações estocásticas que dependem de amostras com distribuição específica, como em problemas de física estatística, dinâmica molecular ou simulações de Monte Carlo. Este programa serve como base para experimentos mais complexos envolvendo desordem estatística, ruído gaussiano, entre outros cenários.

0.3.11 Geração de Números Aleatórios com Distribuição em Lei de Potência

Nesta seção , desejamos gerar números aleatórios que sigam uma distribuição de probabilidade do tipo lei de potência:

$$P(x) \sim x^n, \quad \text{com } x_0 \leq x \leq x_1. \tag{1}$$

Distribuições em lei de potência são comuns em diversos contextos físicos e estatísticos, especialmente em sistemas desordenados e fenômenos críticos. A geração de números aleatórios com essa distribuição requer a inversão da função de distribuição acumulada (CDF). Dado que a CDF $F(x)$ associada a uma distribuição $P(x) = Cx^n$ é:

$$F(x) = \frac{x^{n+1} - x_0^{n+1}}{x_1^{n+1} - x_0^{n+1}}, \tag{2}$$

para $n \neq -1$, a inversa é dada por:

$$x = \left[(x_1^{n+1} - x_0^{n+1}) \cdot r + x_0^{n+1}\right]^{\frac{1}{n+1}}, \tag{3}$$

onde $r \in [0, 1]$ é uma variável aleatória uniforme. Esta é a expressão utilizada no programa a seguir para gerar números com a distribuição desejada.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define RAN() ((double)rand()/(double)(RAND_MAX)) // número aleatório uniforme em [0,1]

int main() {
    FILE *fil, *fil1;
    int i, N1;
    double x0, x1, r1, r2;
```

```
    double M[1000000];
    double nn; // expoente da lei de potência

    // Arquivo de saída para os números gerados
    fil = fopen("Cnumerosaleatoriosleidepotencia.dat", "w");
    // Arquivo de saída para a distribuição (histograma)
    fil1 = fopen("Cdistribuicaonumerosaleatoriosleidepotencia.dat", "w");

    N1 = 500000; // total de números gerados
    x0 = 0.5;    // limite inferior do suporte
    x1 = 1.5;    // limite superior do suporte
    nn = 0.0;    // expoente da lei de potência (ex: 0 para uniforme, -1 para 1/x)

    // Geração dos números com distribuição em lei de potência
    for(i = 1; i <= N1; i++) {
        r1 = RAN();
        r2 = pow(x1, nn+1) - pow(x0, nn+1); // normalização da CDF
        r2 = r2 * r1 + pow(x0, nn+1);        // valor acumulado
        r2 = pow(r2, 1.0 / (nn + 1.0));       // inversa da CDF
        M[i] = r2;
        fprintf(fil, "%20.8g\n", r2); // salva o número gerado
    }

    // Cálculo do histograma da distribuição gerada
    double dx = (x1 - x0) / 80.0; // largura de cada bin

    for(r1 = x0 + dx; r1 <= x1 - dx; r1 += dx) {
        r2 = 0.0;
        for(i = 1; i <= N1; i++) {
            if(fabs(r1 - M[i]) < 0.5 * dx) {
                r2 += 1.0;
            }
        }
        if (r2 > 0.0) {
            fprintf(fil1, "%20.8g %20.8g\n", r1, r2 / (double)N1);
        }
    }

    return 0;
}
```

O programa acima permite gerar um grande conjunto de números aleatórios com distribuição de probabilidade do tipo $P(x) \sim x^n$. A distribuição obtida é então avaliada via histograma (combinando as frequências relativas em intervalos de largura fixa) e salva em um arquivo para posterior análise. Essa técnica é especialmente útil para simulações numéricas em sistemas físicos com desordem não trivial, onde distribuições com caudas longas (como $n < 0$) são relevantes. A normalização da distribuição em si não é garantida diretamente, mas pode ser ajustada posteriormente dividindo pela integral numérica da distribuição (como feito no exemplo anterior com a gaussiana).

0.4 Derivada Numérica

A derivada de uma função $f(x)$ em um ponto x representa a taxa de variação da função em relação à variável x. Geometricamente, corresponde ao coeficiente angular da reta tangente ao gráfico da função naquele ponto.

0.4.1 Definição Matemática

A derivada de uma função $f(x)$ no ponto x é definida como o limite:

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Essa definição formal é fundamental no cálculo, mas em muitas aplicações práticas e computacionais, utilizamos aproximações numéricas desse limite — especialmente quando temos dados discretos ou funções complexas.

0.4.2 Derivada Numérica por Diferenças Finitas

Se tivermos um conjunto de pontos discretos (x_i, y_i), onde $y_i = f(x_i)$, podemos aproximar a derivada usando métodos de diferenças finitas. Os principais métodos são:

- Diferença Progressiva (usada no início dos dados):

$$f'(x_i) \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

- Diferença Regressiva (usada no final dos dados):

$$f'(x_i) \approx \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

- Diferença Central (mais precisa, usada no interior dos dados):

$$f'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}$$

0.4.3 Exemplo 1: Derivada de $f(x) = x^2$ com Diferença Central

Neste exemplo, implementamos em linguagem C a derivada numérica da função $f(x) = x^2$ em um ponto usando o método de diferença central.

```
#include <stdio.h>

// Função f(x) = x^2
double f(double x) {
    return x * x;
}

int main() {
    double h = 1e-5; // Passo pequeno
    double x = 2.0;  // Ponto onde queremos a derivada

    // Diferença central
```

```
    double derivada = (f(x + h) - f(x - h)) / (2 * h);

    printf("Derivada numérica de x^2 em x=%.2f: %.5f\n", x, derivada);
    // Esperado: 2 * x = 4.0
    return 0;
}
```

0.4.4 Exemplo 2: Derivada Numérica de Vetores $x[i], y[i]$

Neste exemplo, utilizamos dois vetores $x[i]$ e $y[i]$, representando os valores discretos da função, e aplicamos a técnica de diferenças finitas para obter a derivada numérica.

```
#include <stdio.h>

#define N 5

int main() {
    double x[N] = {0, 1, 2, 3, 4};
    double y[N] = {0, 1, 4, 9, 16}; // y = x^2
    double dy_dx[N];
    int i;

    for (i = 0; i < N; i++) {
        if (i == 0) {
            dy_dx[i] = (y[i+1] - y[i]) / (x[i+1] - x[i]); // Diferença progressiva
        } else if (i == N-1) {
            dy_dx[i] = (y[i] - y[i-1]) / (x[i] - x[i-1]); // Diferença regressiva
        } else {
            dy_dx[i] = (y[i+1] - y[i-1]) / (x[i+1] - x[i-1]); // Diferença central
        }
    }

    for (i = 0; i < N; i++) {
        printf("x = %.1f, dy/dx = %.2f\n", x[i], dy_dx[i]);
    }

    return 0;
}
```

0.4.5 Exemplo 3: Derivada Numérica de Dados em Arquivo

Suponha que temos um arquivo chamado `dados.dat` com duas colunas: a primeira para os valores de x, a segunda para os valores de y. O programa abaixo lê o arquivo, armazena os dados em vetores e calcula a derivada numérica.

Exemplo de conteúdo do arquivo `dados.dat`:

```
0.0  0.0
1.0  1.0
```

```
2.0  4.0
3.0  9.0
4.0 16.0
```

Código em C:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 1000

int main() {
    double x[MAX], y[MAX], dy_dx[MAX];
    int i, n = 0;

    FILE *fp = fopen("dados.dat", "r");
    if (fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
        return 1;
    }

    // Leitura dos dados do arquivo
    while (fscanf(fp, "%lf %lf", &x[n], &y[n]) == 2 && n < MAX) {
        n++;
    }
    fclose(fp);

    // Cálculo da derivada numérica
    for (i = 0; i < n; i++) {
        if (i == 0) {
            dy_dx[i] = (y[i+1] - y[i]) / (x[i+1] - x[i]);
        } else if (i == n - 1) {
            dy_dx[i] = (y[i] - y[i-1]) / (x[i] - x[i-1]);
        } else {
            dy_dx[i] = (y[i+1] - y[i-1]) / (x[i+1] - x[i-1]);
        }
    }

    // Exibição dos resultados
    printf("x\t dy/dx\n");
    for (i = 0; i < n; i++) {
        printf("%.2f\t %.2f\n", x[i], dy_dx[i]);
    }

    return 0;
}
```

Considerações Finais

- A derivada numérica é uma ferramenta essencial quando não se conhece a forma analítica da função.

- É especialmente útil no tratamento de dados experimentais para estimar velocidades, taxas de variação e outros parâmetros derivados.
- Em regiões com ruído nos dados, convém aplicar técnicas de suavização antes de derivar numericamente.
- Passos muito pequenos h podem introduzir erros de arredondamento por limitações de precisão da máquina.

0.5 Integração Numérica: Aproximação de Integrais com Somatórios em C

A integração numérica é uma técnica essencial para estimar a área sob uma curva, especialmente quando a função não admite uma primitiva elementar ou quando os dados disponíveis são discretos, como em experimentos ou simulações. Nesses casos, em vez de resolver a integral definida por métodos analíticos, utilizamos somatórios que fornecem aproximações cada vez mais precisas conforme o número de subdivisões do intervalo aumenta.

Métodos Clássicos de Integração Numérica

Seja $f(x)$ uma função contínua definida no intervalo $[a, b]$, e seja n o número de subintervalos de comprimento $h = \frac{b-a}{n}$. Os métodos mais comuns para a aproximação da integral definida são:

- **Regra do Retângulo (ponto à esquerda ou ponto médio):**

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

  onde $x_i = a + ih$. Essa é a aproximação mais simples e apresenta erro da ordem de $O(h)$, sendo adequada para estimativas rápidas, porém menos precisas.

- **Regra do Trapézio:**

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

  Essa técnica substitui a curva por segmentos de reta e tem erro de ordem $O(h^2)$, oferecendo uma boa precisão com complexidade computacional moderada.

- **Regra de Simpson (1/3):**

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4\sum_{\substack{i=1 \\ i \text{ ímpar}}}^{n-1} f(x_i) + 2\sum_{\substack{i=2 \\ i \text{ par}}}^{n-2} f(x_i) + f(x_n)\right]$$

  Esta regra exige que n seja par e utiliza aproximações quadráticas (parábolas) em pares de subintervalos. Sua precisão é superior, com erro de ordem $O(h^4)$, sendo ideal quando se busca maior exatidão.

Esses métodos são amplamente utilizados em aplicações científicas e de engenharia, e sua escolha depende do nível de precisão desejado, do comportamento da função, e da quantidade de pontos disponíveis.

0.5.1 Exemplo com função analítica: $f(x) = e^x$ no intervalo $[0, 1]$

```
/* integra_exponencial.c
   Integra numericamente a função f(x) = exp(x)
   no intervalo [0,1] com n=5 subdivisões.
*/

#include <stdio.h>
#include <math.h>

int main() {
    int i, n = 5;
    double a = 0.0, b = 1.0;
    double h = (b - a) / n;
    double integral_ret = 0.0, integral_trap = 0.0, integral_simp = 0.0;
    double xi, x_meio;

    // Método do Retângulo (ponto médio)
    for (i = 0; i < n; i++) {
        xi = a + i * h;
        x_meio = xi + h / 2.0;
        integral_ret += exp(x_meio);
    }
    integral_ret *= h;

    // Método do Trapézio
    integral_trap = 0.5 * (exp(a) + exp(b));
    for (i = 1; i < n; i++) {
        xi = a + i * h;
        integral_trap += exp(xi);
    }
    integral_trap *= h;

    // Método de Simpson (n deve ser par)
    if (n % 2 == 0) {
        integral_simp = exp(a) + exp(b);
        for (i = 1; i < n; i++) {
            xi = a + i * h;
            if (i % 2 == 0)
                integral_simp += 2 * exp(xi);
            else
                integral_simp += 4 * exp(xi);
        }
        integral_simp *= h / 3.0;
    }

    printf("Integral (Retângulo) = %.5f\n", integral_ret);
    printf("Integral (Trapézio)  = %.5f\n", integral_trap);
    if (n % 2 == 0)
        printf("Integral (Simpson)   = %.5f\n", integral_simp);
    else
        printf("Método de Simpson requer número par de subintervalos.\n");
```

```
    return 0;
}
```

0.5.2 Exemplo com dados tabulados salvos em arquivo de dados

O programa a seguir realiza a integração numérica de uma função cujos valores são fornecidos em forma de dados tabulados, armazenados em um arquivo chamado "tabela.dat". Esse tipo de abordagem é extremamente útil quando não se conhece a expressão analítica da função, mas se dispõe de valores experimentais ou simulados da mesma em diferentes pontos do domínio. O arquivo `tabela.dat` contém os seguintes dados :

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

O programa em tela lê os pares $(x_i, f(x_i))$ do arquivo e calcula a integral definida utilizando três métodos clássicos da análise numérica:

1. Método do Retângulo (ponto médio): Aproxima a área sob a curva utilizando retângulos cuja altura é dada pela média dos valores da função nos extremos de cada subintervalo.
2. Método do Trapézio: Substitui a função por segmentos de reta, aproximando a área por uma série de trapézios.
3. Método de Simpson (1/3): Uma técnica mais precisa (quando aplicável), que usa parábolas para aproximar a função em pares de subintervalos. Este método exige que o número de subintervalos seja par, condição que é verificada pelo programa.

Além da implementação das três técnicas, o programa também trata a leitura de dados de forma robusta e informa o usuário se o método de Simpson não puder ser aplicado. Os resultados das integrais são impressos com cinco casas decimais de precisão. Esse programa serve como uma ferramenta prática para análises numéricas aplicadas em física, engenharia e ciências em geral, onde dados experimentais frequentemente precisam ser integrados numericamente.

```
/* integra_dados.c
   Integração numérica com dados tabulados
   lidos do arquivo "tabela.dat"
*/

#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int main() {
```

```
FILE *arq;
double x[MAX], fx[MAX];
int i, n = 0;
double h, integral_ret = 0.0, integral_trap = 0.0, integral_simp = 0.0;

arq = fopen("tabela.dat", "r");
if (arq == NULL) {
    printf("Erro ao abrir o arquivo.\n");
    return 1;
}

while (fscanf(arq, "%lf %lf", &x[n], &fx[n]) == 2) {
    n++;
}
fclose(arq);
n = n - 1; // número de subintervalos
h = x[1] - x[0];

// Método do Retângulo (ponto médio)
for (i = 0; i < n; i++) {
    double f_meio = (fx[i] + fx[i + 1]) / 2.0;
    integral_ret += f_meio;
}
integral_ret *= h;

// Método do Trapézio
integral_trap = fx[0] + fx[n];
for (i = 1; i < n; i++) {
    integral_trap += 2 * fx[i];
}
integral_trap *= h / 2.0;

// Método de Simpson (se n for par)
if (n % 2 == 0) {
    integral_simp = fx[0] + fx[n];
    for (i = 1; i < n; i++) {
        if (i % 2 == 0)
            integral_simp += 2 * fx[i];
        else
            integral_simp += 4 * fx[i];
    }
    integral_simp *= h / 3.0;
}

printf("Integral (Retângulo) = %.5f\n", integral_ret);
printf("Integral (Trapézio)  = %.5f\n", integral_trap);
if (n % 2 == 0)
    printf("Integral (Simpson)   = %.5f\n", integral_simp);
else
```

```
        printf("Método de Simpson requer número par de subintervalos.\n");

    return 0;
}
```

Os exemplos apresentados ilustram claramente como técnicas de integração numérica podem ser implementadas de forma simples e eficiente utilizando a linguagem C. A integração pelo método do retângulo, do trapézio e de Simpson mostra-se bastante útil em diferentes contextos, tanto para funções conhecidas quanto para conjuntos de dados tabulados provenientes de experimentos ou simulações computacionais. No caso de uma função analítica, os métodos permitem comparações com a solução exata, avaliando a eficiência e a precisão de cada técnica. Já para dados tabulados, onde a função não é conhecida explicitamente, a integração numérica torna-se uma ferramenta indispensável, permitindo estimativas confiáveis da integral sem necessidade de ajustes ou aproximações simbólicas da função. Vale destacar que, embora o método de Simpson geralmente forneça resultados mais precisos, ele requer um número par de subintervalos e pressupõe que os pontos estejam uniformemente espaçados. Por outro lado, os métodos do retângulo e do trapézio possuem aplicação mais ampla e simples, mesmo com menor precisão relativa. Essas ferramentas computacionais são amplamente aplicadas nas ciências exatas, como física e engenharia, onde integrais definidas frequentemente aparecem e precisam ser resolvidas numericamente com base em dados discretos. O domínio dessas técnicas representa um passo importante na formação do estudante e na prática profissional de modelagem e análise de sistemas reais.

