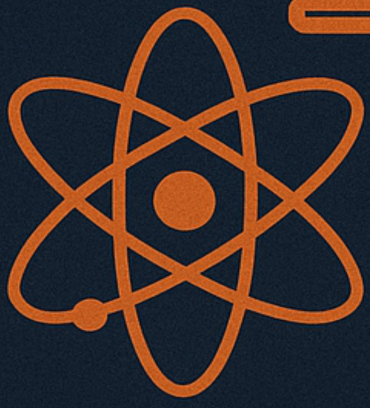
FRANCISCO ANACLETO BARROS FIDELIS DE MOURA

FÍSICA COMPUTACIONAL EM FORTRAN

NOTAS ESSENCIAIS

F



ISBN 978-65-01-75682-0

Física Computacional em Fortran: Notas Essenciais

Francisco Anacleto Barros Fidelis de Moura

Prefácio

A Física Computacional tornou-se uma ferramenta essencial para a investigação e compreensão de fenômenos complexos na física moderna. O presente material surge com o objetivo de fornecer ao estudante uma introdução prática e progressiva às metodologias numéricas utilizadas em simulações físicas, utilizando a linguagem Fortran como instrumento principal.
O curso foi estruturado de forma a permitir um aprendizado gradual. Inicialmente, o estudante é familiarizado com os conceitos fundamentais da linguagem, incluindo estruturas de repetição, manipulação de *arrays* e entrada/saída de dados, bem como técnicas básicas de cálculo, como derivação, integração e análise estatística.
Posteriormente, o foco se desloca para problemas mais complexos, envolvendo equações diferenciais, simulações estocásticas, transformadas de Fourier e métodos de autovalores e autovetores. Este progresso garante que o aluno compreenda tanto a implementação computacional quanto os fundamentos físicos subjacentes aos algoritmos.
Um aspecto central deste material é a transição didática entre Fortran 77 e Fortran 90/95. A abordagem permite que o estudante entenda a lógica de códigos legados e, simultaneamente, explore as funcionalidades avançadas do Fortran moderno, como módulos, alocação dinâmica, tipos de dados de precisão dupla e estruturas de controle mais legíveis.
Além do aprendizado conceitual, o curso introduz boas práticas de programação científica, incluindo modularização, uso de bibliotecas otimizadas e integração com linguagens de *script* para análise e visualização de dados.
Este prefácio pretende contextualizar o leitor, destacando que a prática em Fortran não se limita à execução de algoritmos, mas envolve também compreensão de desempenho, paralelismo e interoperabilidade com outras ferramentas computacionais.
Dessa forma, o material oferece uma ponte entre o conhecimento teórico em física e a capacidade de construir simulações robustas e eficientes, preparando o estudante para desafios de pesquisa e desenvolvimento em ambientes computacionais avançados.

Agradecimentos

Expresso minha profunda gratidão aos estudantes que se envolveram de maneira ativa na construção deste curso, contribuindo com perguntas instigantes, sugestões relevantes e desafios práticos que enriqueceram significativamente o aprendizado coletivo. Agradeço também ao Instituto de Física da minha universidade e aos colegas, amigos e colaboradores, cujo diálogo constante foi fundamental para o desenvolvimento e aperfeiçoamento deste trabalho acadêmico. Finalmente, registro meu reconhecimento à minha família, pelo apoio incondicional e presença diária, elementos essenciais que tornaram possível a realização desta 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 Fortran*, 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. Em seguida, exploraremos aplicações diretas, como o uso de **números complexos em Fortran**, 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 compreender, 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 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. Dessa forma, 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 Fortran

Antes de desenvolver programas mais complexos, é essencial compreender os principais comandos, funções e estruturas da linguagem **Fortran**. O domínio desses elementos permite escrever códigos corretos, eficientes e legíveis, além de facilitar a depuração e manutenção. A tabela a seguir resume as instruções mais importantes, indicando suas formas em **Fortran 77** e **Fortran 90**+, com exemplos e breves descrições. Serve como referência rápida tanto para iniciantes quanto para usuários intermediários.

Comando / Função	**F77 / F90+**	**Descrição / Exemplo**
`program ... end program`	F90+	Delimita um programa principal. Ex.: `program exemplo ... end program exemplo`.
`implicit none`	F90+	Exige declaração explícita de todas as variáveis (boas práticas modernas). F77: comportamento implícito com base na primeira letra do nome da variável.
`integer, real, double precision, complex, logical, character`	F77/F90+	Tipos de dados básicos. F90+ permite `real(kind=8)` (dupla precisão). Ex.: `integer :: i; real(kind=8) :: x`.
`parameter(...)`	F77/F90+	Define constantes. Ex.: `real, parameter :: pi = 3.14159`.
`print*, ...`	F77/F90+	Saída padrão formatada automaticamente. Ex.: `print*, "x =", x`.
`write(unit, fmt) var`	F77/F90+	Saída controlada. Ex.: `write(*, '(F8.3)') x`.
`read*, ...`	F77/F90+	Leitura de entrada padrão. Ex.: `read*, n`.
`if (...) then ... else ... end if`	F90+	Condicional estruturada. F77: `IF (...) GOTO 100` ou `IF (...) THEN ... ENDIF` (sem ELSE).
`select case (...) ... end select`	F90+	Estrutura condicional múltipla. Ex.: `select case(i); case(1); ... case default; ... end select`.
`do i = 1, 10 ... end do`	F77/F90+	Laço com contador. F77 exige rótulos (`DO 100 I=1,10 ... 100 CONTINUE`). F90+ permite fechamento com `end do`.
`do while (...) ... end do`	F90+	Laço condicional.
`exit, cycle`	F90+	Sai do laço (`exit`) ou pula para próxima iteração (`cycle`).
`stop, pause`	F77/F90+	Interrompe execução. `pause` (obsoleta).
`function ... end function`	F77/F90+	Declara uma função que retorna valor. Ex.: `real function f(x); f = x**2; end function f`.
`subroutine ... end subroutine`	F77/F90+	Declara um procedimento sem retorno direto. Chamado com `call nome_sub(... )`.
`call nome_sub(...)`	F77/F90+	Executa uma subrotina.
`return`	F77/F90+	Retorna de função ou subrotina.
`module ... end module`	F90+	Agrupa variáveis, parâmetros e funções reutilizáveis.
`use nome_modulo`	F90+	Importa conteúdo de um módulo.
`common /bloco/ var1, var2`	F77	Compartilha variáveis entre blocos (substituído por `module` no F90+).
`data var /valor/`	F77/F90+	Inicializa variáveis em tempo de compilação.
`allocatable, allocate(), deallocate()`	F90+	Alocação dinâmica de arrays. Ex.: `real, allocatable :: v(:); allocate(v(N)); deallocate(v)`.
`dimension A(10,10)`	F77/F90+	Declara tamanho de arrays. Em F90+, pode-se declarar junto ao tipo: `real :: A(10,10)`.
`intent(in/out/inout)`	F90+	Define modo de passagem de argumentos em funções/subrotinas.
`random_number(x)`	F90+	Gera número aleatório uniforme em [0,1]. F77: usar geradores externos (`RANF`, `RAND`).
`sqrt(), exp(), log(), sin(), cos(), tan(), atan(), abs()`	F77/F90+	Funções matemáticas elementares.
`ceiling(), floor(), nint(), mod(), sign()`	F90+	Arredondamento e manipulação de números reais.
`open(unit, file='dados.txt', status='unknown')`	F77/F90+	Abre arquivo. `close(unit)` fecha.
`inquire(file='dados.txt', exist=ok)`	F90+	Testa existência de arquivo.
`intent(in), intent(out), intent(inout)`	F90+	Define direção de dados em argumentos de subrotina.
`do concurrent (...) ... end do`	F2008+	Laço vetorizado (executável em paralelo, se possível).

Esta tabela amplia o conjunto de comandos básicos, destacando a evolução entre o **Fortran 77** (estruturado, baseado em rótulos) e o **Fortran 90**+ (modular, estruturado, e com suporte a alocação dinâmica e vetorização). Serve como referência prática tanto para consulta rápida durante a programação quanto como material de apoio didático.

2 Exemplos gerais

Operações Matemáticas Simples

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

```
program op_basicas
    implicit none
    real :: a, b

    a = 2.0
    b = 3.0

    print*, "Soma: ", a + b
    print*, "Produto: ", a * b
    print*, "Potência: ", a b
end program op_basicas
```

Esse exemplo mostra como usar operadores básicos e funções de saída com `print*` em Fortran.

Estruturas de Repetição (Loops)

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

```
! Programa que imprime os números de 1 a 10
program loop_for
    implicit none
    integer :: i

    do i = 1, 10
        print*, i
    end do
end program loop_for
```

Esse programa mostra o uso do `do` para repetir instruções de forma eficiente.

Máximo e Mínimo de um Conjunto de Dados

Frequentemente em física computacional e análise de dados, é necessário determinar rapidamente o valor máximo e mínimo de um conjunto de números. Isso é útil, por exemplo, para normalização de dados, detecção de picos em sinais ou verificação de limites físicos em simulações. No exemplo a seguir, mostramos como percorrer um vetor de inteiros e encontrar seus valores máximo e mínimo usando Fortran de forma simples e direta.
O programa inicializa um vetor com alguns valores, atribui os primeiros elementos como candidatos a máximo e mínimo, e percorre o restante do vetor atualizando essas variáveis conforme encontra novos extremos.

```
! Programa que encontra o máximo e o mínimo de um vetor
program max_min
    implicit none
    integer, dimension(5) :: v = (/3, 7, -2, 10, 5/)
    integer :: i, max, min

    ! Inicializa máximo e mínimo com o primeiro elemento do vetor
    max = v(1)
    min = v(1)

    ! Percorre o vetor a partir do segundo elemento
    do i = 2, size(v)
        if (v(i) > max) max = v(i)   ! Atualiza máximo
        if (v(i) < min) min = v(i)   ! Atualiza mínimo
    end do

    ! Imprime os resultados
    print*, "Max = ", max, ", Min = ", min
end program max_min
```

Ao executar este código, o programa retorna os valores máximo e mínimo do vetor, permitindo fácil análise ou uso posterior em cálculos adicionais. A lógica de percorrer cada elemento do vetor e comparar com os valores atuais de máximo e mínimo é fundamental para muitos algoritmos de processamento de dados e pode ser facilmente adaptada para vetores de tamanho maior ou dados reais.

