Notas de aula de física computacional (em C)

Francisco Anacleto Barros Fidelis de Moura

Prefácio

Estas notas de aula surgem como material de apoio ao curso de Física Computacional, com ênfase na implementação de algoritmos numéricos em linguagem C. O objetivo principal é oferecer aos estudantes uma introdução clara, prática, resumida e progressiva às ferramentas computacionais mais usadas na física. Ao longo do texto, conceitos fundamentais de programação e de métodos numéricos são apresentados de forma gradual, sempre acompanhados de exemplos e códigos completos. A **Parte 1** dedica-se aos fundamentos de programação em C, bem como a técnicas iniciais de cálculo numérico e análise de dados. Já a **Parte 2** avança para métodos de integração de equações diferenciais ordinárias, simulações estocásticas, transformadas de Fourier e técnicas de diagonalização de matrizes, essenciais para problemas de autovalores e autovetores em física. A proposta não é apenas fornecer algoritmos prontos, mas também discutir seus fundamentos, limitações, interpretações físicas e aplicações. Assim, o estudante é estimulado a desenvolver tanto o domínio técnico da programação quanto a compreensão conceitual dos métodos utilizados.

Agradecimentos

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

Sumário

Parte 1

A **Parte 1** do curso de Física Computacional tem como objetivo introduzir o estudante às ferramentas fundamentais que servirão de base para todo o desenvolvimento posterior da disciplina. O ponto de partida é a *programação em linguagem C*, que será utilizada ao longo do curso como principal instrumento para a implementação dos algoritmos numéricos. Nesta etapa inicial, o foco estará no aprendizado das estruturas básicas da linguagem e na prática com exemplos simples, que ajudarão a construir familiaridade com a lógica computacional. A seguir, exploraremos aplicações diretas, como o uso de **números complexos em C**, fundamentais em diversas áreas da física, e passaremos à discussão de **técnicas de cálculo numérico elementares**, incluindo derivadas e integrais aproximadas. Esses métodos constituem a base de muitas simulações físicas e permitem entender, em um nível conceitual, as limitações e a precisão das aproximações computacionais. Também será introduzida a geração de **números aleatórios**, ferramenta essencial para métodos de Monte Carlo e para modelagens estocásticas que aparecerão em partes posteriores do curso. Por fim, abordaremos a **regressão linear**, técnica estatística simples mas poderosa, útil para análise de dados experimentais e para testar a eficiência dos algoritmos implementados. Assim, a Parte 1 constitui uma preparação sólida, reunindo tanto fundamentos de programação quanto conceitos numéricos iniciais, fornecendo ao estudante as bases necessárias para compreender e aplicar os métodos mais avançados que serão tratados nas próximas etapas do curso.

1 Introdução à Programação em C

Antes de escrever programas mais complexos, é fundamental compreender os principais comandos, funções e estruturas da linguagem C. Conhecer essas ferramentas permite desenvolver programas corretos, eficientes e legíveis, além de facilitar a depuração e manutenção do código. A tabela a seguir apresenta alguns dos elementos mais importantes da linguagem, com breves exemplos ou descrições de uso, servindo como referência rápida para iniciantes e usuários intermediários.

Comando / Função	**Descrição / Exemplo**
`#include <stdio.h>`	Biblioteca padrão de entrada e saída: `printf`, `scanf`, etc.
`#include <stdlib.h>`	Funções de alocação dinâmica (`malloc`, `free`), números aleatórios (`rand`), conversão de strings (`atoi`, `atof`).
`#include <math.h>`	Funções matemáticas: `sin`, `cos`, `tan`, `exp`, `log`, `pow`, `sqrt`.
`int main()`	Função principal do programa. Ex.: `int main() { ... return 0; }`
`printf()`	Imprime texto ou valores: `printf("x = %f\n", x);`
`scanf()`	Lê valores do usuário: `scanf("%d", &n);`
`for(...)`	Loop com contador: `for(int i=0;i<10;i++) { ... }`
`while(...)`	Loop condicional: `while(x<10) { ... }`
`do { ... } while(...);`	Executa ao menos uma vez antes de checar a condição.
`if(...)`	Condicional simples: `if(x>0) ...`
`else`	Alternativa ao `if`: `else ...`
`switch(...)`	Estrutura múltipla: `switch(n) { case 1: ... break; ... }`
`break`	Encerra loops ou `switch` imediatamente.
`continue`	Pula para a próxima iteração do loop.
`return 0;`	Indica término correto do programa.
`sizeof()`	Retorna tamanho em bytes de tipo ou variável. Ex.: `sizeof(int)`
`typedef`	Define nomes alternativos para tipos. Ex.: `typedef unsigned int uint;`
`struct`	Agrupa variáveis de tipos diferentes. Ex.: `struct Ponto {double x,y;};`
`malloc()/free()`	Aloca e libera memória dinâmica: `double* v = malloc(N*sizeof(double)); free(v);`
`const`	Define variável constante: `const double pi = 3.14159;`
`#define`	Cria constante ou macro: `#define N 100`
`&`	Endereço de variável ou referência de ponteiro: `scanf("%d",&x);`
`*`	Ponteiro ou desreferência: `int *p; p = &x; *p = 5;`
`fopen(), fclose(), fprintf(), fscanf()`	Manipulação de arquivos: leitura e escrita.
`strcpy(), strcat(), strlen(), strcmp()`	Funções de strings: copiar, concatenar, medir e comparar.
`rand(), srand()`	Números aleatórios: `rand()/RAND_MAX` para [0,1], `srand(time(NULL))` define a semente.
`abs(), fabs()`	Valor absoluto para inteiros (`abs`) ou doubles (`fabs`).
`ceil(), floor(), round()`	Arredondamento: teto, piso ou mais próximo.
`exit()`	Encerra o programa imediatamente. Ex.: `exit(1);`
`enum`	Define conjunto de constantes simbólicas. Ex.: `enum Dia {Seg, Ter, Qua};`
`goto`	Salta para um rótulo definido; uso raro.

Esta tabela fornece um resumo das funções e comandos mais usados em C, com exemplos curtos e explicativos. Ela serve como referência rápida para iniciantes e para consultas durante a programação de exercícios ou projetos.

2 Exemplos gerais

Operações Matemáticas Simples

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

```
/* Programa simples em C que faz operações matemáticas básicas */
#include <stdio.h>
#include <math.h>

int main() {
```

```
    double a = 2.0, b = 3.0;
    printf("Soma: %f\n", a + b);
    printf("Produto: %f\n", a * b);
    printf("Potência: %f\n", pow(a, b));
    return 0;
}
```

Esse exemplo mostra como usar a biblioteca `math.h` e funções básicas de saída com `printf` em C.

Estruturas de Repetição (Loops)

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

```
/* Programa que imprime os números de 1 a 10 */
#include <stdio.h>

int main() {
    // Loop que começa em 1 e vai até 10
    for(int i = 1; i <= 10; i++) {
        // Em cada passo, o valor de i é mostrado na tela
        printf("%d\n", i);
    }

    // Indica que o programa terminou corretamente
    return 0;
}
```

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

Máximo e Mínimo de um Conjunto de Dados

Muitas vezes precisamos encontrar os valores extremos de um conjunto de dados. O algoritmo básico percorre todos os elementos e compara com os maiores e menores já encontrados.

```
/* Programa que encontra o máximo e o mínimo de um vetor */
#include <stdio.h>

int main() {
    // Vetor de exemplo com 5 valores
    int v[5] = {3, 7, -2, 10, 5};

    // Inicializa max e min com o primeiro elemento
    int max = v[0], min = v[0];

    // Percorre o vetor comparando cada valor
    for(int i=1; i<5; i++) {
        if(v[i] > max) max = v[i];  // atualiza máximo
        if(v[i] < min) min = v[i];  // atualiza mínimo
    }

    // Exibe os resultados finais
    printf("Max = %d, Min = %d\n", max, min);
    return 0;
}
```

Esse método é simples e eficiente: cada elemento do vetor é verificado apenas uma vez. É uma técnica essencial em análise de dados numéricos.

Séries de Taylor: Aproximação da Exponencial

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

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

Quanto mais termos incluímos, mais precisa é a aproximação.

```
/* Aproximação de e^x usando a série de Taylor */
#include <stdio.h>

int main() {
    double x = 1.0;        // ponto de avaliação
    double soma = 1.0;     // começa com o termo inicial (1)
    double termo = 1.0;    // termo atual da série
    int N = 10;            // número de termos usados

    // Calcula a soma dos termos da série
    for(int n=1; n<=N; n++) {
        termo *= x / n;    // atualiza termo: x^n / n!
        soma += termo;     // adiciona à soma
    }

    // Mostra a aproximação obtida
    printf("Aproximacao de e^x em x=%f: %f\n", x, soma);
    return 0;
}
```

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

Mapa Logístico

O mapa logístico é um modelo simples de dinâmica não-linear, frequentemente usado para estudar comportamento caótico:

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

Neste exemplo em C, vamos gerar uma tabela de x versus r para construir o diagrama de bifurcação. O programa salva os dados em um arquivo de saída.

```
/*
   Mapa logístico: gera dados para diagrama de bifurcação
   x_{n+1} = r * x_n * (1 - x_n)
*/

#include <stdio.h>

int main() {
    double r, x;
    double r_min = 2.5, r_max = 4.0, r_step = 0.01;
    int n_trans = 100;   // iterações de transiente (não salvas)
    int n_iter = 50;     // iterações salvas
    FILE *fp = fopen("bifurcacao.txt", "w"); // abre arquivo de saída



    // Varre o parâmetro r
    for(r = r_min; r <= r_max; r += r_step) {
        x = 0.5; // condição inicial

        // Descarta transientes
        for(int i = 0; i < n_trans; i++) {
            x = r * x * (1 - x);
        }
```

```
        // Salva os próximos valores de x
        for(int i = 0; i < n_iter; i++) {
            x = r * x * (1 - x);
            fprintf(fp, "%f %f\n", r, x);
        }
    }

    fclose(fp); // fecha arquivo

    return 0;
}
```

Após executar este programa, o arquivo `bifurcacao.txt` conterá pares r e x, que podem ser utilizados para plotar o diagrama de bifurcação com qualquer ferramenta gráfica, como Python, gnuplot ou Excel.

3 Números Complexos em C

Em C, a biblioteca padrão `<complex.h>` permite manipular números complexos de forma direta. Podemos representar um número complexo $z = a + ib$ usando o tipo `double complex` e utilizar funções prontas para operações como módulo, conjugado, exponencial e produto.

```
/* Operações básicas com números complexos usando complex.h */

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

int main() {
    // Define dois números complexos
    double complex z1 = 2.0 + 3.0*I;
    double complex z2 = 1.0 - 4.0*I;

    // Soma
    double complex soma = z1 + z2;

    // Produto
    double complex produto = z1 * z2;

    // Conjugado de z1
    double complex conj_z1 = conj(z1);

    // Módulo ao quadrado de z1
    double modulo2 = cabs(z1) * cabs(z1);

    // Exponencial de z1
    double complex exp_z1 = cexp(z1);

    // Exibe os resultados
    printf("z1 = %.2f + %.2fi\n", creal(z1), cimag(z1));
    printf("z2 = %.2f + %.2fi\n\n", creal(z2), cimag(z2));

    printf("Soma: %.2f + %.2fi\n", creal(soma), cimag(soma));
    printf("Produto: %.2f + %.2fi\n", creal(produto), cimag(produto));
    printf("Conjugado de z1: %.2f + %.2fi\n", creal(conj_z1), cimag(conj_z1));
    printf("|z1|^2 = %.2f\n", modulo2);
    printf("Exp(z1) = %.2f + %.2fi\n", creal(exp_z1), cimag(exp_z1));

    return 0;
}
```

Após executar este programa, várias propriedades e operações com números complexos serão exibidas no terminal, incluindo soma, produto, conjugado, módulo ao quadrado e exponencial. Usando `complex.h`, podemos implementar de forma simples qualquer cálculo envolvendo números complexos sem precisar definir estruturas manualmente.

4 Derivada Numérica

A derivada de uma função $f(x)$ pode ser aproximada usando diferenças finitas. Essa técnica é útil quando não temos a expressão analítica da função ou para cálculos rápidos em computador.

Método da Diferença Progressiva

A diferença progressiva aproxima a derivada por:

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

```
/* Derivada numérica: diferença progressiva */
#include <stdio.h>
#include <math.h>

double f(double x) {
    return sin(x); // função a derivar
}

int main() {
    double x = 1.0, h = 0.001;
    double deriv = (f(x+h) - f(x)) / h; // fórmula da diferença progressiva
    printf("Derivada aproximada em x=%f: %f\n", x, deriv);
    return 0;
}
```

O método é simples, mas a precisão depende do tamanho de h. Valores muito pequenos podem gerar erros numéricos.

Diferença Central

Uma versão mais precisa usa valores em torno de x:

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

```
/* Derivada numérica: diferença central */
#include <stdio.h>
#include <math.h>

double f(double x) { return cos(x); } // função a derivar

int main() {
    double x = 1.0, h = 0.001;
    double deriv = (f(x+h) - f(x-h)) / (2*h); // fórmula central
    printf("Derivada (central) em x=%f: %f\n", x, deriv);
    return 0;
}
```

O método central reduz o erro de aproximação em relação à diferença progressiva.

Derivadas a partir de um vetor de dados

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

```
/* Derivada de dados tabelados */
#include <stdio.h>

int main() {
    double x[5] = {0, 1, 2, 3, 4};
    double y[5] = {0, 1, 4, 9, 16}; // exemplo: y = x^2

    // diferenças centrais
    for(int i=1; i<4; i++) {
        double deriv = (y[i+1] - y[i-1]) / (x[i+1] - x[i-1]);
        printf("x=%f -> derivada ~ %f\n", x[i], deriv);
    }
    return 0;
}
```

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

Derivadas a partir de Arquivo de Dados

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

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

```
/* Derivada de dados lidos de arquivo */
#include <stdio.h>

int main() {
   // abrindo arquivo "tabela.dat" no modo leitura
   FILE *fp = fopen("tabela.dat", "r");


    double x[100], y[100]; // arrays para armazenar os dados
    int n = 0;

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

    // calcula derivadas usando diferenças centrais
    for(int i=1; i<n-1; i++) {
        double deriv = (y[i+1] - y[i-1]) / (x[i+1] - x[i-1]);
        printf("x=%.2f -> derivada ~ %.2f\n", x[i], deriv);
    }

    return 0;
}
```

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

5 Integração Numérica

A integral definida pode ser aproximada por métodos numéricos simples. Um deles é a regra dos retângulos, que aproxima a área sob a curva usando retângulos de base h:

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

```
/* Integração numérica usando retângulos */
#include <stdio.h>
#include <math.h>

double f(double x) { return sin(x); } // função a integrar

int main() {
    double a = 0, b = M_PI; // limites de integração
    int N = 100;             // número de subdivisões
    double h = (b-a)/N;
    double soma = 0.0;

    // soma dos valores da função nos pontos da esquerda
    for(int i=0; i<N; i++) {
        soma += f(a + i*h);
    }

    double integral = h * soma;
    printf("Integral aproximada (retângulos): %f\n", integral);
    return 0;
}
```

A regra dos retângulos é simples, mas menos precisa que, por exemplo, a regra do trapézio, especialmente se $f(x)$ varia rapidamente.

Regra do Trapézio

A regra do trapézio melhora a aproximação usando a média entre os valores de $f(x)$ nos extremos de cada subdivisão:

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

```
/* Integração numérica usando trapézio */
#include <stdio.h>
#include <math.h>

double f(double x) { return sin(x); } // função a integrar

int main() {
    double a = 0, b = M_PI; // limites de integração
    int N = 100;             // número de subdivisões
    double h = (b-a)/N;
    double soma = 0.0;

    // soma dos pontos internos
    for(int i=1; i<N; i++) {
        soma += f(a + i*h);
    }

    double integral = h * ( (f(a)+f(b))/2.0 + soma );
    printf("Integral aproximada (trapézio): %f\n", integral);
    return 0;
}
```

A regra do trapézio é mais precisa que a dos retângulos, mantendo o código simples e direto.

Integração a partir de Arquivo de Dados

Quando temos dados experimentais ou de simulações armazenados em um arquivo, podemos calcular a integral aproximada usando a regra do trapézio. O programa abaixo lê um arquivo `tabela2.dat` com duas colunas (valores de x e $y = f(x)$) e calcula a integral.

Exemplo de arquivo `tabela2.dat` :

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

```
/* Integração de dados lidos de arquivo usando regra do trapézio */
#include <stdio.h>

int main() {

    // abrindo arquivo de leitura
    FILE *fp = fopen("tabela2.dat", "r");


    double x[100], y[100]; // arrays para armazenar os dados
    int n = 0;

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

    double soma = 0.0;

    // calcula a integral usando a regra do trapézio
    for(int i=0; i<n-1; i++) {
        soma += (y[i] + y[i+1]) * (x[i+1] - x[i]) / 2.0;
    }

    printf("Integral aproximada: %.2f\n", soma);
    return 0;
}
```

Após executar este programa, a integral aproximada dos valores tabulados será exibida no terminal. Este método é útil para processar dados experimentais ou simulações, quando a função não é conhecida analiticamente.

Integral Dupla

Para integrais em duas variáveis, usamos soma dupla sobre uma grade de pontos.

```
/* Integral dupla simples: f(x,y) = x*y */
#include <stdio.h>

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

int main() {
    int Nx=50, Ny=50;
    double ax=0, bx=1, ay=0, by=1;
    double hx=(bx-ax)/Nx, hy=(by-ay)/Ny;
    double soma=0.0;

    for(int i=0; i<Nx; i++) {
        for(int j=0; j<Ny; j++) {
            double x = ax + i*hx;
```

```
            double y = ay + j*hy;
            soma += f(x,y);
        }
    }

    double integral = soma * hx * hy;
    printf("Integral dupla ~ %f\n", integral);
    return 0;
}
```

Esses métodos permitem calcular derivadas e integrais de funções conhecidas ou de dados discretos, oferecendo uma ferramenta prática para física, engenharia e ciências aplicadas.

6 Introdução aos Números Aleatórios

Computadores não geram números totalmente aleatórios, mas sim **pseudoaleatórios**. Em C, a função padrão para gerar números inteiros pseudoaleatórios é `rand()`.
O código abaixo mostra como gerar alguns números inteiros aleatórios simples:

```
/* Geração de números inteiros pseudoaleatórios */
#include <stdio.h>
#include <stdlib.h>

int main() {
    for(int i = 0; i < 5; i++) {
        int r = rand(); // gera número inteiro pseudoaleatório
        printf("%d\n", r);
    }
    return 0;
}
```

Note que, sem definir a *semente*, a sequência será sempre a mesma a cada execução.