0.6 Interpolação

Interpolação é uma técnica numérica utilizada para estimar valores desconhecidos de uma função a partir de um conjunto conhecido de pontos. Suponha que conhecemos o valor de uma função $f(x)$ em alguns pontos $x_0, x_1, \ldots, x_n$, mas queremos estimar o valor da função em um ponto intermediário x, onde $x_i < x < x_{i+1}$. A interpolação fornece uma maneira de construir uma função simples (geralmente um polinômio) que passa exatamente pelos pontos conhecidos, e então usa essa função para estimar os valores desejados.

Quando usar interpolação:

- Quando não conhecemos a expressão analítica da função, mas temos dados tabulados.
- Quando queremos estimar valores intermediários entre pontos medidos.
- Em diversas aplicações de engenharia, física, computação científica e processamento de sinais.

Tipos de interpolação mais comuns:

- **Interpolação Linear:** usa uma reta entre dois pontos consecutivos.
- **Interpolação Polinomial:** utiliza um polinômio que passa por todos os pontos.
- **Interpolação por Splines:** utiliza funções polinomiais por partes que se conectam suavemente.
- **Interpolação de Lagrange e de Newton:** formas específicas de interpolação polinomial, úteis para implementação computacional.

A escolha do método de interpolação depende da quantidade de dados disponíveis, da precisão desejada e da complexidade computacional. Interpolações simples, como a linear, são fáceis de implementar e bastante eficazes quando os dados são aproximadamente lineares entre si. Já métodos mais sofisticados, como splines cúbicos ou interpolação de Lagrange, fornecem melhores aproximações em casos mais complexos.

0.6.1 Interpolação Simples

A interpolação é uma técnica numérica usada para estimar o valor de uma função em um ponto onde ela não foi calculada explicitamente, com base em pontos conhecidos. O caso mais simples é a **interpolação linear**, que utiliza dois pontos para estimar valores intermediários por meio de uma reta.

Fórmula da interpolação linear:

$$f(x) \approx f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0} \cdot (x - x_0)$$

Onde:

- x_0, x_1 são os dois pontos conhecidos,
- $f(x_0), f(x_1)$ são os valores da função nesses pontos,
- x é o ponto onde se deseja estimar $f(x)$.

Exemplo: Suponha os seguintes dados tabulados:

x	$f(x)$
1.0	2.0
2.0	4.0

Queremos estimar o valor de $f(x)$ para $x = 1.5$.

Programa em C para interpolação linear entre dois pontos:

```
/* Interpolação Linear Simples */
#include <stdio.h>

/* Função que realiza a interpolação linear */
float interpolar(float x0, float x1, float fx0, float fx1, float x) {
    return fx0 + ((fx1 - fx0) / (x1 - x0)) * (x - x0);
}

int main() {
    float x0, x1, fx0, fx1, x, fx;

    // Entrada dos dados conhecidos
    printf("Entre com x0 e f(x0): ");
    scanf("%f %f", &x0, &fx0);

    printf("Entre com x1 e f(x1): ");
    scanf("%f %f", &x1, &fx1);

    // Ponto onde queremos estimar f(x)
    printf("Entre com o valor de x para interpolar: ");
```

```
    scanf("%f", &x);

    // Cálculo da interpolação
    fx = interpolar(x0, x1, fx0, fx1, x);

    printf("Estimativa de f(%.2f) = %.2f\n", x, fx);

    return 0;
}
```

O código apresentado realiza uma interpolação linear simples entre dois pontos conhecidos de uma função. Ele começa pedindo ao usuário que forneça os valores de x_0, x_1 e os correspondentes $f(x_0)$ e $f(x_1)$. Em seguida, solicita o valor de x no qual se deseja estimar a função. A função 'interpolar' calcula o valor aproximado de $f(x)$ usando a fórmula da interpolação linear. Por fim, o programa exibe o resultado da estimativa. Esse tipo de interpolação é útil quando se tem poucos dados e se assume que o comportamento da função entre dois pontos é aproximadamente linear.

Exemplo de execução:

```
Entre com x0 e f(x0): 1.0 2.0
Entre com x1 e f(x1): 2.0 4.0
Entre com o valor de x para interpolar: 1.5
Estimativa de f(1.50) = 3.00
```

Importante:

- A interpolação linear é válida apenas para intervalos pequenos entre os pontos conhecidos.
- Quanto mais próximos os pontos x_0 e x_1, melhor tende a ser a estimativa.
- Para funções mais complexas, pode ser necessário usar métodos de interpolação polinomial ou spline.

A interpolação é uma ferramenta útil em diversas aplicações científicas e de engenharia, permitindo construir aproximações rápidas a partir de dados discretos.

0.6.2 Interpolação Polinomial Simples (Fórmula de Lagrange)

A interpolacção polinomial è um método para encontrar um polinômio de grau n que passe exatamente por $n+1$ pontos dados. Dado um conjunto de pontos $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, o polinômio interpolador de Lagrange $P(x)$ è definido por:

$$P(x) = \sum_{i=0}^{n} y_i \cdot L_i(x) \tag{4}$$

onde $L_i(x)$ é o polinômio base de Lagrange:

$$L_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{5}$$

Cada termo $L_i(x)$ é igual a 1 no ponto x_i e zero nos demais pontos x_j (com $j \neq i$).

Exemplo: Interpolando 3 pontos: $(1, 2), (2, 3), (4, 1)$

Vamos implementar um programa em C que calcula a interpolacção de Lagrange para um valor x fornecido pelo usuário.

Programa em C: Interpolacção de Lagrange

```
#include <stdio.h>

#define N 3

int main() {
    double x[N] = {1.0, 2.0, 4.0};     // Pontos x conhecidos
    double y[N] = {2.0, 3.0, 1.0};     // Valores de y correspondentes
    double xp;                         // Ponto onde queremos interpolar
    double P = 0.0;                    // Resultado da interpolação

    printf("Digite o valor de x para interpolar: ");
    scanf("%lf", &xp);

    for (int i = 0; i < N; i++) {
        double Li = 1.0;
        for (int j = 0; j < N; j++) {
            if (j != i) {
                Li *= (xp - x[j]) / (x[i] - x[j]);
            }
        }
        P += y[i] * Li;
    }

    printf("Valor interpolado em x = %.2lf: %.4lf\n", xp, P);

    return 0;
}
```

Explicação:

- O programa define 3 pontos conhecidos (x_i, y_i).
- O usuário fornece o ponto x no qual deseja interpolar.
- Para cada i, calculamos o polinômio base $L_i(x)$.
- O valor final $P(x)$ é a soma dos produtos $y_i \cdot L_i(x)$.

Importante:

- O método funciona bem para poucos pontos.
- Para muitos pontos, o grau do polinômio aumenta e o cálculo se torna instável numericamente.
- Em casos com muitos pontos, prefira usar interpoladores do tipo *spline*.

0.6.3 Interpolação por Splines

A interpolacão por splines **resolve um problema comum da interpolacão polinomial**: quando usamos polinômios de alto grau para interpolar muitos pontos, o resultado pode oscilar fortemente entre os dados. Para evitar isso, a ideia dos *splines* dividir o intervalo dos dados em vários subintervalos e interpolar cada um com um polinômio de grau baixo, geralmente grau 3 (**splines cúbicos**). Cada spline um polinômio suave que se conecta ao anterior e ao seguinte com continuidade de primeira e segunda derivadas. Isso resulta em uma curva suave que passa por todos os pontos.

Forma Geral de um Spline Cúbico em um Intervalo $[x_i, x_{i+1}]$

Em cada subintervalo $[x_i, x_{i+1}]$, o spline da forma:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 \tag{6}$$

Os coeficientes a_i, b_i, c_i, d_i são determinados impondo:

- $S_i(x_i) = f(x_i)$ e $S_i(x_{i+1}) = f(x_{i+1})$ (o polinômio passa pelos pontos dados);
- $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ (continuidade da derivada de primeira ordem);
- $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ (continuidade da derivada de segunda ordem);
- Condições de contorno, como $S''(x_0) = 0$ e $S''(x_n) = 0$ para *spline natural*.

Essas equações geram um sistema linear tridiagonal para resolver os coeficientes.

Exemplo simples: 3 pontos

Vamos interpolar os pontos $(1, 2)$, $(2, 3)$ e $(3, 5)$ usando splines cúbicos naturais. Como temos 2 intervalos, teremos 2 polinômios. O código a seguir implementa a resolução desse sistema de forma simplificada (valores fixos, resolvidos manualmente):

```
/* Interpolação spline cúbica natural para 3 pontos fixos */
#include <stdio.h>
#include <math.h>

float spline(float x) {
    float resultado;

    if (x <= 2) {
        // Intervalo [1, 2]
        float a = 2.0;
        float b = 0.8125;
        float c = 0.0;
        float d = 0.1875;
        resultado = a + b*(x - 1) + c*pow(x - 1, 2) + d*pow(x - 1, 3);
    } else {
        // Intervalo [2, 3]
        float a = 3.0;
        float b = 1.6875;
        float c = 0.5625;
```

```
        float d = -0.25;
        resultado = a + b*(x - 2) + c*pow(x - 2, 2) + d*pow(x - 2, 3);
    }

    return resultado;
}

int main() {
    float x;
    printf("Digite o valor de x entre 1 e 3: ");
    scanf("%f", &x);

    if (x < 1 || x > 3) {
        printf("x fora do intervalo.\n");
        return 1;
    }

    float y = spline(x);
    printf("Estimativa: f(%.2f) = %.4f\n", x, y);

    return 0;
}
```

Exemplo com leitura de arquivo: spline com 3 pontos de tabela

Neste exemplo, ao invés de digitar os pontos manualmente, o programa lê uma tabela com 3 pares (x, y) a partir de um arquivo chamado `tabela.dat`. O formato desse arquivo é bastante simples: cada linha contém um par de valores reais correspondentes às coordenadas (x,y)(x,y) de um ponto da tabela.

Exemplo de conteúdo do arquivo `tabela.dat`:

```
1.0 2.0
2.0 3.0
3.0 5.0
```

O programa apresentado a seguir é responsável por ler os dados contidos nesse arquivo, realizar a interpolação por spline cúbica natural e gerar um novo arquivo com a curva suavizada, permitindo a posterior visualização gráfica com ferramentas apropriadas.

```
/* Interpolação spline natural para 3 pontos lidos de um arquivo */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float x[3], y[3];

/* Lê dados de um arquivo com três pares (x, y) */
int ler_arquivo(const char *nome_arquivo) {
    FILE *fp = fopen(nome_arquivo, "r");
    if (fp == NULL) {
        printf("Erro ao abrir o arquivo.\n");
```

```
        return 0;
    }

    for (int i = 0; i < 3; i++) {
        if (fscanf(fp, "%f %f", &x[i], &y[i]) != 2) {
            printf("Erro ao ler dados da linha %d.\n", i+1);
            fclose(fp);
            return 0;
        }
    }

    fclose(fp);
    return 1;
}

/* Função para interpolar com spline cúbico aproximado (3 pontos) */
float spline_manual(float x_in) {
    float a, b, c, d;

    if (x_in <= x[1]) {
        // Intervalo [x0, x1]
        a = y[0];
        b = (y[1] - y[0]) / (x[1] - x[0]);
        c = 0;
        d = ((y[2] - y[1]) / (x[2] - x[1]) - b) / (x[2] - x[0]);
        return a + b*(x_in - x[0]) + c*pow(x_in - x[0],2) + d*pow(x_in - x[0],3);
    } else {
        // Intervalo [x1, x2] (aproximação)
        a = y[1];
        b = (y[2] - y[1]) / (x[2] - x[1]);
        c = 0;
        d = (b - (y[1] - y[0]) / (x[1] - x[0])) / (x[2] - x[0]);
        return a + b*(x_in - x[1]) + c*pow(x_in - x[1],2) + d*pow(x_in - x[1],3);
    }
}

int main() {
    float x_interp;

    if (!ler_arquivo("tabela.dat")) {
        return 1;
    }

    printf("Digite o valor de x a interpolar: ");
    scanf("%f", &x_interp);

    float resultado = spline_manual(x_interp);
    printf("f(%.2f) = %.4f\n", x_interp, resultado);

    return 0;
}
```

Este programa permite reutilizar os mesmos dados várias vezes, bastando atualizar o conteúdo do arquivo `tabela.dat`. Embora limitado a 3 pontos, já é uma boa aproximação de uma interpolação suave entre pontos com continuidade de derivadas.

0.6.4 Interpolação por Spline Cúbico Natural com Número Arbitrário de Pontos

A interpolação por splines cúbicos naturais é uma técnica poderosa que utiliza funções polinomiais de terceiro grau em cada subintervalo entre pontos dados. Diferente da interpolação polinomial global, ela evita oscilações excessivas e garante suavidade até a segunda derivada.

Nesta implementação, o programa:

- Lê os dados de um arquivo chamado `tabela.dat` com duas colunas: x e y.
- Determina automaticamente quantos pontos há na tabela.
- Calcula os coeficientes do spline cúbico natural resolvendo um sistema tridiagonal.
- Permite interpolar um valor de entrada arbitrário.

Formato esperado do arquivo `tabela.dat`:

```
1.0 1.0
2.0 4.0
3.0 9.0
4.0 16.0
```

Para facilitar a visualização do resultado da interpolação, o programa apresentado a seguir gera um arquivo de saída contendo tanto os valores originais da tabela quanto os pontos da curva suave obtida por meio da spline cúbica natural. Esse arquivo pode ser utilizado para a construção de gráficos em ferramentas como `gnuplot`, `matplotlib`, `octave`, entre outras, permitindo uma análise visual clara da qualidade da interpolação.

```
/* Interpolação por spline cúbico com escrita dos dados interpolados */
#include <stdio.h>
#include <stdlib.h>

#define MAX 1000

float x[MAX], y[MAX], a[MAX], b[MAX], c[MAX], d[MAX], h[MAX], alpha[MAX];
float l[MAX], mu[MAX], z[MAX];
int n = 0;

int ler_dados(const char *nome) {
    FILE *fp = fopen(nome, "r");
    if (!fp) {
        printf("Erro ao abrir o arquivo.\n");
        return 0;
    }

    while (fscanf(fp, "%f %f", &x[n], &y[n]) == 2) {
        n++;
        if (n >= MAX) break;
    }

    fclose(fp);
    return (n >= 2);
}
```

```
void calcular_spline() {
    for (int i = 0; i < n; i++) a[i] = y[i];

    for (int i = 0; i < n - 1; i++)
        h[i] = x[i + 1] - x[i];

    for (int i = 1; i < n - 1; i++)
        alpha[i] = (3.0 / h[i]) * (a[i + 1] - a[i]) -
                   (3.0 / h[i - 1]) * (a[i] - a[i - 1]);

    l[0] = 1.0; mu[0] = 0.0; z[0] = 0.0;

    for (int i = 1; i < n - 1; i++) {
        l[i] = 2 * (x[i + 1] - x[i - 1]) - h[i - 1] * mu[i - 1];
        mu[i] = h[i] / l[i];
        z[i] = (alpha[i] - h[i - 1] * z[i - 1]) / l[i];
    }

    l[n - 1] = 1.0; z[n - 1] = 0.0; c[n - 1] = 0.0;

    for (int j = n - 2; j >= 0; j--) {
        c[j] = z[j] - mu[j] * c[j + 1];
        b[j] = (a[j + 1] - a[j]) / h[j] -
               h[j] * (c[j + 1] + 2 * c[j]) / 3.0;
        d[j] = (c[j + 1] - c[j]) / (3.0 * h[j]);
    }
}

float spline(float x_in) {
    int i;
    for (i = 0; i < n - 1; i++) {
        if (x_in >= x[i] && x_in <= x[i + 1]) break;
    }
    if (i == n - 1) i--;

    float dx = x_in - x[i];
    return a[i] + b[i]*dx + c[i]*dx*dx + d[i]*dx*dx*dx;
}

void gerar_saida(const char *arquivo_saida) {
    FILE *fp = fopen(arquivo_saida, "w");
    if (!fp) {
        printf("Erro ao criar o arquivo de saída.\n");
        return;
    }

    float xmin = x[0], xmax = x[n-1];
    int pontos = 1000;
    float passo = (xmax - xmin) / (pontos - 1);

    for (int i = 0; i < pontos; i++) {
        float xi = xmin + i * passo;
        float yi = spline(xi);
        fprintf(fp, "%f %f\n", xi, yi);
    }

    fclose(fp);
    printf("Arquivo '%s' gerado com sucesso.\n", arquivo_saida);
```

```
}

int main() {
    if (!ler_dados("tabela.dat")) {
        printf("Erro: não foi possível ler dados suficientes.\n");
        return 1;
    }

    calcular_spline();

    float x_val;
    printf("Digite o valor de x a interpolar: ");
    scanf("%f", &x_val);

    float resultado = spline(x_val);
    printf("f(%.2f)  %.6f\n", x_val, resultado);

    gerar_saida("splinesaida.dat");

    return 0;
}
```

O arquivo `splinesaida.dat` contém uma amostragem suave do spline, permitindo visualizações como:

```
gnuplot> plot "tabela.dat" with points pt 7, "spline_saida.dat" with lines
```

Esse programa é bastante flexível e pode lidar com uma quantidade grande de dados (até 1000 pontos), interpolando suavemente qualquer valor intermediário. O uso de splines cúbicos naturais garante continuidade até a segunda derivada, o que é importante em aplicações físicas, engenharia, gráficos e simulações. Splines cúbicos proporcionam curvas suaves e ajustadas aos dados, mesmo quando há muitos pontos envolvidos, sendo uma ferramenta poderosa em interpolação. Por essa razão, são amplamente utilizados em diversas áreas como computação gráfica, sistemas CAD (desenho assistido por computador), animações digitais e simulações numéricas em geral. Para conjuntos de dados maiores, é recomendável o uso de algoritmos eficientes para a resolução do sistema linear tridiagonal associado, garantindo precisão e desempenho computacional adequado.