Séries de Taylor: Aproximação da Exponencial

A função exponencial e^x é amplamente utilizada em física, matemática e engenharia. Uma forma de aproximá-la numericamente é através da série de Taylor centrada em zero ($x = 0$):

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

A aproximação melhora à medida que aumentamos o número de termos N da série. O código a seguir demonstra como calcular e^x usando uma implementação direta da série de Taylor em Fortran. Cada termo é calculado a partir do anterior, o que evita o uso repetido de potências e fatoriais, tornando o cálculo mais eficiente.

```
! Aproximação de e^x usando a série de Taylor
program taylor_exp
    implicit none
    real :: x, soma, termo
    integer :: n, N

    ! Valor de x para calcular e^x
    x = 1.0

    ! Inicializa soma e primeiro termo da série
    soma = 1.0
    termo = 1.0

    ! Número de termos da série
    N = 10
```

```
    ! Loop para somar os termos da série
    do n = 1, N
        termo = termo * x / n
        soma = soma + termo
    end do

    ! Imprime o resultado da aproximação
    print*, "Aproximacao de e^x em x=", x, ": ", soma
end program taylor_exp
```

Este programa fornece uma aproximação simples e eficiente de e^x. A cada iteração, o termo da série é atualizado multiplicando pelo valor de x e dividindo pelo índice atual, permitindo calcular rapidamente os termos sem fatoriais explícitos. Esse método pode ser facilmente adaptado para outros valores de x ou para aumentar a precisão aumentando o número de termos N. É uma excelente introdução ao uso de séries de Taylor em cálculos numéricos.

Média e Desvio Padrão

O cálculo da média e do desvio padrão é fundamental para análise estatística de conjuntos de dados, permitindo avaliar o valor central e a dispersão dos dados. A média aritmética fornece uma medida central, enquanto o desvio padrão quantifica a variabilidade em torno da média.

O programa abaixo demonstra como calcular a média e o desvio padrão de um vetor de números reais em Fortran. Primeiro, somamos todos os elementos para obter a média. Em seguida, calculamos a variância somando os quadrados das diferenças entre cada elemento e a média, e dividindo pelo número de elementos. O desvio padrão é obtido tomando a raiz quadrada da variância.

```
! Programa que calcula média e desvio padrão de um vetor
program media_desvio
    implicit none
    real, dimension(5) :: v = (/2.0, 4.0, 4.0, 4.0, 5.0/)
    integer :: n, i
    real :: soma, media, var, desvio

    n = size(v)       ! Número de elementos do vetor

    ! Calcula a soma dos elementos
    soma = 0.0
    do i = 1, n
        soma = soma + v(i)
    end do

    ! Calcula a média
    media = soma / n

    ! Calcula a variância
    var = 0.0
    do i = 1, n
        var = var + (v(i) - media) 2
    end do
```

```
    var = var / n

    ! Calcula o desvio padrão
    desvio = sqrt(var)

    ! Imprime os resultados
    print*, "Media = ", media, ", Desvio = ", desvio
end program media_desvio
```

Este código fornece uma implementação direta e didática dos conceitos estatísticos de média e desvio padrão. Pode ser facilmente adaptado para vetores maiores ou para dados de ponto flutuante mais complexos, servindo como base para análise de dados em simulações numéricas e experimentos computacionais.

Tabuada com Loops Aninhados

A tabuada é um exemplo clássico para demonstrar o uso de loops aninhados em programação. Em Fortran, podemos percorrer dois índices simultaneamente usando loops `do`, imprimindo os produtos de cada combinação de valores. Este tipo de estrutura é fundamental para manipulação de matrizes, tabelas de valores e operações combinatórias.
O programa abaixo imprime a tabuada completa de 1 a 10. O loop externo percorre os multiplicandos (i), enquanto o loop interno percorre os multiplicadores (j). A instrução `write` formata a saída de maneira organizada, mostrando cada multiplicação em uma linha.

```
! Programa que imprime a tabuada de 1 a 10 usando loops aninhados
program tabuada
    implicit none
    integer :: i, j

    ! Loop externo para os multiplicandos
    do i = 1, 10
        ! Loop interno para os multiplicadores
        do j = 1, 10
            ! Imprime i x j = resultado formatado
            write(*,'(I2," x ",I2," = ",I3,"   ")') i, j, i*j
        end do
        ! Linha em branco para separar cada tabuada
        print*,''
    end do
end program tabuada
```

Este código demonstra de forma simples como percorrer combinações de valores e gerar saídas estruturadas. Loops aninhados são essenciais em muitas aplicações científicas, incluindo iterações sobre matrizes, simulações físicas em grades 2D ou 3D e algoritmos combinatórios. A tabuada serve como exercício inicial para compreender estas estruturas fundamentais de programação.

Números Primos

A determinação de números primos é um exercício clássico em programação e matemática discreta. Um número primo é aquele que possui exatamente dois divisores distintos: 1 e

ele mesmo. Para testar a primalidade de um número n, não é necessário verificar todos os divisores até $n-1$; basta testar até a raiz quadrada de n, pois qualquer fator maior já teria um correspondente menor.

O programa abaixo implementa um algoritmo simples em Fortran para listar todos os números primos até um limite especificado (50 neste caso). Para cada número n, o programa verifica se existe algum divisor d entre 2 e $\sqrt{n}$. Se algum divisor divide n exatamente, o número não é primo e a verificação é interrompida imediatamente usando a instrução `exit`. Caso contrário, o número é considerado primo e é impresso na tela.

```
! Programa que lista todos os números primos até 50
program primos
    implicit none
    integer :: n, d, limite
    logical :: primo

    limite = 50   ! Define o limite superior para os números primos

    ! Loop sobre todos os números de 2 até o limite
    do n = 2, limite
        primo = .true.  ! Assume que n é primo inicialmente

        ! Testa divisores de 2 até a raiz quadrada de n
        do d = 2, int(sqrt(real(n)))
            if (mod(n,d) == 0) then
                primo = .false.  ! n não é primo
                exit             ! Interrompe o loop interno
            end if
        end do

        ! Se nenhum divisor foi encontrado, imprime n
        if (primo) print*, n
    end do
end program primos
```

Este código fornece uma implementação eficiente para números relativamente pequenos, aproveitando a propriedade de que só precisamos verificar divisores até $\sqrt{n}$. A instrução `exit` evita verificações desnecessárias, tornando o algoritmo mais rápido. Para limites maiores, algoritmos mais avançados, como a Crivo de Eratóstenes, podem ser utilizados para melhorar a eficiência.

Sequência de Fibonacci

A sequência de Fibonacci é uma das séries mais conhecidas em matemática, definida pelos termos iniciais $F_0 = 0$ e $F_1 = 1$, e pela relação de recorrência:

$$F_{n+1} = F_n + F_{n-1}, \quad n \geq 1.$$

Cada termo subsequente é a soma dos dois termos anteriores. Esta sequência possui inúmeras aplicações em ciência, engenharia e computação, além de conexões com proporções áureas e crescimento exponencial.

O programa abaixo gera os primeiros 10 termos da sequência de Fibonacci utilizando um método iterativo em Fortran. Inicialmente, definimos os dois primeiros termos ($f0$ e $f1$), imprimimos esses valores e, em seguida, usamos um loop para calcular cada termo subsequente, atualizando as variáveis a cada iteração. Este procedimento evita chamadas recursivas, sendo eficiente e direto para sequências pequenas e médias.

```
! Programa que gera os primeiros termos da sequência de Fibonacci
program fibonacci
    implicit none
    integer :: N, i
    integer :: f0, f1, fn

    N = 10       ! Número de termos desejados
    f0 = 0       ! Primeiro termo
    f1 = 1       ! Segundo termo

    ! Imprime os dois primeiros termos
    print*, f0, f1,

    ! Loop para calcular os próximos termos
    do i = 2, N-1
        fn = f0 + f1         ! Próximo termo da sequência
        print*, fn,          ! Imprime o termo calculado
        f0 = f1              ! Atualiza f0 para o próximo cálculo
        f1 = fn              ! Atualiza f1 para o próximo cálculo
    end do

    print*,''
end program fibonacci
```

Este código ilustra como sequências recursivas podem ser construídas iterativamente , economizando memória e evitando sobrecarga de chamadas recursivas. Além disso, a implementação direta em Fortran demonstra conceitos de loops, atualização de variáveis e impressão formatada, essenciais para o aprendizado de programação científica.

Leitura e Escrita em Arquivos

A manipulação de arquivos é essencial em programação científica, permitindo salvar resultados de cálculos, registrar dados experimentais ou reutilizar informações em diferentes etapas de um programa. Em Fortran, podemos abrir arquivos para escrita ou leitura , escrever dados formatados, e posteriormente lê-los para processamentos adicionais, como somatórios, médias ou análises estatísticas.

O programa a seguir demonstra um exemplo simples: ele escreve os números de 1 a 5 em um arquivo chamado `dados.txt`, fecha o arquivo, reabre-o para leitura e soma todos os valores presentes. A instrução `iostat` é utilizada para detectar o fim do arquivo, permitindo que o loop de leitura seja interrompido corretamente.

```
! Programa que escreve e lê números de um arquivo em Fortran
program arquivo_simples
    implicit none
```

```
    integer :: i, x, soma
    character(len=20) :: nome_arquivo

    nome_arquivo = 'dados.txt'  ! Nome do arquivo

    ! ---------- Escrita ----------
    open(unit=10, file=nome_arquivo, status='replace', action='write')
    do i = 1, 5
        write(10,*) i          ! Escreve cada número no arquivo
    end do
    close(10)

    ! ---------- Leitura e soma ----------
    soma = 0
    open(unit=10, file=nome_arquivo, status='old', action='read')
    do
        read(10,*,iostat=i) x  ! Lê um número do arquivo
        if (i /= 0) exit        ! Sai do loop ao atingir o fim do arquivo
        soma = soma + x         ! Acumula a soma
    end do
    close(10)

    print*, "Soma dos numeros = ", soma
end program arquivo_simples
```

Este exemplo ilustra de forma didática como abrir, escrever e ler arquivos em Fortran, utilizando blocos de escrita e leitura separados, tratamento de fim de arquivo e acumulação de resultados. Esses conceitos são fundamentais para programas que processam grandes volumes de dados ou armazenam resultados de simulações para análise posterior.

Estatística de dados guardados em um Arquivo

Em muitas aplicações de física e ciência de dados, é comum armazenar tabelas com várias colunas de resultados experimentais ou simulações. A manipulação dessas tabelas permite calcular estatísticas descritivas que caracterizam a distribuição dos dados. No exemplo a seguir, vamos considerar a leitura da segunda coluna de um arquivo chamado `dados.dat` e calcular as seguintes medidas:

- **Média:** $\bar{x} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N} x_i$, que indica o valor médio da coluna.
- **Média do quadrado:** $\overline{x^2} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N} x_i^2$, usada para cálculo de variância e outras estatísticas de ordem superior.
- **Curtose:** $\kappa = \dfrac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^4}{\left(\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2\right)^2} - 3$, que indica o achatamento da distribuição em relação à gaussiana.

- **Skewness (assimetria):** $\gamma = \dfrac{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^3}{\left(\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2\right)^{3/2}}$, que indica se a distribuição é simétrica ou inclinada para algum lado.