Definindo a Semente do Gerador

Para variar a sequência, podemos usar `srand()` com o tempo atual:

```
/* Definindo a semente para gerar sequências diferentes */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL)); // inicializa a semente com o tempo atual
    for(int i = 0; i < 5; i++) {
        printf("%d\n", rand()); // números diferentes a cada execução
    }
    return 0;
}
```

Distribuição Uniforme no Intervalo [0,1]

Para obter números pseudoaleatórios em ponto flutuante no intervalo $[0, 1]$, usamos:

$$x = \frac{\texttt{rand()}}{RAND\_MAX}$$

```
/* Números uniformes entre 0 e 1 */
#include <stdio.h>
#include <stdlib.h>

int main() {
    for(int i = 0; i < 5; i++) {
        double x = (double)rand() / RAND_MAX; // normaliza para [0,1]
```

```
        printf("%f\n", x);
    }
    return 0;
}
```

Esses valores podem ser usados para simulações, sorteios ou métodos de Monte Carlo, mantendo a simplicidade e controle sobre a sequência pseudoaleatória.

Histograma de Números Aleatórios

Um histograma é uma representação gráfica da frequência de valores em uma sequência de números. Para números aleatórios uniformes, podemos dividir o intervalo $[0, 1]$ em "bins" (subintervalos) e contar quantos números caem em cada bin.

```
/* Histograma de números aleatórios uniformes */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int N = 1000;     // quantidade de números a gerar
    int bins = 10;     // número de subintervalos
    int hist[10] = {0}; // vetor para contar frequências

    // gera números uniformes e incrementa o bin correspondente
    for(int i=0; i<N; i++) {
        double u = (double)rand()/RAND_MAX; // número aleatório em [0,1]
        int k = (int)(u * bins);            // identifica o bin
        if(k >= bins) k = bins-1;           // garante que k está dentro do intervalo
        hist[k]++;
    }

    // exibe o histograma
    for(int i=0; i<bins; i++) {
        printf("Bin %d: %d\n", i, hist[i]);
    }

    return 0;
}
```

Após a execução, cada linha mostra quantos números caíram em cada intervalo. Este método permite visualizar a distribuição dos números gerados e é útil para verificar uniformidade ou outras propriedades estatísticas.

Distribuição $P(x) = Cx^n$

Podemos gerar números aleatórios com distribuições diferentes aplicando transformações sobre números uniformes $u \in [0, 1]$. Por exemplo, para obter $P(x) \sim x^n$, usamos a transformação:

$$x = u^{1/(n+1)}$$

A seguir, geramos uma sequência de números com esta distribuição, salvamos em arquivo e calculamos um histograma simples:

```
/* Geração de números com P(x) ~ x^n e histograma */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    int n = 2;              // expoente da distribuição
    int N = 1000;            // quantidade de números
    int bins = 10;           // número de bins do histograma
    int hist[10] = {0};      // vetor para contar frequências

    FILE *fp = fopen("numeros.dat", "w"); // arquivo para salvar números
```

```
    for(int i=0; i<N; i++) {
        double u = (double)rand()/RAND_MAX;     // número uniforme
        double x = pow(u, 1.0/(n+1));          // transformação
        fprintf(fp, "%f\n", x);                // grava no arquivo

        int k = (int)(x * bins);               // calcula bin
        if(k >= bins) k = bins-1;              // corrige limite superior
        hist[k]++;
    }

    fclose(fp);

    FILE *fh = fopen("histograma.dat", "w");   // arquivo para histograma
    for(int i=0; i<bins; i++) {
        fprintf(fh, "%d %d\n", i, hist[i]);   // bin, frequência
    }
    fclose(fh);

    return 0;
}
```

Após a execução, o arquivo `numeros.dat` conterá os valores gerados com distribuição $P(x) \sim x^n$, e `histograma.dat` mostrará a contagem de cada bin, permitindo visualizar a forma da distribuição.

Geradores Lineares Congruentes

Um dos métodos clássicos para gerar números pseudoaleatórios é o **gerador linear congruente**:

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

A seguir, geramos uma sequência de números pseudoaleatórios normalizados em [0,1] e gravamos em arquivo:

```
/* Gerador linear congruente com saída em arquivo */
#include <stdio.h>
#include <stdlib.h>

int main() {
    long a = 1664525, c = 1013904223, m = 2147483648;
    long x = 12345;           // semente inicial
    int N = 1000;             // quantidade de números
    FILE *fp = fopen("lcg.dat", "w"); // arquivo de saída

    for(int i=0; i<N; i++) {
        x = (a*x + c) % m;
        double u = (double)x / m;
        fprintf(fp, "%f\n", u);
    }

    fclose(fp);
    return 0;
}
```

Após executar, o arquivo `lcg.dat` conterá os números pseudoaleatórios gerados, que podem ser usados em simulações ou análises estatísticas.

Distribuição Gaussiana (Box-Muller)

A distribuição normal ou Gaussiana tem densidade de probabilidade:

$$P(z) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\Big[-\frac{(z-\mu)^2}{2\sigma^2}\Big]$$

com média μ e desvio padrão σ. O método Box-Muller transforma dois números uniformes $u_1, u_2 \in [0, 1]$ em dois números gaussianos independentes:

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

```
/* Números gaussianos e histograma */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    int N = 1000;       // quantidade de números
    int bins = 20;       // número de bins do histograma
    int hist[20] = {0};

    FILE *fp = fopen("gauss.dat","w");   // arquivo para números gaussianos

    for(int i=0; i<N; i++) {
        double u1 = (double)rand()/RAND_MAX;
        double u2 = (double)rand()/RAND_MAX;
        double z = sqrt(-2*log(u1))*cos(2*M_PI*u2);

        fprintf(fp, "%f\n", z);

        int k = (int)((z+4)/8*bins); // aproximação para bin entre [-4,4]
        if(k<0) k=0; if(k>=bins) k=bins-1;
        hist[k]++;
    }
    fclose(fp);

    FILE *fh = fopen("histograma_gauss.dat","w"); // arquivo do histograma
    for(int i=0;i<bins;i++) {
        fprintf(fh, "%d %d\n", i, hist[i]);
    }
    fclose(fh);

    return 0;
}
```

Após a execução, o arquivo `gauss.dat` conterá os números gaussianos, e `histograma_gauss.dat` mostrará a frequência de cada bin, permitindo visualizar a distribuição normal.

Distribuição Lorentziana

A distribuição de Lorentz possui caudas largas, o que significa que valores extremos são mais prováveis do que em uma distribuição Gaussiana. Podemos gerar números Lorentzianos usando a transformação:

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

onde u é uniforme em [0,1]. Também podemos criar um histograma para visualizar a frequência dos valores.

```
/* Números aleatórios Lorentzianos e histograma */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main() {
    int N = 1000;          // quantidade de números
    int bins = 20;          // número de bins para o histograma
    double hist[20] = {0};  // vetor para contagem
    double x_min = -10.0, x_max = 10.0;  // limites para o histograma
    double dx = (x_max - x_min)/bins;

    FILE *fp = fopen("lorentz.dat", "w");    // arquivo para números
```

```
    FILE *fh = fopen("hist_lorentz.dat", "w"); // arquivo para histograma

    for(int i=0; i<N; i++) {
        double u = (double)rand()/RAND_MAX;
        double x = tan(M_PI*(u-0.5)); // número Lorentziano
        fprintf(fp, "%f\n", x);

        // incrementa bin se dentro do intervalo
        if(x >= x_min && x < x_max) {
            int k = (int)((x - x_min)/dx);
            hist[k]++;
        }
    }

    fclose(fp);

    // escreve histograma em arquivo
    for(int i=0; i<bins; i++) {
        double bin_center = x_min + (i+0.5)*dx;
        fprintf(fh, "%f %f\n", bin_center, hist[i]);
    }
    fclose(fh);

    return 0;
}
```

Após a execução, o arquivo `lorentz.dat` conterá os números gerados, e `hist_lorentz.dat` mostrará a frequência de cada bin. O histograma permite visualizar a característica de caudas largas típica da distribuição de Lorentz.

Autocorrelação Simples

Para uma série x_i, $i = 1, 2, \ldots, N$, a autocorrelação no lag k é definida por:

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

onde $C(0)$ é a variância da série. Este método indica se valores da série estão correlacionados com seus vizinhos a uma distância k.

```
/* Autocorrelação de uma série maior (N=2000) */
#include <stdio.h>
#include <stdlib.h>

#define N 2000
#define MAX_LAG 50

int main() {
    double x[N], media=0.0;
    double C[MAX_LAG+1];

    // gera números pseudoaleatórios uniformes em [0,1]
    for(int i=0; i<N; i++) {
        x[i] = (double)rand()/RAND_MAX;
        media += x[i];
    }
    media /= N;

    // calcula autocorrelação para lags 0 a MAX_LAG
    for(int k=0; k<=MAX_LAG; k++) {
        double soma = 0.0;
        for(int i=0; i<N-k; i++) {
            soma += (x[i]-media)*(x[i+k]-media);
        }
        C[k] = soma / (N-k);
        printf("Lag %d: C = %f\n", k, C[k]);
    }

    return 0;
}
```

O programa acima calcula a autocorrelação de uma série com $N = 2000$ para lags até 50. Lag 0 fornece a variância, e lags maiores mostram possíveis correlações entre elementos distantes da série. Este método é amplamente usado em análise de séries temporais e em simulações de Monte Carlo.

Caminhante Aleatório Clássico

O modelo de caminhante aleatório em 1D representa um processo de difusão simples: a cada passo, o caminhante move-se para a esquerda ou direita com igual probabilidade. O comportamento típico é que o deslocamento médio quadrático cresce linearmente com o número de passos:

$$\langle x^2 \rangle = N \cdot (\text{passo})^2$$

```
/* Caminhante Aleatório em 1D com <x^2> */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int Npassos = 1000;       // número total de passos
    int pos = 0;               // posição inicial
    double x2_acum = 0.0;      // acumula x^2 para cada passo

    FILE *fp = fopen("x2.dat", "w"); // arquivo para salvar <x^2>

    for(int i=1; i<=Npassos; i++) {
        int r = rand()%2;      // 0 ou 1
        if(r==0) pos--; else pos++;
        x2_acum += pos*pos;

        // escreve o passo e <x^2> médio até agora
        fprintf(fp, "%d %f\n", i, x2_acum/i);
    }

    fclose(fp);
    return 0;
}
```

Após executar o programa, o arquivo `x2.dat` conterá o número de passos e o deslocamento médio quadrático correspondente. Plotando $\langle x^2 \rangle$ versus o número de passos, observa-se o crescimento linear característico da difusão clássica.

7 Regressão Linear

A regressão linear é uma técnica para ajustar uma reta aos dados (x_i, y_i) de forma que a soma dos quadrados dos desvios seja mínima. A equação da reta é:

$$y = a + bx$$

com coeficientes calculados por:

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

O código a seguir calcula os coeficientes a e b para um conjunto de dados discretos.

```
/* Regressão linear simples em C */
#include <stdio.h>

int main() {
    // Dados de exemplo
    double x[5] = {1, 2, 3, 4, 5};
    double y[5] = {2, 4, 5, 4, 5};
```

```
    int n = 5;

    // Calcula médias de x e y
    double soma_x=0, soma_y=0;
    for(int i=0;i<n;i++){
        soma_x += x[i];
        soma_y += y[i];
    }
    double media_x = soma_x/n;
    double media_y = soma_y/n;

    // Cálculo da inclinação b e intercepto a
    double num=0, den=0;
    for(int i=0;i<n;i++){
        num += (x[i]-media_x)*(y[i]-media_y);
        den += (x[i]-media_x)*(x[i]-media_x);
    }
    double b = num/den;
    double a = media_y - b*media_x;

    printf("Ajuste linear: y = %f + %f x\n", a, b);
    return 0;
}
```

Exemplo com Lei de Potência:
Se os dados seguem aproximadamente $y = Cx^n$, podemos linearizar tomando logaritmo:

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

Aplicando regressão linear sobre $(\ln x_i, \ln y_i)$, obtemos o expoente n e constante C.

```
/* Regressão linear para lei de potência */
#include <stdio.h>
#include <math.h>

int main() {
    // Dados exemplo: y = x^2
    double x[6] = {1,2,3,4,5,6};
    double y[6] = {1,4,9,16,25,36};
    int n = 6;

    // Transformação logarítmica
    double ln_x[6], ln_y[6];
    for(int i=0;i<n;i++){
        ln_x[i] = log(x[i]);
        ln_y[i] = log(y[i]);
    }

    // Médias
    double soma_x=0, soma_y=0;
    for(int i=0;i<n;i++){
        soma_x += ln_x[i];
        soma_y += ln_y[i];
    }
    double media_x = soma_x/n;
    double media_y = soma_y/n;

    // Cálculo da inclinação n e ln(C)
    double num=0, den=0;
    for(int i=0;i<n;i++){
        num += (ln_x[i]-media_x)*(ln_y[i]-media_y);
        den += (ln_x[i]-media_x)*(ln_x[i]-media_x);
    }
    double n_exp = num/den;                  // expoente da lei de potência
    double lnC = media_y - n_exp*media_x;    // ln(C)
    double C = exp(lnC);

    printf("Lei de potência ajustada: y = %f * x^%f\n", C, n_exp);
    return 0;
}
```

Este método permite identificar a relação de tipo potência em dados experimentais ou simulados, transformando o problema em uma regressão linear clássica sobre os logaritmos.

8 Interpolação

A interpolação é uma técnica fundamental em física computacional para estimar valores de uma função a partir de um conjunto discreto de pontos conhecidos. O método mais simples é a interpolação linear, que conecta pares de pontos por retas. Versões mais sofisticadas incluem a interpolação polinomial, que ajusta um único polinômio a vários pontos, e as splines cúbicas, que utilizam polinômios por trechos de forma suave e estável. Cada abordagem apresenta vantagens e limitações, sendo escolhida de acordo com a precisão desejada e a natureza dos dados.

Interpolação linear

A interpolação linear é o método mais simples: entre dois pontos (x_0, y_0) e (x_1, y_1), o valor interpolado em x é dado por:

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

```
/* Interpolacao Linear em C */
#include <stdio.h>

/* Funcao que faz interpolacao linear */
double interp_linear(double x, double x0, double y0, double x1, double y1) {
    return y0 + ( (y1 - y0) / (x1 - x0) ) * (x - x0);
}

int main() {
    double x0 = 1.0, y0 = 2.0;
    double x1 = 3.0, y1 = 6.0;
    double x = 2.0; // ponto onde queremos interpolar

    double y = interp_linear(x, x0, y0, x1, y1);
    printf("Interpolacao linear em x=%.2f -> y=%.2f\n", x, y);

    return 0;
}
```

Este exemplo mostra a forma mais simples de interpolação: traçar uma reta entre dois pontos conhecidos e estimar o valor em um ponto intermediário. Embora seja limitado em precisão para funções muito curvas, o método linear é rápido, fácil de implementar e bastante eficaz quando os pontos estão próximos.

Interpolação Polinomial (Lagrange)

A interpolação polinomial busca construir um único polinômio que passe exatamente por todos os pontos conhecidos de um conjunto de dados. A ideia é substituir a função original por uma expressão polinomial que seja simples de avaliar e manipular numericamente. Para três pontos $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, obtemos um polinômio de grau 2, chamado polinômio de Lagrange. A forma geral é

$$P(x) = \sum_{i=0}^{2} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j},$$

que garante que $P(x_i) = y_i$ para cada ponto do conjunto.

```
/* Interpolacao Polinomial de Lagrange (3 pontos) */
#include <stdio.h>

/* Funcao Lagrange para 3 pontos */
double interp_lagrange(double x, double x0,double y0,
                               double x1,double y1,
                               double x2,double y2) {
    double L0 = ((x - x1)*(x - x2))/((x0 - x1)*(x0 - x2));
    double L1 = ((x - x0)*(x - x2))/((x1 - x0)*(x1 - x2));
    double L2 = ((x - x0)*(x - x1))/((x2 - x0)*(x2 - x1));
    return y0*L0 + y1*L1 + y2*L2;
}

int main() {
    // pontos de exemplo: y = x^2
    double x0=1,y0=1, x1=2,y1=4, x2=3,y2=9;

    double x = 2.5;
    double y = interp_lagrange(x,x0,y0,x1,y1,x2,y2);

    printf("Interpolacao Lagrange em x=%.2f -> y=%.2f\n", x, y);
    return 0;
}
```

O código acima implementa a forma mais simples do polinômio de Lagrange para três pontos, gerando uma parábola que passa exatamente por eles. Esse método permite capturar curvaturas da função original de forma mais precisa que a interpolação linear, mas pode introduzir oscilações indesejadas quando aplicado a muitos pontos.

Interpolação por Splines Cúbicas

As splines cúbicas utilizam polinômios de grau 3 em cada intervalo entre os pontos de dados, garantindo continuidade da função, da primeira derivada e da segunda derivada. Isso torna o método mais suave e estável do que a interpolação polinomial simples ou linear, evitando oscilações indesejadas em grandes conjuntos de pontos. Matematicamente, se temos $n + 1$ pontos $(x_0, y_0), \ldots, (x_n, y_n)$, a spline cúbica $S(x)$ é definida por polinômios cúbicos $S_i(x)$ em cada intervalo $[x_i, x_{i+1}]$:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad i = 0, \ldots, n - 1$$

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

1. $S_i(x_i) = y_i$ e $S_i(x_{i+1}) = y_{i+1}$ — a spline passa pelos pontos.
2. $S_i'(x_{i+1}) = S_{i+1}'(x_{i+1})$ — continuidade da primeira derivada.
3. $S_i''(x_{i+1}) = S_{i+1}''(x_{i+1})$ — continuidade da segunda derivada.

4. Condições de borda (por exemplo, derivada segunda nula nas extremidades para a spline natural).

Como casos especiais:

- Para **dois pontos**, a spline cúbica degenera na **interpolação linear**, já que não há intervalos intermediários para curvatura.
- Para **três pontos**, é possível construir diretamente os polinômios cúbicos usando os sistemas lineares resultantes das condições acima, obtendo uma aproximação suave mesmo com poucos pontos.