0.7 Regressão Linear

Regressão linear é uma técnica estatística utilizada para modelar a relação entre duas variáveis, assumindo que essa relação possa ser aproximada por uma linha reta. Em outras palavras, dados um conjunto de pontos experimentais (x_i, y_i), a ideia é encontrar a melhor reta $z(x) = ax + b$ que aproxima esses dados no sentido dos mínimos quadrados, ou seja, aquela que minimiza o erro total entre os valores reais y_i e os valores estimados $z(x_i)$.

Quando usar regressão linear:

- Quando os dados parecem alinhar-se aproximadamente ao longo de uma reta.
- Quando desejamos prever o valor de uma variável com base em outra.
- Em análises exploratórias de dados, modelagem de fenômenos físicos, engenharia, ciências sociais, etc.

Fórmula da regressão linear: O modelo ajustado por regressão linear é da forma:

$$z(x) = ax + b$$

onde:

- a é o coeficiente angular (inclinação da reta),
- b é o coeficiente linear (intercepto com o eixo y).

Objetivo: Dado um conjunto de n observações (x_i, y_i), queremos encontrar os coeficientes a e b que melhor ajustam os dados no sentido dos mínimos quadrados. Isso significa minimizar o erro quadrático total entre os valores observados y_i e os valores ajustados $z_i = ax_i + b$:

$$E(a, b) = \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$

Essa é a função de erro a ser minimizada. Para encontrar os valores ótimos de a e b, derivamos $E(a, b)$ com relação a a e b e igualamos as derivadas a zero:

$$\frac{\partial E}{\partial a} = -2 \sum_{i=1}^{n} x_i (y_i - ax_i - b) = 0$$

$$\frac{\partial E}{\partial b} = -2 \sum_{i=1}^{n} (y_i - ax_i - b) = 0$$

Dividindo ambas as equações por 2 e reorganizando, obtemos o sistema:

$$\sum x_i y_i = a \sum x_i^2 + b \sum x_i \tag{1}$$

$$\sum y_i = a \sum x_i + bn \tag{2}$$

Multiplicando a equação (2) por $\sum x_i$:

$$\left(\sum y_i\right)\left(\sum x_i\right) = a\left(\sum x_i\right)^2 + bn\sum x_i \tag{3}$$

Subtraindo (3) da equação (1) multiplicada por n:

$$n\sum x_i y_i - \left(\sum y_i\right)\left(\sum x_i\right) = a\left(n\sum x_i^2 - \left(\sum x_i\right)^2\right)$$

Isolando a:

$$a = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - \left(\sum x_i\right)^2}$$

Com o valor de a, substituímos em (2) para encontrar b:

$$b = \frac{1}{n}\left(\sum y_i - a\sum x_i\right)$$

Resumo das fórmulas da regressão linear:

$$a = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{n\sum x_i^2 - \left(\sum x_i\right)^2}$$

$$b = \frac{\sum y_i - a\sum x_i}{n}$$

Essas expressões permitem ajustar uma reta $z(x) = ax + b$ aos dados (x_i, y_i) de modo que o erro quadrático total seja minimizado.

Exemplo 1: Regressão Linear Simples

Suponha que temos um conjunto de dados armazenado no arquivo `dados.dat`, contendo duas colunas correspondentes às variáveis x e y. Queremos ajustar uma reta aos dados e gerar um novo arquivo `regressao.dat`, contendo três colunas: os valores originais de x, y, e os valores $z(x)$ previstos pela regressão linear.

Código em C para regressão linear simples:

```
/* Regressão Linear Simples */
#include <stdio.h>
#define N 1000  // número máximo de pontos

float x[N], y[N];

int main() {
    FILE *fin, *fout;
    int n = 0;
    float soma_x = 0, soma_y = 0, soma_xy = 0, soma_x2 = 0;
```

```
    float a, b;

    // Abre arquivo com dados de entrada
    fin = fopen("dados.dat", "r");
    if (fin == NULL) {
        printf("Erro ao abrir dados.dat\n");
        return 1;
    }

    // Leitura dos dados
    while (fscanf(fin, "%f %f", &x[n], &y[n]) == 2) {
        soma_x += x[n];
        soma_y += y[n];
        soma_xy += x[n] * y[n];
        soma_x2 += x[n] * x[n];
        n++;
    }
    fclose(fin);

// Cálculo dos coeficientes da reta: y = a*x + b
    a = (n * soma_xy - soma_x * soma_y) / (n * soma_x2 - soma_x * soma_x);
    b = (soma_y - a * soma_x) / n;

// Grava os dados com a reta ajustada (x,y,z(x)) onde z(x)=ax+b
    fout = fopen("regressao.dat", "w");
    for (int i = 0; i < n; i++) {
        float z = a * x[i] + b;
        fprintf(fout, "%f %f %f\n", x[i], y[i], z);
    }
    fclose(fout);

    printf("Regressão concluída: z(x) = %.4f*x + %.4f\n", a, b);
    return 0;
}
```

Exemplo 2: Regressão para Lei de Potência

Muitos fenômenos físicos seguem leis do tipo $y \sim x^A$, conhecidas como leis de potência. Para identificar esse tipo de dependência, é comum aplicar logaritmo nos dados, obtendo uma relação linear: $\log y = A \log x + B$. Dessa forma, podemos aplicar a regressão linear nos dados transformados $(\log x, \log y)$ e recuperar os parâmetros da lei original.

Código em C para regressão log-log:

```
/* Regressão Log-Log para Lei de Potência */
#include <stdio.h>
#include <math.h>
#define N 1000

float x[N], y[N];

int main() {
    FILE *fin, *fout;
    int n = 0;
    float soma_logx = 0, soma_logy = 0, soma_logxlogy = 0, soma_logx2 = 0;
    float A, B;
```

```
    // Leitura dos dados do arquivo
    fin = fopen("dadospot.dat", "r");
    if (fin == NULL) {
        printf("Erro ao abrir dadospot.dat\n");
        return 1;
    }

    while (fscanf(fin, "%f %f", &x[n], &y[n]) == 2) {
        float logx = log(x[n]);
        float logy = log(y[n]);

        soma_logx += logx;
        soma_logy += logy;
        soma_logxlogy += logx * logy;
        soma_logx2 += logx * logx;
        n++;
    }
    fclose(fin);

    // Ajuste linear: log(y) = A*log(x) + B
    A = (n * soma_logxlogy - soma_logx * soma_logy) / (n * soma_logx2 - soma_logx * soma_logx);
    B = (soma_logy - A * soma_logx) / n;

    // Grava os resultados ajustados
    fout = fopen("potencia_ajustada.dat", "w");
    for (int i = 0; i < n; i++) {
        float z = exp(B) * pow(x[i], A);  // z(x) = e^B * x^A
        fprintf(fout, "%f %f %f\n", x[i], y[i], z);
    }
    fclose(fout);

    printf("Ajuste concluído: y  %.4f * x^%.4f\n", exp(B), A);
    return 0;
}
```

Esses dois exemplos ilustram como a regressão linear pode ser aplicada em diferentes contextos: tanto para relações lineares diretas quanto para identificar leis de potência por meio da transformação logarítmica. A análise gráfica dos arquivos de saída pode ajudar a visualizar a qualidade dos ajustes.

0.8 Mapas Caóticos e Dinâmica Não Linear

Sistemas dinâmicos discretos têm sido amplamente utilizados como modelos simplificados para descrever a evolução temporal de diversas quantidades em contextos físicos, biológicos, econômicos e computacionais. Em particular, *mapas não lineares* representam uma ferramenta poderosa para o estudo de comportamentos complexos a partir de regras matemáticas simples. Um mapa dinâmico é uma função iterativa que relaciona o estado de um sistema em um instante n com seu estado no instante seguinte $n+1$, por meio de uma equação do tipo:

$$x_{n+1} = f(x_n), \tag{7}$$

onde x_n representa o estado do sistema no tempo discreto n, e f é uma função (geralmente não linear). A análise desses sistemas revela fenômenos surpreendentes, como bifurcações, ciclos periódicos, atratores estranhos e caos determinístico. Um sistema caótico, embora completamente determinístico, exibe sensibilidade extrema às condições iniciais — um pequeno desvio na configuração inicial pode levar a evoluções drasticamente diferentes no futuro. Isso torna impossível prever seu comportamento a longo prazo, mesmo com conhecimento completo das equações de evolução.
Um dos exemplos mais emblemáticos de mapa caótico é o **mapa logístico**, que, apesar de sua forma simples, apresenta uma transição clara da ordem ao caos à medida que o parâmetro de controle é variado. Devido a essa riqueza de comportamentos, o mapa logístico tem sido extensivamente estudado como protótipo de sistemas não lineares e caóticos. Nos parágrafos seguintes, discutiremos com mais detalhes a estrutura do mapa logístico e apresentaremos uma simulação numérica que permite visualizar seu diagrama de bifurcação.

0.8.1 O Mapa Logístico

O mapa logístico é uma equação de recorrência amplamente estudada em teoria do caos e sistemas dinâmicos não lineares. Ele é definido pela relação:

$$x_{n+1} = r x_n (1 - x_n), \tag{8}$$

onde $x_n \in [0, 1]$ representa a população (normalizada) no tempo discreto n, e r é um parâmetro real que controla a taxa de crescimento da população. Apesar de sua simplicidade, essa equação mostra uma rica variedade de comportamentos conforme o valor de r varia:

- Para $0 < r < 1$, a população tende a zero.
- Para $1 < r < 3$, a população converge para um valor fixo.
- Para $3 < r < 3.57$, ocorrem bifurcações sucessivas, onde o valor final alterna entre dois, quatro, oito, etc., pontos.
- Para $r > 3.57$, o sistema entra em um regime caótico, embora ainda existam janelas de periodicidade.

Esse comportamento torna o mapa logístico um modelo paradigmático para o estudo de transições da ordem ao caos.

A seguir, apresentamos um programa em linguagem C que simula a dinâmica do mapa logístico para vários valores do parâmetro $r \in [2.5, 3.99]$ e várias condições iniciais $x_0 \in [0.01, 0.99]$. O

programa realiza 100.000 iterações para cada par (r, x_0), descartando os transientes e registrando o valor final da variável x em um arquivo chamado `cmapa.dat`. Esses dados podem ser utilizados para construir, por exemplo, o diagrama de bifurcação do mapa.

```
/* Programa em C para simular o mapa logístico e gerar
   dados para o diagrama de bifurcação.
   Cada linha do arquivo de saída contém o valor de r
   e o valor final de x após muitas iterações. */

#include<stdio.h>
#include<math.h>

int main() {
    FILE *fil;

    double x2, x1, x0, r;
    int n;

    // Nome do arquivo de saída
    char filename[100] = "cmapa.dat";

    // Abre o arquivo para escrita
    fil = fopen(filename, "w");

    // Loop externo: percorre valores do parâmetro r
    for(r = 2.5; r <= 3.99; r = r + 0.002) {

        // Loop interno: percorre diferentes condições iniciais x0
        for(x0 = 0.01; x0 <= 0.99; x0 = x0 + 0.02) {

            x1 = x0; // Condição inicial

            // Executa a dinâmica do mapa logístico
            for(n = 1; n <= 100000; n = n + 1) {
                x2 = r * x1 * (1. - x1); // Aplicação do mapa
                x1 = x2;
            }

            // Escreve r e o valor final de x no arquivo
            fprintf(fil, "%13.8f %13.8f\n", r, x1);
        }
    }

    // Fecha o arquivo e termina o programa
    fclose(fil);
    return 0;
}
```

Uma vez gerado o arquivo `cmapa.dat`, é possível utilizar ferramentas como `gnuplot`, `matplotlib` ou `octave` para visualizar o diagrama de bifurcação, revelando as estruturas periódicas e caóticas

do sistema conforme o parâmetro r é variado.

0.8.2 O Mapa de Verhulst

O mapa de Verhulst, também conhecido como mapa logístico generalizado, é uma variação do mapa logístico clássico. Ele foi originalmente proposto por Pierre François Verhulst no século XIX como modelo para o crescimento populacional limitado por recursos. A versão discreta do modelo assume a forma:

$$x_{n+1} = rx_n - \beta x_n^2, \tag{9}$$

onde x_n representa a população normalizada no tempo discreto n, r é um parâmetro de crescimento e β representa a intensidade da competição intraespecífica (isto é, o efeito limitante do crescimento). Essa forma evidencia de maneira explícita a competição entre o crescimento linear e o termo de saturação quadrático. Com a escolha $\beta = r$, a equação assume a forma do mapa logístico usual:

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

No entanto, ao permitir valores diferentes de β, o mapa de Verhulst fornece uma generalização útil para o estudo de dinâmicas não lineares, incluindo bifurcações e caos, em contextos ecológicos, econômicos e computacionais. A seguir, apresentamos um programa simples em linguagem C que simula o mapa de Verhulst para valores do parâmetro $r \in [2.5, 3.99]$, assumindo $\beta = 1.0$. O valor final da população x após muitas iterações é registrado para diferentes condições iniciais. Os dados são salvos no arquivo `verhulst.dat`.

```
/* Simulação do mapa de Verhulst em C.
   O programa gera dados para o diagrama de bifurcação,
   variando o parâmetro r e condições iniciais. */

#include<stdio.h>
#include<math.h>

int main() {
    FILE *fil;

    double x2, x1, x0, r, beta;
    int n;

    // Define o parâmetro de competição intraespecífica
    beta = 1.0;

    // Nome do arquivo de saída
    char filename[100] = "verhulst.dat";

    // Abre o arquivo para escrita
    fil = fopen(filename, "w");

    // Loop externo: varia o parâmetro r
```

```
    for(r = 2.5; r <= 3.99; r = r + 0.002) {

        // Loop interno: diferentes condições iniciais x0
        for(x0 = 0.01; x0 <= 0.99; x0 = x0 + 0.02) {

            x1 = x0; // Condição inicial

            // Dinâmica iterativa do mapa de Verhulst
            for(n = 1; n <= 100000; n = n + 1) {
                x2 = r * x1 - beta * x1 * x1;
                x1 = x2;
            }

            // Salva os dados: r e valor final de x
            fprintf(fil, "%13.8f %13.8f\n", r, x1);
        }
    }

    // Fecha o arquivo
    fclose(fil);
    return 0;
}
```

Assim como no caso do mapa logístico, o arquivo de saída pode ser utilizado para construir o diagrama de bifurcação, que revela como o sistema evolui de regimes estacionários para dinâmicas periódicas e, eventualmente, para o caos à medida que o parâmetro r cresce. Ferramentas de visualização como `gnuplot`, `matplotlib` e `octave` permitem uma análise gráfica eficiente desses dados.