Considere o arquivo `dados.dat` com três colunas, por exemplo:

X	Y	Z
1.0	2.3	5.1
2.0	3.5	4.8
3.0	2.8	5.0
4.0	3.1	4.9
5.0	2.7	5.2

O código em Fortran a seguir realiza a leitura da segunda coluna e calcula as medidas estatísticas acima:

```
! Programa de estatistica: media, quadrado, curtose e skewness
program estatistica_coluna
    implicit none
    integer :: i, n, iostat_val
    real(8) :: x, soma, soma2, soma3, soma4
    real(8) :: media, media2, curtose, skewness
    real(8) :: y_dummy, z_dummy
    character(len=20) :: nome_arquivo
    integer :: unit_val

    nome_arquivo = 'dados.dat'
    n = 0
    soma = 0.0d0
    soma2 = 0.0d0
    soma3 = 0.0d0
    soma4 = 0.0d0
    unit_val = 10

    open(unit=unit_val, file=nome_arquivo, status='old', action='read')
    do
        read(unit_val,*,iostat=iostat_val) x, y_dummy, z_dummy
        if (iostat_val /= 0) exit
        n = n + 1
        soma = soma + x
        soma2 = soma2 + x 2
        soma3 = soma3 + x 3
        soma4 = soma4 + x 4
    end do
    close(unit_val)

    media = soma / n
    media2 = soma2 / n
    curtose = (soma4/n) / ((soma2/n) 2) - 3.0d0
    skewness = (soma3/n) / ((soma2/n) 1.5d0)

    print*, 'Numero de dados = ', n
    print*, 'Media = ', media
    print*, 'Media do quadrado = ', media2
```

```
    print*, 'Curtose = ', curtose
    print*, 'Skewness = ', skewness
end program estatistica_coluna
```

Neste programa:

- Abrimos o arquivo `dados.dat` e lemos cada linha, separando as três colunas;
- Acumulamos somatórios de x, x^2, x^3 e x^4 para calcular médias e estatísticas de ordem superior;
- Calculamos a média $\bar{x}$, a média do quadrado $\overline{x^2}$, a curtose κ e a skewness γ;
- Por fim, imprimimos os resultados.

Este procedimento é fundamental para análise estatística de dados experimentais, e pode ser facilmente adaptado para arquivos maiores, mais colunas ou diferentes estatísticas, permitindo um estudo sistemático da distribuição dos dados.

Contagem de Bases em Sequências de DNA

Em bioinformática e física estatística de sistemas biológicos, é comum lidar com grandes tabelas contendo sequências de bases de DNA, compostas pelas letras `A`, `T`, `C` e `G`. Um procedimento elementar, mas fundamental, é determinar a frequência de ocorrência de cada base — por exemplo, o número total de bases `G` em um conjunto de sequências.
Suponha que temos um arquivo texto (`sequencias.txt`) com várias linhas, cada uma representando uma sequência de DNA, como no exemplo abaixo:

```
CGTAGC
GGGTTT
AACGTA
TGCGGG
```

O programa em Fortran a seguir realiza a leitura de todas as linhas do arquivo, independentemente do número total de linhas, e conta quantas vezes a letra `G` aparece no conjunto completo de sequências. O código utiliza leitura sequencial até o fim do arquivo e compara caractere por caractere de cada linha.

```
! Contagem de bases 'G' em um arquivo de sequências de DNA
program conta_g
  implicit none
  character(len=200) :: linha
  integer :: i, total_g, ios, tam

  total_g = 0

  ! Abre o arquivo contendo as sequências
  open(unit=10, file='sequencias.txt', status='old', action='read')
```

```
  ! Lê cada linha até o fim do arquivo
  do
     read(10, '(A)', iostat=ios) linha
     if (ios /= 0) exit        ! Sai do loop ao atingir EOF
     tam = len_trim(linha)     ! Comprimento útil da linha

     ! Varre os caracteres e conta os 'G'
     do i = 1, tam
        if (linha(i:i) == 'G') total_g = total_g + 1
     end do
  end do

  close(10)

  print *, 'Número total de G =', total_g
end program conta_g
```

A lógica do programa é direta: cada linha é lida como uma *string* de caracteres, e o loop interno percorre a sequência para contar as ocorrências da base desejada. O comando `iostat` controla a leitura, interrompendo o laço quando o final do arquivo é atingido. Assim, o código é robusto mesmo quando o número de linhas não é conhecido previamente.
Esse tipo de procedimento é frequentemente utilizado em análises de composição de genomas, permitindo estimar proporções de bases e correlações entre sequências. Além disso, serve como uma introdução ao tratamento de dados textuais em Fortran — um passo importante antes da análise estatística de sequências biológicas ou simulações de modelos genéticos.

Busca de Motivo em Sequências de DNA

Um **motivo** (*motif*) é um pequeno padrão recorrente em uma sequência biológica, frequentemente associado a funções específicas, como promotores, sítios de ligação de proteínas ou regiões regulatórias. Detectar a presença de um motivo é uma das tarefas fundamentais da bioinformática.
O código a seguir implementa uma busca simples de um motivo (padrão) dentro de cada linha de um arquivo contendo sequências de DNA. O programa conta quantas vezes o motivo aparece em cada sequência, mesmo quando o número total de linhas é desconhecido.

```
! Busca de um motivo (padrão) em sequências de DNA
program busca_motivo
  implicit none
  character(len=200) :: linha
  character(len=20)  :: motivo
  integer :: ios, i, tam_seq, tam_mot, contagem, pos, nseq

  motivo = 'ATG'    ! Motivo a ser procurado
  tam_mot = len_trim(motivo)
  nseq = 0

  open(unit=10, file='sequencias.txt', status='old', action='read')

  do
```

```
      read(10, '(A)', iostat=ios) linha
      if (ios /= 0) exit
      tam_seq = len_trim(linha)
      contagem = 0

      ! Varre a sequência procurando o motivo
      do i = 1, tam_seq - tam_mot + 1
         if (linha(i:i+tam_mot-1) == motivo) contagem = contagem + 1
      end do

      nseq = nseq + 1
      print *, 'Sequência', nseq, ':', trim(linha)
      print *, 'Ocorrências de "', trim(motivo), '" =', contagem
   end do

   close(10)
end program busca_motivo
```

Neste exemplo, o programa lê cada sequência e realiza uma varredura caractere a caractere, comparando trechos de mesmo comprimento do motivo procurado. O método é uma implementação direta da chamada *busca de força bruta* (*brute-force search*) — suficiente para sequências curtas e clara do ponto de vista didático.
Versões mais sofisticadas desse algoritmo utilizam técnicas como o método de Knuth–Morris–Pratt (KMP) ou o algoritmo de Boyer–Moore, capazes de acelerar substancialmente a busca em genomas longos. Entretanto, o código acima captura a essência da análise de padrões biológicos de forma clara e acessível.

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 Fortran, 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
program mapa_logistico
    implicit none
    real :: r, x, r_min, r_max, r_step
    integer :: i, j, n_trans, n_iter
    character(len=30) :: nome_arquivo

    r_min = 2.5
    r_max = 4.0
    r_step = 0.01
    n_trans = 100
    n_iter = 50
    nome_arquivo = 'bifurcacao.txt'

    open(unit=10, file=nome_arquivo, status='replace', action='write')
```

```
    r = r_min
    do while (r <= r_max)
        x = 0.5

        ! Descarta transientes
        do i = 1, n_trans
            x = r * x * (1.0 - x)
        end do

        ! Salva próximos valores
        do i = 1, n_iter
            x = r * x * (1.0 - x)
            write(10,'(F6.3,1x,F6.3)') r, x
        end do

        r = r + r_step
    end do

    close(10)
end program mapa_logistico
```

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 Operações Básicas com Matrizes em Fortran

O tratamento de matrizes é fundamental em Física Computacional, tanto em métodos numéricos lineares quanto na simulação de sistemas físicos complexos. Nesta seção, veremos como o Fortran lida com operações básicas envolvendo matrizes e vetores (ou "matrizes coluna"), desde multiplicações simples até a exponenciação matricial aproximada.

Multiplicação de um Escalar por Matrizes

Multiplicar um escalar por uma matriz é uma das operações mais simples e serve como base para outras manipulações numéricas. Essa operação aparece em diversos contextos físicos e matemáticos, como na normalização de vetores, ajuste de unidades ou construção de operadores lineares. Em Fortran, a multiplicação escalar é feita elemento a elemento, o que permite controle total sobre o cálculo e facilita a compreensão do funcionamento interno das rotinas matriciais.

(a) Escalar por Matriz Coluna

No exemplo a seguir, multiplicamos um vetor (ou matriz coluna) por um número real a, produzindo um novo vetor y cujos elementos são simplesmente o produto de a por cada elemento de x:

$$y_i = a\, x_i$$

```
! Multiplicação de escalar por matriz coluna
program escalar_vetor
    implicit none
    integer, parameter :: n = 4
    real :: a
    real, dimension(n) :: x, y
    integer :: i

    a = 2.5
    x = (/ 1.0, 2.0, 3.0, 4.0 /)

    do i = 1, n
        y(i) = a * x(i)
    end do

    print *, "Vetor resultante:"
    print *, y
end program escalar_vetor
```

O programa define um vetor de quatro componentes e um escalar $a = 2.5$. Em seguida, cada elemento do vetor x é multiplicado por a, resultando em um novo vetor y. O laço `do` percorre todos os elementos, garantindo a operação elemento a elemento. No final, o resultado é exibido na tela, mostrando o vetor multiplicado pelo escalar.

(b) Escalar por Matriz Quadrada

A multiplicação de um escalar por uma matriz quadrada segue o mesmo princípio, sendo aplicada a cada elemento individualmente. Este tipo de operação é comum, por exemplo, na definição de operadores escalares em álgebra linear e em transformações lineares uniformes.

$$B_{ij} = a\, A_{ij}$$

O código abaixo ilustra como realizar essa operação em Fortran, utilizando uma matriz 3×3:

```
! Multiplicação de escalar por matriz quadrada
program escalar_matriz
    implicit none
    integer, parameter :: n = 3
    real :: a
    real, dimension(n,n) :: A, B
    integer :: i, j

    a = 0.5
    A = reshape([1.0, 2.0, 3.0, &
                 4.0, 5.0, 6.0, &
                 7.0, 8.0, 9.0], shape(A))

    do i = 1, n
        do j = 1, n
            B(i,j) = a * A(i,j)
```

```
        end do
    end do

    print *, "Matriz resultante:"
    print '(3F8.3)', B
end program escalar_matriz
```

Neste programa, a matriz A é criada manualmente com o comando `reshape`, que reorganiza os elementos em formato bidimensional. Cada elemento de A é então multiplicado pelo escalar $a = 0.5$, resultando na matriz B. Os dois laços aninhados percorrem todas as linhas e colunas, aplicando a operação de forma sistemática. O resultado impresso exibe a matriz reduzida à metade dos valores originais, confirmando a operação esperada.

Multiplicação de Matriz Quadrada por Matriz Coluna

A multiplicação de uma matriz quadrada por um vetor (ou matriz coluna) é uma das operações mais fundamentais da álgebra linear. Ela representa a aplicação de uma transformação linear A sobre o vetor x, resultando em um novo vetor y:

$$y_i = \sum_j A_{ij} x_j$$

Em Fortran, essa operação pode ser implementada de forma direta usando a função interna `matmul`, que executa automaticamente o produto matricial com a devida compatibilidade de dimensões.