Assim, mesmo em exemplos simples, a spline cúbica garante que a curva resultante seja suave, com transições contínuas de inclinação e curvatura, ao contrário de polinômios de grau alto que podem oscilar de forma instável.

```
/* Spline Cubica muito simples (3 pontos, sem derivadas) */
#include <stdio.h>

/* Funcao spline cubica simples:
   Aqui usamos apenas uma forma aproximada
   (na pratica, bibliotecas proprias sao usadas). */
double spline3(double x, double x0,double y0,
                         double x1,double y1,
                         double x2,double y2) {
    // Ajusta um polinomio quadratico como "mini spline"
    double L0 = ((x - x1)*(x - x2))/((x0 - x1)*(x0 - x2));
    double L1 = ((x - x0)*(x - x2))/((x1 - x0)*(x1 - x2));
    double L2 = ((x - x0)*(x - x1))/((x2 - x0)*(x2 - x1));
    return y0*L0 + y1*L1 + y2*L2;
}

int main() {
    // exemplo com pontos y = x^2
    double x0=0,y0=0, x1=1,y1=1, x2=2,y2=4;

    double x = 1.5;
    double y = spline3(x,x0,y0,x1,y1,x2,y2);

    printf("Spline 'cubica' simples em x=%.2f -> y=%.2f\n", x, y);
    return 0;
}
```

O exemplo acima mostra uma implementação muito simplificada de spline cúbica para apenas três pontos. Na prática, usamos um polinômio quadrático como aproximação, sem considerar derivadas nas bordas. Apesar de simples, ele já ilustra como os polinômios de Lagrange podem ser usados para interpolação. Para conjuntos maiores de pontos, é recomendável utilizar bibliotecas próprias que garantem continuidade de primeira e segunda derivadas. Este código serve como uma introdução didática ao conceito de spline cúbica.

Parte 2

Estas próximas notas de aula introduzem os temas que serão desenvolvidos na **Parte 2** do curso de Física Computacional. O foco recai sobre algumas das principais técnicas numéricas utilizadas em física, apresentadas de maneira conceitual, com suas formulações discretas, exemplos aplicados e códigos em linguagem C. Iniciaremos com métodos para a solução de equações diferenciais ordinárias (EDOs), utilizando como exemplo central o sistema massa–mola sem atrito. Esse modelo simples e clássico permitirá discutir aspectos fundamentais como estabilidade, conservação de energia e precisão numérica. Na sequência, exploraremos métodos de **diagonalização de matrizes**, essenciais para o estudo de problemas de autovalores em física, como a resolução de Hamiltonianos e a análise de modos normais. O objetivo geral desta parte é fornecer um material direto, autoexplicativo e útil para estudantes, enfatizando não apenas a aplicação prática dos algoritmos, mas também suas interpretações físicas, vantagens e limitações.

9 Método de Euler (Explícito)

O método de Euler *explícito* é o mais simples para resolver EDOs numericamente. Ele se baseia na aproximação da derivada por uma diferença finita:

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

É um método de primeira ordem, com erro local da ordem de $\mathcal{O}(\Delta t^2)$ e erro global de ordem $\mathcal{O}(\Delta t)$. Apesar de simples, pode ser instável para integrações longas ou sistemas rígidos.

Equações para o sistema massa-mola

Considere um sistema massa-mola sem atrito:

$$\frac{d^2x}{dt^2} = -\omega^2 x \quad \Rightarrow \quad \begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\omega^2 x \end{cases}$$

Quebrar a equação de segunda ordem em duas de primeira ordem permite aplicar métodos numéricos padrão desenvolvidos para EDOs de primeira ordem(como Euler ou Runge-Kutta). Além disso, essa reformulação facilita a implementação computacional e a análise da dinâmica do sistema. Aplicando Euler explícito, temos:

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

Código em C: Massa-Mola com Euler

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

#define N 1000       // número de passos de tempo
#define DT 0.01      // passo de tempo
#define OMEGA 1.0    // frequência natural

int main() {
    double x = 1.0;   // posição inicial
    double v = 0.0;   // velocidade inicial
    double t;

    FILE *f = fopen("massa_mola_euler.dat", "w");

    for (int i = 0; i < N; i++) {
        t = i * DT;
        fprintf(f, "%f %f %f\n", t, x, v);

        // Atualização via método de Euler
        double x_novo = x + DT * v;
        double v_novo = v - DT * OMEGA * OMEGA * x;

        // Atualiza as variáveis para o próximo passo
        x = x_novo;
        v = v_novo;
    }

    fclose(f);
    return 0;
}
```

10 Diferença Finita Centradas no Tempo

Outro método eficiente para resolver EDOs de segunda ordem, como o sistema massa-mola, é o método das diferenças finitas centradas no tempo. Ele utiliza três pontos consecutivos da malha temporal para estimar a derivada de segunda ordem:

Equação Geral

Para uma equação do tipo:

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

A aproximação de segunda ordem para a derivada é dada por:

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

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

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

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

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

Esse é um método explícito de segunda ordem, com erro local de $\mathcal{O}(\Delta t^3)$ e erro global de $\mathcal{O}(\Delta t^2)$.

Inicialização

Como a fórmula depende de x_{n-1}, precisamos de dois valores iniciais:
- $x_0 = x(t=0)$
- x_1, que pode ser obtido usando o método de Euler ou a expansão de Taylor:

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

Código em C: Massa-Mola com Diferença Finita

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

#define N 1000        // número de passos de tempo
#define DT 0.01       // passo de tempo
#define OMEGA 1.0     // frequência natural

int main() {
    double x[N];      // posições
    double v0 = 0.0;  // velocidade inicial
    double t;

    // Condições iniciais
    x[0] = 1.0;  // posição inicial

    // Inicializa x[1] com expansão de Taylor
    x[1] = x[0] + DT * v0 - 0.5 * DT * DT * OMEGA * OMEGA * x[0];

    FILE *f = fopen("massa_mola_dif_finitas.dat", "w");
```

```
    for (int i = 0; i < N; i++) {
        t = i * DT;
        fprintf(f, "%f %f\n", t, x[i]);

        // Atualiza posição futura usando fórmula de diferenças finitas
        if (i >= 1 && i < N - 1) {
            x[i+1] = 2 * x[i] - x[i-1] - DT * DT * OMEGA * OMEGA * x[i];
        }
    }

    fclose(f);
    return 0;
}
```

Observações

- Esse método é mais estável que o de Euler explícito para sistemas oscilatórios.
- A velocidade v_n não é calculada diretamente, mas pode ser estimada como:

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

- A conservação de energia é melhor preservada do que no método de Euler.

11 Método de Runge-Kutta de 4ª Ordem (RK4)

O método de Runge-Kutta de 4ª ordem é um dos métodos mais precisos e estáveis para resolver EDOs. Ele calcula quatro estimativas da derivada por passo e combina essas estimativas de forma ponderada para obter uma aproximação de alta precisão:

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

Equações para o sistema massa-mola

O sistema massa-mola sem atrito é descrito por um sistema de duas EDOs de primeira ordem:

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

Para aplicar o método de Runge-Kutta de 4ª ordem (RK4), devemos tratar esse sistema como um vetor de variáveis:

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

Com o método RK4, a atualização da solução se dá por:

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

onde os vetores $\mathbf{k}_i$ são:

$$\mathbf{k}_1 = \mathbf{f}(\mathbf{y}_n, t_n) = \begin{pmatrix} v_n \\ -\omega^2 x_n \end{pmatrix}$$

$$\mathbf{k}_2 = \mathbf{f}\left(\mathbf{y}_n + \frac{\Delta t}{2}\mathbf{k}_1, t_n + \frac{\Delta t}{2}\right) = \begin{pmatrix} v_n + \frac{\Delta t}{2} \cdot (-\omega^2 x_n) \\ -\omega^2 \left(x_n + \frac{\Delta t}{2} v_n\right) \end{pmatrix}$$

$$\mathbf{k}_3 = \mathbf{f}\left(\mathbf{y}_n + \frac{\Delta t}{2}\mathbf{k}_2, t_n + \frac{\Delta t}{2}\right) = \begin{pmatrix} v_n + \frac{\Delta t}{2} \cdot k_{2,v} \\ -\omega^2 \left(x_n + \frac{\Delta t}{2} k_{2,x}\right) \end{pmatrix}$$

$$\mathbf{k}_4 = \mathbf{f}\left(\mathbf{y}_n + \Delta t \cdot \mathbf{k}_3, t_n + \Delta t\right) = \begin{pmatrix} v_n + \Delta t \cdot k_{3,v} \\ -\omega^2 \left(x_n + \Delta t \cdot k_{3,x}\right) \end{pmatrix}$$

Finalmente, atualizamos as variáveis x e v separadamente:

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

Onde usamos as notações $k_{i,x}$ e $k_{i,v}$ para as componentes de posição e velocidade dos vetores $\mathbf{k}_i$, respectivamente. Esse procedimento fornece uma aproximação muito precisa da trajetória oscilatória da partícula presa à mola, preservando bem a energia total do sistema ao longo do tempo.

Código em C: Massa-Mola com RK4

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

#define N 1000
#define DT 0.01
#define OMEGA 1.0

int main() {
    double x = 1.0, v = 0.0;
    FILE *f = fopen("massa_mola_rk4.dat", "w");

    for (int i = 0; i < N; i++) {
        double t = i * DT;
        fprintf(f, "%f %f %f\n", t, x, v);

        // Cálculo dos coeficientes k para x e v
        double k1x = DT * v;
        double k1v = -DT * OMEGA * OMEGA * x;

        double k2x = DT * (v + 0.5 * k1v);
        double k2v = -DT * OMEGA * OMEGA * (x + 0.5 * k1x);

        double k3x = DT * (v + 0.5 * k2v);
        double k3v = -DT * OMEGA * OMEGA * (x + 0.5 * k2x);

        double k4x = DT * (v + k3v);
        double k4v = -DT * OMEGA * OMEGA * (x + k3x);

        x += (k1x + 2*k2x + 2*k3x + k4x) / 6.0;
        v += (k1v + 2*k2v + 2*k3v + k4v) / 6.0;
    }

    fclose(f);
    return 0;
}
```

12 Método de Adams-Bashforth (2ª Ordem)

O método de Adams-Bashforth é um método explícito multi-passo, que usa os valores de $f(t, y)$ em passos anteriores para estimar o próximo valor da solução. A fórmula de Adams-Bashforth de 2ª ordem vem da aproximação da integral $y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(t)\,dt$ por interpolação linear de $f(t)$ nos pontos t_{n-1} e t_n. Usando o polinômio de Lagrange de grau 1 e integrando no intervalo $[t_n, t_{n+1}]$, obtemos:

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

Substituindo na equação de evolução, resulta:

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

Para sistemas de EDOs, como o massa-mola:

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

O método de Adams-Bashforth pode ser aplicado separadamente a cada equação:

$$\begin{aligned} x_{n+1} &= x_n + \frac{\Delta t}{2}\,(3v_n - v_{n-1}) \\ v_{n+1} &= v_n + \frac{\Delta t}{2}\,\left(-3\omega^2 x_n + \omega^2 x_{n-1}\right) \end{aligned}$$

Inicialização

Como o método usa dois passos, precisamos de x_0, x_1 e v_0, v_1. Esses podem ser obtidos com o método de Euler:

$$\begin{cases} x_1 = x_0 + \Delta t \cdot v_0 \\ v_1 = v_0 - \Delta t \cdot \omega^2 x_0 \end{cases}$$

Código em C: Massa-Mola com Adams-Bashforth (2ª ordem)

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

#define N 1000
#define DT 0.01
#define OMEGA 1.0

int main() {
    double x[N], v[N];
    double t;

    // Condições iniciais
    x[0] = 1.0;
    v[0] = 0.0;

    // Inicialização com método de Euler
    x[1] = x[0] + DT * v[0];
    v[1] = v[0] - DT * OMEGA * OMEGA * x[0];

    FILE *f = fopen("massa_mola_adams.dat", "w");
```

```
    for (int i = 1; i < N - 1; i++) {
        t = i * DT;
        fprintf(f, "%f %f %f\n", t, x[i], v[i]);

        x[i+1] = x[i] + 0.5 * DT * (3 * v[i] - v[i-1]);
        v[i+1] = v[i] + 0.5 * DT * (-3 * OMEGA * OMEGA * x[i] + OMEGA * OMEGA * x[i-1]);
    }

    fclose(f);
    return 0;
}
```

13 Método de Taylor de 2ª Ordem

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

$$y_{n+1} = y_n + \Delta t \cdot y'_n + \frac{\Delta t^2}{2} \cdot y''_n \tag{3}$$

Aplicando ao sistema massa-mola, com:

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\omega^2 x \end{cases} \Rightarrow \frac{d^2x}{dt^2} = \frac{dv}{dt} = -\omega^2 x \quad ; \quad \frac{d^2v}{dt^2} = -\omega^2 v$$

A aproximação de Taylor para cada variável é:

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

Código em C: Massa-Mola com Taylor (2ª ordem)

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

#define N 1000
#define DT 0.01
#define OMEGA 1.0

int main() {
    double x = 1.0;
    double v = 0.0;
    double t;

    FILE *f = fopen("massa_mola_taylor.dat", "w");

    for (int i = 0; i < N; i++) {
        t = i * DT;
        fprintf(f, "%f %f %f\n", t, x, v);

        double x_novo = x + DT * v - 0.5 * DT * DT * OMEGA * OMEGA * x;
        double v_novo = v - DT * OMEGA * OMEGA * x - 0.5 * DT * DT * OMEGA * OMEGA * v;

        x = x_novo;
        v = v_novo;
    }

    fclose(f);
    return 0;
}
```

14 Método de Verlet com Velocidade (Velocity-Verlet)

O método *Velocity-Verlet* é muito utilizado em simulações de dinâmica molecular e sistemas oscilatórios, como o sistema massa-mola. Ele oferece boa precisão (ordem 2) e estabilidade numérica, especialmente para sistemas conservativos (sem dissipação). Esse método atualiza as posições x e velocidades v de maneira acoplada, usando a aceleração a que depende da posição. Para o sistema massa-mola sem atrito, a aceleração é dada por:

$$a(t) = \frac{dv}{dt} = -\omega^2 x(t)$$

A ideia do método é:

1. Atualizar a posição usando a posição e velocidade atuais e a aceleração no instante atual;
2. Calcular a nova aceleração baseada na nova posição;
3. Atualizar a velocidade usando a média entre a aceleração atual e a nova.

O algoritmo discreto completo para o sistema massa-mola é:

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

Passo a passo para implementar:

1. Comece com valores iniciais para posição x_0, velocidade v_0 e calcule $a_0 = -\omega^2 x_0$;
2. Use a equação da posição para obter x_1;
3. Calcule a nova aceleração $a_1 = -\omega^2 x_1$;
4. Atualize a velocidade usando $v_1 = v_0 + \frac{1}{2}(a_0 + a_1) \cdot \Delta t$;
5. Repita o processo para os próximos passos.

Este método é especialmente bom para o sistema massa-mola porque conserva aproximadamente a energia do sistema ao longo do tempo, ao contrário do método de Euler explícito, que tende a gerar crescimento ou decaimento espúrio na energia total.

Código em C: Massa-Mola com Velocity-Verlet

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

#define N 1000
#define DT 0.01
#define OMEGA 1.0

int main() {
    double x = 1.0, v = 0.0;
    double a = -OMEGA * OMEGA * x;

    FILE *f = fopen("massa_mola_verlet.dat", "w");

    for (int i = 0; i < N; i++) {
        double t = i * DT;
```

```
        fprintf(f, "%f %f %f\n", t, x, v);

        // Atualiza posição com aceleração atual
        double x_new = x + v * DT + 0.5 * a * DT * DT;

        // Calcula nova aceleração com nova posição
        double a_new = -OMEGA * OMEGA * x_new;

        // Atualiza velocidade com média das acelerações
        double v_new = v + 0.5 * (a + a_new) * DT;

        // Prepara para próximo passo
        x = x_new;
        v = v_new;
        a = a_new;
    }

    fclose(f);
    return 0;
}
```

15 Método Leap-Frog

O método *Leap-Frog* (em português, "salto de sapo") é um integrador numérico de segunda ordem muito utilizado para simular sistemas oscilatórios, como o sistema massa-mola, principalmente quando se deseja preservar a energia ao longo do tempo. Sua principal característica é que as atualizações das variáveis de posição e velocidade ocorrem em instantes de tempo intercalados (desfasados em meio passo de tempo), o que confere estabilidade e boa conservação de energia para sistemas hamiltonianos. Para o sistema massa-mola sem atrito, a aceleração é dada por:

$$a(t) = \frac{dv}{dt} = -\omega^2 x(t)$$

O algoritmo discreto segue os seguintes passos:

$$\begin{aligned}
v_{n+\frac{1}{2}} &= v_n + \frac{\Delta t}{2} \cdot a_n \quad \text{(meio passo da velocidade)} \\
x_{n+1} &= x_n + \Delta t \cdot v_{n+\frac{1}{2}} \quad \text{(passo completo da posição)} \\
a_{n+1} &= -\omega^2 x_{n+1} \quad \text{(nova aceleração)} \\
v_{n+1} &= v_{n+\frac{1}{2}} + \frac{\Delta t}{2} \cdot a_{n+1} \quad \text{(completa a velocidade)}
\end{aligned}$$

Como aplicar no problema massa-mola:
1. Comece com os valores iniciais x_0 e v_0, e calcule a aceleração inicial $a_0 = -\omega^2 x_0$;
2. Faça um meio passo de velocidade: $v_{1/2} = v_0 + \frac{\Delta t}{2} a_0$;
3. Atualize a posição: $x_1 = x_0 + \Delta t \cdot v_{1/2}$;
4. Calcule a nova aceleração: $a_1 = -\omega^2 x_1$;
5. Complete o passo da velocidade: $v_1 = v_{1/2} + \frac{\Delta t}{2} a_1$;
6. Repita o processo usando x_1, v_1, a_1 para o próximo passo.

Observação: Como as velocidades e posições estão defasadas no tempo, pode ser conveniente manter uma variável auxiliar para a velocidade em "meio passo", $v_{n+1/2}$, que será atualizada diretamente a cada passo do algoritmo. Este método é bastante eficiente e robusto para longas integrações de sistemas oscilatórios, com excelente conservação da energia, o que o torna ideal

para simulações físicas realistas.