0.9 Caminhante Aleatório em Uma Dimensão

O **caminhante aleatório** é um dos modelos mais fundamentais e influentes na física estatística, matemática aplicada e ciência dos materiais. Seu estudo remonta ao século XIX, quando foi inicialmente proposto por Karl Pearson como uma forma de descrever o movimento de partículas em suspensão (movimento browniano). Desde então, o modelo tem sido amplamente utilizado para compreender fenômenos que envolvem difusão, transporte de cargas, dinâmica de preços no mercado financeiro, crescimento de interfaces, algoritmos de otimização e processos estocásticos em geral. O modelo representa, em sua forma mais simples, o movimento de uma partícula em uma linha unidimensional, onde a cada passo ela escolhe aleatoriamente entre mover-se para a esquerda ou para a direita. Apesar de sua simplicidade, esse sistema exibe uma série de propriedades estatísticas ricas e universais, que aparecem em contextos muito mais complexos.

0.9.1 Descrição do Modelo

Consideramos um caminhante que parte da origem $x_0 = 0$ e realiza uma sequência de movimentos discretos ao longo de uma linha. A cada instante de tempo, o caminhante se move uma unidade para a esquerda ou para a direita com igual probabilidade:

- Probabilidade $p = 0.5$ de um passo para a direita $(+1)$;

- Probabilidade $1 - p = 0.5$ de um passo para a esquerda (-1).

A posição do caminhante após j passos é dada pela relação de recorrência:

$$x_j = x_{j-1} + \delta x_j,$$

onde $\delta x_j \in \{-1, +1\}$ é uma variável aleatória que representa o deslocamento no passo j, e $x_0 = 0$ é a condição inicial.

0.9.2 Importância do Modelo

O estudo do caminhante aleatório fornece uma base conceitual para compreender diversos fenômenos naturais e artificiais, como:

- **Difusão de partículas**: Em fluidos ou meios porosos;
- **Movimento browniano**: Descrito por Einstein e observado em partículas microscópicas suspensas em líquidos;
- **Dinâmica de mercado**: Variações de preços podem ser modeladas como passeios aleatórios em certas aproximações;
- **Biologia**: Movimento de organismos simples, migração celular, ou busca por alimento;
- **Informação e computação**: Algoritmos randômicos e otimização estocástica.

Mesmo sendo um modelo de primeira aproximação, o caminhante aleatório capta as propriedades estatísticas essenciais de diversos sistemas complexos e serve como ponto de partida para generalizações mais sofisticadas, como caminhantes com viés, em várias dimensões, ou em ambientes com obstáculos e desordem. Nos tópicos seguintes, apresentaremos uma implementação computacional do modelo e analisaremos propriedades estatísticas como a média e o desvio padrão da posição, além da distribuição de probabilidade das posições finais.
onde:

- $x_0 = 0$ é a posição inicial;
- $\delta x_j \in \{-1, +1\}$ é o deslocamento no passo j, com probabilidade $p = 0.5$ para cada direção.

Durante a simulação, calculamos três quantidades principais:

- **Média da posição**: $\langle x \rangle$, média aritmética das posições;
- **Desvio padrão**: $\sigma = \sqrt{\langle x^2 \rangle - \langle x \rangle^2}$, que mede a dispersão das posições;
- **Histograma**: Distribuição de frequências das posições finais do caminhante após todos os passos.

O código a seguir implementa a simulação do caminhante aleatório para muitas amostras. Ele calcula o desvio padrão da posição a cada passo e salva a distribuição final em um histograma.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PASSOS 500        // Número de passos por caminhante
#define AMOSTRAS 400000   // Número de caminhantes
#define PROB 0.5          // Probabilidade de passo para a direita

float pos[PASSOS][AMOSTRAS]; // Matriz de posições

int main() {
    int i, j;
    float media, media2, sigma;
    FILE *fpSigma, *fpHist, *fpPos;

    // Abrir arquivos de saída
    fpSigma = fopen("sigmacaminhante.dat", "w");
    fpHist = fopen("histograma.dat", "w");
    fpPos = fopen("posicoescaminhante.dat", "w");

    // Inicialização das posições iniciais
    for (j = 0; j < AMOSTRAS; j++) {
        pos[0][j] = 0.0;
    }

    // Simulação dos passos
    for (j = 0; j < AMOSTRAS; j++) {
        for (i = 1; i < PASSOS; i++) {
            float r = rand() / (float)RAND_MAX;
            int passo = (r > PROB) ? 1 : -1;
            pos[i][j] = pos[i - 1][j] + passo;
        }
        fprintf(fpPos, "%f\n", pos[PASSOS - 1][j]);
    }

    // Cálculo do desvio padrão por passo
    for (i = 1; i < PASSOS; i++) {
        media = 0.0;
        media2 = 0.0;
        for (j = 0; j < AMOSTRAS; j++) {
            media += pos[i][j];
            media2 += pos[i][j] * pos[i][j];
        }
        media /= AMOSTRAS;
        media2 /= AMOSTRAS;
        sigma = sqrt(media2 - media * media);
        fprintf(fpSigma, "%d %f\n", i, sigma);
    }
```

```
    // Histograma das posições finais
    for (int x = -100; x <= 100; x += 4) {
        int contador = 0;
        for (j = 0; j < AMOSTRAS; j++) {
            if (fabs(pos[PASSOS - 1][j] - x) < 2) {
                contador++;
            }
        }
        if (contador > 0) {
            fprintf(fpHist, "%d %f\n", x, contador / (float)AMOSTRAS);
        }
    }

    // Fechar arquivos
    fclose(fpSigma);
    fclose(fpHist);
    fclose(fpPos);

    return 0;
}
```

O código acima simula uma grande quantidade de caminhantes (400.000), cada um realizando 500 passos. Ele fornece três saídas principais:

- `sigmacaminhante.dat`: contém o desvio padrão da posição ao longo do tempo. Espera-se um crescimento proporcional a $\sqrt{j}$, típico do passeio aleatório.
- `posicoescaminhante.dat`: lista a posição final de cada amostra.
- `histograma.dat`: mostra a distribuição de probabilidades das posições finais, que se aproxima de uma gaussiana centrada em zero.

Este modelo pode ser estendido para incluir viés, várias dimensões, barreiras e muitas outras variações relevantes em modelos de transporte, difusão, algoritmos de busca e mais.

0.10 Estatística Básica: Cálculo de Médias, Variância, Skewness e Curtose

Nesta seção, exploramos algumas das principais medidas estatísticas associadas a um conjunto de dados. Em diversas áreas do conhecimento, é comum lidar com conjuntos de dados provenientes de experimentos, simulações ou observações. Para entender o comportamento desses dados, é essencial calcular certas quantidades estatísticas que descrevem suas características fundamentais.

As medidas estatísticas básicas nos ajudam a responder perguntas como: qual é o valor médio observado? Quão dispersos estão os dados em torno da média? A distribuição é simétrica ou enviesada? Os dados apresentam caudas mais pesadas do que uma distribuição normal?

Suponha que temos uma série de dados $\{x_i\}$, com $i = 1, 2, \ldots, N$. As principais quantidades que vamos calcular são:

- **Média (valor esperado)**: representa a tendência central dos dados, isto é, o valor médio em torno do qual os dados se distribuem.

$$m_1 = \frac{1}{N}\sum_{i=1}^{N} x_i$$

- **Variância (dispersão)**: mede a dispersão dos dados em relação à média. Uma variância maior indica maior espalhamento dos dados.

$$m_2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - m_1)^2$$

- **Skewness (assimetria)**: indica o grau de simetria da distribuição dos dados. Valores positivos indicam uma cauda mais longa à direita, enquanto valores negativos apontam para uma cauda mais longa à esquerda.

$$a = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{x_i - m_1}{\sqrt{m_2}}\right)^3 = \frac{m_3}{m_2^{3/2}}$$

onde

$$m_3 = \frac{1}{N}\sum_{i=1}^{N}(x_i - m_1)^3$$

- **Curtose**: avalia o grau de "achatamento" da distribuição. Uma curtose elevada indica a presença de caudas mais pesadas (mais dados extremos), enquanto uma curtose menor sugere uma distribuição mais uniforme.

$$b = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{x_i - m_1}{\sqrt{m_2}}\right)^4 = \frac{m_4}{m_2^2}$$

onde

$$m_4 = \frac{1}{N}\sum_{i=1}^{N}(x_i - m_1)^4$$

O cálculo dessas quantidades fornece uma visão abrangente sobre o comportamento estatístico dos dados e constitui uma etapa inicial fundamental em qualquer análise quantitativa. Elas são amplamente utilizadas na validação de modelos, detecção de anomalias e tomada de decisões baseadas em dados.

0.10.1 Medidas estatísticas em uma série aleatória

Neste exemplo, ilustramos o cálculo de algumas medidas estatísticas fundamentais — média, variância, skewness (assimetria) e curtose — a partir de uma série de dados gerados aleatoriamente. Para isso, utilizamos um programa simples em linguagem C que gera uma sequência de $N = 1000$ números pseudoaleatórios com distribuição uniforme no intervalo $[0, 1]$. O objetivo é mostrar como essas quantidades podem ser computadas diretamente a partir dos dados, utilizando suas definições matemáticas. A média fornece uma estimativa do valor central da série, a variância mede a dispersão em torno da média, a skewness revela possíveis assimetrias da distribuição, enquanto a

curtose fornece informações sobre o formato das caudas da distribuição. Esse tipo de análise estatística é amplamente utilizado em contextos experimentais e computacionais para caracterizar propriedades fundamentais de conjuntos de dados, especialmente em sistemas físicos, econômicos ou biológicos onde padrões ocultos podem emergir a partir de flutuações estatísticas. O código a seguir implementa essas ideias.

```
/* estatisticas.c - Calcula média, variância, skewness e curtose */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 1000

double v[N];

int main() {
    int i;
    double m1 = 0.0, m2 = 0.0, m3 = 0.0, m4 = 0.0;
    double skewness, curtose;

    // Gerar dados aleatórios uniformes entre 0 e 1
    for (i = 0; i < N; i++) {
        v[i] = (double)rand() / RAND_MAX;
        m1 += v[i];
    }

    m1 /= N;

    for (i = 0; i < N; i++) {
        double delta = v[i] - m1;
        m2 += delta * delta;
        m3 += delta * delta * delta;
        m4 += delta * delta * delta * delta;
    }

    m2 /= N;
    m3 /= N;
    m4 /= N;

    skewness = m3 / pow(m2, 1.5);
    curtose = m4 / (m2 * m2);

    printf("Média     = %f\n", m1);
    printf("Variância = %f\n", m2);
    printf("Skewness  = %f\n", skewness);
    printf("Curtose   = %f\n", curtose);

    return 0;
}
```

0.10.2 Normalização de uma série de dados a partir de um arquivo

Neste exemplo, lidamos com o problema prático de normalizar uma série de dados numéricos previamente armazenados em um arquivo. Suponha que temos uma tabela chamada `dados.dat`, contendo uma única coluna com N números reais aleatórios distribuídos uniformemente entre 0 e 1. O objetivo é processar essa tabela da seguinte forma:

1. Ler os dados do arquivo e armazená-los em um vetor.
2. Calcular a média e a variância dos dados.
3. Aplicar a normalização em cada elemento da série, isto é, transformar os dados para que tenham média zero e variância igual a um.
4. Salvar os dados normalizados em um novo arquivo chamado `dadosnormalizados.dat`.

A normalização de dados é uma etapa importante em muitas aplicações de análise estatística e aprendizado de máquina, pois elimina unidades e facilita comparações, além de garantir estabilidade numérica em algoritmos que dependem da escala dos dados. O programa a seguir, escrito em C básico e bem comentado, realiza todas as etapas acima. Ele supõe que o arquivo `dados.dat` já existe e contém exatamente N linhas com um número real por linha.

```
/* normaliza_arquivo.c - Lê dados de um arquivo, normaliza e salva em outro */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 1000  // número de elementos esperados

double v[N], vnorm[N];

int main() {
    FILE *fin, *fout;
    int i;
    double media = 0.0, variancia = 0.0, desvio;

    // Abrir arquivo de entrada
    fin = fopen("dados.dat", "r");
    if (fin == NULL) {
        printf("Erro ao abrir o arquivo dados.dat\n");
        return 1;
    }

    // Ler os dados e calcular a soma para a média
    for (i = 0; i < N; i++) {
        if (fscanf(fin, "%lf", &v[i]) != 1) {
            printf("Erro na leitura dos dados\n");
            fclose(fin);
            return 1;
        }
        media += v[i];
    }
```

```
    fclose(fin);

    media /= N;

    // Calcular a variância
    for (i = 0; i < N; i++) {
        variancia += (v[i] - media) * (v[i] - media);
    }
    variancia /= N;
    desvio = sqrt(variancia);

    // Normalizar os dados
    for (i = 0; i < N; i++) {
        vnorm[i] = (v[i] - media) / desvio;
    }

    // Escrever os dados normalizados em outro arquivo
    fout = fopen("dadosnormalizados.dat", "w");
    if (fout == NULL) {
        printf("Erro ao criar o arquivo dadosnormalizados.dat\n");
        return 1;
    }

    for (i = 0; i < N; i++) {
        fprintf(fout, "%f\n", vnorm[i]);
    }
    fclose(fout);



    return 0;
}
```

Esses dois exemplos ilustram como calcular de forma simples e eficiente as principais quantidades estatísticas utilizando a linguagem C. O domínio dessas ferramentas é fundamental para qualquer pessoa que trabalhe com análise de dados, seja em contextos acadêmicos ou profissionais. O cálculo de medidas como média, variância, skewness e curtose permite descrever e interpretar adequadamente o comportamento de conjuntos de dados, identificando padrões, tendências e possíveis anomalias. Essas estatísticas básicas são a base para análises mais complexas e ajudam a construir uma compreensão sólida sobre os dados antes de aplicar qualquer modelo ou método mais avançado. A normalização dos dados, por sua vez, é uma etapa essencial em muitas áreas, como aprendizado de máquina, análise de séries temporais e processamento de sinais. Ao transformar os dados para que tenham média zero e variância unitária, garantimos que diferentes variáveis contribuam de forma equilibrada nas análises, evitando que escalas distintas distorçam os resultados ou prejudiquem a convergência de algoritmos. Saber implementar essas operações em uma linguagem como C proporciona controle total sobre o fluxo de dados e o desempenho dos cálculos, o que é especialmente importante em aplicações que exigem eficiência computacional. Além disso, exercita o pensamento algorítmico e reforça a compreensão conceitual dos métodos estatísticos.

0.10.3 Cálculo da Autocorrelação

Considere uma série de dados $\{x_i\}$ com $i = 1, 2, 3, ..., N$. A autocorrelação intrínseca nesta série é definida por:

$$C(r) = \langle x_i x_{i+r} \rangle - \langle x_i \rangle \langle x_{i+r} \rangle \tag{10}$$

Temos que:

$$\langle x_i x_{i+r} \rangle = \frac{1}{N-r} \sum_{i=1}^{N-r} x_i x_{i+r} \tag{11}$$

e

$$\langle x_i \rangle = \langle x_{i+r} \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{12}$$

Logo:

$$C(r) = \frac{1}{N-r} \sum_{i=1}^{N-r} x_i x_{i+r} - \left(\frac{1}{N} \sum_{i=1}^{N} x_i \right)^2 \tag{13}$$

Considere que os dados $\{x_i\}$ são números aleatórios Gaussianos com $\langle x_i \rangle = 0$ e $\langle x_i^2 \rangle = 1$. Temos que para $r = 0$ a autocorrelação fica $C(0) = 1$ e para $r > 0$ temos que $C(r) \to 0$. A seguir, apresentamos um programa simples em linguagem C que gera uma série de números aleatórios usando um gerador congruente linear, e em seguida calcula a autocorrelação da série para $r = 0$ até $r = 20$.