```
! Multiplicação de matriz quadrada por matriz coluna
program matriz_por_vetor
    implicit none
    integer, parameter :: n = 3
    real, dimension(n,n) :: A
    real, dimension(n) :: x, y
    integer :: i, j

    ! Definindo a matriz e o vetor
    A = reshape([1.0, 2.0, 3.0, &
                 0.0, 1.0, 4.0, &
                 5.0, 6.0, 0.0], shape(A))
    x = (/ 1.0, 2.0, 3.0 /)

    !-----------------------------------------
    ! Produto usando a função matmul (F90+)
    !-----------------------------------------
    y = matmul(A, x)

    print *, "Resultado de A * x (usando matmul):"
    print '(3F10.3)', y

    !-----------------------------------------
    ! Produto manual de matriz por vetor (como em F77)
    !-----------------------------------------
    ! Em Fortran 77 não havia a função matmul, então
    ! fazemos a multiplicação usando dois loops aninhados:
```

```
    ! y_manual(i) = soma_j A(i,j)*x(j)
    ! Comentado aqui apenas para referência:

    ! real, dimension(n) :: y_manual
    ! y_manual = 0.0
    ! do i = 1, n
    !     do j = 1, n
    !         y_manual(i) = y_manual(i) + A(i,j)*x(j)
    !     end do
    ! end do
    ! print *, "Resultado de A * x (manual, F77-style):"
    ! print '(3F10.3)', y_manual

end program matriz_por_vetor
```

Neste exemplo, a matriz A de ordem 3 é multiplicada pelo vetor x também de dimensão 3. O comando `matmul(A, x)` realiza automaticamente o somatório implícito da expressão matemática do produto. O resultado é impresso na tela como um novo vetor y, que representa a transformação de x pela matriz A.

Multiplicação de Matrizes Quadradas

A multiplicação de duas matrizes quadradas é uma generalização natural da operação anterior. Ela é usada em inúmeros contextos — da composição de operadores lineares à evolução temporal em sistemas físicos discretos. O produto matricial é definido como:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Em Fortran, novamente podemos utilizar a função `matmul`, que realiza o produto completo de forma eficiente.

```
! Multiplicação de duas matrizes quadradas
program matriz_por_matriz
    implicit none
    integer, parameter :: n = 3
    real, dimension(n,n) :: A, B, C
    integer :: i, j, k

    ! Definindo as matrizes
    A = reshape([1.0, 2.0, 3.0, &
                 0.0, 1.0, 4.0, &
                 5.0, 6.0, 0.0], shape(A))

    B = reshape([1.0, 0.0, 2.0, &
                 0.0, 1.0, 3.0, &
                 4.0, 0.0, 1.0], shape(B))

    !-----------------------------------------
    ! Produto usando a função matmul (F90+)
    !-----------------------------------------
    C = matmul(A, B)
```

```
    print *, "Produto A * B (usando matmul):"
    print '(3F10.3)', C

    !-------------------------------------------
    ! Produto manual de matrizes (como em F77)
    !-------------------------------------------
    ! Em Fortran 77 não havia a função matmul, então
    ! fazemos a multiplicação usando três loops aninhados:
    ! C_manual(i,j) = soma_k A(i,k)*B(k,j)
    ! Comentado aqui apenas para referência:

    ! real, dimension(n,n) :: C_manual
    ! C_manual = 0.0
    ! do i = 1, n
    !     do j = 1, n
    !         do k = 1, n
    !             C_manual(i,j) = C_manual(i,j) + A(i,k)*B(k,j)
    !         end do
    !     end do
    ! end do
    ! print *, "Produto A * B (manual, F77-style):"
    ! print '(3F10.3)', C_manual

end program matriz_por_matriz
```

A função `reshape` é novamente usada para definir as matrizes de entrada. O produto `C = matmul(A, B)` realiza o cálculo elemento a elemento de acordo com a definição matemática acima. A saída impressa exibe a matriz resultante C, produto das duas matrizes iniciais.

Traço e Normalização de Matrizes

O **traço** e a **normalização** são operações importantes em física e matemática aplicada. O traço mede a soma dos elementos da diagonal principal, e aparece, por exemplo, no cálculo de médias quânticas e invariantes de operadores. A normalização, por sua vez, é essencial em muitos contextos numéricos, garantindo que vetores e matrizes tenham magnitude controlada.

$$\mathrm{Tr}(A) = \sum_i A_{ii}$$

Já a normalização consiste em dividir todos os elementos de um vetor ou matriz por sua norma (geralmente euclidiana), isto é, a raiz quadrada da soma dos quadrados de seus elementos.

(a) Traço de uma Matriz

O programa abaixo calcula o traço de uma matriz quadrada percorrendo apenas os elementos da diagonal principal.

```
! Cálculo do traço de uma matriz
program traco_matriz
```

```
    implicit none
    integer, parameter :: n = 3
    real, dimension(n,n) :: A
    real :: traco
    integer :: i

    A = reshape([1.0, 2.0, 3.0, &
                 0.0, 1.0, 4.0, &
                 5.0, 6.0, 2.0], shape(A))

    traco = 0.0
    do i = 1, n
        traco = traco + A(i,i)
    end do

    print *, "Traço da matriz =", traco
end program traco_matriz
```

O laço `do` percorre as posições diagonais da matriz (onde $i = j$) e acumula a soma desses valores. O resultado final, exibido na tela, representa o traço $\mathrm{Tr}(A)$.

(b) Normalização de Vetor e Matriz

O código a seguir mostra como normalizar tanto um vetor quanto uma matriz. A normalização é feita dividindo cada elemento pelo módulo total, calculado como a raiz quadrada da soma dos quadrados.

```
! Normalização de vetor e matriz
program normalizacao
    implicit none
    integer, parameter :: n = 3
    real, dimension(n) :: v
    real, dimension(n,n) :: M
    real :: norma
    integer :: i, j

    !-----------------------------------------
    ! Normalização usando sum (F90+)
    !-----------------------------------------
    v = (/ 1.0, 2.0, 3.0 /)
    norma = sqrt(sum(v**2))
    v = v / norma
    print *, "Vetor normalizado:"
    print '(3F8.4)', v

    M = reshape([1.0, 2.0, 3.0, &
                 4.0, 5.0, 6.0, &
                 7.0, 8.0, 9.0], shape(M))
    norma = sqrt(sum(M**2))
    M = M / norma
    print *, "Matriz normalizada:"
    print '(3F8.4)', M

    !-----------------------------------------
    ! Normalização manual (em F77)
    !-----------------------------------------
```

```
    ! Em Fortran 77 não havia a função sum, então calculamos a norma manualmente:
    ! Para vetor:
    ! norma_v = 0.0
    ! do i = 1, n
    !     norma_v = norma_v + v(i)**2
    ! end do
    ! norma_v = sqrt(norma_v)
    ! do i = 1, n
    !     v(i) = v(i) / norma_v
    ! end do
    !
    ! Para matriz:
    ! norma_M = 0.0
    ! do i = 1, n
    !     do j = 1, n
    !         norma_M = norma_M + M(i,j)**2
    !     end do
    ! end do
    ! norma_M = sqrt(norma_M)
    ! do i = 1, n
    !     do j = 1, n
    !         M(i,j) = M(i,j) / norma_M
    !     end do
    ! end do

end program normalizacao
```

A função `sum` soma os quadrados dos elementos, e `sqrt` calcula a raiz quadrada para obter a norma euclidiana. A operação de divisão é feita de forma vetorizada: cada elemento é dividido pela norma total. Isso produz um vetor e uma matriz com módulo unitário, úteis em simulações numéricas e operações de álgebra linear normalizadas.

Exponencial de uma Matriz

A exponencial de uma matriz A pode ser definida pela série de Taylor:

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \cdots$$

O exemplo a seguir mostra uma aproximação truncada dessa série.

```
program mat_exp_taylor
  implicit none
  integer, parameter :: dp = kind(1.0d0)
  integer, parameter :: N = 3, NTERM = 10
  integer :: i, j, k, n2
  real(dp) :: fact
  real(dp), dimension(N,N) :: A, M, term, expA, Id

  !-------------------------------------------------------
  ! Exemplo: matriz A
  !-------------------------------------------------------
  A = reshape([0.0d0, 1.0d0, 0.0d0, &
               -1.0d0, 0.0d0, 0.0d0, &
                0.0d0, 0.0d0, 0.5d0], shape(A))
```

```
  ! Matriz identidade
  Id = 0.0d0
  do i = 1, N
     Id(i,i) = 1.0d0
  end do

  expA = Id
  M = Id     ! termo corrente da série
  fact = 1.0d0

  !------------------------------------------------------
  ! Série de Taylor: exp(A)=\sum A^n / n!
  !------------------------------------------------------
  do n2 = 1, NTERM
     M = matmul(M, A)
     fact = fact * real(n2, dp)
     term = M / fact
     expA = expA + term
  end do

  !------------------------------------------------------
  ! Resultado
  !------------------------------------------------------
  print *, 'Exponencial aproximada da matriz A (NTERM=', NTERM, '):'
  do i = 1, N
     write(*,'(3F12.6)') expA(i,1:N)
  end do

end program mat_exp_taylor
```

Este código usa uma aproximação de ordem N_{TERM} para a série de Taylor, suficiente para matrizes com norma pequena. Para aplicações mais avançadas, recomenda-se o uso de aproximações de Padé e técnicas de "scaling and squaring", vistas em seções posteriores.

4 Números Complexos em Fortran

Fortran possui suporte nativo a números complexos através do tipo `complex`. Isso permite realizar operações matemáticas diretamente sobre números da forma

$$z = a + ib,$$

onde a é a parte real e b a parte imaginária. O Fortran oferece funções integradas para calcular módulo (`abs`), conjugado (`conjg`), exponencial (`cexp`), entre outras operações úteis em física, engenharia e matemática aplicada.
O exemplo a seguir demonstra a manipulação básica de números complexos: definição de variáveis, soma, produto, conjugado, módulo e exponencial. É uma forma direta de visualizar operações complexas sem necessidade de escrever funções adicionais.

```
program complexos
    implicit none
    ! Declaração de variáveis complexas
```

```
    complex :: z1, z2, soma, produto, conj_z1, exp_z1
    real :: modulo_z1

    ! Inicialização de números complexos
    z1 = (2.0, 3.0)       ! z1 = 2 + 3i
    z2 = (1.0, -4.0)      ! z2 = 1 - 4i

    ! Operações básicas
    soma = z1 + z2
    produto = z1 * z2
    conj_z1 = conjg(z1)   ! Conjugado de z1
    modulo_z1 = abs(z1)   ! Módulo de z1
    exp_z1 = cexp(z1)     ! Exponencial de z1

    ! Impressão dos resultados
    print*, "z1 = ", z1
    print*, "z2 = ", z2
    print*, "Soma = ", soma
    print*, "Produto = ", produto
    print*, "Conjugado de z1 = ", conj_z1
    print*, "|z1| = ", modulo_z1
    print*, "Exp(z1) = ", exp_z1
end program complexos
```

Este código demonstra como realizar operações matemáticas comuns com números complexos de forma simples e direta em Fortran. Ele mostra a praticidade do tipo `complex` e das funções nativas, permitindo trabalhar com cálculos complexos sem necessidade de decompor os números em partes reais e imaginárias manualmente. Essa abordagem é especialmente útil em problemas de análise de sinais, física quântica e transformadas de Fourier.

5 Derivada Numérica

A derivada de uma função $f(x)$ pode ser aproximada numericamente utilizando diferentes esquemas de diferenças finitas. A escolha do método depende da precisão desejada, do tipo de dados disponíveis e do contexto da aplicação. Métodos simples, como a diferença progressiva, são fáceis de implementar, enquanto esquemas mais sofisticados, como a diferença central ou derivadas a partir de vetores e arquivos de dados, oferecem maior precisão e flexibilidade.

Diferença Progressiva

O método de diferença progressiva estima a derivada usando o valor da função no ponto x e no ponto seguinte $x + h$:

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

Este método possui erro truncamento de ordem $O(h)$, sendo adequado para casos em que apenas o ponto atual e o seguinte estão disponíveis.