Código em C: Massa-Mola com Leap-Frog

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

#define N 1000
#define DT 0.01
#define OMEGA 1.0

int main() {
    double x = 1.0;
    double v = 0.0;
    double a = -OMEGA * OMEGA * x;

    // Primeiro passo: meio passo de velocidade
    double v_half = v + 0.5 * DT * a;

    FILE *f = fopen("massa_mola_leapfrog.dat", "w");

    for (int i = 0; i < N; i++) {
        double t = i * DT;
        fprintf(f, "%f %f %f\n", t, x, v_half - 0.5 * DT * a);

        // Atualiza posição
        x += DT * v_half;

        // Calcula nova aceleração
        a = -OMEGA * OMEGA * x;

        // Atualiza velocidade no próximo meio passo
        v_half += 0.5 * DT * a;
    }

    fclose(f);
    return 0;
}
```

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

Equações diferenciais estocásticas (EDEs) descrevem a evolução de sistemas que sofrem flutuações aleatórias no tempo, frequentemente associadas a ruídos térmicos, ambientais ou quânticos. Uma EDE típica possui a forma:

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

onde:

- $f(y,t)$ é o termo determinístico (como em uma EDO comum);
- $g(y,t)$ modula a intensidade do ruído;
- $\xi(t)$ é um processo estocástico (geralmente ruído branco gaussiano).

Como essas equações envolvem variáveis aleatórias, a solução exata é frequentemente impossível. Utilizamos então métodos numéricos como o **Euler-Maruyama**, uma extensão do método de Euler explícito para o caso estocástico.

Método de Euler-Maruyama

O método de Euler-Maruyama aproxima a solução da EDE de forma discreta, incorporando o ruído em cada passo de tempo:

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

onde ΔW_n é uma variável aleatória com distribuição normal:

$$\Delta W_n = \sqrt{\Delta t} \cdot \mathcal{N}(0, 1)$$

Esse termo simula a integral estocástica (movimento browniano) entre os tempos t_n e t_{n+1}.

Aplicação: Equação de Langevin com Ruído Térmico

A equação de Langevin descreve o movimento de uma partícula sujeita a uma força de atrito e a flutuações térmicas. A forma simplificada (com ruído aditivo) é:

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

onde:

- γ é o coeficiente de atrito viscoso;
- m é a massa da partícula;
- $\xi(t)$ é o ruído branco gaussiano, modelando colisões aleatórias com moléculas do meio.

Discretização via Euler-Maruyama

Para implementar essa equação numericamente, usamos:

$$\begin{aligned} \xi^n &= \sqrt{2\gamma k_B T / \Delta t} \cdot \mathcal{N}(0, 1) \\ v_{n+1} &= v_n - \frac{\gamma}{m} v_n \cdot \Delta t + \frac{1}{m} \cdot \xi^n \cdot \Delta t \\ x_{n+1} &= x_n + v_{n+1} \cdot \Delta t \end{aligned}$$

ou, de forma equivalente, considerando a integral estocástica explicitamente:

$$\begin{aligned} v_{n+1} &= v_n - \frac{\gamma}{m} v_n \cdot \Delta t + \frac{1}{m} \cdot \sqrt{2\gamma k_B T \Delta t} \cdot \eta_n \\ x_{n+1} &= x_n + v_{n+1} \cdot \Delta t \end{aligned}$$

onde $\eta_n \sim \mathcal{N}(0, 1)$ é uma variável aleatória gerada a cada passo.

Observações importantes

- As equações estocásticas não possuem uma única solução, mas sim uma distribuição de soluções. Portanto, é comum simular várias trajetórias e calcular médias estatísticas.
- O passo de tempo Δt deve ser suficientemente pequeno para capturar corretamente as escalas do ruído.
- O termo $\sqrt{\Delta t} \cdot \eta_n$ garante que o ruído tem variância proporcional a Δt, como esperado para um processo de Wiener (movimento browniano).

Média quadrática do deslocamento

Um observável relevante no estudo de difusão estocástica é o deslocamento quadrático médio:

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

onde R é o número de trajetórias simuladas, e $x_r(t)$ é a posição da partícula na trajetória r no instante t. Esse valor cresce linearmente com o tempo para o movimento browniano, refletindo difusão normal. Esse comportamento foi previsto por Albert Einstein em seu trabalho seminal de 1905 sobre o movimento browniano, onde ele relacionou o deslocamento quadrático médio ao coeficiente de difusão. Sua análise teórica confirmou que $\langle x^2(t) \rangle \propto t$ para partículas em suspensão, validando a natureza estatística do movimento microscópico. Esse resultado foi fundamental para a consolidação da teoria cinética dos fluidos.

Código em C: Langevin via Euler-Maruyama

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

#define N 10000     // número de passos de tempo
#define DT 0.01     // passo de tempo
#define GAMMA 1.0   // coeficiente de atrito
#define KB 1.0      // constante de Boltzmann
#define T 1.0       // temperatura
#define M 1.0       // massa
#define R 1000      // número de trajetórias

// Função que gera números aleatórios gaussianos (Box-Muller)
double gaussrand() {
    static int pronto = 0;
    static double u, v;

    if (pronto) {
        pronto = 0;
        return sqrt(u) * sin(v);
    }

    pronto = 1;
    u = rand() / ((double) RAND_MAX);
    if (u < 1e-100) u = 1e-100; // evita log(0)
    u = -2.0 * log(u);
    v = (rand() / ((double) RAND_MAX)) * 2.0 * M_PI;
    return sqrt(u) * cos(v);
}

int main() {
    srand(time(NULL));  // inicializa gerador de números aleatórios

    double x[N] = {0};  // acumulador para <x^2(t)>
```

```
    for (int r = 0; r < R; r++) {
        double v = 0.0, xloc = 0.0;

        for (int i = 0; i < N; i++) {
            double eta = gaussrand();  // ruído gaussiano
            double dW = sqrt(2.0 * GAMMA * KB * T * DT) * eta;

            // Atualiza velocidade e posição
            v += -GAMMA * v / M * DT + dW / M;
            xloc += v * DT;

            // Acumula x² para média
            x[i] += xloc * xloc;
        }
    }

    // Escreve <x^2(t)> no arquivo
    FILE *f = fopen("langevin_euler.dat", "w");
    for (int i = 0; i < N; i++) {
        fprintf(f, "%f %f\n", i * DT, x[i] / R);
    }
    fclose(f);

    return 0;
}
```

17 Integração Numérica via Monte Carlo

O método de **Monte Carlo** é uma técnica estocástica de integração numérica baseada em amostragem aleatória. Em vez de usar somatórios determinísticos como nos métodos de trapézio ou Simpson, o método Monte Carlo estima a integral a partir da média estatística de uma função avaliada em pontos escolhidos aleatoriamente dentro do domínio de integração.

Formulação Geral

Se quisermos calcular a integral definida de uma função $f(x)$ no intervalo $[a, b]$:

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

geramos N pontos aleatórios $x_i \in [a, b]$ com distribuição uniforme e estimamos a integral como:

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

O erro típico dessa aproximação diminui como $\sim \frac{1}{\sqrt{N}}$, o que é mais lento do que os métodos determinísticos de ordem elevada, mas o método é muito poderoso em altas dimensões e para domínios complexos.

Exemplo: Estimativa de π usando Monte Carlo

Considere um quadrado de lado $L = 1$ e um quarto de círculo de raio $R = 1$ inscrito nele, no primeiro quadrante. A razão entre a área do quarto de círculo e a área do quadrado é:

$$\frac{\pi R^2/4}{R^2} = \frac{\pi}{4}$$

Se sorteamos pontos (x, y) aleatórios com $x, y \in [0, 1]$, a fração deles que cai dentro do círculo satisfaz $x^2 + y^2 \leq 1$. Assim, podemos estimar:

$$\pi \approx 4 \cdot \frac{\text{número de pontos dentro do círculo}}{N}$$

Código em C: Estimativa de π com Monte Carlo

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

#define N 1000000  // número de pontos

int main() {
    int dentro = 0;
    double x, y;

    srand(time(NULL));  // inicializa o gerador de números aleatórios

    for (int i = 0; i < N; i++) {
        x = (double) rand() / RAND_MAX;  // x em [0,1]
        y = (double) rand() / RAND_MAX;  // y em [0,1]

        if (x*x + y*y <= 1.0) {
            dentro++;
        }
    }

    double pi_est = 4.0 * dentro / N;

    printf("Estimativa de pi = %.6f\n", pi_est);
    return 0;
}
```

Aplicações e Observações

- O método Monte Carlo é especialmente útil em altas dimensões, onde métodos determinísticos tornam-se inviáveis (problema da maldição da dimensionalidade).
- Pode ser usado para estimar volumes, integrais múltiplas, valores esperados de distribuições, e problemas de física estatística e mecânica quântica.
- A convergência é lenta ($\sim 1/\sqrt{N}$), mas pode ser melhorada com técnicas como *importance sampling*, *stratified sampling*, ou *Markov Chain Monte Carlo* (MCMC).

17.1 Exemplo Simples de Integração Monte Carlo em Alta Dimensão

Considere a integral da função

$$f(\mathbf{x}) = \exp\left(-\sum_{i=1}^{d} x_i^2\right)$$

no hiper-cubo unitário $[0,1]^d$, para uma dimensão d arbitrariamente alta. Esta é uma função multidimensional suave, cuja integral é dada por:

$$I_d = \int_0^1 \cdots \int_0^1 e^{-\sum_{i=1}^{d} x_i^2}\, dx_1 \cdots dx_d$$

Calculá-la analiticamente para dimensões grandes pode ser difícil, mas o método Monte Carlo permite estimar esse valor amostrando pontos aleatórios no hiper-cubo e calculando a média das avaliações de $f(\mathbf{x})$.

Estimativa via Monte Carlo Geramos N pontos $\mathbf{x}^{(j)} = (x_1^{(j)}, \ldots, x_d^{(j)})$ uniformemente em $[0,1]^d$, e estimamos

$$I_d \approx \frac{1}{N} \sum_{j=1}^{N} f(\mathbf{x}^{(j)})$$

Note que o volume do domínio é 1, então não multiplicamos por nenhum fator adicional.

Comentário sobre a convergência Para alta dimensão, métodos determinísticos como quadratura se tornam impraticáveis devido ao número exponencialmente crescente de pontos necessários. Já o erro do método Monte Carlo depende apenas de N, independentemente da dimensão d, tornando-o mais eficiente para altas dimensões.

Código em C: Integração Monte Carlo em dimensão d

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

#define N 1000000   // número de pontos Monte Carlo
#define D 10        // dimensão do integral

int main() {
    double soma = 0.0;
    double x;
    srand(time(NULL));

    for (int i = 0; i < N; i++) {
        double prod = 0.0;
        // gera vetor d-dimensional e calcula soma dos quadrados
        for (int j = 0; j < D; j++) {
            x = (double) rand() / RAND_MAX;  // amostra uniforme em [0,1]
            prod += x*x;
        }
        soma += exp(-prod);
    }

    double integral = soma / N;

    printf("Estimativa da integral em %d dimensoes = %.6f\n", D, integral);
    return 0;
}
```

A estimativa da integral em 10 dimensões foi de 0.053945. Note que aumentando N a estimativa fica mais precisa, e o método permanece computacionalmente viável mesmo para $d \gg 10$. Este exemplo simples ilustra como o método Monte Carlo pode ser aplicado em integrais multidimensionais complexas onde métodos tradicionais se tornam inviáveis.

18 Transformada Discreta de Fourier (DFT)

A Transformada Discreta de Fourier (DFT) é uma ferramenta fundamental para analisar a composição em frequências de um sinal discreto, real ou complexo. Para um conjunto de N pontos reais x_n, a DFT transforma os dados do domínio do tempo para o domínio da frequência:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{4}$$

onde X_k são os coeficientes complexos que indicam a amplitude e fase das componentes de frequência. No caso de dados reais, a DFT apresenta simetria conjugada, o que pode ser explorado para otimização, mas aqui consideraremos a forma direta para maior clareza.

Código em C: Cálculo direto da DFT para dados reais

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

#define N 256
#define PI 3.14159265358979323846

int main() {
    double x[N];
    double complex X[N];
    int n, k;

    // Sinal de entrada: seno simples
    for (n = 0; n < N; n++) {
        x[n] = sin(2.0 * PI * 5 * n / N);
    }

    // Calcula DFT usando complexos
    for (k = 0; k < N; k++) {
        X[k] = 0.0 + 0.0*I;
        for (n = 0; n < N; n++) {
            double angle = 2.0 * PI * k * n / N;
            X[k] += x[n] * cexp(-I * angle);
        }
    }

    // Salva módulo e índice
    FILE *f = fopen("dft_complex.dat", "w");
    for (k = 0; k < N; k++) {
        double magnitude = cabs(X[k]);
        fprintf(f, "%d %f\n", k, magnitude);
    }
    fclose(f);

    return 0;
}
```

19 Solução Numérica de Sistemas Lineares

Sistemas de equações lineares da forma

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

são fundamentais em diversas áreas da física computacional. Aqui, A é uma matriz quadrada $N \times N$, $\mathbf{x}$ é o vetor incógnita e $\mathbf{b}$ é o vetor de termos constantes. Para resolver numericamente esse sistema, um método direto simples e eficiente é a **eliminação de Gauss**. A eliminação de Gauss consiste em transformar a matriz A em uma matriz triangular superior por operações elementares, seguida de uma substituição regressiva para obter $\mathbf{x}$. Embora existam métodos mais avançados, essa abordagem é didática e eficaz para matrizes de tamanho moderado.

Algoritmo da Eliminação de Gauss

Dado um sistema linear $A\mathbf{x} = \mathbf{b}$, com $A \in \mathbb{R}^{N\times N}$, queremos resolver esse sistema transformando-o em um sistema equivalente triangular superior. A seguir, descrevemos o algoritmo básico (sem pivotamento):

1. **Para** $k = 0$ até $N - 2$ (varre as colunas):

   (a) Para cada linha $i = k + 1$ até $N - 1$:

      i. Calcule o multiplicador: $m_{ik} = \frac{A_{ik}}{A_{kk}}$

      ii. Atualize a linha i da matriz A: $A_{ij} \leftarrow A_{ij} - m_{ik}A_{kj}, \quad$ para $j = k$ até $N - 1$

      iii. Atualize o vetor $\mathbf{b}$: $b_i \leftarrow b_i - m_{ik}b_k$

2. Após a matriz ter sido transformada em triangular superior, faça **substituição regressiva** para obter $x_N, x_{N-1}, \ldots, x_1$:

   (a) Para $i = N - 1$ até 0 (de trás para frente):

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

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

```
#include <stdio.h>

int main() {
    int N = 3;
    double A[3][3] = {
        {2, -1, 1},
        {3, 3, 9},
        {3, 3, 5}
    };
    double b[3] = {8, 0, -6};
    double x[3];
    int i, j, k;

    // Eliminação de Gauss
    for (k = 0; k < N-1; k++) {
        for (i = k+1; i < N; i++) {
            double fator = A[i][k] / A[k][k];
            for (j = k; j < N; j++) {
                A[i][j] -= fator * A[k][j];
            }
            b[i] -= fator * b[k];
        }
    }

    // Substituição regressiva
    for (i = N-1; i >= 0; i--) {
        double soma = 0.0;
        for (j = i+1; j < N; j++) {
            soma += A[i][j] * x[j];
        }
        x[i] = (b[i] - soma) / A[i][i];
    }

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

    return 0;
}
```

19.1 Solução Numérica de Sistemas Lineares Tridiagonais

Sistemas lineares tridiagonais possuem matrizes A com elementos diferentes de zero apenas na diagonal principal e nas diagonais imediatamente acima e abaixo dela:

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

Um exemplo de sistema tridiagonal é:

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

Esse sistema pode ser resolvido por métodos genéricos como a eliminação de Gauss, ou de forma mais eficiente pelo **método de Thomas**, que aproveita a estrutura tridiagonal para reduzir o custo computacional para $O(N)$.

Código em C: Eliminação de Gauss para sistema 4×4

```
#include <stdio.h>

int main() {
    int N = 4;
    double A[4][4] = {
        {4, 1, 0, 0},
        {1, 4, 1, 0},
        {0, 1, 4, 1},
        {0, 0, 1, 3}
    };
    double b[4] = {15, 15, 15, 10};
    double x[4];
    int i, j, k;

    // Eliminação de Gauss
    for (k = 0; k < N-1; k++) {
        for (i = k+1; i < N; i++) {
            double fator = A[i][k] / A[k][k];
            for (j = k; j < N; j++) {
                A[i][j] -= fator * A[k][j];
            }
            b[i] -= fator * b[k];
        }
    }

    // Substituição regressiva
    for (i = N-1; i >= 0; i--) {
        double soma = 0.0;
        for (j = i+1; j < N; j++) {
            soma += A[i][j] * x[j];
        }
        x[i] = (b[i] - soma) / A[i][i];
    }

    // Imprime solução
    printf("Solução (Eliminação de Gauss):\n");
    for (i = 0; i < N; i++) {
        printf("x[%d] = %f\n", i, x[i]);
    }

    return 0;
}
```

Código em C: Método de Thomas para sistema tridiagonal

O método de Thomas é uma versão simplificada da eliminação de Gauss, desenvolvida especificamente para sistemas lineares com matrizes tridiagonais da forma:

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad \text{para } i = 1, 2, \ldots, N$$

com $a_1 = 0$ e $c_N = 0$. O algoritmo possui duas fases:

- Eliminação direta (forward sweep): modifica os coeficientes b_i e d_i para eliminar os termos a_i, transformando o sistema em triangular superior.
- Substituição regressiva (back substitution): resolve o sistema triangular resultante, começando de x_N até x_1.

Esse método tem complexidade linear $\mathcal{O}(N)$, sendo extremamente eficiente e estável para matrizes tridiagonais diagonais dominantes.