```
#include <stdio.h>
#include <stdlib.h>
#define N 10000
#define A 1664525
#define C 1013904223
#define M 4294967296 // 2^32

unsigned int seed = 12345; // semente inicial

// Gerador congruente linear
double lcg() {
    seed = (A * seed + C) % M;
    return (double)seed / (double)M;
}

int main() {
    double x[N];
    double media = 0.0;
    int i, r;

    // Gera a série aleatória
    for (i = 0; i < N; i++) {
        x[i] = lcg();
        media += x[i];
    }
    media = media / N;
```

```
    // Calcula a autocorrelação até r=20
    for (r = 0; r <= 20; r++) {
        double soma = 0.0;
        for (i = 0; i < N - r; i++) {
            soma += (x[i] - media) * (x[i + r] - media);
        }
        soma = soma / (N - r);
        printf("r = %2d, C(r) = %f\n", r, soma);
    }

    return 0;
}
```

O programa acima mostra que para $r = 0$, a autocorrelação $C(0)$ corresponde à variância da série (próxima de 1 no caso de dados normalizados). Para valores maiores de r, espera-se que $C(r)$ se aproxime de zero para dados aleatórios não correlacionados. Isso serve como um teste numérico da aleatoriedade da série gerada e é um instrumento fundamental em análises de ruído e desordem em sistemas físicos.

0.11 Desordem Correlacionada

A presença de desordem em sistemas físicos é um tema de grande relevância, especialmente em contextos como localização de Anderson, transporte em materiais desordenados, vidros de spin, e mais recentemente, em simulações de matéria condensada e informação quântica. No entanto, a maioria dos estudos clássicos trata da desordem como puramente *não correlacionada*, ou seja, valores aleatórios independentes em cada ponto do sistema. Em muitos sistemas reais, entretanto, a desordem apresenta correlações espaciais. Essa **desordem correlacionada** pode afetar drasticamente as propriedades físicas do sistema, como os espectros de energia, a condutividade elétrica e o comportamento dinâmico de partículas.

0.11.1 Definição e Função de Correlação

Considere uma série de variáveis aleatórias $\{X_i\}$ definidas ao longo de um sistema unidimensional. Para caracterizar estatisticamente a dependência espacial entre essas variáveis, introduzimos a **função de correlação de dois pontos**, definida como:

$$C(r) = \langle X_i X_{i+r} \rangle - \langle X_i \rangle^2,$$

onde r representa a distância entre dois pontos da série, e $\langle \cdot \rangle$ denota uma média sobre o índice i, assumindo que o processo seja estacionário — isto é, que as propriedades estatísticas não dependem da posição absoluta i, mas apenas da separação relativa r. A função $C(r)$ quantifica o grau de correlação entre os valores da série separados por uma distância r: valores próximos de zero indicam ausência de correlação, enquanto valores positivos ou negativos significativos revelam correlação direta ou inversa, respectivamente. Em diversos contextos físicos, a desordem presente em um sistema não é puramente aleatória, mas apresenta correlações espaciais que podem influenciar fortemente suas propriedades dinâmicas e espectrais. A seguir, descrevemos alguns dos principais modelos utilizados para gerar séries com desordem correlacionada, juntamente com suas respectivas implementações numéricas.

0.11.2 Modelo AR(1) (Auto-Regressivo de Primeira Ordem)

Um dos modelos mais simples e amplamente utilizados para gerar séries com desordem correlacionada é o **modelo autorregressivo de ordem 1** (AR(1)). Nesse modelo, a série $\{X_i\}$ é definida recursivamente pela relação:

$$X_i = \phi X_{i-1} + \epsilon_i,$$

onde ϵ_i são variáveis aleatórias independentes e identicamente distribuídas, com média zero e variância σ^2. O parâmetro ϕ controla o grau de correlação entre os termos consecutivos da série e deve satisfazer $|\phi| < 1$ para garantir a estacionariedade do processo. A função de correlação $C(r)$ para esse modelo decai exponencialmente com a distância r, sendo dada por:

$$C(r) = \sigma^2 \frac{\phi^r}{1 - \phi^2}.$$

Esse comportamento exponencial reflete o fato de que os termos da série são mais fortemente correlacionados quando próximos, com a correlação diminuindo rapidamente à medida que r aumenta. O modelo AR(1) é frequentemente utilizado como ponto de partida para estudos de desordem com correlações de curto alcance.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 1000

double phi = 0.9; // parâmetro de correlação
double X[N];

int main() {
    srand(time(NULL));
    X[0] = ((double)rand()/RAND_MAX)*2 - 1; // valor inicial entre -1 e 1

    for (int i = 1; i < N; i++) {
        double noise = ((double)rand()/RAND_MAX)*2 - 1; // ruído branco
        X[i] = phi * X[i-1] + noise;
    }

    FILE *f = fopen("ar1.dat", "w");
    for (int i = 0; i < N; i++) {
        fprintf(f, "%d %f\n", i, X[i]);
    }
    fclose(f);
    return 0;
}
```

0.11.3 Modelo AR(2) (Auto-Regressivo de Segunda Ordem)

O modelo autorregressivo de segunda ordem (AR(2)) é uma generalização do modelo AR(1), permitindo que cada termo da série dependa linearmente dos dois termos anteriores. A série $\{X_i\}$ é definida recursivamente pela equação:

$$X_i = a_1 X_{i-1} + a_2 X_{i-2} + \epsilon_i,$$

onde a_1 e a_2 são os coeficientes autorregressivos e ϵ_i representa uma sequência de variáveis aleatórias independentes, com média zero e variância σ^2. A inclusão do segundo termo autorregressivo (X_{i-2}) permite que o modelo capture estruturas de correlação mais complexas do que aquelas reproduzidas por um modelo AR(1), incluindo correlações oscilatórias ou amortecidas, dependendo dos valores dos coeficientes a_1 e a_2. A função de correlação $C(r)$ neste caso não possui uma forma analítica simples, mas pode exibir comportamentos oscilatórios, exponencialmente amortecidos ou críticos. A estabilidade e estacionariedade do processo AR(2) requerem que as raízes dessa equação estejam dentro do círculo unitário no plano complexo. Esse modelo é especialmente útil quando se deseja simular desordem com padrões mais ricos de correlação.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 1000

double a1 = 1.2, a2 = -0.7;
```

```
double X[N];

int main() {
    srand(time(NULL));
    X[0] = ((double)rand()/RAND_MAX)*2 - 1;
    X[1] = ((double)rand()/RAND_MAX)*2 - 1;

    for (int i = 2; i < N; i++) {
        double noise = ((double)rand()/RAND_MAX)*2 - 1;
        X[i] = a1 * X[i-1] + a2 * X[i-2] + noise;
    }

    FILE *f = fopen("ar2.dat", "w");
    for (int i = 0; i < N; i++) {
        fprintf(f, "%d %f\n", i, X[i]);
    }
    fclose(f);
    return 0;
}
```

0.11.4 Desordem com Correlação Exponencial: $C(r) \sim e^{-r/L}$

Uma maneira eficaz de gerar uma série aleatória com função de correlação exponencial é por meio da convolução de ruído branco com um **filtro gaussiano**. A função de correlação desejada é dada por:

$$C(r) = \exp\left(-\frac{r}{L}\right),$$

onde L representa o comprimento de correlação — ou seja, a escala típica ao longo da qual os elementos da série permanecem correlacionados. O procedimento pode ser descrito em três etapas principais:

1. **Geração do ruído branco:** cria-se uma sequência $\{\eta_i\}$ de variáveis aleatórias independentes com média zero e variância unitária.

2. **Aplicação do filtro gaussiano:** cada ponto X_i da nova série é obtido como uma média ponderada dos elementos η_j, com pesos dados por uma gaussiana centrada em i:

$$X_i = \sum_j \eta_j \exp\left(-\frac{(i-j)^2}{2L^2}\right).$$

   Esse processo suaviza o ruído branco e introduz correlações espaciais com decaimento exponencial.

3. **Normalização:** por fim, a série $\{X_i\}$ é normalizada para garantir que tenha média zero e variância igual a 1. Isso é feito subtraindo-se a média da série e dividindo-se pelo desvio padrão.

Esse método é amplamente utilizado na modelagem de desordem espacialmente correlacionada em sistemas físicos, e o parâmetro L pode ser ajustado para controlar o grau de correlação entre os elementos da série.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 1000
#define L 10.0

double eta[N], X[N];

int main() {
    srand(time(NULL));

    // Ruído branco
    for (int i = 0; i < N; i++) {
        eta[i] = ((double)rand()/RAND_MAX)*2 - 1;
    }

    // Aplicação do filtro gaussiano
    for (int i = 0; i < N; i++) {
        X[i] = 0.0;
        for (int j = 0; j < N; j++) {
            double weight = exp(-pow(i - j, 2)/(2*L*L));
            X[i] += eta[j] * weight;
        }
    }

    // Normalização: média 0, variância 1
    double mean = 0.0;
    for (int i = 0; i < N; i++) mean += X[i];
    mean /= N;

    double var = 0.0;
    for (int i = 0; i < N; i++) {
        X[i] -= mean;
        var += X[i] * X[i];
    }
    var = sqrt(var / N);

    for (int i = 0; i < N; i++) X[i] /= var;

    FILE *f = fopen("exp_corr.dat", "w");
    for (int i = 0; i < N; i++) {
        fprintf(f, "%d %f\n", i, X[i]);
    }
    fclose(f);
    return 0;
}
```

0.11.5 Desordem com Correlação Exponencial via Decomposição de Cholesky

Uma abordagem alternativa para gerar desordem com correlação espacial é utilizar a **decomposição de Cholesky** da matriz de covariância. Esse método é mais direto do ponto de vista matemático e permite gerar uma série com correlação prescrita, assumindo que a matriz de correlação seja definida positiva. Consideramos novamente a função de correlação exponencial:

$$C(r) = \exp\left(-\frac{|i-j|}{L}\right),$$

onde L é o comprimento de correlação. A matriz de covariância $C_{ij} = C(|i-j|)$ é então construída para um tamanho N fixado. O procedimento é o seguinte:

1. Construir a matriz de covariância $C_{ij} = \exp(-|i-j|/L)$;
2. Obter a decomposição de Cholesky $C = LL^T$;
3. Gerar um vetor η de ruído branco com média zero e variância unitária;
4. Obter a série correlacionada $X = L\eta$;
5. Normalizar a série para média zero e variância 1.

Abaixo apresentamos um exemplo básico em linguagem C para gerar desordem com essa técnica.

```
/* Gera uma série de desordem com correlação exponencial usando Cholesky */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 100     // Tamanho da série
#define L 10.0    // Comprimento de correlação

double C[N][N];   // Matriz de covariância
double Lmat[N][N]; // Matriz de Cholesky
double eta[N];    // Ruído branco
double X[N];      // Série correlacionada

// Função para gerar ruído branco com média 0 e variância 1
void gerar_ruido_branco() {
    for (int i = 0; i < N; i++) {
        double u1 = (rand() + 1.0) / (RAND_MAX + 2.0);
        double u2 = (rand() + 1.0) / (RAND_MAX + 2.0);
        eta[i] = sqrt(-2 * log(u1)) * cos(2 * M_PI * u2); // Box-Muller
    }
}

// Constrói a matriz de covariância com correlação exponencial
void construir_covariancia() {
```

```
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = exp(-fabs(i - j) / L);
        }
    }
}

// Decomposição de Cholesky: C = L * L^T
void cholesky() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j <= i; j++) {
            double soma = C[i][j];
            for (int k = 0; k < j; k++)
                soma -= Lmat[i][k] * Lmat[j][k];
            if (i == j)
                Lmat[i][j] = sqrt(soma);
            else
                Lmat[i][j] = soma / Lmat[j][j];
        }
    }
}

// Calcula X = L * eta
void gerar_desordem() {
    for (int i = 0; i < N; i++) {
        X[i] = 0.0;
        for (int j = 0; j <= i; j++) {
            X[i] += Lmat[i][j] * eta[j];
        }
    }
}

// Normaliza X para média zero e variância 1
void normalizar() {
    double soma = 0.0, soma2 = 0.0;
    for (int i = 0; i < N; i++) soma += X[i];
    double media = soma / N;

    for (int i = 0; i < N; i++) {
        X[i] -= media;
        soma2 += X[i] * X[i];
    }

    double desvio = sqrt(soma2 / N);
    for (int i = 0; i < N; i++) X[i] /= desvio;
}

int main() {
    gerar_ruido_branco();
```

```
    construir_covariancia();
    cholesky();
    gerar_desordem();
    normalizar();

    FILE *f = fopen("desordem_cholesky.dat", "w");
    for (int i = 0; i < N; i++)
        fprintf(f, "%d %f\n", i, X[i]);
    fclose(f);

    return 0;
}
```

Esse código gera uma série com correlação exponencial controlada pelo parâmetro L, utilizando o método de Cholesky para aplicar a correlação diretamente via a matriz de covariância. O resultado é salvo no arquivo `desordem_cholesky.dat`.

0.11.6 Comparação entre Séries AR(1) e Desordem Correlacionada via Decomposição de Cholesky

Nesta sub-seção , vamos comparar duas formas distintas de gerar séries com correlação espacial : uma baseada em processos auto-regressivos de primeira ordem (AR(1)) e outra fundamentada na geração de desordem correlacionada por meio da decomposição de Cholesky de uma matriz de covariância. O objetivo é mostrar que, embora ambos os métodos gerem séries com correlações exponenciais, suas estruturas internas e o controle da correlação diferem significativamente. A decomposição de Cholesky é uma técnica matemática que permite gerar vetores aleatórios com covariância conhecida. A matriz de covariância utilizada tem a forma $C_{ij} = \exp(-|i-j|/L)$, onde L é o comprimento de correlação. Por outro lado, o modelo AR(1) é uma forma iterativa de gerar uma série onde cada novo valor depende linearmente do valor anterior mais um termo de ruído branco. O parâmetro de correlação ρ é escolhido de modo que a correlação entre vizinhos imediatos seja compatível com a obtida via Cholesky. O programa abaixo, escrito em linguagem C, realiza as seguintes tarefas:

- Geração de ruído branco com média zero e variância unitária.
- Construção da matriz de covariância e sua decomposição de Cholesky.
- Geração da série com correlação espacial via multiplicação da matriz de Cholesky pelo ruído branco.
- Geração da série AR(1) com o mesmo comprimento de correlação.
- Normalização de ambas as séries para média zero e variância 1.
- Cálculo da função de autocorrelação de ambas as séries, até uma distância máxima definida.
- Escrita dos resultados em arquivos de dados.
- Visualização comparativa da função de autocorrelação das duas séries utilizando Gnuplot.