```
! Derivada numérica: diferença progressiva
program deriv_progressiva
    implicit none
```

```
    real :: x, h, deriv

    ! Valor do ponto e passo
    x = 1.0
    h = 0.001

    ! Cálculo da derivada usando diferença progressiva
    deriv = (sin(x+h) - sin(x)) / h

    ! Resultado
    print*, "Derivada aproximada em x=", x, " : ", deriv
end program deriv_progressiva
```

O programa acima demonstra de forma direta como calcular a derivada aproximada em um ponto específico usando apenas o valor seguinte da função. É ideal para introdução à derivada numérica ou situações em que os dados são fornecidos sequencialmente.

Diferença Central

A diferença central utiliza pontos à direita e à esquerda de x:

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

Este método apresenta erro truncamento de ordem $O(h^2)$, oferecendo maior precisão em comparação com a diferença progressiva, desde que seja possível avaliar a função em ambos os lados do ponto de interesse.

```
! Derivada numérica: diferença central
program deriv_central
    implicit none
    real :: x, h, deriv

    ! Valor do ponto e passo
    x = 1.0
    h = 0.001

    ! Cálculo da derivada usando diferença central
    deriv = (sin(x+h) - sin(x-h)) / (2.0*h)

    ! Resultado
    print*, "Derivada central em x=", x, " : ", deriv
end program deriv_central
```

Este exemplo ilustra como a diferença central melhora a precisão ao considerar informações simétricas ao redor do ponto de interesse, sendo uma escolha frequente em aplicações científicas.

Derivada a partir de vetor de dados

Quando os valores de uma função são armazenados em vetores discretos, a derivada pode ser calculada usando diferenças centrais nos pontos internos do vetor:

$$f'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{x_{i+1} - x_{i-1}}, \quad 2 \leq i \leq n-1.$$

```
! Derivada de dados tabelados
program deriv_vetor
    implicit none
    real, dimension(5) :: x = (/0,1,2,3,4/)
    real, dimension(5) :: y = (/0,1,4,9,16/)
    integer :: i
    real :: deriv

    ! Calcula derivadas usando diferença central
    do i = 2, 4
        deriv = (y(i+1) - y(i-1)) / (x(i+1) - x(i-1))
        print*, "x=", x(i), " -> derivada ~ ", deriv
    end do
end program deriv_vetor
```

Este código mostra como calcular derivadas aproximadas para conjuntos discretos de dados, permitindo analisar tendências e taxas de variação em séries discretas ou simuladas.

Derivada a partir de arquivo de dados

Para dados experimentais ou de simulações armazenados em arquivos, podemos ler os valores e aplicar o mesmo esquema de diferenças centrais para obter a derivada. Este método automatiza o processamento de grandes volumes de dados.

```
! Derivada de dados lidos de arquivo
program deriv_arquivo
    implicit none
    real, dimension(100) :: x, y
    integer :: i, n, iostat
    character(len=20) :: arquivo

    arquivo = 'tabela.dat'
    n = 0

    ! Leitura do arquivo
    open(unit=10, file=arquivo, status='old', action='read')
    do
        read(10,*,iostat=iostat) x(n+1), y(n+1)
        if (iostat /= 0) exit
        n = n + 1
    end do
    close(10)

    ! Cálculo da derivada por diferença central
    do i = 2, n-1
        print*, "x=", x(i), " -> derivada ~ ", (y(i+1)-y(i-1))/(x(i+1)-x(i-1))
    end do
end program deriv_arquivo
```

Esses exemplos cobrem métodos básicos de derivadas numéricas em Fortran, desde a análise de funções conhecidas até o processamento de vetores e arquivos de dados. Eles fornecem ferramentas práticas para estimar taxas de variação em experimentos, simulações e análises de séries temporais, sendo fundamentais em física computacional, engenharia e ciência de dados.

6 Integração Numérica

A integração numérica é uma ferramenta essencial para aproximar integrais de funções quando uma solução analítica não é viável. Em Fortran, podemos implementar métodos simples como a regra dos retângulos, do trapézio, integrais a partir de dados tabulados e integrais múltiplas.

Regra dos Retângulos

O método dos retângulos consiste em dividir o intervalo de integração em N subintervalos iguais e aproximar a função por sua avaliação no ponto esquerdo de cada subintervalo:

$$\int_a^b f(x)\,dx \approx h \sum_{i=0}^{N-1} f(a+ih), \quad h = \frac{b-a}{N}.$$

Este é um método simples, mas com precisão limitada ($O(h)$).

```
! Integração numérica usando retângulos
program integracao_retangulos
    implicit none
    real :: a, b, h, soma, integral
    integer :: N, i

    ! Intervalo de integração e número de pontos
    a = 0.0
    b = 3.14159265
    N = 100
    h = (b-a)/N
    soma = 0.0

    ! Soma dos valores da função nos pontos do subintervalo
    do i = 0, N-1
        soma = soma + sin(a + i*h)
    end do

    ! Multiplica pelo passo para obter integral aproximada
    integral = h * soma
    print*, "Integral aproximada (retangulos) = ", integral
end program integracao_retangulos
```

O programa acima demonstra a integração numérica da função $\sin(x)$ no intervalo $[0, \pi]$ utilizando a regra dos retângulos. A cada passo, o valor da função é avaliado no início de cada subintervalo e somado à soma acumulada. Ao final, a soma é multiplicada pelo tamanho

do passo (h) para obter a integral aproximada. Embora simples, este método fornece uma estimativa razoável da área sob a curva, sendo especialmente útil para introdução à integração numérica e para funções suaves, embora sua precisão seja inferior à da regra do trapézio ou métodos de maior ordem.

Regra do Trapézio

A regra do trapézio melhora a precisão usando a média dos valores da função nos extremos de cada subintervalo:

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

O erro é menor que o dos retângulos, da ordem $O(h^2)$, mantendo a implementação simples.

```
! Integração numérica usando trapézio
program integracao_trapezio
    implicit none
    real :: a, b, h, soma, integral
    integer :: N, i

    ! Intervalo e número de divisões
    a = 0.0
    b = 3.14159265
    N = 100
    h = (b-a)/N
    soma = 0.0

    ! Soma dos valores da função nos pontos internos
    do i = 1, N-1
        soma = soma + sin(a + i*h)
    end do

    ! Integral aproximada usando extremos
    integral = h * ( (sin(a)+sin(b))/2.0 + soma )
    print*, "Integral aproximada (trapezio) = ", integral
end program integracao_trapezio
```

O código acima implementa a integração numérica da função $\sin(x)$ no intervalo $[0, \pi]$ utilizando a regra do trapézio. A cada iteração, somamos os valores da função nos pontos internos do intervalo e, ao final, adicionamos metade das contribuições dos extremos para calcular a integral aproximada. Esse método proporciona maior precisão em comparação à regra dos retângulos, mantendo a implementação simples e eficiente. A abordagem é especialmente adequada para funções contínuas e suaves, permitindo estimativas confiáveis da área sob a curva mesmo com um número moderado de subdivisões.

Integração a partir de Arquivo de Dados

Quando temos dados experimentais ou de simulações armazenados em arquivo, podemos calcular a integral aproximada utilizando a regra do trapézio aplicada aos pontos tabulados.

O arquivo `tabela2.dat` deve conter duas colunas: x e $y = f(x)$.

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
program integra_arquivo
    implicit none
    integer :: i, n, iostat
    double precision :: soma
    double precision, dimension(100) :: x, y
    character(len=100) :: arquivo

    arquivo = "tabela2.dat"
    n = 0

    ! Leitura dos dados do arquivo
    open(unit=10, file=arquivo, status="old", action="read")
    do
        read(10,*,iostat=iostat) x(n+1), y(n+1)
        if (iostat /= 0) exit
        n = n + 1
    end do
    close(10)

    ! Aplicação da regra do trapézio
    soma = 0.0
    do i = 1, n-1
        soma = soma + 0.5*(y(i)+y(i+1))*(x(i+1)-x(i))
    end do

    print*, "Integral aproximada:", soma
end program integra_arquivo
```

O programa acima demonstra como realizar a integração numérica de dados discretos armazenados em um arquivo utilizando a regra do trapézio. Primeiramente, os valores de (x) e (y=f(x)) são lidos do arquivo `tabela2.dat` e armazenados em vetores. Em seguida, a integral aproximada é calculada somando a área de cada trapézio formado pelos pontos consecutivos, multiplicando a média das alturas pelo comprimento do intervalo. Este método é especialmente útil para analisar dados experimentais ou resultados de simulações, permitindo estimar a integral sem conhecimento explícito da função subjacente. A abordagem é direta, eficiente e facilmente adaptável a arquivos maiores ou a funções de múltiplas variáveis.

Integral Dupla

Para funções de duas variáveis, a integral pode ser aproximada por uma soma dupla sobre uma grade regular:

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

```
! Integral dupla simples: f(x,y) = x*y
program integral_dupla
    implicit none
    integer :: i, j, Nx, Ny
    double precision :: ax, bx, ay, by, hx, hy, soma, x, y

    Nx = 50
    Ny = 50
    ax = 0.0; bx = 1.0
    ay = 0.0; by = 1.0
    hx = (bx-ax)/Nx
    hy = (by-ay)/Ny
    soma = 0.0

    ! Soma dupla sobre a grade de pontos
    do i = 0, Nx-1
        x = ax + i*hx
        do j = 0, Ny-1
            y = ay + j*hy
            soma = soma + x*y
        end do
    end do

    print*, "Integral dupla ~", soma*hx*hy
end program integral_dupla
```

Esses exemplos ilustram como implementar integrais numéricas simples em Fortran, desde métodos básicos para funções contínuas até integrais baseadas em dados experimentais ou integrais duplas em duas variáveis. A abordagem é direta e facilmente adaptável para problemas mais complexos.

7 Introdução aos Números Aleatórios

Computadores não geram números verdadeiramente aleatórios, mas sim números **pseudo-aleatórios**, que simulam propriedades estatísticas de números aleatórios. Em Fortran, a função intrínseca `random_number()` permite gerar números reais uniformemente distribuídos no intervalo $[0, 1]$.

```
! Números pseudoaleatórios
program numeros_aleatorios
  implicit none
  integer :: i
  real :: r

  ! Gera 5 números aleatórios uniformes em [0,1]
```

```
  do i = 1, 5
     call random_number(r)
     print*, r
  end do
end program numeros_aleatorios
```

Este programa simples demonstra a geração de números pseudoaleatórios. Cada execução gera uma sequência diferente, desde que a semente interna do gerador seja alterada. Este método é útil para simulações rápidas, testes e experimentos numéricos.