```
#include <stdio.h>

int main() {
    int N = 4;
    double a[4] = {0, 1, 1, 1};   // subdiagonal (a_1 não usada)
    double b[4] = {4, 4, 4, 3};   // diagonal principal
    double c[4] = {1, 1, 1, 0};   // superdiagonal (c_4 não usada)
    double d[4] = {15, 15, 15, 10};
    double c_prime[4], d_prime[4];
    double x[4];
    int i;

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

    for (i = 1; i < N; i++) {
        double m = b[i] - a[i] * c_prime[i-1];
        c_prime[i] = (i < N-1) ? c[i] / m : 0;
        d_prime[i] = (d[i] - a[i] * d_prime[i-1]) / m;
    }

    // Back substitution
    x[N-1] = d_prime[N-1];
    for (i = N-2; i >= 0; i--) {
        x[i] = d_prime[i] - c_prime[i] * x[i+1];
    }

    // Imprime solução
    printf("Solução (Método de Thomas):\n");
    for (i = 0; i < N; i++) {
        printf("x[%d] = %f\n", i, x[i]);
    }

    return 0;
}
```

20 Método da Bisseção

O método da bisseção é uma técnica numérica simples e confiável para encontrar uma raiz real de uma equação do tipo $f(x) = 0$, desde que a função f seja contínua e mude de sinal em um intervalo $[a, b]$, ou seja:

$$f(a) \cdot f(b) < 0 \tag{5}$$

Baseado no Teorema do Valor Intermediário, o método consiste em dividir o intervalo ao meio repetidamente até que a raiz seja localizada com a precisão desejada. A cada passo, o ponto médio $c = \frac{a+b}{2}$ é calculado e substitui a ou b, dependendo do sinal de $f(c)$. O processo é repetido até que o tamanho do intervalo seja menor que uma tolerância ϵ.

Equações do Método

$$c = \frac{a+b}{2}$$
$$\text{Se } f(c) = 0 \Rightarrow \text{Raiz encontrada}$$
$$\text{Se } f(a) \cdot f(c) < 0 \Rightarrow b = c$$
$$\text{Se } f(c) \cdot f(b) < 0 \Rightarrow a = c$$

A convergência é garantida para funções contínuas, mas o método é relativamente lento (convergência linear).

Aplicação: Raiz de um Polinômio de 4º grau

Considere o polinômio:

$$f(x) = x^4 - 3x^3 - 7x^2 + 27x - 18 \tag{6}$$

Para aplicar o método da bisseção, devemos primeiro encontrar um intervalo $[a, b]$ tal que $f(a) \cdot f(b) < 0$. Após testes, nota-se que o intervalo $[0, 1]$ satisfaz essa condição.

Código em C: Bisseção para Polinômio de 4º grau

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

#define EPSILON 1e-6
#define MAX_IT 100

// Função polinomial f(x) = x^4 - 3x^3 - 7x^2 + 27x - 18
double f(double x) {
    return x*x*x*x - 3*x*x*x - 7*x*x + 27*x - 18;
}

int main() {
    double a = 0.0, b = 1.0, c;
    int iter = 0;

    if (f(a) * f(b) >= 0) {
        printf("Escolha de intervalo inválida: f(a) * f(b) >= 0\n");
        return 1;
    }

    FILE *fp = fopen("bissecao.dat", "w");

    while ((b - a) > EPSILON && iter < MAX_IT) {
        c = (a + b) / 2.0;
        fprintf(fp, "%d %.10f %.10f\n", iter, c, f(c));

        if (fabs(f(c)) < EPSILON)
            break;

        if (f(a) * f(c) < 0)
            b = c;
        else
            a = c;

        iter++;
    }

    fclose(fp);

    printf("Raiz aproximada: %.10f\n", c);
    printf("f(c) = %.10e\n", f(c));
    return 0;
}
```

O programa armazena os valores de cada iteração em um arquivo `bissecao.dat`, permitindo análise posterior. O resultado final é uma aproximação da raiz no intervalo dado, com precisão controlada por `EPSILON`.

21 Autovalor e Autovetor : Combinação dos Métodos de Bisseção e Thomas

Nesta seção, aplicaremos dois métodos numéricos em sequência para determinar um autovalor e o autovetor correspondente de uma matriz tridiagonal simétrica H, que, em contextos de mecânica quântica, está associada ao Modelo de Anderson unidimensional. A matriz considerada é:

$$H = \begin{bmatrix} 0.5 & -1 & 0 & 0 \\ -1 & -1.2 & -1 & 0 \\ 0 & -1 & 0.7 & -1 \\ 0 & 0 & -1 & -0.3 \end{bmatrix}$$

O problema de autovalores consiste em resolver:

$$H\vec{v} = \lambda\vec{v} \quad \Rightarrow \quad (H - \lambda I)\vec{v} = 0$$

A estratégia usada será:

1. Usar o **método da bisseção** para encontrar um valor de λ tal que $\det(H - \lambda I) = 0$, ou seja, um autovalor.
2. Usar o **método de Thomas** para resolver o sistema linear $(H - \lambda I)\vec{v} = 0$, encontrando o autovetor correspondente.

Como o sistema é homogêneo, fixamos arbitrariamente $v_0 = 1$ para evitar a solução trivial.

Código em C: Bisseção + Thomas

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

#define N 4
#define EPSILON 1e-6
#define A -1.0  // sub/super diagonal constante

// Função para calcular o determinante tridiagonal de (H - lambda I)
double det(double lambda) {
    double d[N] = {0.5, -1.2, 0.7, -0.3};
    double b[N];

    // Construção da diagonal modificada
    for (int i = 0; i < N; i++)
        b[i] = d[i] - lambda;

    // Eliminação para determinante
    for (int i = 1; i < N; i++) {
        double m = A / b[i-1];
        b[i] = b[i] - m * A;
    }

    return b[N-1];
}

int main() {
```

```
    double a = 1.5, b = 2.0, c, f_c;
    int max_it = 100, iter = 0;

    // Bisseção para encontrar autovalor
    while ((b - a) > EPSILON && iter < max_it) {
        c = 0.5 * (a + b);
        f_c = det(c);

        if (det(a) * f_c < 0)
            b = c;
        else
            a = c;

        iter++;
    }

    double lambda = 0.5 * (a + b);
    printf("Autovalor aproximado: %.6f\n", lambda);

    // Construção do sistema (H - lambda I)
    double d[N] = {0.5 - lambda, -1.2 - lambda, 0.7 - lambda, -0.3 - lambda};
    double bp[N-1], dp[N-1], rhs[N] = {0};

    // Fixamos v[0] = 1 => ajustamos rhs[1] = -A * v[0]
    rhs[1] = -A * 1.0;

    // Thomas para resolver sistema 3x3 para v[1], v[2], v[3]
    bp[0] = d[1];
    dp[0] = rhs[1];

    for (int i = 1; i < N-1; i++) {
        double m = A / bp[i-1];
        bp[i] = d[i+1] - m * A;
        dp[i] = -m * dp[i-1];
    }

    // Substituição regressiva
    double v[N];
    v[0] = 1.0;
    v[3] = dp[2] / bp[2];
    v[2] = (dp[1] - A * v[3]) / bp[1];
    v[1] = (dp[0] - A * v[2]) / bp[0];

    // Normalização do autovetor
double norm = 0.0;
for (int i = 0; i < N; i++)
    norm += v[i] * v[i];
norm = sqrt(norm);

printf("Autovetor normalizado:\n");
for (int i = 0; i < N; i++)
    printf("v[%d] = %f\n", i, v[i] / norm);



    return 0;
}
```

Esse programa retorna um autovalor naproximado $\lambda \approx 1.7$ encontrado via bisseção e o autovetor associado. O autovetor encontrado via método de Thomas foi normalizado. Ao final do programa, realizamos uma verificação numérica da relação fundamental $H\vec{v} = \lambda\vec{v}$, comparando o produto da matriz H pelo autovetor normalizado com o resultado da multiplicação escalar $\lambda\vec{v}$. Pequenas discrepâncias são esperadas devido aos erros numéricos acumulados no processo de bisseção e na resolução do sistema via método de Thomas.

22 Autovetor e Autovalor : Método da Potência

Vamos apresentar agora o chamado "método da potência"que é um formalismo interessante para estimar o maior autovalor (em módulo) e seu autovetor de uma matriz Hermitiana/-

Simétrica. Dada uma matriz $H \in \mathbb{R}^{N\times N}$ simétrica (ou Hermitiana), seja $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_N|$ a ordenação por módulo dos autovalores. Para um vetor inicial $\mathbf{v}^{(0)}$ com projecção não nula no autovetor dominante $\mathbf{u}_1$, a iteração

$$\mathbf{w}^{(n+1)} = H\,\mathbf{v}^{(n)}, \quad \mathbf{v}^{(n+1)} = \frac{\mathbf{w}^{(n+1)}}{\|\mathbf{w}^{(n+1)}\|}$$

converge para $\mathbf{u}_1$. O autovalor associado é estimado pelo quociente de Rayleigh

$$\lambda^{(n)} = \frac{(\mathbf{v}^{(n)})^T H\,\mathbf{v}^{(n)}}{(\mathbf{v}^{(n)})^T\mathbf{v}^{(n)}} = (\mathbf{v}^{(n)})^T H\,\mathbf{v}^{(n)} \quad (\text{com } \|\mathbf{v}^{(n)}\| = 1).$$

Critério de parada. Pare quando $\dfrac{|\lambda^{(n)} - \lambda^{(n-1)}|}{|\lambda^{(n)}|} < \tau$ ou quando $\|\mathbf{v}^{(n)} - \mathbf{v}^{(n-1)}\| < \tau$.

Algoritmo (passo a passo)

1. Escolha $\mathbf{v}^{(0)} \neq 0$ e normalize.
2. Para $n = 0, 1, 2, \ldots$:
   (a) Compute $\mathbf{w}^{(n+1)} = H\,\mathbf{v}^{(n)}$.
   (b) Normalize: $\mathbf{v}^{(n+1)} = \mathbf{w}^{(n+1)}/\|\mathbf{w}^{(n+1)}\|$.
   (c) Estime $\lambda^{(n+1)} = (\mathbf{v}^{(n+1)})^T H\,\mathbf{v}^{(n+1)}$.
   (d) Teste o critério de parada.

Exemplo de uma matriz simétrica 3×3

Considere

$$H_3 = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

Seus autovalores são $\{1, 2, 4\}$, logo o maior é $\lambda_{\max} = 4$. Vamos aplicar o método com $\mathbf{v}^{(0)} = \frac{1}{\sqrt{3}}(1,1,1)^T$.

Iteração 1.

$$\mathbf{w}^{(1)} = H_3\,\mathbf{v}^{(0)} = \begin{pmatrix} 1.73205081 \\ 2.88675135 \\ 1.73205081 \end{pmatrix}, \quad \|\mathbf{w}^{(1)}\| \approx 3.78593890.$$

$$\mathbf{v}^{(1)} = \frac{\mathbf{w}^{(1)}}{\|\mathbf{w}^{(1)}\|} = \begin{pmatrix} 0.45749571 \\ 0.76249285 \\ 0.45749571 \end{pmatrix}, \quad \lambda^{(1)} = (\mathbf{v}^{(1)})^T H_3\,\mathbf{v}^{(1)} \approx 3.97674419.$$

Iteração 2.

$$\mathbf{w}^{(2)} = H_3\,\mathbf{v}^{(1)} = \begin{pmatrix} 1.67748427 \\ 3.20246998 \\ 1.67748427 \end{pmatrix}, \quad \|\mathbf{w}^{(2)}\| \approx 3.98543861.$$

$$\mathbf{v}^{(2)} = \frac{\mathbf{w}^{(2)}}{\|\mathbf{w}^{(2)}\|} = \begin{pmatrix} 0.42090330 \\ 0.80354267 \\ 0.42090330 \end{pmatrix}, \quad \lambda^{(2)} \approx 3.99853587.$$

Após poucas iterações $\lambda^{(n)} \to 4$ e $\mathbf{v}^{(n)}$ se aproxima do autovetor dominante (proporcional a $(1,\,2,\,1)^T$ normalizado).

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

#define N 3              // tamanho da matriz
#define MAX_IT 1000      // máximo de iterações
#define TOL 1e-10        // tolerância para convergência

// Multiplica matriz A (NxN) por vetor v (N) -> resultado w (N)
void matvec(double A[N][N], double v[N], double w[N]) {
    for (int i=0; i<N; i++) {
        w[i] = 0.0;
        for (int j=0; j<N; j++) {
            w[i] += A[i][j]*v[j];
        }
    }
}

// Norma Euclidiana de vetor
double norm(double v[N]) {
    double s = 0.0;
    for (int i=0; i<N; i++) s += v[i]*v[i];
    return sqrt(s);
}

// Normaliza vetor v
void normalize(double v[N]) {
    double n = norm(v);
    for (int i=0; i<N; i++) v[i] /= n;
}

// Produto interno
double dot(double u[N], double v[N]) {
    double s = 0.0;
    for (int i=0; i<N; i++) s += u[i]*v[i];
    return s;
}

int main() {
    // Matriz simétrica 3x3 do exemplo
    double H[N][N] = {
        {2, 1, 0},
        {1, 3, 1},
        {0, 1, 2}
    };

    // Vetor inicial normalizado
    double v[N] = {1.0, 1.0, 1.0};
    normalize(v);

    double w[N], lambda_old = 0.0, lambda = 0.0;

    for (int it = 0; it < MAX_IT; it++) {
        // w = H v
        matvec(H, v, w);
        // Rayleigh quotient: lambda \approx  v^T H v
        lambda = dot(v, w);
        // Normaliza para próxima iteração
        normalize(w);
        for (int i=0; i<N; i++) v[i] = w[i];

        if (fabs(lambda - lambda_old) < TOL) break;
        lambda_old = lambda;
    }

    printf("Autovalor dominante \approx  %.10f\n", lambda);
    printf("Autovetor correspondente \approx  (");
    for (int i=0; i<N; i++) printf(" %.10f", v[i]);
    printf(" )\n");

    // Teste Hv \approx  \lambda v
    double Hv[N];
    matvec(H, v, Hv);
    printf("\nTeste Hv e \lambda v:\n");
    for (int i=0; i<N; i++) {
        printf("Hv[%d]=%.10f   \lambda v[%d]=%.10f\n", i, Hv[i], i, lambda*v[i]);
    }
```

```
    return 0;
}
```

Esse código anterior implementa o método da potência para encontrar o maior autovalor e o autovetor correspondente de uma matriz simétrica 3×3. Primeiro, ele define funções auxiliares para multiplicar matriz por vetor, calcular a norma euclidiana, normalizar vetores e computar o produto interno. A matriz H usada é a do exemplo dado, e o vetor inicial é escolhido como $(1, 1, 1)$, que é normalizado antes do início das iterações. Em cada passo, o programa calcula o novo vetor $w = Hv$, atualiza a estimativa do autovalor pelo quociente de Rayleigh $\lambda \approx v^T Hv$, e normaliza o resultado para continuar o processo. O loop continua até que a diferença entre valores consecutivos de λ seja menor que a tolerância pré-definida. Ao final, o código imprime o autovalor dominante encontrado e o autovetor correspondente. Além disso, ele realiza um teste direto, mostrando os componentes de Hv e λv, para verificar numericamente que o vetor obtido é realmente um autovetor associado ao autovalor calculado. Dessa forma, o programa confirma a convergência e a consistência do método.

Exemplo de uma matriz simétrica 4×4

Considere

$$H_4 = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}.$$

Os autovalores são $\{1,\ 1.5858\ldots,\ 3,\ 4.4142\ldots\}$, portanto $\lambda_{\max} \approx 4.41421356$. Com $\mathbf{v}^{(0)} = \frac{1}{2}(1, 1, 1, 1)^T$:

Iteração 1.

$$\mathbf{w}^{(1)} = H_4\,\mathbf{v}^{(0)} = \begin{pmatrix} 1.5 \\ 2.5 \\ 2.5 \\ 1.5 \end{pmatrix}, \quad \|\mathbf{w}^{(1)}\| \approx 4.12310563.$$

$$\mathbf{v}^{(1)} = \begin{pmatrix} 0.36380344 \\ 0.60633906 \\ 0.60633906 \\ 0.36380344 \end{pmatrix}, \quad \lambda^{(1)} \approx 4.35294118.$$

Iteração 2.

$$\mathbf{w}^{(2)} = H_4\,\mathbf{v}^{(1)} = \begin{pmatrix} 1.33394594 \\ 2.78915969 \\ 2.78915969 \\ 1.33394594 \end{pmatrix}, \quad \|\mathbf{w}^{(2)}\| \approx 4.37237316.$$

$$\mathbf{v}^{(2)} = \begin{pmatrix} 0.30508511 \\ 0.63790523 \\ 0.63790523 \\ 0.30508511 \end{pmatrix}, \quad \lambda^{(2)} \approx 4.40615385.$$

Novamente observa-se a rápida convergência para $\lambda_{\max} \approx 4.4142$.

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

#define N 4
#define MAX_IT 1000
#define TOL 1e-12

// Multiplica matriz A (NxN) por vetor v (N) -> w
void matvec(double A[N][N], double v[N], double w[N]) {
    for (int i=0; i<N; i++) {
        w[i] = 0.0;
        for (int j=0; j<N; j++) {
            w[i] += A[i][j]*v[j];
        }
    }
}

// Norma Euclidiana
double norm(double v[N]) {
    double s = 0.0;
    for (int i=0; i<N; i++) s += v[i]*v[i];
    return sqrt(s);
}

// Normaliza vetor
void normalize(double v[N]) {
    double n = norm(v);
    for (int i=0; i<N; i++) v[i] /= n;
}

// Produto interno
double dot(double u[N], double v[N]) {
    double s = 0.0;
    for (int i=0; i<N; i++) s += u[i]*v[i];
    return s;
}

int main() {
    // Matriz simétrica 4x4 do exemplo
    double H[N][N] = {
        {2,1,0,0},
        {1,3,1,0},
        {0,1,3,1},
        {0,0,1,2}
    };

    // Vetor inicial normalizado: (1,1,1,1)^T / 2
    double v[N] = {0.5, 0.5, 0.5, 0.5};
    normalize(v);

    double w[N], lambda=0.0, lambda_old=0.0;

    for (int it=0; it<MAX_IT; it++) {
        // w = H v
        matvec(H,v,w);

        // Aproximação do autovalor
        lambda = dot(v,w);

        // Normaliza
        normalize(w);
        for (int i=0; i<N; i++) v[i] = w[i];

        if (fabs(lambda - lambda_old) < TOL) break;
        lambda_old = lambda;
    }

    printf("Autovalor dominante \approx  %.10f\n", lambda);
    printf("Autovetor correspondente \approx  (");
    for (int i=0; i<N; i++) printf(" %.10f", v[i]);
    printf(" )\n");

    // Teste Hv \approx  \lambda v
    double Hv[N];
    matvec(H, v, Hv);
    printf("\nTeste Hv e \lambda v:\n");
```