A seguir, apresentamos o código-fonte completo do programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 5000     // Tamanho da série
#define L 20.0     // Comprimento de correlação
#define MAX_LAG 25 // Máxima distância para autocorrelação

double C[N][N];     // Matriz de covariância
double Lmat[N][N];  // Matriz de Cholesky
double eta[N];      // Ruído branco
double X1[N];       // Série com correlação via Cholesky
double X2[N];       // Série AR(1)

void gerar_ruido_branco(double *v) {
    for (int i = 0; i < N; i++) {
        double u1 = (rand() + 1.0) / (RAND_MAX + 2.0);
        double u2 = (rand() + 1.0) / (RAND_MAX + 2.0);
        v[i] = sqrt(-2 * log(u1)) * cos(2 * M_PI * u2);
    }
}

void construir_covariancia() {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            C[i][j] = exp(-fabs(i - j) / L);
}

void cholesky() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j <= i; j++) {
            double soma = C[i][j];
            for (int k = 0; k < j; k++)
                soma -= Lmat[i][k] * Lmat[j][k];
            if (i == j)
                Lmat[i][j] = sqrt(soma);
            else
                Lmat[i][j] = soma / Lmat[j][j];
        }
    }
}

void gerar_desordem_cholesky() {
    for (int i = 0; i < N; i++) {
        X1[i] = 0.0;
        for (int j = 0; j <= i; j++)
            X1[i] += Lmat[i][j] * eta[j];
    }
}
```

```
void normalizar(double *X) {
    double soma = 0.0, soma2 = 0.0;
    for (int i = 0; i < N; i++) soma += X[i];
    double media = soma / N;

    for (int i = 0; i < N; i++) {
        X[i] -= media;
        soma2 += X[i] * X[i];
    }

    double desvio = sqrt(soma2 / N);
    for (int i = 0; i < N; i++) X[i] /= desvio;
}

void autocorrelacao(double *X, const char *filename) {
    FILE *f = fopen(filename, "w");
    for (int lag = 0; lag <= MAX_LAG; lag++) {
        double soma = 0.0;
        for (int i = 0; i < N - lag; i++)
            soma += X[i] * X[i + lag];
        soma /= (N - lag);
        fprintf(f, "%d %f\n", lag, soma);
    }
    fclose(f);
}

void gerar_AR1(double rho) {
    gerar_ruido_branco(eta);
    X2[0] = eta[0];
    for (int i = 1; i < N; i++)
        X2[i] = rho * X2[i - 1] + eta[i];
}

int main() {
    gerar_ruido_branco(eta);
    construir_covariancia();
    cholesky();
    gerar_desordem_cholesky();
    normalizar(X1);
    autocorrelacao(X1, "autocor_cholesky.dat");

    double rho = exp(-1.0 / L);
    gerar_AR1(rho);
    normalizar(X2);
    autocorrelacao(X2, "autocor_ar1.dat");

    FILE *f1 = fopen("serie_cholesky.dat", "w");
    FILE *f2 = fopen("serie_ar1.dat", "w");
    for (int i = 0; i < N; i++) {
        fprintf(f1, "%d %f\n", i, X1[i]);
        fprintf(f2, "%d %f\n", i, X2[i]);
    }
    fclose(f1);
```

```
    fclose(f2);

    FILE *gp = popen("gnuplot -persist", "w");
    fprintf(gp,
        "set title 'Autocorrelação: Cholesky vs AR(1)'\n"
        "set xlabel 'Distância'\n"
        "set ylabel 'C(r)'\n"
        "set grid\n"
        "plot 'autocor_cholesky.dat' with lines title 'Cholesky', \\\n"
        "     'autocor_ar1.dat' with lines title 'AR(1)'\n");
    pclose(gp);

    return 0;
}
```

A execução deste programa resultará na criação de quatro arquivos de dados: duas séries (`serie_cholesky.da` e `serie_ar1.dat`) e duas autocorrelações (`autocor_cholesky.dat` e `autocor_ar1.dat`). Além disso, o programa invoca o `gnuplot` para exibir, automaticamente, um gráfico comparativo das funções de autocorrelação de ambas as séries. Essa visualização ajuda a verificar se as propriedades estatísticas são compatíveis entre os dois métodos.

Desordem correlacionada aparece em diversos contextos físicos, como em materiais semicondutores amorfos, cadeias de spin quânticas com acoplamentos espaciais variáveis, sistemas ópticos e eletrônicos com estruturas fabricadas artificialmente (como quasicristais e cristais fotônicos), além de modelos de localização quântica nos quais a presença de correlação na desordem pode impedir a localização total das funções de onda. Notavelmente, a introdução de correlação espacial pode ainda levar à emergência de fases críticas e transições de fase, mesmo em sistemas unidimensionais, o que seria proibido segundo o cenário clássico de Anderson para desordem puramente aleatória.

0.11.7 Desordem correlacionada com densidade espectral do tipo lei de potência

Nesta sub-seção, apresentaremos o formalismo para gerar uma sequência temporal aleatória ϵ_i com densidade espectral do tipo lei de potência $S(\omega) \propto 1/\omega^{\alpha}$. Tal abordagem é útil para introduzir correlações de longo alcance na desordem de modelos como o de Anderson. A posição da partícula (ou, neste caso, a desordem nos sítios ϵ_i) é observada em tempos discretos $t_i = i\tau$, com $i = 0, 1, \ldots, N-1$. A densidade espectral desejada é imposta na série por meio da soma de componentes harmônicas com fases aleatórias, conforme:

$$\epsilon_i = \sum_{k=1}^{N/2} \frac{\mathcal{C}(\alpha)}{k^{\alpha/2}} \cos\left(\frac{2\pi i k}{N} + \phi_k\right), \tag{14}$$

onde as fases ϕ_k são variáveis aleatórias distribuídas uniformemente em $[0, 2\pi]$, e a constante $\mathcal{C}(\alpha)$ é escolhida de forma que a sequência ϵ_i tenha média zero e variância unitária:

$$\langle \epsilon_i \rangle = 0, \quad \Delta\epsilon_i = \sqrt{\langle \epsilon_i^2 \rangle - \langle \epsilon_i \rangle^2} = 1. \tag{15}$$

A seguir, apresentamos um programa simples em linguagem C que gera essa sequência de desordem correlacionada. O valor de α é fornecido pelo usuário na linha de comando. O programa gera as N energias ϵ_i e as escreve no arquivo `desordem.dat`.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 1000  // Tamanho da cadeia
#define PI 3.141592653589793

double epsilon[N];  // Sequência de desordem

// Função para gerar número aleatório entre 0 e 2pi
double random_phase() {
    return ((double) rand() / RAND_MAX) * 2.0 * PI;
}

// Função principal
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Uso: %s <alpha>\n", argv[0]);
        return 1;
    }

    double alpha = atof(argv[1]);
    srand(time(NULL));  // Inicializa gerador de números aleatórios

    // Gera a sequência bruta
    for (int i = 0; i < N; i++) {
        epsilon[i] = 0.0;
        for (int k = 1; k <= N/2; k++) {
            double phi = random_phase();
            double coef = 1.0 / pow((double)k, alpha/2.0);
            epsilon[i] += coef * cos(2.0 * PI * i * k / N + phi);
        }
    }

    // Normalização: média zero
    double soma = 0.0;
    for (int i = 0; i < N; i++) soma += epsilon[i];
    double media = soma / N;
    for (int i = 0; i < N; i++) epsilon[i] -= media;

    // Normalização: variância 1
    double soma2 = 0.0;
    for (int i = 0; i < N; i++) soma2 += epsilon[i] * epsilon[i];
    double desvio = sqrt(soma2 / N);
    for (int i = 0; i < N; i++) epsilon[i] /= desvio;
```

```
    // Salva os dados no arquivo
    FILE *fp = fopen("desordem.dat", "w");
    if (fp == NULL) {
        perror("Erro ao abrir arquivo");
        return 1;
    }
    for (int i = 0; i < N; i++) {
        fprintf(fp, "%d\t%f\n", i, epsilon[i]);
    }
    fclose(fp);

    printf("Sequência gerada com média = 0 e variância = 1 no arquivo 'desordem.dat'.\n");

    return 0;
}
```

Este código é uma implementação direta da Eq.(14), com normalização da média e da variância ao final da geração da série. A série gerada é útil para simulações de modelos com desordem espacialmente correlacionada, como em sistemas quânticos unidimensionais desordenados.

0.12 Integração de Monte Carlo

A integração de Monte Carlo é uma técnica numérica utilizada para estimar integrais por meio de amostragem aleatória. É especialmente útil para integrais de alta dimensão ou em domínios complexos, onde métodos tradicionais tornam-se ineficazes. A ideia é estimar a integral de uma função $f(x)$ pela média dos valores da função em pontos aleatórios dentro do intervalo de integração. A seguir, apresentamos o procedimento básico para estimar uma integral do tipo:

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

usando o método de Monte Carlo padrão:

1. Gerar N pontos aleatórios $x_i \in [a, b]$.
2. Calcular $f(x_i)$ para cada ponto.
3. Estimar a integral como:

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

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

Vamos estimar numericamente a integral:

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

O seguinte código em linguagem C implementa essa estimativa pelo método de Monte Carlo padrão:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 10000
double a = 0.0, b = 2.0;

double f(double x) {
    return x * x;
}

double monte_carlo_integracao() {
    double soma = 0.0, x;
    for (int i = 0; i < N; i++) {
        x = a + (b - a) * ((double) rand() / RAND_MAX);
        soma += f(x);
    }
    return (b - a) * soma / N;
}
```

```
int main() {
    srand(time(NULL));
    double resultado = monte_carlo_integracao();
    printf("Estimativa da integral de x^2 em [0,2] (Monte Carlo): %.5f\n", resultado);
    return 0;
}
```

Com um número suficientemente grande de amostras N, o resultado se aproxima de $\frac{8}{3}$. A simplicidade e generalidade do método tornam-no ideal para aplicações em diversas áreas da ciência e engenharia.

0.12.2 Método da Amostragem por Rejeição

A amostragem por rejeição é uma variação do método de Monte Carlo que utiliza uma abordagem geométrica para estimar integrais. A ideia é gerar pontos (x, y) aleatórios dentro de um retângulo que contém a área sob a curva da função $f(x)$, e contar quantos pontos caem sob a curva.

1. Definimos um retângulo $[a, b] \times [0, M]$, onde $M \geq \max f(x)$.
2. Geramos N pares (x, y), com $x \in [a, b]$ e $y \in [0, M]$.
3. Contamos quantos pontos satisfazem $y \leq f(x)$.
4. A área sob a curva é estimada por:

$$I \approx \frac{n_{\text{aceitos}}}{N} \cdot (b - a) \cdot M$$

O código C abaixo implementa esse método para a função $f(x) = x^2$ em $[0, 2]$, usando $M = 4$, pois $x^2 \leq 4$ nesse intervalo:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 10000
double a = 0.0, b = 2.0;
double M = 4.0;

double f(double x) {
    return x * x;
}

double amostragem_rejeicao() {
    int aceitos = 0;
    double x, y;
    for (int i = 0; i < N; i++) {
        x = a + (b - a) * ((double) rand() / RAND_MAX);
        y = M * ((double) rand() / RAND_MAX);
```

```
        if (y <= f(x)) {
            aceitos++;
        }
    }
    double area_retangulo = (b - a) * M;
    return area_retangulo * ((double) aceitos / N);
}

int main() {
    srand(time(NULL));
    double resultado = amostragem_rejeicao();
    printf("Estimativa da integral de x^2 em [0,2] (Amostragem por Rejeição): %.5f\n", res
    return 0;
}
```

Esse método pode ser útil para funções complicadas ou em casos onde não se conhece a primitiva da função. No entanto, ele pode ser menos eficiente que o método padrão quando $f(x)$ ocupa uma pequena fração do retângulo $[a, b] \times [0, M]$. Ambos os métodos apresentados ilustram bem a flexibilidade da técnica de Monte Carlo para integração. O método padrão é geralmente mais eficiente em uma dimensão, mas o método de rejeição pode ser estendido para gerar amostras com distribuições complexas, além de servir para visualizar regiões sob curvas. A precisão pode ser melhorada com o aumento de N, mas há um custo computacional associado.

0.13 Transformada de Fourier: Fundamentos Teóricos e Aplicações

A **Transformada de Fourier** é uma das ferramentas matemáticas mais poderosas e versáteis para a análise de funções, sinais e sistemas físicos. Ela permite decompor uma função arbitrária no domínio do tempo (ou do espaço) em uma soma de ondas harmônicas, revelando como diferentes frequências contribuem para a formação do sinal original. Isso é particularmente útil em áreas como física, engenharia elétrica, análise de séries temporais, mecânica quântica e processamento de imagens.

0.13.1 Definição Matemática

Seja $f(x)$ uma função complexa (ou real) integrável, a transformada de Fourier contínua é definida como:

$$F(k) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i k x}\, dx$$

onde:

- x representa a variável original (tempo, posição, etc.);
- k é a frequência espacial ou temporal (ou número de onda, dependendo do contexto físico);
- i é a unidade imaginária, $i^2 = -1$;
- $F(k)$ é a amplitude complexa da componente de frequência k.

A transformada inversa, que reconstrói a função original a partir do espectro de frequências, é dada por:

$$f(x) = \int_{-\infty}^{\infty} F(k)\, e^{2\pi i k x}\, dk$$

Essas expressões revelam a dualidade entre os domínios do tempo/espaço e da frequência. Em sistemas físicos, a transformada de Fourier é frequentemente usada para resolver equações diferenciais, estudar oscilações e analisar espectros de energia.

0.13.2 Transformada de Fourier em Dados Discretos

Na prática, muitas vezes dispomos apenas de uma amostra discreta de uma função, isto é, conhecemos valores $\{x_i, y_i = f(x_i)\}$ em um conjunto finito de pontos. Nesses casos, a versão contínua da transformada é aproximada numericamente por uma soma discreta:

$$F(f) \approx \sum_{i=1}^{N} y_i\, e^{-i x_i f}$$

onde:

- f é a frequência contínua que percorre um intervalo definido;
- x_i são os pontos da variável independente (não necessariamente uniformemente espaçados);
- y_i são os valores da função nesses pontos;

- o termo $e^{-ix_i f}$ realiza a projeção da função em componentes harmônicas de frequência f.

Essa expressão representa uma versão não-uniforme da transformada de Fourier, pois não pressupõe espaçamento constante entre os x_i. Esse tipo de abordagem é útil, por exemplo, em dados experimentais com espaçamento irregular, ou em métodos espectrais com malhas adaptativas. O programa em linguagem C apresentado abaixo implementa precisamente essa aproximação para a transformada de Fourier. Ele percorre um intervalo de frequências $f \in [0.05, 2.0]$, com passo $\Delta f = 0.025$, e calcula o módulo da transformada (isto é, a magnitude do espectro em cada frequência). Os dados de entrada são lidos de um arquivo chamado `entrada.dat`, contendo ao menos quatro colunas. O programa utiliza apenas as duas primeiras colunas x_i e y_i, ignorando as demais. O resultado, contendo os pares $(f, |F(f)|)$, é gravado em um arquivo chamado `saida.dat`. Esse método direto não utiliza FFT (Fast Fourier Transform), pois permite lidar com pontos x_i arbitrários. Embora menos eficiente para grandes volumes de dados igualmente espaçados, ele é mais flexível para aplicações gerais e análise espectral precisa.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <complex.h>

int main() {
    FILE *fil, *fil1;
    double r1, r2, f, df;
    double _Complex cc1, ic;
    double x[300000], y[300000];
    int i, N;

    // Abrir arquivo de entrada com dados (x, y, ... , ...)
    fil = fopen("entrada.dat", "r");
    if (fil == NULL) {
        printf("Erro ao abrir entrada.dat\n");
        return 1;
    }

    // Abrir arquivo de saída
    fil1 = fopen("saida.dat", "w");
    if (fil1 == NULL) {
        printf("Erro ao abrir saida.dat\n");
        return 1;
    }

    ic = 0.0 + I; // Unidade imaginária

    // Leitura dos dados do arquivo (ignorando 3ª e 4ª colunas)
    i = 1;
    while ((fscanf(fil, "%lf %lf %lf %lf\n", &x[i], &y[i], &r1, &r2)) != EOF) {
        i = i + 1;
    }
    N = i - 1; // Número de pontos lidos

    // Parâmetro da frequência
    df = 0.025;
```

```
    // Loop sobre as frequências f
    for (f = 0.05; f <= 2.0; f += df) {
        cc1 = 0.0 + I*0.0;