Histograma de Números Aleatórios

Para analisar a distribuição de números aleatórios, podemos construir um histograma. No exemplo abaixo, dividimos o intervalo $[0, 1]$ em 10 bins e contamos quantos números caem em cada intervalo.

```
! Histograma de números aleatórios uniformes
program histograma
  implicit none
  integer :: i, k, N, bins
  real :: u
  integer, dimension(10) :: hist = 0

  N = 1000     ! número de amostras
  bins = 10    ! número de intervalos

  ! Geração de números aleatórios e contagem nos bins
  do i = 1, N
     call random_number(u)
     k = int(u*bins) + 1
     if (k > bins) k = bins
     hist(k) = hist(k) + 1
  end do

  ! Exibição do histograma
  do i = 1, bins
     print*, "Bin", i, ":", hist(i)
  end do
end program histograma
```

O histograma permite visualizar a uniformidade da distribuição gerada. Este conceito é fundamental em análises estatísticas e em simulações de Monte Carlo, fornecendo uma maneira simples de verificar a qualidade do gerador de números aleatórios.

Distribuição $P(x) \sim x^n$

Para gerar números com distribuições diferentes da uniforme, aplicamos transformações sobre números uniformes. Por exemplo, para $P(x) \sim x^n$, usamos a relação $x = u^{1/(n+1)}$, onde u é uniforme em $[0, 1]$.

```
! Números com P(x) ~ x^n e histograma
program distrib_xn
```

```
  implicit none
  integer :: i, N, bins, n, k
  real :: u, x
  integer, dimension(10) :: hist = 0

  N = 1000
  n = 2
  bins = 10

  open(unit=10, file="numeros.dat", status="replace")
  do i = 1, N
     call random_number(u)
     x = u  (1.0/(n+1))   ! transformação para P(x) ~ x^n
     write(10,*) x

     k = int(x*bins) + 1
     if (k > bins) k = bins
     hist(k) = hist(k) + 1
  end do
  close(10)

  open(unit=11, file="histograma.dat", status="replace")
  do i = 1, bins
     write(11,*) i, hist(i)
  end do
  close(11)
end program distrib_xn
```

Após executar este programa, dois arquivos são gerados:

- `numeros.dat`: lista de números pseudoaleatórios distribuídos conforme $P(x) \sim x^n$.
- `histograma.dat`: contagem de números em cada bin, útil para plotar a distribuição.

Esse procedimento mostra como transformar números uniformes em qualquer distribuição desejada e como construir histogramas simples, sendo uma técnica básica mas poderosa em simulações de Monte Carlo.

Geradores Lineares Congruentes

Um dos métodos clássicos para gerar números pseudoaleatórios é o **gerador linear congruente** (LCG), definido pela recorrência:

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

A sequência gerada é então normalizada em $[0, 1]$ para uso em simulações e experimentos numéricos. Segue um exemplo em Fortran:

```
! Gerador Linear Congruente (LCG)
program lcg
  implicit none
  integer :: i, N
  integer(kind=8) :: a, c, m, x
```

```
  real :: u
  integer, parameter :: dp = selected_real_kind(15,307)

  N = 1000          ! número de números aleatórios
  a = 1664525
  c = 1013904223
  m = 2147483648
  x = 12345         ! semente inicial

  open(unit=10, file="lcg.dat")
  do i = 1, N
     x = mod(a*x + c, m)
     u = real(x, kind=dp)/real(m, kind=dp)
     write(10,*) u
  end do
  close(10)
end program lcg
```

O gerador linear congruente é eficiente e simples de implementar, mas deve ser usado com cuidado, pois apresenta correlações de longo período e limitações de qualidade em algumas aplicações. Para análises estatísticas mais rigorosas, é recomendável combinar o LCG com outras técnicas ou utilizar geradores de qualidade superior disponíveis em bibliotecas modernas.

Estes exemplos fornecem uma base sólida para trabalhar com números pseudoaleatórios em Fortran, preparar histogramas, gerar distribuições específicas e implementar geradores clássicos, sendo essenciais para simulações computacionais, testes estatísticos e experimentos numéricos.

Distribuição Gaussiana (Box-Muller)

A distribuição normal ou gaussiana é uma das mais importantes em estatística. O método **Box-Muller** permite gerar números gaussianos a partir de números uniformes $u_1, u_2 \in [0, 1]$. As fórmulas são:

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

O código abaixo gera N números gaussianos e constrói um histograma simples:

```
program box_muller
  implicit none
  integer :: i, N, bins, k
  real(kind=8) :: u1, u2, z
  integer :: hist(20)
  N = 1000
  bins = 20
  hist = 0

  open(unit=10, file="gauss.dat")
  do i = 1, N
     call random_number(u1)
     call random_number(u2)
```

```
    z = sqrt(-2.0d0*log(u1)) * cos(2.0d0*acos(-1.0d0)*u2)
    write(10,*) z

    k = int((z+4.0d0)/8.0d0*bins)
    if(k<0) k=0
    if(k>=bins) k=bins-1
    hist(k+1) = hist(k+1) + 1
  end do
  close(10)

  open(unit=11, file="histograma_gauss.dat")
  do k = 1, bins
    write(11,*) k, hist(k)
  end do
  close(11)
end program box_muller
```

Este programa cria dois arquivos: `gauss.dat`, com os valores gaussianos gerados, e `histograma_gauss.dat` que contém a contagem de valores em cada bin. O Box-Muller é útil em simulações de Monte Carlo e modelagem de fenômenos com ruído gaussiano.

Distribuição Lorentziana

A distribuição de Lorentz (ou Cauchy) apresenta caudas mais pesadas que a gaussiana. Podemos gerar números Lorentzianos a partir de números uniformes $u \in (0, 1)$ usando:

$$x = \tan\left[\pi(u - 0.5)\right]$$

O código abaixo gera números Lorentzianos e constrói um histograma:

```
program lorentz
  implicit none
  integer :: i, N, bins, k
  real(kind=8) :: u, x, x_min, x_max, dx
  real(kind=8) :: hist(20)
  N = 1000
  bins = 20
  hist = 0
  x_min = -10.0d0
  x_max = 10.0d0
  dx = (x_max - x_min)/bins

  open(unit=10, file="lorentz.dat")
  do i = 1, N
    call random_number(u)
    x = tan(acos(-1.0d0)*(u-0.5d0))
    write(10,*) x

    if(x>=x_min .and. x<x_max) then
      k = int((x - x_min)/dx)
      hist(k+1) = hist(k+1) + 1
    end if
  end do
  close(10)
```

```
  open(unit=11, file="hist_lorentz.dat")
  do k = 1, bins
     write(11,*) x_min + (k-0.5d0)*dx, hist(k)
  end do
  close(11)
end program lorentz
```

O arquivo `lorentz.dat` contém os números gerados, enquanto `hist_lorentz.dat` mostra o histograma. A distribuição Lorentziana é útil para estudar fenômenos com grande variabilidade e caudas longas, como certas distribuições financeiras ou espectros de ressonância.

Distribuição de Potência $P(f) \sim f^{-B}$

Distribuições de potência são comuns em fenômenos com grande variabilidade, como terremotos, flutuações financeiras e redes complexas. Podemos gerar números com distribuição

$$P(f) \sim f^{-B}, \quad f \in [f_{\min}, f_{\max}], \quad B > 0$$

a partir de números uniformes $u \in [0, 1]$ usando a transformação inversa:

$$f = \left[f_{\min}^{1-B} + u\big(f_{\max}^{1-B} - f_{\min}^{1-B}\big) \right]^{\frac{1}{1-B}}$$

O código abaixo gera N números com distribuição de potência e constrói um histograma:

```
program power_law
  implicit none
  integer :: i, N, bins, k
  real(kind=8) :: u, f, f_min, f_max, B, df
  real(kind=8) :: hist(20)
  N = 1000
  bins = 20
  hist = 0
  f_min = 1.0d0
  f_max = 100.0d0
  B = 2.5d0
  df = (f_max - f_min)/bins

  open(unit=10, file="power_law.dat")
  do i = 1, N
     call random_number(u)
     f = (f_min  (1.0d0-B) + u*(f_max  (1.0d0-B) - f_min  (1.0d0-B)))  (1.0d0/(1.0d0-B))
     write(10,*) f

     if(f>=f_min .and. f<f_max) then
        k = int((f - f_min)/df)
        if(k>=bins) k=bins-1
        hist(k+1) = hist(k+1) + 1
     end if
  end do
  close(10)

  open(unit=11, file="hist_power_law.dat")
  do k = 1, bins
```

```
    write(11,*) f_min + (k-0.5d0)*df, hist(k)
  end do
  close(11)
end program power_law
```

O arquivo `power_law.dat` contém os números gerados segundo a distribuição f^{-B}, enquanto `hist_power_law.dat` apresenta o histograma. Esse tipo de distribuição é útil para estudar fenômenos com grande variabilidade e escalas sem característica típica, onde eventos raros podem ter grande impacto.

Autocorrelação Simples

A autocorrelação mede a dependência entre elementos de uma série ao longo de diferentes intervalos de lag k:

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

O código abaixo gera uma série de números aleatórios e calcula a autocorrelação para lags até um valor máximo:

```
program autocorrel
  implicit none
  integer, parameter :: N=2000, MAX_LAG=50
  real(kind=8) :: x(N), media, C(0:MAX_LAG)
  integer :: i, k

  ! Geração da série e cálculo da média
  media = 0.0d0
  do i = 1, N
     call random_number(x(i))
     media = media + x(i)
  end do
  media = media / N

  ! Cálculo da autocorrelação
  do k = 0, MAX_LAG
     C(k) = 0.0d0
     do i = 1, N-k
        C(k) = C(k) + (x(i)-media)*(x(i+k)-media)
     end do
     C(k) = C(k)/(N-k)
     print *, 'Lag', k, ': C =', C(k)
  end do
end program autocorrel
```

A autocorrelação para lag 0 corresponde à variância da série. Lags maiores revelam a presença de dependências ou correlações em distâncias temporais. Esta análise é amplamente utilizada em séries temporais, física estatística e simulações de Monte Carlo para avaliar a independência dos números gerados.