```
    for (int i=0; i<N; i++) {
        printf("Hv[%d] = %.10f   \lambda v[%d] = %.10f\n",
               i, Hv[i], i, lambda*v[i]);
    }

    return 0;
}
```

Esse código aplica o método da potência a uma matriz simétrica 4×4 com estrutura tridiagonal simétrica. Ele começa implementando funções auxiliares: multiplicação matriz–vetor, cálculo da norma euclidiana, normalização de vetores e produto interno. A matriz escolhida é simples, representando um sistema de acoplamentos locais entre vizinhos próximos, e o vetor inicial é $(1,1,1,1)^T$, normalizado antes do início do processo iterativo. Em cada iteração, o código calcula $w = Hv$, estima o autovalor dominante pelo quociente de Rayleigh $\lambda = v^T Hv$, e normaliza o vetor resultante para gerar a próxima aproximação do autovetor. A convergência é controlada pela diferença entre valores consecutivos de λ, interrompendo o processo quando a tolerância é satisfeita. Ao final, o programa imprime o autovalor dominante aproximado e o autovetor correspondente, mostrando explicitamente sua convergência. Como verificação, ele compara os resultados de Hv com λv, demonstrando numericamente a consistência da solução.

22.1 Método da potência inversa com deslocamento para matriz simétrica 4×4

O **método da potência inversa com deslocamento** é utilizado para encontrar o autovalor de uma matriz que esteja próximo de um valor inicial μ escolhido como *chute*. A ideia é construir a matriz deslocada

$$A_{\text{shifted}} = H - \mu I,$$

onde I é a matriz identidade, e então aplicar a potência inversa a A_{shifted}. O maior autovalor da matriz inversa $(H-\mu I)^{-1}$ corresponde ao autovalor de H mais próximo de μ.
No código abaixo, a resolução do sistema linear $(H-\mu I)w = v$ é feita usando o método de eliminação de Gauss simples, que consiste em:

1. Montar a matriz aumentada $[A|b]$ do sistema $Ax = b$.
2. Transformar a matriz aumentada em uma forma triangular superior (eliminação para zerar elementos abaixo do pivô) percorrendo as linhas e usando operações lineares.
3. Realizar substituição regressiva, a partir da última linha, para obter as incógnitas x_i.

O procedimento passo a passo pode ser ilustrado considerando a matriz 4×4 do exemplo:

$$H - \mu I = \begin{pmatrix} 4-\mu & 1 & 2 & 0 \\ 1 & 3-\mu & 0 & 1 \\ 2 & 0 & 5-\mu & 1 \\ 0 & 1 & 1 & 2-\mu \end{pmatrix}, \quad v = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

1. **Montagem da matriz aumentada** Primeiro, construímos a matriz aumentada $[A|v]$, adicionando o vetor v como coluna extra:

$$[A|v] = \begin{pmatrix} 4-\mu & 1 & 2 & 0 & 1 \\ 1 & 3-\mu & 0 & 1 & 1 \\ 2 & 0 & 5-\mu & 1 & 1 \\ 0 & 1 & 1 & 2-\mu & 1 \end{pmatrix}.$$

2. **Eliminação para obter triangular superior** Percorremos cada linha para zerar os elementos abaixo da diagonal:

   - Escolhemos o pivô da primeira coluna, $a_{11} = 4 - \mu$.
   - Para as linhas 2, 3 e 4, subtraímos múltiplos da primeira linha de modo a zerar a_{21}, a_{31}, a_{41}.
   - Repetimos o mesmo procedimento para os pivôs seguintes a_{22}, a_{33} até obter uma matriz triangular superior, com zeros abaixo da diagonal.

3. **Substituição regressiva** Com a matriz triangular superior, resolvemos as incógnitas de baixo para cima:

$$x_4 = \frac{b'_4}{a'_{44}}, \quad x_3 = \frac{b'_3 - a'_{34}x_4}{a'_{33}}, \quad x_2 = \frac{b'_2 - a'_{23}x_3 - a'_{24}x_4}{a'_{22}}, \quad x_1 = \frac{b'_1 - a'_{12}x_2 - a'_{13}x_3 - a'_{14}x_4}{a'_{11}}.$$

   Aqui, b'_i e a'_{ij} são os elementos da matriz triangular superior obtida após a eliminação.

Ao final, o vetor w obtido é usado na iteração do método da potência inversa, e normalizado antes de calcular o autovalor aproximado via quociente de Rayleigh:

$$\lambda \approx \frac{w^T H w}{w^T w}.$$

Este procedimento permite encontrar o autovalor de H mais próximo do chute μ, mesmo para matrizes cheias e não triangulares.
Segue o código completo em C:

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

#define N 4
#define MAX_IT 1000
#define TOL 1e-12

// Resolve Ax = b pelo método de Gauss (simples, sem pivoteamento)
void gauss(double A[N][N], double b[N], double x[N]) {
    double M[N][N+1];
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++) M[i][j]=A[i][j];
        M[i][N]=b[i];
    }
    for(int k=0;k<N;k++){
        double pivot=M[k][k];
        for(int j=k;j<=N;j++) M[k][j]/=pivot;
        for(int i=k+1;i<N;i++){
            double f=M[i][k];
            for(int j=k;j<=N;j++) M[i][j]-=f*M[k][j];
        }
    }
    for(int i=N-1;i>=0;i--){
        x[i]=M[i][N];
        for(int j=i+1;j<N;j++) x[i]-=M[i][j]*x[j];
    }
}

// Norma Euclidiana
double norm(double v[N]){
    double s=0.0;
    for(int i=0;i<N;i++) s+=v[i]*v[i];
    return sqrt(s);
}

// Normaliza vetor
void normalize(double v[N]){
    double n=norm(v);
```

```
    for(int i=0;i<N;i++) v[i]/=n;
}

// Produto interno
double dot(double u[N], double v[N]){
    double s=0.0;
    for(int i=0;i<N;i++) s+=u[i]*v[i];
    return s;
}

int main(){
    // Matriz simétrica cheia 4x4
    double H[N][N] = {
        {4,1,2,0},
        {1,3,0,1},
        {2,0,5,1},
        {0,1,1,2}
    };

    double mu = 6.5; // chute inicial próximo do autovalor desejado
    double v[N] = {1,1,1,1};
    normalize(v);

    double w[N], lambda_old=0.0, lambda=0.0;

    double I[N][N];
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++) I[i][j] = (i==j)?1.0:0.0;
    }

    double A_shifted[N][N]; // H - mu*I
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            A_shifted[i][j] = H[i][j] - mu*I[i][j];
        }
    }

    for(int it=0;it<MAX_IT;it++){
        // Resolve (H - mu I) w = v
        gauss(A_shifted, v, w);
        normalize(w);

        // Aproximação do autovalor
        double Hv[N];
        for(int i=0;i<N;i++){
            Hv[i]=0.0;
            for(int j=0;j<N;j++) Hv[i]+=H[i][j]*w[j];
        }
        lambda = dot(w,Hv)/dot(w,w);

        for(int i=0;i<N;i++) v[i]=w[i];

        if(fabs(lambda - lambda_old)<TOL) break;
        lambda_old=lambda;
    }

    printf("Autovalor próximo de %.3f \approx  %.10f\n", mu, lambda);
    printf("Autovetor correspondente \approx  (");
    for(int i=0;i<N;i++) printf(" %.10f", v[i]);
    printf(" )\n");

    // Teste Hv \approx \lambda v
    double Hv[N];
    for(int i=0;i<N;i++){
        Hv[i]=0.0;
        for(int j=0;j<N;j++) Hv[i]+=H[i][j]*v[j];
    }
    printf("\nTeste Hv e \lambda v:\n");
    for(int i=0;i<N;i++){
        printf("Hv[%d] = %.10f   \lambda v[%d] = %.10f\n", i, Hv[i], i, lambda*v[i]);
    }

    return 0;
}
```

O código inicializa a matriz H e um vetor inicial v normalizado. O chute inicial μ é usado para

construir a matriz deslocada $H - \mu I$. Em cada iteração, o sistema linear é resolvido usando o método de Gauss simples, obtendo o vetor w que é normalizado. O autovalor aproximado é calculado pelo quociente de Rayleigh $\lambda = \frac{w^T H w}{w^T w}$. A cada iteração, v é atualizado com w até que a diferença de λ entre iterações consecutivas seja menor que a tolerância. No final, o código imprime o autovalor aproximado, o autovetor e verifica a relação $Hv \approx \lambda v$, garantindo que o resultado está correto. Este procedimento permite obter o autovalor de H mais próximo do chute μ, mesmo para matrizes cheias não triangulares.

23 Autovalor e Autovetor : Método da Potência Inversa com Deslocamento aplicado no Modelo de Anderson 1d

O método da potência inversa com deslocamento é uma técnica iterativa muito utilizada para encontrar autovalores de uma matriz próximos a um valor estimado μ. Essa abordagem é especialmente útil quando queremos autovalores internos, isto é, não necessariamente os de maior módulo, e pode ser aplicada eficientemente em matrizes tridiagonais, como as que aparecem no Modelo de Anderson 1D (caso que foi abordado no exemplo anterior).

Princípio do Método

Dado um operador linear $H \in \mathbb{R}^{N\times N}$, um deslocamento μ, e um vetor inicial não nulo $\vec{x}_0$, a ideia é iterar a seguinte etapa:

$$(H - \mu I)\vec{y}_{k+1} = \vec{x}_k$$

ou seja, a cada passo resolvemos um sistema linear e definimos

$$\vec{x}_{k+1} = \frac{\vec{y}_{k+1}}{\|\vec{y}_{k+1}\|}$$

O vetor $\vec{x}_k$ converge para o autovetor associado ao autovalor mais próximo de μ. O autovalor estimado na iteração k pode ser calculado pelo quociente de Rayleigh:

$$\lambda_k = \vec{x}_k^{\top} H \vec{x}_k$$

A convergência ocorre quando $|\lambda_k - \lambda_{k-1}|$ fica abaixo de um critério de tolerância.

Aplicação à matriz do Modelo de Anderson

Considere a matriz tridiagonal simétrica:

$$H = \begin{bmatrix} 0.5 & -1 & 0 & 0 \\ -1 & -1.2 & -1 & 0 \\ 0 & -1 & 0.7 & -1 \\ 0 & 0 & -1 & -0.3 \end{bmatrix}$$

Queremos encontrar um autovalor próximo de $\mu = 1.7$ e seu autovetor associado.

Como $H - \mu I$ é uma matriz tridiagonal, o sistema linear $(H - \mu I)\vec{y} = \vec{x}$ pode ser resolvido eficientemente pelo método de Thomas.

O código abaixo implementa o método da potência inversa com deslocamento, utilizando o método de Thomas para a resolução dos sistemas tridiagonais a cada passo:

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

#define N 4
#define MAX_IT 1000
#define TOL 1e-8

double H[N][N] = {
    {0.5, -1.0,  0.0,  0.0},
    {-1.0, -1.2, -1.0, 0.0},
    {0.0, -1.0,  0.7, -1.0},
    {0.0,  0.0, -1.0, -0.3}
};

// Método de Thomas para resolver sistema tridiagonal
void thomas(int n, double *a, double *b, double *c, double *d, double *x) {
    double c_prime[n-1], d_prime[n];
    c_prime[0] = c[0]/b[0];
    d_prime[0] = d[0]/b[0];
    for (int i=1; i<n-1; i++) {
        double m = b[i] - a[i-1]*c_prime[i-1];
        c_prime[i] = c[i]/m;
        d_prime[i] = (d[i] - a[i-1]*d_prime[i-1])/m;
    }
    d_prime[n-1] = (d[n-1] - a[n-2]*d_prime[n-2])/(b[n-1] - a[n-2]*c_prime[n-2]);
    x[n-1] = d_prime[n-1];
    for (int i=n-2; i>=0; i--)
        x[i] = d_prime[i] - c_prime[i]*x[i+1];
}

// Produto matriz-vetor
void mat_vec(int n, double A[n][n], double *x, double *y) {
    for (int i=0; i<n; i++) {
        y[i] = 0.0;
        for (int j=0; j<n; j++)
            y[i] += A[i][j]*x[j];
    }
}

// Norma euclidiana
double norm(int n, double *x) {
    double s=0.0;
    for (int i=0; i<n; i++) s += x[i]*x[i];
    return sqrt(s);
}

// Produto interno
double dot(int n, double *x, double *y) {
    double s=0.0;
    for (int i=0; i<n; i++) s += x[i]*y[i];
    return s;
}

int main() {
    double mu = 1.7;
    double a[N-1], b[N], c[N-1];
    double x[N], y[N];
    double lambda = 0.0, lambda_old = 0.0;

    // Inicializa vetor x com valores não nulos
    for (int i=0; i<N; i++) x[i] = 1.0;

    // Monta tridiagonal A = H - mu I
    for (int i=0; i<N; i++) {
        b[i] = H[i][i] - mu;
        if (i>0) a[i-1] = H[i][i-1];
        if (i<N-1) c[i] = H[i][i+1];
    }

    for (int iter=0; iter<MAX_IT; iter++) {
        // Resolve A y = x
        thomas(N, a, b, c, x, y);

        // Normaliza y
        double y_norm = norm(N, y);
        for (int i=0; i<N; i++) y[i] /= y_norm;
```

```
        // Calcula autovalor aproximado lambda = x^T H x
        double Hx[N];
        mat_vec(N, H, y, Hx);
        lambda = dot(N, y, Hx);

        // Critério de parada
        if (fabs(lambda - lambda_old) < TOL)
            break;

        lambda_old = lambda;

        // Atualiza x para próxima iteração
        for (int i=0; i<N; i++) x[i] = y[i];
    }

    printf("Autovalor aproximado perto de %.2f: %.8f\n", mu, lambda);
    printf("Autovetor correspondente (normalizado):\n");
    for (int i=0; i<N; i++)
        printf("v[%d] = % .8f\n", i, y[i]);

    return 0;
}
```

Observações finais

- O método converge rapidamente para o autovalor e autovetor associados ao valor de deslocamento μ.
- O sistema tridiagonal é resolvido eficientemente pelo método de Thomas, reduzindo o custo computacional.
- A normalização do autovetor é feita a cada passo para evitar estouros numéricos.
- O resultado final pode ser verificado conferindo se $H\vec{v} \approx \lambda\vec{v}$.

24 Autovalor e Autovetor : método da potência inversa com deslocamento aplicado no Modelo de Anderson em 2D

O cálculo de autovalores e autovetores é uma tarefa central em diversas áreas da física, especialmente em sistemas quânticos com muitos graus de liberdade. Quando a matriz associada ao Hamiltoniano do sistema não é tridiagonal, como acontece em muitos modelos de rede com interações de vizinhança, métodos diretos como a diagonalização completa podem ser computacionalmente custosos. Como já debatemos, o método da potência inversa com deslocamento é uma técnica eficiente para encontrar autovalores próximos de um *chute inicial* μ, mesmo em matrizes não tridiagonais ou densas. A ideia principal é transformar o problema $Hv = \lambda v$ em $(H - \mu I)^{-1}v = \tilde{\lambda}v$, de modo que os autovalores próximos a μ se tornem dominantes na iteração da potência.

Como exemplo prático, utilizamos o **modelo de Anderson em 2D**, que descreve os estados eletrônicos em uma rede quadrada $L \times L$ de sítios; cada sítio, localizado na posição i, j, tem energia potencial aleatória ($\varepsilon_{i,j}$) e hopping constante t entre vizinhos próximos. A figura 1 ilustra essa rede de sítios: os círculos representam os sítios e as linhas conectando-os correspondem aos termos de *hopping*, que descrevem a probabilidade de um elétron se deslocar entre sítios vizinhos (equivalente ao termo cinético do Hamiltoniano). Este modelo gera uma matriz Hamiltoniana um pouco mais densa e simétrica, perfeita para demonstrar a aplicação do

método da potência inversa com deslocamento em matrizes não tridiagonais. O Hamiltoniano do Modelo de Anderson 2d do sistema é dado por:

$$H = \sum_{i,j} \varepsilon_{i,j} |i,j\rangle\langle i,j| + t \sum_{\langle ij,op\rangle} (|i,j\rangle\langle o,p| + |o,p\rangle\langle i,j|)$$

Como já mencionamos anteriormente, cada sítio possui uma energia $\varepsilon_{i,j}$, distribuída aleatoriamente, tipicamente no intervalo $[-W/2, W/2]$, representando a desordem local causada por impurezas ou imperfeições na rede cristalina. Alguns regimes especiais podem ser destacados:

- $W = 0$: cristal perfeito, sem desordem; todas as energias dos sítios são iguais, e os estados eletrônicos são estendidos.
- $W \gg t$: desordem forte; os elétrons tendem a se localizar nos sítios, caracterizando a *localização de Anderson*.
- $W \approx t$: desordem intermediária; há competição entre mobilidade dos elétrons (hopping t) e desordem local.

Para o propósito deste estudo, nos interessamos principalmente na forma matricial do Hamiltoniano. Para uma rede $L \times L$, podemos escrever o Hamiltoniano H como uma matriz $N \times N$, com $N = L^2$, em que:

- os elementos diagonais contêm as energias aleatórias dos sítios $\varepsilon_{i,j}$,
- os elementos fora da diagonal representam o hopping t entre sítios vizinhos imediatos.

A seguir, mostramos uma instância do Hamiltoniano de Anderson para $L = 3$:

$$H = \begin{pmatrix} \varepsilon_{1,1} & t & 0 & t & 0 & 0 & 0 & 0 & 0 \\ t & \varepsilon_{1,2} & t & 0 & t & 0 & 0 & 0 & 0 \\ 0 & t & \varepsilon_{1,3} & 0 & 0 & t & 0 & 0 & 0 \\ t & 0 & 0 & \varepsilon_{2,1} & t & 0 & t & 0 & 0 \\ 0 & t & 0 & t & \varepsilon_{2,2} & t & 0 & t & 0 \\ 0 & 0 & t & 0 & t & \varepsilon_{2,3} & 0 & 0 & t \\ 0 & 0 & 0 & t & 0 & 0 & \varepsilon_{3,1} & t & 0 \\ 0 & 0 & 0 & 0 & t & 0 & t & \varepsilon_{3,2} & t \\ 0 & 0 & 0 & 0 & 0 & t & 0 & t & \varepsilon_{3,3} \end{pmatrix}$$

Esta forma ilustra claramente como os termos de hopping conectam apenas vizinhos imediatos na rede e como a desordem local é representada na diagonal. Na prática, o modelo de Anderson serve como um exemplo didático para entender efeitos de desordem em sistemas quânticos discretos, fornecendo uma plataforma para estudar localização de estados e transições de fase metal-isolante em redes bidimensionais. A forma matricial apresentada torna possível implementar algoritmos numéricos que operam diretamente sobre matrizes densas, mesmo quando o tamanho do sistema cresce rapidamente com L.

A implementação em C que apresentaremos utiliza alocação dinâmica de memória e ponteiros para lidar com matrizes e vetores de tamanho N variável.

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

#define MAX_IT 1000
```