        // Cálculo da soma: Transformada de Fourier
        for (i = 1; i <= N; i++) {
            cc1 += y[i] * cexp(-ic * x[i] * f);
        }

        r1 = cabs(cc1); // Módulo da transformada

        // Escrita no arquivo de saída
        fprintf(fil1, "%20.8lf %20.8lf\n", f, r1);
    }

    fclose(fil);
    fclose(fil1);
    return 0;
}
```

Para utilizar o programa apresentado, siga os passos abaixo:

1. Compile o código utilizando o compilador `gcc` com o seguinte comando:

   ```
   gcc fourier.c -O1 -o fourier -lm
   ```

2. Prepare um arquivo de entrada chamado `entrada.dat`, contendo ao menos quatro colunas em cada linha. O programa utilizará apenas as duas primeiras colunas, correspondentes aos pares (x_i, y_i), ignorando as demais.

3. Execute o programa com o comando:

   ```
   ./fourier
   ```

4. Após a execução, será gerado um arquivo chamado `saida.dat`, contendo dois valores por linha: a frequência f e o módulo da transformada $|F(f)|$. Esses dados podem ser visualizados ou analisados com ferramentas como `Gnuplot`, `Python (Matplotlib)`, `Mathematica`, entre outras.

Este procedimento fornece uma estimativa direta e precisa do espectro de frequências presente em um conjunto arbitrário de dados amostrados. Embora não utilize algoritmos rápidos como a Transformada Rápida de Fourier (FFT), essa abordagem tem a vantagem de funcionar mesmo quando os pontos x_i não são uniformemente espaçados, o que é comum em dados experimentais ou simulações adaptativas. Além disso, a implementação simples torna o código acessível, facilmente modificável e aplicável em diversas situações práticas de análise espectral.

0.14 Introdução à Solução Numérica de Sistemas Lineares

A resolução de sistemas lineares é uma das tarefas mais fundamentais em matemática aplicada, ciência da computação, engenharia e física. Um sistema linear pode ser representado na forma matricial $A\vec{x} = \vec{b}$, onde A é uma matriz de coeficientes conhecida, $\vec{b}$ é um vetor de termos independentes, e $\vec{x}$ é o vetor incógnita que se deseja determinar. Embora existam métodos analíticos para resolver sistemas pequenos, como substituição e comparação, para sistemas de grande dimensão — que aparecem com frequência em aplicações científicas e computacionais — torna-se necessário empregar métodos numéricos. Esses métodos podem ser classificados, de forma geral, em duas categorias: métodos diretos e métodos iterativos. Os **métodos diretos**, como a eliminação de Gauss e a fatoração LU, visam obter a solução exata (dentro da precisão da máquina) após um número finito de operações. São amplamente utilizados quando a matriz é densa e de tamanho moderado. Já os **métodos iterativos**, como os métodos de Jacobi, Gauss-Seidel e gradiente conjugado, constroem uma sequência de aproximações para a solução e são especialmente úteis para sistemas grandes, esparsos ou mal condicionados. A escolha do método adequado depende de diversas características do sistema, como o tamanho da matriz, sua estrutura (simetria, esparsidade, dominância diagonal), e o custo computacional envolvido. Nesta seção, apresentaremos os principais métodos numéricos para resolver sistemas lineares, discutindo suas vantagens, limitações e implementações computacionais.

0.14.1 Método de Eliminação de Gauss

O **método de eliminação de Gauss** é uma técnica algébrica sistemática para resolver sistemas lineares do tipo:

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

onde A é uma matriz $n \times n$, $\vec{x}$ é o vetor incógnita e $\vec{b}$ é o vetor de termos constantes. A ideia é transformar a matriz aumentada $[A \,|\, \vec{b}]$ em uma matriz triangular superior utilizando operações elementares nas linhas. Uma vez que o sistema está na forma triangular superior, as variáveis podem ser encontradas pela substituição retroativa (back-substitution). As operações elementares permitidas são:

- Trocar duas linhas;
- Multiplicar uma linha por uma constante não nula;
- Somar um múltiplo de uma linha a outra.

Exemplo

Considere novamente o sistema:

$$\begin{cases} 10x + 2y + z = 14 \\ 2x + 10y + 3z = 14 \\ x + 3y + 10z = 14 \end{cases}$$

Escrevemos a matriz aumentada:

$$\left[\begin{array}{ccc|c} 10 & 2 & 1 & 14 \\ 2 & 10 & 3 & 14 \\ 1 & 3 & 10 & 14 \end{array}\right]$$

Aplicando o método de Gauss:

1. Eliminar o elemento a_{21}:
$$L_2 \leftarrow L_2 - 2L_1$$

2. Eliminar o elemento a_{31}:
$$L_3 \leftarrow L_3 + L_1$$

3. Continuar eliminando para obter a forma triangular superior.

Após essas etapas, fazemos substituição retroativa para encontrar z, depois y, e finalmente x. A seguir, temos um programa simples em C que resolve um sistema linear de n equações com n incógnitas utilizando o método de eliminação de Gauss com substituição retroativa.

```
/* Programa em C: Método de Eliminação de Gauss
 * Autor: [Seu Nome]
 * Descrição: Resolve um sistema linear Ax = b
 */

#include <stdio.h>
#include <math.h>

#define N 3  // Número de equações e variáveis

float A[N][N+1]; // Matriz aumentada [A|b]
float x[N];      // Solução

void gauss() {
    int i, j, k;
    float m;

    // Eliminação direta
    for (k = 0; k < N - 1; k++) {
        for (i = k + 1; i < N; i++) {
            if (A[k][k] == 0) {
                printf("Divisão por zero!\n");
                return;
            }
            m = A[i][k] / A[k][k];
            for (j = k; j <= N; j++) {
                A[i][j] -= m * A[k][j];
            }
        }
    }

    // Substituição retroativa
    for (i = N - 1; i >= 0; i--) {
        x[i] = A[i][N];
        for (j = i + 1; j < N; j++) {
            x[i] -= A[i][j] * x[j];
        }
        x[i] /= A[i][i];
```

```
    }
}

int main() {
    int i, j;

    // Exemplo do sistema:
    // 2x + 3y - z = 5
    // 4x + 4y -3z = 3
    // -2x + 3y + 2z = 7

    float dados[N][N+1] = {
        {10, 2, 1, 14},
        {2, 10, 3, 14},
        {1, 3, 10, 14}
    };

    // Copia os dados para a matriz A
    for (i = 0; i < N; i++)
        for (j = 0; j <= N; j++)
            A[i][j] = dados[i][j];

    gauss();

    // Exibe a solução
    printf("Solução do sistema:\n");
    for (i = 0; i < N; i++) {
        printf("x[%d] = %f\n", i + 1, x[i]);
    }

    return 0;
}
```

O método de eliminação de Gauss é confiável e eficiente para sistemas pequenos e médios. No entanto, em alguns casos pode ocorrer instabilidade numérica, especialmente quando o elemento pivô é muito pequeno. Para lidar com isso, é comum utilizar o pivotamento parcial, que consiste em trocar linhas de modo a garantir que o maior elemento (em módulo) na coluna atual seja usado como pivô. Além disso, para sistemas muito grandes ou mal condicionados, o método direto pode se tornar ineficiente ou impreciso, sendo mais adequado recorrer a métodos iterativos, como Gauss-Seidel ou gradiente conjugado, ou ainda a técnicas baseadas em fatorações, como a fatoração LU.

0.14.2 Método de Gauss-Seidel

O **método de Gauss-Seidel** é um método iterativo utilizado para resolver sistemas lineares do tipo:

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

onde A é uma matriz $n \times n$, $\vec{x}$ é o vetor de incógnitas e $\vec{b}$ é o vetor de constantes.
Dado um sistema linear:

$$a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1, a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2, \vdots a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n,$$

a ideia do método de Gauss-Seidel é reescrever cada equação isolando uma variável:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right),$$

onde $x_i^{(k)}$ representa o valor da variável x_i na k-ésima iteração. Ao contrário do método de Jacobi, o Gauss-Seidel utiliza os valores mais recentes assim que eles são computados, o que pode acelerar a convergência.

Exemplo

Considere novamente o sistema:

$$\begin{cases} 10x + 2y + z = 14 \\ 2x + 10y + 3z = 14 \\ x + 3y + 10z = 14 \end{cases}$$

Reescrevemos cada equação isolando as variáveis:

$$x^{(k+1)} = \frac{1}{2}(5 - 3y^{(k)} + z^{(k)})$$
$$y^{(k+1)} = \frac{1}{4}(3 - 4x^{(k+1)} + 3z^{(k)})$$
$$z^{(k+1)} = \frac{1}{2}(7 + 2x^{(k+1)} - 3y^{(k+1)})$$

A cada iteração, atualizamos os valores das variáveis usando os valores mais recentes disponíveis. Abaixo temos um programa simples em C que implementa o método de Gauss-Seidel para resolver o sistema dado.

```
/* Programa em C: Método de Gauss-Seidel
 * Autor: [Seu Nome]
 * Descrição: Resolve um sistema linear Ax = b iterativamente
 * usando o método de Gauss-Seidel.
 */

#include <stdio.h>
#include <math.h>

// Número de variáveis
#define N 3

// Tolerância para critério de parada
#define TOL 1e-6

// Número máximo de iterações
#define MAX_IT 100

// Matriz dos coeficientes A
float A[N][N] = {
    {10, 2, 1},
```

```
    {2, 10, 3},
    {1, 3, 10}
};

// Vetor dos termos independentes b
float b[N] = {14, 14, 14};

// Vetor solução (inicializado com chute inicial)
float x[N] = {1, 1, 1};

// Vetor para armazenar a solução da iteração anterior
float x_old[N];

// Função que implementa o método de Gauss-Seidel
void gauss_seidel() {
    int i, j, k;
    float sum, erro;

    // Loop principal de iteração
    for (k = 0; k < MAX_IT; k++) {
        // Salva o vetor anterior
        for (i = 0; i < N; i++) {
            x_old[i] = x[i];
        }

        // Atualiza cada variável x[i]
        for (i = 0; i < N; i++) {
            sum = b[i];
            for (j = 0; j < N; j++) {
                if (j != i) {
                    if (j < i)
                        sum -= A[i][j] * x[j];      // usa o valor já atualizado
                    else
                        sum -= A[i][j] * x_old[j];   // usa o valor antigo
                }
            }
            x[i] = sum / A[i][i];
        }

        // Calcula o erro (soma das diferenças absolutas)
        erro = 0;
        for (i = 0; i < N; i++) {
            erro += fabs(x[i] - x_old[i]);
        }

        // Verifica se o erro é menor que a tolerância
        if (erro < TOL) break;
    }
}
```

```
// Função principal
int main() {
    int i;

    gauss_seidel();

    // Exibe a solução aproximada
    printf("Solução aproximada com o método de Gauss-Seidel:\n");
    for (i = 0; i < N; i++) {
        printf("x[%d] = %f\n", i + 1, x[i]);
    }

    return 0;
}
```

O método de Gauss-Seidel é especialmente eficaz quando a matriz A é estritamente diagonal dominante ou simétrica definida positiva, condições que garantem a convergência do processo iterativo. É um método simples e eficiente para sistemas de grande porte e esparsos. Entretanto, nem todos os sistemas garantem convergência com Gauss-Seidel, e sua taxa de convergência pode ser lenta se o sistema não for bem condicionado.

Se faz importante salientar que , embora o método seja simples e eficaz, ele não garante convergência para qualquer sistema linear. A convergência depende fortemente das propriedades da matriz A. Em particular, o método converge se pelo menos uma das seguintes condições for satisfeita:

1. **Matriz diagonalmente dominante:** uma matriz A é diagonalmente dominante se:

$$|a_{ii}| \geq \sum_{j \neq i} |a_{ij}| \quad \text{para todo } i,$$

   e estritamente dominante para pelo menos uma linha. Isso garante convergência.

2. **Matriz simétrica positiva definida (SPD):** se A é simétrica ($A = A^T$) e $\vec{x}^T A\vec{x} > 0$ para todo vetor $\vec{x} \neq 0$, então A é positiva definida. Neste caso, o método também converge.

3. **Outras condições espectrais:** mesmo que A não seja diagonalmente dominante ou SPD, o método pode convergir se o espectro da matriz de iteração (o raio espectral) for menor que 1. Porém, isso exige análise mais técnica.

0.15 Método de Newton-Raphson para Sistemas de Equações Não Lineares

O método de Newton-Raphson é uma das técnicas mais conhecidas para resolver sistemas de equações não lineares. Ele é uma generalização do método de Newton de uma variável para o caso multivariado. Sua eficiência depende fortemente de uma boa aproximação inicial da solução. Considere um sistema de n equações não lineares com n incógnitas:

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \quad \vdots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases}$$

Definimos o vetor das funções:

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, \ldots, x_n) \\ f_2(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{bmatrix}$$

E a matriz Jacobiana $J(\mathbf{x})$ como:

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

A iteração de Newton é dada por:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - J^{-1}(\mathbf{x}^{(k)}) \cdot \mathbf{F}(\mathbf{x}^{(k)})$$

Ou, de forma equivalente, resolvendo um sistema linear a cada passo:

$$J(\mathbf{x}^{(k)}) \cdot \Delta\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)})$$

E então:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$

O processo se repete até que $\|\Delta\mathbf{x}^{(k)}\|$ ou $\|\mathbf{F}(\mathbf{x}^{(k)})\|$ seja menor que uma tolerância pré-estabelecida.

Exemplo Numérico

Considere o sistema:

$$\begin{cases} f_1(x, y) = x^2 + y^2 - 1 = 0 \\ f_2(x, y) = e^x + y - 1 = 0 \end{cases}$$

As derivadas parciais são:

$$J(x,y) = \begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ e^x & 1 \end{bmatrix}$$

A seguir, um programa simples em linguagem C para resolver esse sistema pelo método de Newton-Raphson. Usamos o método de substituição de Gauss para resolver o sistema linear interno.

```
#include <stdio.h>
#include <math.h>

#define TOL 1e-6
#define MAX_ITER 100

// Funções do sistema
double f1(double x, double y) {
    return x*x + y*y - 1;
}

double f2(double x, double y) {
    return exp(x) + y - 1;
}

// Derivadas parciais (Jacobiano)
double df1dx(double x, double y) {
    return 2*x;
}

double df1dy(double x, double y) {
    return 2*y;
}

double df2dx(double x, double y) {
    return exp(x);
}

double df2dy(double x, double y) {
    return 1;
}

int main() {
    double x = 0.5, y = 0.5; // chute inicial
    double dx, dy;
    int iter = 0;

    while (iter < MAX_ITER) {
        // Calcula o sistema linear J * [dx dy]^T = -F
        double J[2][2] = {
            {df1dx(x, y), df1dy(x, y)},
```

```
            {df2dx(x, y), df2dy(x, y)}
        };
        double F[2] = {-f1(x, y), -f2(x, y)};

        // Eliminação de Gauss (2x2)
        double det = J[0][0]*J[1][1] - J[0][1]*J[1][0];
        if (fabs(det) < 1e-12) {
            printf("Jacobiano singular.\n");
            break;
        }

        dx = (F[0]*J[1][1] - F[1]*J[0][1]) / det;
        dy = (J[0][0]*F[1] - J[1][0]*F[0]) / det;

        x += dx;
        y += dy;

        if (sqrt(dx*dx + dy*dy) < TOL) break;
        iter++;
    }

    printf("Solução encontrada: x = %.6f, y = %.6f (em %d iterações)\n", x, y, iter);
    return 0;
}
```

Este programa realiza uma iteração de Newton para um sistema 2×2. Algumas observações importantes:

- O método exige o cálculo simbólico ou automático das derivadas parciais (Jacobiano).
- O sistema linear interno foi resolvido manualmente com a fórmula de Cramer (válido apenas para 2×2). Em sistemas maiores, seria melhor usar uma rotina de decomposição LU ou método de Gauss.
- A escolha do ponto inicial $(x_0, y_0) = (0.5, 0.5)$ é essencial. Outras escolhas podem não convergir.
- A convergência é rápida se o chute inicial for bom e o sistema for suave.

O método de Newton-Raphson é uma ferramenta poderosa para a resolução de sistemas não lineares, mas seu uso exige atenção a diversos aspectos cruciais. É fundamental garantir que o Jacobiano seja não-singular em cada iteração, que o ponto inicial esteja suficientemente próximo da solução e que o número de iterações permaneça dentro de limites razoáveis. Quando bem aplicado, o método oferece convergência rápida e resultados precisos, sendo uma técnica indispensável no estudo de sistemas não lineares em áreas como ciência computacional, física e engenharia.

0.16 Resolvendo Numericamente uma Equação Integral

Nesta seção, apresentamos um exemplo simples de resolução numérica de uma equação integral. O problema que queremos resolver é dado por:

$$x = \int_0^1 (x + s)\,\phi(s)\,ds,$$

onde x é um valor conhecido e nosso objetivo é determinar a função $\phi(s)$. Esse tipo de equação aparece com frequência em aplicações de física e engenharia, e seu tratamento numérico é essencial quando não há solução analítica disponível.
Para resolver numericamente a equação integral, utilizamos um processo de discretização. Dividimos o intervalo $[0, 1]$ em N subintervalos de tamanho $h = \frac{1}{N}$. Com isso, a integral é aproximada por uma soma de Riemann:

$$x \approx h \sum_{j=0}^{N-1} (x + s_j)\,\phi(s_j),$$

onde $s_j = jh$ são os pontos de discretização, e $\phi_j = \phi(s_j)$. Isso transforma a equação integral em um sistema linear:

$$x = h \sum_{j=0}^{N-1} (x + s_j)\,\phi_j.$$

Neste exemplo, simplificamos ainda mais o problema assumindo que a solução pode ser obtida por um processo direto, com cada ϕ_j calculado separadamente. A seguir, apresentamos um programa em linguagem C que implementa o método descrito, assumindo $N = 10$ e $x = 0.5$. O código está comentado para facilitar a compreensão.