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 deslocamento médio quadrático cresce linearmente com o número de passos:

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

```
      PROGRAM RANDOM_WALK
      INTEGER Npassos, i, pos, r
      REAL x2_acum
      Npassos = 1000
      pos = 0
      x2_acum = 0.0

      OPEN(10, FILE='x2.dat')
      DO 10 i = 1, Npassos
         r = MOD(INT(RANDOM()*2),2)
         IF (r .EQ. 0) THEN
            pos = pos - 1
         ELSE
            pos = pos + 1
         ENDIF
         x2_acum = x2_acum + pos*pos
         WRITE(10,*) i, x2_acum/i
10    CONTINUE
      CLOSE(10)
      END
```

O código acima implementa um modelo simples de **caminhante aleatório clássico em uma dimensão** utilizando Fortran. Cada passo do caminhante consiste em mover-se para a esquerda ou para a direita com probabilidade igual, simulando um processo de difusão discreta. O algoritmo acompanha a posição atual do caminhante (`pos`) e calcula o **deslocamento médio quadrático** $\langle x^2 \rangle$, que é a média do quadrado da posição ao longo do tempo. O deslocamento médio quadrático é atualizado a cada passo e armazenado na variável `x2_acum`. Dividindo-se pela quantidade de passos já realizados (`i`), obtemos $\langle x^2 \rangle$ para aquele instante, que é então gravado no arquivo `x2.dat` para posterior análise ou plotagem. Esse procedimento permite observar que, para um caminhar aleatório simétrico, o crescimento de $\langle x^2 \rangle$ é linear com o número de passos (N), confirmando a relação teórica $\langle x^2 \rangle = N \cdot (\text{passo})^2$. O uso da função `RANDOM()` gera números aleatórios uniformemente distribuídos no intervalo $[0, 1]$. Multiplicando por 2 e aplicando `INT()` seguido de `MOD(...,2)`, convertemos o valor contínuo em um inteiro 0 ou 1, determinando a direção do passo. O bloco condicional `IF` atualiza a posição de acordo com o resultado aleatório. Embora o código seja relativamente simples, ele exemplifica conceitos centrais de física estatística e mecânica estatística: processos estocásticos, difusão e crescimento do deslocamento médio quadrático. Além disso, ilustra aspectos importantes de programação em Fortran: uso de loops `DO`, manipulação de variáveis inteiras e reais, escrita em arquivos e acumulação de estatísticas. O modelo pode ser facilmente expandido: simulações em 2D ou 3D, caminhantes múltiplos, ou inclusão de

passos assimétricos e probabilidades variadas. Tais extensões permitem explorar fenômenos de difusão anômala, espalhamento aleatório em redes ou propriedades de transporte em materiais desordenados. Em resumo, este programa fornece uma ferramenta didática para visualizar e quantificar o comportamento médio de sistemas estocásticos, consolidando a relação entre simulação computacional e previsões teóricas da física de processos aleatórios.

8 Regressão Linear e Ajuste de Dados

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

$$y = a + bx,$$

onde a é o intercepto e b é a inclinação da reta. Os coeficientes podem ser 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 programa a seguir implementa a regressão linear em Fortran para um conjunto simples de dados:

```
      PROGRAM REGRESSAO_LINEAR
      IMPLICIT NONE
      INTEGER N, I
      PARAMETER (N=5)
      REAL X(N), Y(N), MEDIA_X, MEDIA_Y, SOMA_X, SOMA_Y
      REAL NUM, DEN, A, B

      DATA X /1.0, 2.0, 3.0, 4.0, 5.0/
      DATA Y /2.0, 4.0, 5.0, 4.0, 5.0/

      ! Calcula médias de X e Y
      SOMA_X = 0.0
      SOMA_Y = 0.0
      DO 10 I = 1, N
         SOMA_X = SOMA_X + X(I)
         SOMA_Y = SOMA_Y + Y(I)
10    CONTINUE
      MEDIA_X = SOMA_X / N
      MEDIA_Y = SOMA_Y / N

      ! Cálculo de B e A
      NUM = 0.0
      DEN = 0.0
      DO 20 I = 1, N
         NUM = NUM + (X(I)-MEDIA_X)*(Y(I)-MEDIA_Y)
         DEN = DEN + (X(I)-MEDIA_X)  2
20    CONTINUE

      B = NUM / DEN
      A = MEDIA_Y - B*MEDIA_X
```

```
      PRINT *, 'Ajuste linear: Y = ', A, ' + ', B, '* X'

      END
```

Neste programa, criamos arrays para armazenar os dados (x_i, y_i), calculamos as médias, e depois aplicamos a fórmula da regressão linear. O resultado é a reta que melhor representa os dados segundo o critério dos mínimos quadrados.

Ajuste de Lei de Potência

Muitas vezes os dados seguem uma relação de potência do tipo

$$y = Cx^n.$$

Tomando o logaritmo natural, transformamos o problema em regressão linear:

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

permitindo usar as mesmas fórmulas da regressão linear para determinar n e C.
O código Fortran abaixo realiza esse ajuste:

```
      PROGRAM LEI_POTENCIA
      IMPLICIT NONE
      INTEGER N, I
      PARAMETER (N=6)
      REAL X(N), Y(N), LN_X(N), LN_Y(N)
      REAL SOMA_X, SOMA_Y, MEDIA_X, MEDIA_Y
      REAL NUM, DEN, N_EXP, LN_C, C

      DATA X /1.0, 2.0, 3.0, 4.0, 5.0, 6.0/
      DATA Y /1.0, 4.0, 9.0, 16.0, 25.0, 36.0/

      ! Transformação logarítmica
      DO 10 I = 1, N
         LN_X(I) = LOG(X(I))
         LN_Y(I) = LOG(Y(I))
10    CONTINUE

      ! Médias
      SOMA_X = 0.0
      SOMA_Y = 0.0
      DO 20 I = 1, N
         SOMA_X = SOMA_X + LN_X(I)
         SOMA_Y = SOMA_Y + LN_Y(I)
20    CONTINUE
      MEDIA_X = SOMA_X / N
      MEDIA_Y = SOMA_Y / N

      ! Cálculo da inclinação (n) e ln(C)
      NUM = 0.0
      DEN = 0.0
      DO 30 I = 1, N
         NUM = NUM + (LN_X(I)-MEDIA_X)*(LN_Y(I)-MEDIA_Y)
         DEN = DEN + (LN_X(I)-MEDIA_X)  2
```

```
30    CONTINUE

      N_EXP = NUM / DEN
      LN_C = MEDIA_Y - N_EXP*MEDIA_X
      C = EXP(LN_C)

      PRINT *, 'Lei de potência ajustada: Y = ', C, ' * X^', N_EXP

      END
```

Explicações e Observações:

- Primeiro, os dados são armazenados em arrays e transformados para escala logarítmica.
- As médias de $\ln x$ e $\ln y$ são calculadas.
- Os coeficientes n (expoente) e $\ln C$ são determinados via somatórios das diferenças em relação às médias.
- O valor de C é obtido exponenciando $\ln C$.
- Essa técnica converte um problema de ajuste não-linear em um problema de regressão linear clássico, permitindo fácil implementação em Fortran.
- O método é aplicável tanto a dados experimentais quanto a resultados de simulações numéricas.

Dessa forma, tanto a regressão linear quanto o ajuste de lei de potência fornecem ferramentas simples e eficientes para análise de dados e extração de parâmetros físicos ou estatísticos a partir de séries discretas.

9 Interpolação

A interpolação é uma ferramenta fundamental em física computacional, especialmente quando se trabalha com dados discretos ou resultados de simulações numéricas, como aqueles obtidos a partir de integrações de equações diferenciais ou medições experimentais. Em muitas situações, não conhecemos a função contínua subjacente, mas dispomos de valores em pontos específicos; a interpolação permite estimar a função em pontos intermediários com base nesses dados. O método mais simples e direto é a interpolação linear, que aproxima a função por segmentos de reta entre pontos consecutivos. Em Fortran básico, isso pode ser implementado facilmente com vetores que armazenam os pontos (x_i, y_i) e uma regra que calcula o valor interpolado y entre dois pontos vizinhos. Embora simples, a interpolação linear não seja suave, pois a derivada da função interpolada sofre descontinuidades nos pontos de junção. Para maior precisão e suavidade, utiliza-se a interpolação polinomial, na qual um único polinômio de grau adequado é ajustado a todos os pontos disponíveis. Este método garante que o polinômio passe exatamente por todos os pontos, mas em Fortran

básico seu uso pode ser limitado por instabilidades numéricas quando o número de pontos aumenta, devido ao fenômeno de Runge. A escolha do grau do polinômio é crítica: polinômios de grau muito alto podem oscilar excessivamente entre os pontos. Uma alternativa mais estável é a interpolação por splines cúbicas. Nesse método, cada intervalo entre pontos consecutivos é aproximado por um polinômio cúbico, e condições de suavidade garantem que a função interpolada e suas derivadas de primeira e segunda ordem sejam contínuas. Em Fortran, implementações básicas podem resolver pequenos sistemas lineares tridiagonais para determinar os coeficientes das splines. Esse método combina precisão, suavidade e estabilidade, sendo adequado mesmo para conjuntos de dados grandes. Ao escolher o método de interpolação em Fortran básico, deve-se considerar a finalidade: se o objetivo é apenas estimar valores intermediários rapidamente, a interpolação linear é suficiente; se deseja-se suavidade e boa aproximação global, splines cúbicas são preferíveis. Além disso, é sempre importante verificar o comportamento da interpolação nos extremos do intervalo, pois métodos polinomiais de alto grau podem apresentar grandes erros fora da faixa dos dados conhecidos. Em resumo, a interpolação em Fortran básico permite implementar técnicas simples de aproximação, compreender a relação entre discretização e continuidade da função, e estudar o trade-off entre complexidade computacional, precisão e estabilidade numérica. Com prática, torna-se possível aplicar esses métodos a problemas de física, engenharia e ciências aplicadas, desde gráficos de funções simuladas até análise de resultados experimentais discretos.

Interpolação Linear

A interpolação linear é a forma mais simples, conectando dois pontos (x_0, y_0) e (x_1, y_1) por uma reta. O valor estimado em x é dado por:

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

```
PROGRAM INTERP_LINEAR
REAL x0, y0, x1, y1, x, y

x0 = 1.0
y0 = 2.0
x1 = 3.0
y1 = 6.0
x = 2.0

! Cálculo da interpolação linear
y = y0 + ((y1 - y0)/(x1 - x0))*(x - x0)
PRINT *, 'Interpolacao linear em x=', x, '-> y=', y
END
```

A interpolação linear é simples, rápida e suficiente quando os pontos são próximos e a função varia de forma quase linear entre eles.

Interpolação Polinomial de Lagrange (3 pontos)

Para mais pontos, podemos usar interpolação polinomial de Lagrange. Com três pontos (x_0, y_0), (x_1, y_1), (x_2, y_2), o polinômio de Lagrange é:

$$y = y_0 L_0(x) + y_1 L_1(x) + y_2 L_2(x),$$

onde

$$L_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}, \quad i = 0, 1, 2.$$

```
PROGRAM LAGRANGE
REAL x0, y0, x1, y1, x2, y2, x, y
REAL L0, L1, L2

x0 = 1.0; y0 = 1.0
x1 = 2.0; y1 = 4.0
x2 = 3.0; y2 = 9.0
x = 2.5

! Cálculo dos polinômios de base de Lagrange
L0 = ((x - x1)*(x - x2))/((x0 - x1)*(x0 - x2))
L1 = ((x - x0)*(x - x2))/((x1 - x0)*(x1 - x2))
L2 = ((x - x0)*(x - x1))/((x2 - x0)*(x2 - x1))

! Interpolação polinomial
y = y0*L0 + y1*L1 + y2*L2
PRINT *, 'Interpolacao Lagrange em x=', x, '-> y=', y
END
```

O polinômio de Lagrange fornece uma aproximação exata nos pontos dados e é útil para poucos pontos. Para muitos pontos, pode ser instável e gerar oscilações indesejadas (fenômeno de Runge).