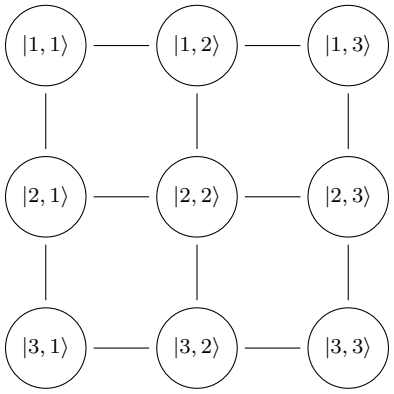

Figura 1: Diagrama esquemático da geometria bidimensional para o modelo de Anderson, bem como a ordenação dos orbitais $|i,j\rangle$ utilizados. O Hamiltoniano é escrito em uma rede 3×3, onde cada sítio é representado pelos índices i, j. Em cada sítio da rede temos um orbital atômico $|i,j\rangle$.

```
#define TOL 1e-10
#define HOPPING 1

// Aloca matriz NxN dinamicamente
double** alloc_matrix(int N) {
    double **mat = (double**) malloc(N * sizeof(double*));
    for(int i=0;i<N;i++)
        mat[i] = (double*) malloc(N * sizeof(double));
    return mat;
}

// Libera matriz
void free_matrix(double **mat, int N){
    for(int i=0;i<N;i++) free(mat[i]);
    free(mat);
}


void gauss(int N, double **A, double *b, double *x){
    // cria matriz aumentada Nx(N+1)
    double **M = (double**) malloc(N * sizeof(double*));
    for(int i=0;i<N;i++){
        M[i] = (double*) malloc((N+1)*sizeof(double));
        for(int j=0;j<N;j++) M[i][j] = A[i][j];
        M[i][N] = b[i];
    }

    // eliminação
    for(int k=0;k<N;k++){
        double pivot = M[k][k];
        for(int j=k;j<=N;j++) M[k][j]/=pivot;
        for(int i=k+1;i<N;i++){
            double f = M[i][k];
            for(int j=k;j<=N;j++) M[i][j] -= f*M[k][j];
        }
    }

    // substituição regressiva
    for(int i=N-1;i>=0;i--){
        x[i] = M[i][N];
        for(int j=i+1;j<N;j++) x[i]-= M[i][j]*x[j];
    }

    for(int i=0;i<N;i++) free(M[i]);
    free(M);
}



// Vetor: norma, normalize, dot
double norm(int N, double *v){
    double s=0;
    for(int i=0;i<N;i++) s+=v[i]*v[i];
    return sqrt(s);
```

```
}
void normalize(int N, double *v){
    double n = norm(N,v);
    for(int i=0;i<N;i++) v[i]/=n;
}
double dot(int N, double *u, double *v){
    double s=0;
    for(int i=0;i<N;i++) s+=u[i]*v[i];
    return s;
}

int main(){
    int L;
    printf("Digite L (numero de sitios na aresta): ");
    scanf("%d",&L);
    int N=L*L;

    // Aloca matrizes e vetores dinamicamente
    double **H = alloc_matrix(N);
    double **I = alloc_matrix(N);
    double **A_shifted = alloc_matrix(N);
    double *v = (double*) malloc(N*sizeof(double));
    double *w = (double*) malloc(N*sizeof(double));

    srand(0);
    double W = 1.0;

    // Hamiltoniano de Anderson 2D
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
            H[i][j]=0;
            if(i==j) H[i][j] = W*(rand()/(double)RAND_MAX - 0.5);
        }
    }
    for(int x=0;x<L;x++){
        for(int y=0;y<L;y++){
            int site = x*L + y;
            if(y+1<L){
                int right=x*L+(y+1);
                H[site][right]=H[right][site]=HOPPING;
            }
            if(x+1<L){
                int down=(x+1)*L+y;
                H[site][down]=H[down][site]=HOPPING;
            }
        }
    }

    // Inicializa vetores e matrizes auxiliares
    for(int i=0;i<N;i++){
        v[i]=1.0;
        for(int j=0;j<N;j++){
            I[i][j]=(i==j)?1.0:0.0;
        }
    }

    double mu=0.0;
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++)
            A_shifted[i][j] = H[i][j]-mu*I[i][j];
    }
    normalize(N,v);

    double lambda=0, lambda_old=0;
    for(int it=0;it<MAX_IT;it++){
        gauss(N,A_shifted,v,w);
        normalize(N,w);

        double *Hv = (double*) malloc(N*sizeof(double));
        for(int i=0;i<N;i++){
            Hv[i]=0;
            for(int j=0;j<N;j++) Hv[i]+=H[i][j]*w[j];
        }
        lambda = dot(N,w,Hv)/dot(N,w,w);
        free(Hv);
```

```
        for(int i=0;i<N;i++) v[i]=w[i];
        if(fabs(lambda-lambda_old)<TOL) break;
        lambda_old=lambda;
    }

    printf("Autovalor próximo de %.3f \approx  %.10f\n", mu, lambda);
    printf("Autovetor correspondente \approx  (");
    for(int i=0;i<N;i++) printf(" %.10f", v[i]);
    printf(" )\n");

    // Teste Hv \approx \lambda v
    double *Hv = (double*) malloc(N*sizeof(double));
    for(int i=0;i<N;i++){
        Hv[i]=0;
        for(int j=0;j<N;j++) Hv[i]+=H[i][j]*v[j];
    }
    printf("\nTeste Hv e \lambda v:\n");
    for(int i=0;i<N;i++)
        printf("Hv[%d]=%.10f   \lambda v[%d]=%.10f\n", i,Hv[i],i,lambda*v[i]);
    free(Hv);

    // Libera memória
    free_matrix(H,N);
    free_matrix(I,N);
    free_matrix(A_shifted,N);
    free(v); free(w);

    return 0;
}
```

Vamos debater ponto a ponto a implementação em C do método da potência inversa com deslocamento, aplicada ao Hamiltoniano de Anderson em duas dimensões. Como já mencionamos diversas vezes, o objetivo é encontrar autovalores e autovetores próximos de um chute inicial μ em matrizes não tridiagonais. O código faz uso de alocação dinâmica de memória e ponteiros, permitindo simular redes $L \times L$ de tamanho arbitrário.

Para criar matrizes $N \times N$ em tempo de execução (com $N = L^2$), usamos ponteiros para ponteiros:

```
// Aloca matriz NxN dinamicamente
double** alloc_matrix(int N) {
    double **mat = (double**) malloc(N * sizeof(double*));
    for(int i=0;i<N;i++)
        mat[i] = (double*) malloc(N * sizeof(double));
    return mat;
}

// Libera matriz
void free_matrix(double **mat, int N){
    for(int i=0;i<N;i++) free(mat[i]);
    free(mat);
}
```

Cada linha da matriz é um ponteiro para um vetor de doubles, o que permite criar matrizes de tamanho variável. Vetores como v e w também são alocados dinamicamente:

```
double *v = (double*) malloc(N*sizeof(double));
double *w = (double*) malloc(N*sizeof(double));
```

O Hamiltoniano H é composto por dois termos:

- **Diagonal**: energias aleatórias $\varepsilon_{i,j} \in [-W/2, W/2]$ simulando desordem.
- **Hopping**: conecta cada sítio aos vizinhos imediatos na grade $L \times L$ com amplitude $t = \text{HOPPING}$.

```
// Hamiltoniano de Anderson 2D
for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
        H[i][j]=0;
        if(i==j) H[i][j] = W*(rand()/(double)RAND_MAX - 0.5);
    }
}
for(int x=0;x<L;x++){
    for(int y=0;y<L;y++){
        int site = x*L + y;
        if(y+1<L){
            int right=x*L+(y+1);
            H[site][right]=H[right][site]=HOPPING;
        }
        if(x+1<L){
            int down=(x+1)*L+y;
            H[site][down]=H[down][site]=HOPPING;
        }
    }
}
```

O método da potência inversa com deslocamento requer resolver sistemas lineares $(H-\mu I)w = v$ a cada iteração. Para isso, implementamos a eliminação de Gauss com matrizes aumentadas:

```
void gauss(int N, double **A, double *b, double *x){
    // Matriz aumentada Nx(N+1)
    double **M = alloc_matrix(N);
    for(int i=0;i<N;i++){
        for(int j=0;j<N;j++) M[i][j] = A[i][j];
        M[i][N] = b[i];
    }

    // Eliminação para triangular superior
    for(int k=0;k<N;k++){
        double pivot = M[k][k];
        for(int j=k;j<=N;j++) M[k][j]/=pivot;
        for(int i=k+1;i<N;i++){
            double f = M[i][k];
            for(int j=k;j<=N;j++) M[i][j] -= f*M[k][j];
        }
    }

    // Substituição regressiva
    for(int i=N-1;i>=0;i--){
        x[i] = M[i][N];
        for(int j=i+1;j<N;j++) x[i]-= M[i][j]*x[j];
    }

    free_matrix(M,N);
}
```

Funções auxiliares calculam a norma euclidiana, normalizam vetores e realizam produtos internos:

```
double norm(int N, double *v){
    double s=0;
    for(int i=0;i<N;i++) s+=v[i]*v[i];
    return sqrt(s);
}

void normalize(int N, double *v){
    double n = norm(N,v);
    for(int i=0;i<N;i++) v[i]/=n;
}

double dot(int N, double *u, double *v){
    double s=0;
    for(int i=0;i<N;i++) s+=u[i]*v[i];
    return s;
}
```

Lembramos que o procedimento principal consiste em:

1. Escolher um chute inicial μ para o autovalor desejado.

2. Inicializar o vetor v com valores não nulos e normalizar.

3. Iterar: resolver $(H - \mu I)w = v$ usando Gauss, normalizar w, calcular $\lambda = \frac{w^T H w}{w^T w}$ e atualizar $v \leftarrow w$.

4. Repetir até que $|\lambda - \lambda_{\text{old}}| < \text{TOL}$.

Após convergência, o código calcula Hv e verifica que $Hv \approx \lambda v$, garantindo que o autovalor e autovetor encontrados são consistentes.
Este código ilustra como o método da potência inversa com deslocamento pode ser aplicado a Hamiltonianos de Anderson 2D de forma escalável. A utilização de ponteiros e alocação dinâmica permite:

- Simular redes $L \times L$ de tamanho arbitrário.

- Resolver sistemas lineares grandes sem conhecimento prévio do tamanho.

- Obter autovalores próximos de chutes iniciais, mesmo em matrizes não tridiagonais.

Essa abordagem combina simplicidade conceitual e flexibilidade computacional, permitindo aplicar o método da potência inversa com deslocamento a Hamiltonianos de Anderson 2D de tamanho arbitrário. Ela é bastante útil para estudos exploratórios de desordem quântica e localização eletrônica em redes bidimensionais.
Entretanto, existem algumas limitações importantes:

- O método é sensível ao chute inicial μ; autovalores muito distantes podem não convergir.

- Convergência pode ser lenta, especialmente em sistemas com espectro denso ou desordem intermediária.

- Resolver o sistema linear $(H - \mu I)w = v$ usando eliminação de Gauss simples não é eficiente para matrizes grandes. Métodos iterativos como GMRES ou LU com pivotamento seriam mais adequados.

- A memória necessária cresce rapidamente com $N = L^2$, limitando o tamanho máximo da rede que pode ser simulada em computadores comuns.

- O método obtém apenas um autovalor/autovetor por vez; calcular múltiplos autovalores próximos requer múltiplas execuções ou variantes mais sofisticadas (como deflation ou Lanczos).

Apesar dessas limitações, a implementação serve como excelente ferramenta didática, permitindo visualizar efeitos da desordem e familiarizar-se com técnicas de álgebra linear aplicadas a problemas de física computacional.

25 Autovalores e autovetores : Método da potência inversa com deslocamento otimizado para o cálculo de autovalores e autovetores do Modelo de Anderson 1D com correlações gaussianas na desordem

Nesta seção apresentamos uma estratégia prática para implementar o método da potência inversa com deslocamento, totalmente adaptada ao Modelo de Anderson unidimensional (matriz tridiagonal) com desordem correlacionada de forma gaussiana. Em vez de montar explicitamente uma matriz quadrada densa, trabalhamos apenas com os termos não nulos — ou seja, com os vetores que representam a diagonal principal e as sub-/super-diagonais. Essa abordagem reduz drasticamente o custo de memória e operações, pois todas as multiplicações e resoluções de sistemas são realizadas em complexidade $\mathcal{O}(N)$ e armazenamento $\mathcal{O}(N)$. Utilizando o método de Thomas para resolver $(H - \mu I)y = x$ e operadores produto-matriz-vetor implementados através dessas três bandas, evitamos a alocação e manipulação de matrizes $N \times N$. O ganho prático é significativo: torna-se viável estudar sistemas muito maiores (por exemplo $N > 10^5$) com tempo de execução e uso de memória compatíveis com máquinas modernas. Além da eficiência, essa formulação minimiza custos de cópia de dados, facilita a vetorização/paralelização e preserva a robustez numérica necessária para a convergência da iteração inversa. Combinada com a geração de hopping com correlação gaussiana, a implementação permite investigar de forma escalável propriedades de localização, como a *participation ratio*, em regimes de grande tamanho e amostragem estatística. Apresentarei inicialmente o código completo e, em seguida, discutirei cada uma de suas partes separadamente.

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

#define MAX_IT 1000
#define TOL 1e-8
#define PI 3.14159265358979323846

// Gera número aleatório entre 0 e 1
double rand01() {
    return ((double)rand() ) / ((double)RAND_MAX);
}

// Gera número gaussiano ~ N(0,1) usando Box-Muller
double gauss_rand() {
    double u = rand01();
    double v = rand01();
    return sqrt(-2.0 * log(u)) * cos(2.0 * PI * v);
}

// Gera ruído branco gaussiano
void gera_ruido_branco(double *ruido, int N1) {
    for (int i = 0; i < N1; i++) {
        double u = ((double)rand() + 1.0) / ((double)RAND_MAX + 2.0);
        double v = ((double)rand() + 1.0) / ((double)RAND_MAX + 2.0);
        ruido[i] = sqrt(-2.0 * log(u)) * cos(2.0 * M_PI * v);  // Box-Muller
    }
}

// Convolui vetor com kernel gaussiano para criar correlação de longo alcance
void gera_correlacao_gaussiana(double *vetor, int N1, double sigma) {
    double *ruido = malloc(N1 * sizeof(double));
    gera_ruido_branco(ruido, N1);
    double soma, soma_abs, rr1;

    // Aplica convolução
    for (int i = 0; i < N1; i++) {
        soma = 0.;
        for (int j = 0; j < N1; j++) {
```

```
            soma += ruido[j] * exp(-pow((double)(i - j), 2.) / (sigma * sigma));
        }
        vetor[i] = soma;
    }

    soma = 0.;
    soma_abs = 0.;
    rr1 = (double)N1;
    for (int i = 0; i < N1; i++) {
        soma += vetor[i];
        soma_abs += vetor[i] * vetor[i];
    }
    soma = soma / rr1;
    soma_abs = soma_abs / rr1;

    for (int i = 0; i < N1; i++) {
        vetor[i] = (vetor[i] - soma) / sqrt(soma_abs - soma * soma);
    }
    for (int i = 0; i < N1; i++) {
        vetor[i] = 0.5 * tanh(vetor[i]) + 1.;
    }

    free(ruido);
}

// Método de Thomas para sistema tridiagonal A*x = d
void thomas(int n, double *a, double *b, double *c, double *d, double *x) {
    double *c_prime = malloc((n - 1) * sizeof(double));
    double *d_prime = malloc(n * sizeof(double));

    c_prime[0] = c[0] / b[0];
    d_prime[0] = d[0] / b[0];

    for (int i = 1; i < n - 1; i++) {
        double m = b[i] - a[i - 1] * c_prime[i - 1];
        c_prime[i] = c[i] / m;
        d_prime[i] = (d[i] - a[i - 1] * d_prime[i - 1]) / m;
    }
    d_prime[n - 1] = (d[n - 1] - a[n - 2] * d_prime[n - 2]) / (b[n - 1] - a[n - 2] * c_prime[n - 2]);

    x[n - 1] = d_prime[n - 1];
    for (int i = n - 2; i >= 0; i--)
        x[i] = d_prime[i] - c_prime[i] * x[i + 1];

    free(c_prime);
    free(d_prime);
}

// Produto matriz-vetor y = H*x, adaptado para matriz tridiagonal
void mat_vec_tridiagonal(int n, double *b_diag, double *a_sub, double *c_sup, double *x, double *y) {
    // Primeiro elemento
    y[0] = b_diag[0] * x[0] + c_sup[0] * x[1];
    // Elementos do meio
    for (int i = 1; i < n - 1; i++) {
        y[i] = a_sub[i - 1] * x[i - 1] + b_diag[i] * x[i] + c_sup[i] * x[i + 1];
    }
    // Último elemento
    y[n - 1] = a_sub[n - 2] * x[n - 2] + b_diag[n - 1] * x[n - 1];
}

// Norma Euclidiana
double norm(int n, double *x) {
    double s = 0.0;
    for (int i = 0; i < n; i++) s += x[i] * x[i];
    return sqrt(s);
}

// Produto interno
double dot(int n, double *x, double *y) {
    double s = 0.0;
    for (int i = 0; i < n; i++) s += x[i] * y[i];
    return s;
}

int main() {
    int N, nmed, ym;
```

```
 double ll2, mu, gg1;
 double par, parm;
 printf("Digite o tamanho do sistema N: ");
 scanf("%d", &N);
 printf("Medias: ");
 scanf("%d", &nmed);
 printf("L: ");
 scanf("%lf", &gg1);
 printf("E Alvo: ");
 scanf("%lf", &mu);

// srand(time(NULL));
 srand(1);
 // Vetores que definem a matriz H (tridiagonal)
 double *V = malloc(N * sizeof(double));
 double *b_diag = malloc(N * sizeof(double));
 double *a_sub = malloc((N - 1) * sizeof(double)); // Subdiagonal
 double *c_sup = malloc((N - 1) * sizeof(double)); // Superdiagonal

 // Vetores para a Iteração Inversa
 double *x = malloc(N * sizeof(double));
 double *y = malloc(N * sizeof(double));
 double *Hx = malloc(N * sizeof(double));

 // Vetores para o Thomas Algorithm
 double *a_thomas = malloc((N - 1) * sizeof(double));
 double *b_thomas = malloc(N * sizeof(double));
 double *c_thomas = malloc((N - 1) * sizeof(double));

 char filename[50];
 snprintf(filename, sizeof(filename), "ParhoppxNesparsaN%dE%.3f_MED%dL%.2f.dat", N, mu, nmed, gg1);
 FILE *f = fopen(filename, "w");

 parm = 0.;
 ll2 = 0.;
 for (ym = 1; ym <= nmed; ym++) {

     // Gera os termos de "hopping" correlacionados
     gera_correlacao_gaussiana(V, N, gg1);

     // Constrói os vetores para a matriz H
     for (int i = 0; i < N; i++) {
         b_diag[i] = 0.0; // Termos na diagonal são zero
         if (i < N - 1) {
             a_sub[i] = V[i];
             c_sup[i] = V[i];
         }
     }