```
#include <stdio.h>
#include <math.h>

#define N 10      // Número de subdivisões do intervalo [0,1]
#define X_VAL 0.5 // Valor conhecido de x na equação

// Função que resolve a equação integral discretizada
void solve_equation(double phi[]) {
    double h = 1.0 / N;  // Tamanho do passo
    double s[N];         // Pontos de discretização
    double A[N];         // Soma dos coeficientes do sistema
    double B[N];         // Lado direito da equação (constante x)
    int i, j;

    // Inicializa os pontos s_j
    for (i = 0; i < N; i++) {
        s[i] = i * h;
    }

    // Monta o sistema linear para cada ponto
    for (i = 0; i < N; i++) {
```

```
        B[i] = X_VAL;  // Valor fixo de x no lado direito
        A[i] = 0.0;
        for (j = 0; j < N; j++) {
            A[i] += h * (X_VAL + s[j]);
        }
    }

    // Solução direta simplificada para cada phi_j
    for (i = 0; i < N; i++) {
        phi[i] = B[i] / A[i];
    }
}

int main() {
    double phi[N]; // Vetor solução phi_j
    int i;

    // Resolve a equação integral
    solve_equation(phi);

    // Imprime os resultados
    printf("Resultado para phi(s):\n");
    for (i = 0; i < N; i++) {
        printf("phi[%d] = %.5f\n", i, phi[i]);
    }

    return 0;
}
```

O programa acima demonstra como transformar uma equação integral simples em um problema algébrico por meio da discretização. Embora o método utilizado seja elementar, ele oferece uma introdução prática aos métodos numéricos aplicados à resolução de equações integrais. Em aplicações reais e problemas mais complexos, pode ser necessário utilizar técnicas mais sofisticadas, como quadratura de Gauss, métodos iterativos ou estratégias baseadas em elementos finitos.

0.16.1 Equações Integrais: Um Exemplo com Transferência Radiativa

Nesta seção do nosso ebook, apresentamos um exemplo importante de equação integral aplicado à física: a equação de transferência radiativa em um meio absorvente com fonte emissiva constante. Este tipo de equação surge naturalmente na descrição de como a radiação se propaga e interage com a matéria.

Forma Analítica da Equação

A equação de transferência radiativa para um meio com coeficiente de absorção constante κ, fonte de emissão constante S_0 e intensidade inicial I_0, è dada por:

$$I(x) = I_0 e^{-\kappa x} + \int_0^x S_0 e^{-\kappa(x-s)}\, ds.$$

Neste caso, queremos determinar numericamente o valor da intensidade $I(x)$ para um dado ponto x.

Discretização Numérica da Integral

Para resolver numericamente a equação integral, utilizamos a método de discretização do intervalo $[0, x]$ em N subintervalos de tamanho:

$$h = \frac{x}{N}.$$

A integral $\int_0^x S_0 e^{-\kappa(x-s)}\, ds$ é então aproximada por uma soma:

$$\int_0^x S_0 e^{-\kappa(x-s)}\, ds \approx h \sum_{i=0}^{N-1} S_0 e^{-\kappa(x-s_i)},$$

onde $s_i = ih$ representa os pontos de discretização. Essa abordagem transforma a equação integral em um problema computacional simples. A seguir, apresentamos um programa simples em linguagem C que implementa o método descrito.

```
#include <stdio.h>
#include <math.h>

#define N 100      // Numero de subdivisoes
#define KAPPA 1.0  // Coeficiente de absorcao
#define S0 1.0     // Fonte de emissao
#define I0 1.0     // Intensidade inicial

// Funcao que resolve numericamente a equacao de transferencia radiativa
double solve_radiative_transfer(double x) {
    double h = x / N;         // Tamanho do passo
    double integral = 0.0;    // Inicializa o valor da integral
    double s;                 // Variavel auxiliar para o ponto s_i

    // Loop para calcular a integral numericamente via soma de Riemann
    for (int i = 0; i < N; i++) {
        s = i * h;  // Ponto s_i
        integral += S0 * exp(-KAPPA * (x - s)) * h;
    }

    // Termo da intensidade inicial atenuado pela absorcao
    return I0 * exp(-KAPPA * x) + integral;
}

int main() {
    double x = 1.0;           // Ponto em que queremos calcular I(x)
    double intensity;

    // Chama a funcao que calcula a intensidade
    intensity = solve_radiative_transfer(x);

    // Imprime o resultado final
    printf("Intensidade em x = %.2f: I(x) = %.5f\n", x, intensity);

    return 0;
}
```

O programa apresentado computa a intensidade $I(x)$ para um valor arbitrário de x, considerando a atenuação da radiação inicial e a contribuição da fonte S_0 ao longo do caminho. O método de soma de Riemann utilizado é suficiente para problemas simples como este, onde as funções envolvidas são suaves e bem comportadas. Este exemplo ilustra como equações integrais surgem naturalmente em problemas físicos e como podemos aplicar métodos numéricos simples para resolvê-las. Problemas mais complexos podem exigir técnicas mais sofisticadas, como quadratura adaptativa, métodos espectrais ou técnicas iterativas, mas a compreensão do processo de discretização é essencial para avançar nesses tópicos.

0.17 Resolução Numérica de Equações Transcendentais: Um Exemplo com o Método da Bisseção

Equações transcendentais aparecem com frequência em diversas áreas da matemática, física e engenharia. Diferentemente das equações algébricas, que envolvem apenas polinômios, as transcendentais incluem funções como exponenciais, logaritmos, senos, cossenos, tangentes, entre outras. Como regra geral, essas equações não possuem soluções analíticas exatas, exigindo métodos numéricos para a obtenção de aproximações precisas das raízes. Nesta seção, exploraremos um exemplo concreto de equação transcendental:

$$x + 1 = \tan(x),$$

que, rearranjada, pode ser escrita como:

$$f(x) = x + 1 - \tan(x) = 0.$$

Como $f(x)$ envolve a função tangente, que possui descontinuidades em múltiplos de $\frac{\pi}{2}$, devemos escolher o intervalo com cuidado para garantir que a função seja contínua dentro dele e que exista uma raiz.

0.17.1 O Método da Bisseção

Um dos métodos mais simples e confiáveis para encontrar raízes de equações contínuas é o método da bisseção. Baseado no Teorema do Valor Intermediário, ele garante a existência de uma raiz em um intervalo $[a, b]$ desde que $f(a) \cdot f(b) < 0$, ou seja, a função muda de sinal dentro do intervalo.

O procedimento básico é o seguinte:

1. Escolhe-se um intervalo inicial $[a, b]$ tal que $f(a) \cdot f(b) < 0$.
2. Calcula-se o ponto médio $c = \frac{a+b}{2}$.
3. Avalia-se $f(c)$. Se for suficientemente próximo de zero, c é uma boa aproximação da raiz.
4. Se não, escolhe-se o subintervalo $[a, c]$ ou $[c, b]$ no qual a função muda de sinal, e repete-se o processo.

O critério de parada ocorre quando a largura do intervalo for menor que uma tolerância previamente especificada ou quando $f(c)$ estiver suficientemente próximo de zero. A seguir, apresentamos uma implementação do método da bisseção para resolver a equação $x + 1 = \tan(x)$ usando linguagem C. A tolerância numérica foi fixada em 10^{-6}, e o intervalo inicial considerado é [1,1, 1,2], no qual sabemos que há uma raiz.

```c
#include <stdio.h>
#include <math.h>

#define TOLERANCE 1e-6
#define MAX_ITERATIONS 1000

// Função f(x) = x + 1 - tan(x)
double f(double x) {
    return x + 1 - tan(x);
}

// Método da Bisseção
double bisection(double a, double b) {
    double fa = f(a);
    double fb = f(b);

    if (fa * fb > 0) {
        printf("Erro: f(a) e f(b) devem ter sinais opostos.\n");
        return NAN;
    }

    double c, fc;
    int iterations = 0;

    while ((b - a) / 2 > TOLERANCE && iterations < MAX_ITERATIONS) {
        c = (a + b) / 2;
        fc = f(c);

        if (fabs(fc) < TOLERANCE) {
            return c; // Encontrou a raiz
        }

        if (fa * fc < 0) {
            b = c;
            fb = fc;
        } else {
            a = c;
            fa = fc;
        }

        iterations++;
    }

    if (iterations == MAX_ITERATIONS) {
        printf("Aviso: Número máximo de iterações atingido.\n");
    }

    return (a + b) / 2;
}
```

```
int main() {
    double a = 1.1, b = 1.2; // Intervalo inicial
    double root = bisection(a, b);

    if (!isnan(root)) {
        printf("A raiz encontrada é: %.6f\n", root);
    }

    return 0;
}
```

Executando o programa acima, obtemos como saída:

```
A raiz encontrada é: 1.132268
```

Este valor corresponde à raiz da equação no intervalo considerado, com precisão de seis casas decimais. A escolha do intervalo foi crucial para garantir a convergência do método, pois evitamos os pontos onde $\tan(x)$ diverge.

0.17.2 Resolvendo a equação $xe^x = 2$ com o método de Newton-Raphson

Equações transcendentais, como $xe^x = 2$, não podem ser resolvidas de forma exata usando apenas funções elementares. Para encontrar uma solução numérica, podemos empregar o método iterativo de Newton-Raphson, que oferece rápida convergência desde que uma boa aproximação inicial seja escolhida. Como já foi mencionado anteriormente, a ideia do método de Newton-Raphson é utilizar a aproximação linear da função em um ponto inicial x_0, ou seja, sua reta tangente, para obter uma melhor aproximação da raiz. A fórmula iterativa é dada por:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Etapas do algoritmo:

1. Escolher um valor inicial x_0.
2. Calcular x_{n+1} usando a fórmula acima.
3. Verificar se a diferença $|x_{n+1} - x_n|$ é menor que uma tolerância ϵ.
4. Repetir os passos até atingir a tolerância ou um número máximo de iterações.

Aplicando à equação $xe^x = 2$

Podemos reescrever a equação como:

$$f(x) = xe^x - 2,$$

cuja derivada é:

$$f'(x) = e^x(x + 1).$$

A seguir, apresentamos uma implementação simples do método de Newton-Raphson em linguagem C, com comentários explicativos para facilitar o entendimento:

```
#include <stdio.h>
#include <math.h>

// Define a tolerância e o número máximo de iterações
#define TOLERANCIA 1e-6
#define MAX_ITER 100

// Função f(x) = x * e^x - 2
double f(double x) {
    return x * exp(x) - 2;
}

// Derivada f'(x) = e^x * (x + 1)
double df(double x) {
    return exp(x) * (x + 1);
}

int main() {
    double x0 = 0.5;  // Aproximação inicial
    double x1;
    int iteracoes = 0;

    // Laço principal do método de Newton-Raphson
    while (iteracoes < MAX_ITER) {
        x1 = x0 - f(x0) / df(x0);  // Fórmula de Newton-Raphson
        if (fabs(x1 - x0) < TOLERANCIA) {  // Critério de parada
            break;
        }
        x0 = x1;  // Atualiza a aproximação
        iteracoes++;
    }

    // Exibe o resultado
    if (iteracoes < MAX_ITER) {
        printf("Raiz encontrada: %.6f\n", x1);
    } else {
        printf("O método não convergiu.\n");
    }

    return 0;
}
```

Configurações importantes:

- A função $f(x)$ representa a equação $xe^x - 2$.
- A derivada $f'(x)$ é computada diretamente como $e^x(x + 1)$.
- O valor inicial $x_0 = 0.5$ foi escolhido com base em uma análise gráfica simples.
- A tolerância foi fixada em 10^{-6}, o que garante boa precisão.

Com a escolha de $x_0 = 0.5$, o método converge rapidamente para uma raiz aproximada de $x \approx 0.852605$ em poucas iterações. O número exato de passos depende da tolerância escolhida. O método pode falhar em convergir se a derivada for nula ou próxima de zero, ou se o valor inicial estiver muito distante da raiz verdadeira. O método de Newton-Raphson é extremamente útil para resolver numericamente equações que não possuem soluções fechadas. Seu uso é simples, eficiente e, como mostrado no exemplo acima, pode ser facilmente implementado em C. A chave para o sucesso do método está na escolha adequada da aproximação inicial e na verificação da derivada da função.

0.18 Considerações Finais

Este eBook foi concebido com o propósito de introduzir estudantes de graduação aos métodos numéricos fundamentais, utilizando a linguagem de programação C como ferramenta principal para a implementação prática dos conceitos abordados. Ao longo dos capítulos, exploramos tópicos essenciais como geração de números aleatórios, resolução de sistemas lineares, estatística básica, transformadas de Fourier e, finalmente, uma introdução a mapas caóticos — todos apresentados de forma gradual e com exemplos simples, voltados à construção de uma base sólida no pensamento computacional e científico.

A escolha dos temas não foi feita ao acaso. Eles refletem tanto a relevância no contexto das ciências exatas e das engenharias quanto a possibilidade de desenvolver, desde cedo, uma intuição numérica nos estudantes. Métodos numéricos não são apenas uma ferramenta complementar à matemática analítica — eles são, cada vez mais, o caminho natural para estudar sistemas complexos, modelar fenômenos reais e explorar propriedades que, muitas vezes, escapam à solução exata.

A programação em linguagem C foi escolhida por sua estrutura próxima ao hardware, pelo controle que oferece sobre o fluxo de dados e pela clareza conceitual em relação à lógica algorítmica. Apesar de existirem linguagens mais modernas e de mais alto nível, como Python ou Julia, o C permanece uma linguagem valiosa para quem deseja entender os fundamentos da computação numérica sem abstrações excessivas.

Esperamos que os leitores tenham compreendido que muitos dos métodos aqui apresentados são apenas o ponto de partida. Por exemplo, os geradores de números aleatórios discutidos são versões básicas, mas abrem portas para estudos mais profundos sobre estatística computacional, simulações de Monte Carlo e modelagem estocástica. A resolução de sistemas lineares pode ser expandida para métodos iterativos, decomposições sofisticadas e aplicações em álgebra computacional. A transformada de Fourier, por sua vez, é a base de técnicas modernas de processamento de sinais, imagem e análise espectral.

Da mesma forma, a breve incursão pelo mundo dos sistemas caóticos tem como intuito despertar o interesse pela não linearidade, sensibilidade às condições iniciais e comportamento complexo que surgem mesmo em sistemas determinísticos simples. Esses tópicos fazem a ponte entre a computação e áreas como a física, biologia, economia e ciências sociais, sendo centrais para a compreensão de fenômenos naturais e sociais no século XXI.

Por fim, vale lembrar que a aprendizagem de métodos numéricos não se dá apenas pela leitura ou memorização de fórmulas, mas principalmente pela prática constante, pelo erro e pela tentativa. Cada linha de código escrita, cada *bug* encontrado e cada gráfico gerado são passos fundamentais na formação de um pensamento analítico e computacional maduro.

Esperamos que este material possa servir como um guia introdutório, mas também como uma inspiração para aprofundamentos futuros. Que os leitores sintam-se encorajados a explorar novos métodos, aprender outras linguagens e, sobretudo, aplicar o conhecimento numérico de forma crítica, criativa e consciente.

Boa jornada nos estudos — e que seus algoritmos sejam sempre estáveis, suas matrizes bem condicionadas, e suas séries convergentes!