     // Constrói os vetores para a matriz (H - mu*I) para o Thomas Algorithm
     for (int i = 0; i < N; i++) {
         b_thomas[i] = b_diag[i] - mu;
     }
     for (int i = 0; i < N - 1; i++) {
         a_thomas[i] = a_sub[i];
         c_thomas[i] = c_sup[i];
     }

     // Inicializa o vetor de chute para a Iteração Inversa
     for (int i = 0; i < N; i++) {
         x[i] = 1.0;
     }

     double lambda = 0.0, lambda_old = 0.0;

     // Iteração inversa
     for (int iter = 0; iter < MAX_IT; iter++) {
         // resolve (H - mu*I) * y = x
         thomas(N, a_thomas, b_thomas, c_thomas, x, y);

         double y_norm = norm(N, y);
         for (int i = 0; i < N; i++) y[i] /= y_norm;

         // Calcula H*y
         mat_vec_tridiagonal(N, b_diag, a_sub, c_sup, y, Hx);
         lambda = dot(N, y, Hx);
```

```
            if (fabs(lambda - lambda_old) < TOL) break;
            lambda_old = lambda;

            for (int i = 0; i < N; i++) x[i] = y[i];
        }

        ll2 += lambda;

        par = 0.;
        for (int i = 0; i < N; i++) {
            par += pow(fabs(y[i]), 4.);
        }
        par = 1. / par;
        parm += par;
    }

    ll2 /= (double)nmed;
    parm /= (double)nmed;

    fprintf(f, "%.8f %.8f %.8f\n", (double)N, parm, ll2);

    fclose(f);

    // Liberação da memória
    free(V); free(b_diag); free(a_sub); free(c_sup);
    free(x); free(y); free(Hx);
    free(a_thomas); free(b_thomas); free(c_thomas);

    return 0;
}
```

Vamos agora debater o código em partes:

Geração de Ruído Correlacionado

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

#define MAX_IT 1000
#define TOL 1e-8
#define PI 3.14159265358979323846

// Gera número aleatório entre 0 e 1
double rand01() {
    return ((double)rand() ) / ((double)RAND_MAX);
}

// Gera número gaussiano ~ N(0,1) usando Box-Muller
double gauss_rand() {
    double u = rand01();
    double v = rand01();
    return sqrt(-2.0 * log(u)) * cos(2.0 * PI * v);
}

// Gera ruído branco gaussiano
void gera_ruido_branco(double *ruido, int N1) {
    for (int i = 0; i < N1; i++) {
        double u = ((double)rand() + 1.0) / ((double)RAND_MAX + 2.0);
        double v = ((double)rand() + 1.0) / ((double)RAND_MAX + 2.0);
        ruido[i] = sqrt(-2.0 * log(u)) * cos(2.0 * M_PI * v);  // Box-Muller
    }
}

// Convolui vetor com kernel gaussiano para criar correlação de longo alcance
void gera_correlacao_gaussiana(double *vetor, int N1, double sigma) {
    double *ruido = malloc(N1 * sizeof(double));
    gera_ruido_branco(ruido, N1);
```

```
    double soma, soma_abs, rr1;

    // Aplica convolução
    for (int i = 0; i < N1; i++) {
        soma = 0.;
        for (int j = 0; j < N1; j++) {
            soma += ruido[j] * exp(-pow((double)(i - j), 2.) / (sigma * sigma));
        }
        vetor[i] = soma;
    }

    soma = 0.;
    soma_abs = 0.;
    rr1 = (double)N1;
    for (int i = 0; i < N1; i++) {
        soma += vetor[i];
        soma_abs += vetor[i] * vetor[i];
    }
    soma = soma / rr1;
    soma_abs = soma_abs / rr1;

    for (int i = 0; i < N1; i++) {
        vetor[i] = (vetor[i] - soma) / sqrt(soma_abs - soma * soma);
    }
    for (int i = 0; i < N1; i++) {
        vetor[i] = 0.5 * tanh(vetor[i]) + 1.;
    }

    free(ruido);
}
```

A primeira parte do código é responsável por gerar as desordens que entram no Hamiltoniano. Inicialmente, é definida uma função para produzir números aleatórios uniformes no intervalo $[0, 1]$, a partir da qual se constrói um gerador de variáveis gaussianas $\mathcal{N}(0, 1)$ pelo método de *Box-Muller*. Esse procedimento é implementado nas funções `rand01` e `gauss_rand`. Em seguida, a função `gera_ruido_branco` constrói um vetor de ruído branco gaussiano, isto é, uma sequência de números aleatórios independentes e distribuídos normalmente. Para introduzir correlação espacial de longo alcance, utilizamos a função `gera_correlacao_gaussiana`: o vetor de ruído branco é convoluído com um núcleo gaussiano $\exp(-(i-j)^2/\sigma^2)$, de forma que cada valor no sítio i recebe contribuições de todos os sítios j, ponderadas pela distância. Posteriormente, o vetor resultante é normalizado para ter média nula e variância unitária, garantindo homogeneidade estatística. Finalmente, aplica-se uma transformação não linear do tipo tanh, deslocada e reescalada, de modo a limitar os valores e gerar uma distribuição suave dos elementos de *hopping* correlacionados. Esse procedimento assegura que o modelo incorpore desordens com estrutura não trivial, mais realistas do que o ruído puramente aleatório.

Solução de sistemas tridiagonais: implementação do método de Thomas

A seguir incluímos o trecho do código que implementa o método de Thomas adaptado para matrizes tridiagonais (usando diagonal e subdiagonais):

```
// Método de Thomas para sistema tridiagonal A*x = d
void thomas(int n, double *a, double *b, double *c, double *d, double *x) {
    double *c_prime = malloc((n - 1) * sizeof(double));
    double *d_prime = malloc(n * sizeof(double));

    c_prime[0] = c[0] / b[0];
    d_prime[0] = d[0] / b[0];

    for (int i = 1; i < n - 1; i++) {
        double m = b[i] - a[i - 1] * c_prime[i - 1];
```

```
        c_prime[i] = c[i] / m;
        d_prime[i] = (d[i] - a[i - 1] * d_prime[i - 1]) / m;
    }
    d_prime[n - 1] = (d[n - 1] - a[n - 2] * d_prime[n - 2]) / (b[n - 1] - a[n - 2] * c_prime[n - 2]);

    x[n - 1] = d_prime[n - 1];
    for (int i = n - 2; i >= 0; i--)
        x[i] = d_prime[i] - c_prime[i] * x[i + 1];

    free(c_prime);
    free(d_prime);
}
```

Explicação detalhada e adaptação para armazenamento em bandas Neste algoritmo assumimos que a matriz tridiagonal A está armazenada em três vetores:

$$\begin{aligned} b &= (b_0, \ldots, b_{n-1}) \quad \text{(diagonal principal)}, \\ a &= (a_0, \ldots, a_{n-2}) \quad \text{(subdiagonal, } A_{i,i-1} = a_{i-1}), \\ c &= (c_0, \ldots, c_{n-2}) \quad \text{(superdiagonal, } A_{i,i+1} = c_i). \end{aligned}$$

O vetor d é o lado direito do sistema $Ax = d$ e x é a solução procurada. A grande vantagem deste arranjo é que não existe alocação da matriz $N \times N$: só armazenamos os termos não nulos (as três bandas), reduzindo memória para $\mathcal{O}(N)$.

Idéia do método (varredura para frente e retro-substituição) O método de Thomas realiza duas fases:

1. *Forward sweep* (eliminação direta): calcula-se coeficientes modificados c'_i e d'_i que correspondem à eliminação gaussiana especializada para a estrutura tridiagonal. As fórmulas usadas no código são

   $$c'_0 = \frac{c_0}{b_0}, \qquad d'_0 = \frac{d_0}{b_0},$$

   e, para $i = 1, \ldots, n-2$,

   $$m = b_i - a_{i-1}\, c'_{i-1}, \qquad c'_i = \frac{c_i}{m}, \qquad d'_i = \frac{d_i - a_{i-1}\, d'_{i-1}}{m}.$$

   Finalmente,

   $$d'_{n-1} = \frac{d_{n-1} - a_{n-2}\, d'_{n-2}}{b_{n-1} - a_{n-2}\, c'_{n-2}}.$$

   Essas expressões resultam de eliminar progressivamente a subdiagonal a, sem tocar em elementos que são zero por construção.

2. *Back substitution* (retro-substituição): com os coeficientes modificados obtemos a solução por

   $$x_{n-1} = d'_{n-1}, \qquad x_i = d'_i - c'_i\, x_{i+1} \quad (i = n-2, \ldots, 0).$$

Propriedades numéricas e de eficiência

- **Complexidade:** o algoritmo executa $\mathcal{O}(N)$ operações e usa $\mathcal{O}(N)$ de memória adicional (aqui alocado em `c_prime` e `d_prime`). Isso contrasta com a eliminação gaussiana densa, que usaria $\mathcal{O}(N^3)$ tempo e $\mathcal{O}(N^2)$ memória.
- **Não é necessário montar a matriz completa:** como só referenciamos a, b, c, todas as operações são feitas diretamente nas bandas; isso permite escalar para $N \gg 10^5$.

- **Robustez:** é imprescindível garantir que os pivôs b_0 e $m = b_i - a_{i-1}c'_{i-1}$ nunca sejam zero (ou muito pequenos). O método de Thomas não faz pivotamento; se ocorrer um pivô nulo, torna-se necessário usar permutações de linhas (solver mais geral) ou aplicar um pequeno deslocamento numérico na diagonal para restaurar a estabilidade.

- **Economias adicionais:** para reduzir ainda mais memória pode-se sobrescrever os vetores `c` e `d` com `c_prime` e `d_prime` (in-place), evitando alocações extras. O código atual usa buffers separados por clareza.

- **Paralelização:** a varredura é inerentemente sequencial (cada etapa depende do passo anterior), o que limita paralelização fina; no entanto, o custo linear e a baixa utilização de memória tornam-no extremamente eficiente na prática em CPUs modernas.

Integração com a iteração inversa No contexto do programa principal, para resolver $(H - \mu I)y = x$ basta construir os vetores $\tilde{b} = b - \mu$ (diagonal deslocada) e passar $\tilde{b}, a, c$ à função `thomas`. Dessa forma, cada iteração inversa requer apenas uma solução tridiagonal com custo $\mathcal{O}(N)$, o que torna a busca por autovalores próximos de μ bastante eficiente para grandes tamanhos de sistema.

Produto Matriz-Vetor

```
// Produto matriz-vetor y = H*x, adaptado para matriz tridiagonal
void mat_vec_tridiagonal(int n, double *b_diag, double *a_sub, double *c_sup, double *x, double *y) {
    // Primeiro elemento
    y[0] = b_diag[0] * x[0] + c_sup[0] * x[1];
    // Elementos do meio
    for (int i = 1; i < n - 1; i++) {
        y[i] = a_sub[i - 1] * x[i - 1] + b_diag[i] * x[i] + c_sup[i] * x[i + 1];
    }
    // Último elemento
    y[n - 1] = a_sub[n - 2] * x[n - 2] + b_diag[n - 1] * x[n - 1];
}

// Norma Euclidiana
double norm(int n, double *x) {
    double s = 0.0;
    for (int i = 0; i < n; i++) s += x[i] * x[i];
    return sqrt(s);
}

// Produto interno
double dot(int n, double *x, double *y) {
    double s = 0.0;
    for (int i = 0; i < n; i++) s += x[i] * y[i];
    return s;
}
```

O trecho acima implementa três funções fundamentais para os cálculos envolvendo o Hamiltoniano do modelo de Anderson em uma base de tamanho N. A função `mat_vec_tridiagonal` realiza o produto matriz-vetor $y = Hx$, explorando o fato de que H é tridiagonal e pode ser representado apenas pelas três diagonais não nulas: a diagonal principal (`b_diag`), a subdiagonal (`a_sub`) e a superdiagonal (`c_sup`). Isso evita a necessidade de manipular uma matriz densa $N \times N$, reduzindo o custo computacional e o uso de memória de $\mathcal{O}(N^2)$ para $\mathcal{O}(N)$.

Além disso, são definidas duas rotinas auxiliares: `norm`, que calcula a norma euclidiana de um vetor, e `dot`, que computa o produto interno entre dois vetores. Ambas serão utilizadas em diferentes etapas da iteração inversa, como normalização dos autovetores e cálculo de resíduos.

A parte inicial do "main"do código

```
int main() {
    int N, nmed, ym;
    double ll2, mu, gg1;
    double par, parm;
    printf("Digite o tamanho do sistema N: ");
    scanf("%d", &N);
    printf("Medias: ");
    scanf("%d", &nmed);
    printf("L: ");
    scanf("%lf", &gg1);
    printf("E Alvo: ");
    scanf("%lf", &mu);

   // srand(time(NULL));
    srand(1);
    // Vetores que definem a matriz H (tridiagonal)
    double *V = malloc(N * sizeof(double));
    double *b_diag = malloc(N * sizeof(double));
    double *a_sub = malloc((N - 1) * sizeof(double)); // Subdiagonal
    double *c_sup = malloc((N - 1) * sizeof(double)); // Superdiagonal

    // Vetores para a Iteração Inversa
    double *x = malloc(N * sizeof(double));
    double *y = malloc(N * sizeof(double));
    double *Hx = malloc(N * sizeof(double));

    // Vetores para o Thomas Algorithm
    double *a_thomas = malloc((N - 1) * sizeof(double));
    double *b_thomas = malloc(N * sizeof(double));
    double *c_thomas = malloc((N - 1) * sizeof(double));

    char filename[50];
    snprintf(filename, sizeof(filename), "ParhoppxNesparsaN%dE%.3f_MED%dL%.2f.dat", N, mu, nmed, gg1);
    FILE *f = fopen(filename, "w");

    parm = 0.;
    ll2 = 0.;
    for (ym = 1; ym <= nmed; ym++) {

        // Gera os termos de "hopping" correlacionados
        gera_correlacao_gaussiana(V, N, gg1);
```

O trecho acima corresponde à função `main`, responsável por controlar a execução do programa. Inicialmente, o usuário fornece como entrada: o tamanho do sistema N, o número de médias estatísticas `nmed`, o parâmetro de correlação `L` e a energia alvo μ.
Em seguida, são alocados dinamicamente os vetores necessários para representar a matriz tridiagonal H (diagonal principal, subdiagonal e superdiagonal), além dos vetores usados no método de Iteração Inversa e no algoritmo de Thomas.
Por fim, abre-se o arquivo de saída onde os resultados serão armazenados, e inicia-se o loop sobre o número de médias. Em cada iteração, a função `gera_correlacao_gaussiana` é chamada para construir os termos de "hopping" correlacionados, que modelam o desordem com correlação espacial no sistema de Anderson.

Construção da matriz tridiagonal H

```
    // Constrói os vetores para a matriz H
    for (int i = 0; i < N; i++) {
        b_diag[i] = 0.0; // Termos na diagonal são zero
        if (i < N - 1) {
            a_sub[i] = V[i];
            c_sup[i] = V[i];
        }
    }
```

Neste trecho, definimos explicitamente a estrutura da matriz tridiagonal H. A diagonal principal (`b_diag`) é inicializada com zeros, enquanto a subdiagonal (`a_sub`) e a superdiagonal

(`c_sup`) recebem os valores do vetor `V`, previamente gerado com correlação espacial. Assim, a matriz H é construída de forma compacta, sem necessidade de armazenar a matriz quadrada completa. Isso reduz o custo de memória e permite simulações para tamanhos muito grandes ($N > 10^5$).

Iteração inversa com deslocamento

```
// Constrói os vetores para a matriz (H - mu*I) para o Thomas Algorithm
for (int i = 0; i < N; i++) {
    b_thomas[i] = b_diag[i] - mu;
}
for (int i = 0; i < N - 1; i++) {
    a_thomas[i] = a_sub[i];
    c_thomas[i] = c_sup[i];
}

// Inicializa o vetor de chute para a Iteração Inversa
for (int i = 0; i < N; i++) {
    x[i] = 1.0;
}

double lambda = 0.0, lambda_old = 0.0;

// Iteração inversa
for (int iter = 0; iter < MAX_IT; iter++) {
    // resolve (H - mu*I) * y = x
    thomas(N, a_thomas, b_thomas, c_thomas, x, y);

    double y_norm = norm(N, y);
    for (int i = 0; i < N; i++) y[i] /= y_norm;

    // Calcula H*y
    mat_vec_tridiagonal(N, b_diag, a_sub, c_sup, y, Hx);
    lambda = dot(N, y, Hx);

    if (fabs(lambda - lambda_old) < TOL) break;
    lambda_old = lambda;

    for (int i = 0; i < N; i++) x[i] = y[i];
}

l12 += lambda;
```

Neste trecho, implementamos o núcleo do método de iteração inversa com deslocamento para encontrar o autovalor de H mais próximo de uma energia alvo μ.

Primeiro, construímos os vetores correspondentes à matriz $(H - \mu I)$, utilizando apenas os termos não nulos da tridiagonal. Isso permite resolver o sistema linear eficientemente com o método de Thomas.

Em seguida, inicializamos o vetor de chute `x` com valores iguais a 1 e realizamos a iteração inversa:

- resolve-se $(H - \mu I)y = x$ com Thomas;
- normaliza-se y;
- calcula-se o autovalor aproximado $\lambda = y^T H y$;
- verifica-se a convergência comparando com o valor anterior;
- atualiza-se $x \leftarrow y$ para a próxima iteração.

Ao final, `l12` acumula a média dos autovalores obtidos para cada configuração de desordem.

Cálculo do *Participation Ratio* e finalização

```
    par = 0.;
    for (int i = 0; i < N; i++) {
        par += pow(fabs(y[i]), 4.);
    }
    par = 1. / par;
    parm += par;
}

ll2 /= (double)nmed;
parm /= (double)nmed;

fprintf(f, "%.8f %.8f %.8f\n", (double)N, parm, ll2);

fclose(f);

// Liberação da memória
free(V); free(b_diag); free(a_sub); free(c_sup);
free(x); free(y); free(Hx);
free(a_thomas); free(b_thomas); free(c_thomas);

return 0;
}
```

Nesta última etapa, calculamos o *Participation Ratio* (PR) da função de onda obtida em cada realização de desordem. O PR é definido como $PR = 1/\sum_i |y_i|^4$ e fornece uma medida do grau de localização do autovetor: valores baixos indicam estados localizados, enquanto valores próximos de N indicam estados espalhados.

Após calcular o PR (`par`) para cada realização, acumulamos os resultados em `parm` e, ao final, fazemos a média sobre o número de médias estatísticas (`nmed`) para obter valores representativos.

Finalmente, os resultados são gravados em arquivo, todos os vetores alocados dinamicamente são liberados, e o programa retorna 0, concluindo a execução de forma limpa e eficiente.