Spline Cúbica Simples (3 pontos)

As splines cúbicas oferecem uma interpolação suave e contínua de primeira e segunda derivadas. Para três pontos, uma aproximação simples pode ser obtida usando a mesma base do polinômio de Lagrange ou uma forma quadrática simplificada:

```
PROGRAM SPLINE_SIMPLE
REAL x0, y0, x1, y1, x2, y2, x, y
REAL L0, L1, L2

x0 = 0.0; y0 = 0.0
x1 = 1.0; y1 = 1.0
x2 = 2.0; y2 = 4.0
x = 1.5

! Polinômios de base para spline simples
L0 = ((x - x1)*(x - x2))/((x0 - x1)*(x0 - x2))
L1 = ((x - x0)*(x - x2))/((x1 - x0)*(x1 - x2))
L2 = ((x - x0)*(x - x1))/((x2 - x0)*(x2 - x1))
```

```
! Avaliação da spline
y = y0*L0 + y1*L1 + y2*L2
PRINT *, 'Spline simples em x=', x, '-> y=', y
END
```

Este método produz uma interpolação mais suave que o polinômio único de Lagrange quando aplicado localmente. Em casos reais, para muitas amostras, o uso de splines cúbicas completas com sistemas tridiagonais garante continuidade das derivadas e evita oscilações. Esses exemplos ilustram três métodos de interpolação em Fortran: linear, polinomial de Lagrange e spline cúbica simples. Eles fornecem ferramentas práticas para estimar valores de funções em pontos intermediários, sendo adequados para aprendizado, experimentação didática e pequenas aplicações numéricas.

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 **Fortran básico**. 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** e resolução de problemas de autovalores, essenciais para o estudo de Hamiltonianos e modos normais. Todos os exemplos serão apresentados com códigos em Fortran básico, enfatizando técnicas de álgebra linear, vetores, matrizes fixas ou alocáveis de forma simples, iterações e normalizações, de modo a tornar os conceitos acessíveis mesmo a quem inicia na programação científica. O objetivo geral desta parte é fornecer um material direto, autoexplicativo e útil para estudantes, destacando não apenas a aplicação prática dos algoritmos, mas também suas interpretações físicas, vantagens e limitações, sempre com implementações básicas em Fortran que podem ser testadas e modificadas pelos alunos.

10 Método de Euler

O método de Euler explícito aproxima a derivada por uma diferença finita:

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

Aplicando ao sistema massa-mola sem atrito:

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

O método de Euler fornece a recorrência:

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

```
      PROGRAM MASSA_MOLA_EULER
      INTEGER N, i
      PARAMETER (N=1000)
      REAL DT, OMEGA, x, v, t
      DT = 0.01
      OMEGA = 1.0
      x = 1.0
      v = 0.0

      OPEN(10, FILE='massa_mola_euler.dat')
      DO 20 i = 0, N-1
         t = i*DT
         WRITE(10,*) t, x, v
         x = x + DT*v
         v = v - DT*OMEGA*OMEGA*x
20    CONTINUE
      CLOSE(10)
      END
```

O método de Euler utilizado neste exemplo é simples e direto, permitindo integrar a equação do movimento de um sistema massa–mola sem atrito. Entretanto, ele apresenta limitações importantes: a precisão depende do tamanho do passo de tempo Δt, e valores muito grandes podem causar instabilidade numérica. Além disso, o método não conserva exatamente a energia do sistema, levando a erros acumulativos ao longo do tempo. Para simulações longas ou sistemas mais complexos, métodos de integração mais avançados, como o de Runge-Kutta ou algoritmos symplectic, oferecem melhor estabilidade e precisão. Mesmo com essas limitações, o método de Euler é útil como primeira introdução à solução numérica de EDOs, ajudando a visualizar qualitativamente o comportamento dinâmico do sistema. Ele também permite entender conceitos de convergência e erro numérico, servindo de base para técnicas mais sofisticadas em física computacional.

11 Diferenças Finitas Centradas no Tempo

O método das diferenças finitas centradas no tempo, é uma técnica simples e eficiente para integrar numericamente equações diferenciais de segunda ordem, como as do oscilador harmônico. A ideia principal é aproximar a segunda derivada da posição em relação ao tempo usando três pontos consecutivos:

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

Substituindo na equação do oscilador harmônico sem atrito,

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

obtemos a fórmula de atualização discreta:

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

Essa relação nos permite calcular a posição no passo seguinte x_{n+1} a partir das duas posições anteriores x_n e x_{n-1}. Note que o método é *simétrico no tempo* e apresenta boa conservação de energia para sistemas oscilatórios.

Passo Inicial

Para iniciar a iteração, precisamos das posições nos dois primeiros instantes x_0 e x_1. Sabemos a posição inicial x_0 e a velocidade inicial v_0. Usando uma expansão de Taylor em $t = 0$:

$$x(\Delta t) = x_0 + \Delta t\, v_0 + \frac{1}{2}(\Delta t)^2 \frac{d^2x}{dt^2}\Big|_{t=0} + O(\Delta t^3),$$

e substituindo $\frac{d^2x}{dt^2} = -\omega^2 x_0$, obtemos:

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

A partir de x_0 e x_1, podemos aplicar a fórmula de diferenças finitas centradas para calcular todas as posições subsequentes $x_2, x_3, \ldots, x_N$. Este método é particularmente útil para sistemas oscilatórios simples como a massa-mola, pois combina simplicidade computacional com boa precisão para passos de tempo moderados.

Resumo do Algoritmo

1. Definir x_0 (posição inicial) e v_0 (velocidade inicial).

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

3. Para cada passo $n \geq 1$, calcular:

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

4. Repetir até alcançar o instante final desejado.

Este método é simples de implementar em qualquer linguagem de programação e fornece uma excelente aproximação da oscilação da massa presa à mola, com boa conservação da energia total do sistema ao longo do tempo. A implementação pode ser encontrada logo a seguir:

```
      PROGRAM MASSA_MOLA_DIF
      INTEGER N, i
      PARAMETER (N=1000)
      REAL DT, OMEGA, x(0:N-1), v0, t
      DT = 0.01
      OMEGA = 1.0
      v0 = 0.0
      x(0) = 1.0
      x(1) = x(0) + DT*v0 - 0.5*DT*DT*OMEGA*OMEGA*x(0)

      OPEN(10, FILE='massa_mola_dif_finitas.dat')
      DO 20 i = 0, N-1
         t = i*DT
         WRITE(10,*) t, x(i)
         IF (i .GE. 1 .AND. i .LT. N-1) THEN
            x(i+1) = 2*x(i) - x(i-1) - DT*DT*OMEGA*OMEGA*x(i)
         ENDIF
20    CONTINUE
      CLOSE(10)
      END
```

Esse método é mais estável que Euler explícito para oscilações harmônicas. A velocidade pode ser estimada por

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

sem necessidade de cálculos adicionais. Além disso, ele preserva a energia total do sistema melhor do que o Euler, evitando crescimento ou dissipação artificial de energia, sendo adequado para simulações de osciladores harmônicos e problemas que requerem conservação física.

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

O método de Runge-Kutta de 4ª ordem (RK4) é amplamente utilizado para resolver equações diferenciais ordinárias (EDOs) com alta precisão e estabilidade. A ideia é calcular quatro estimativas da derivada por passo de tempo e combiná-las de forma ponderada para aproximar a solução:

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

Sistema Massa-Mola sem Atrito

Consideramos o oscilador harmônico simples, cuja equação diferencial de segunda ordem pode ser escrita como um sistema de duas EDOs de primeira ordem:

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

Tratamos x e v como um vetor de variáveis:

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

Aplicando RK4, a atualização do sistema é feita com os vetores $\mathbf{k}_i$ para cada componente de posição e velocidade, exatamente como descrito nas equações acima para C.

Código em Fortran: Massa-Mola com RK4

O programa Fortran abaixo implementa RK4 para o oscilador harmônico simples. O código é básico, comentado e armazena os resultados em arquivo para análise posterior:

```
program massa_mola_rk4
  implicit none
  integer, parameter :: N = 1000
  real(8), parameter :: dt = 0.01, omega = 1.0
  real(8) :: x, v, t
  real(8) :: k1x, k2x, k3x, k4x
  real(8) :: k1v, k2v, k3v, k4v
  integer :: i
  open(unit=10, file="massa_mola_rk4.dat")

  ! Condições iniciais
  x = 1.0d0
  v = 0.0d0

  do i = 1, N
     t = (i-1) * dt
     write(10,*) t, x, v

     ! Cálculo dos coeficientes k
     k1x = dt * v
     k1v = -dt * omega  2 * x
```

```
    k2x = dt * (v + 0.5d0*k1v)
    k2v = -dt * omega  2 * (x + 0.5d0*k1x)

    k3x = dt * (v + 0.5d0*k2v)
    k3v = -dt * omega  2 * (x + 0.5d0*k2x)

    k4x = dt * (v + k3v)
    k4v = -dt * omega  2 * (x + k3x)

    ! Atualização das variáveis
    x = x + (k1x + 2.0d0*k2x + 2.0d0*k3x + k4x)/6.0d0
    v = v + (k1v + 2.0d0*k2v + 2.0d0*k3v + k4v)/6.0d0
  end do

  close(10)
end program massa_mola_rk4
```

Neste programa:

- As variáveis `x` e `v` representam a posição e a velocidade da massa.
- Os coeficientes $\mathbf{k}_i$ são calculados separadamente para cada componente, seguindo o esquema clássico de RK4.
- O laço `do` percorre todos os passos de tempo, armazenando os resultados no arquivo `massa_mola_rk4.dat`.
- Este método preserva a energia do sistema de forma aproximada e produz uma trajetória oscilatória precisa.

O arquivo gerado permite a visualização da posição e velocidade ao longo do tempo, sendo útil para estudo de osciladores harmônicos e validação de métodos numéricos.

13 Método Velocity-Verlet

O método *Velocity-Verlet* atualiza posições e velocidades de forma acoplada usando a aceleração $a(t) = -\omega^2 x(t)$. É de segunda ordem e muito estável para sistemas conservativos, como o oscilador harmônico simples.
As fórmulas discretas são:

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

Passo a passo:

1. Inicializar x_0, v_0 e $a_0 = -\omega^2 x_0$;

2. Atualizar a posição $x_1 = x_0 + v_0\Delta t + \frac{1}{2}a_0\Delta t^2$;

3. Calcular a nova aceleração $a_1 = -\omega^2 x_1$;

4. Atualizar a velocidade $v_1 = v_0 + \frac{1}{2}(a_0 + a_1)\Delta t$;

5. Repetir o procedimento para os próximos passos.

Código em Fortran : Velocity-Verlet

```
      PROGRAM MASSA_MOLA_VERLET
      INTEGER N, i
      PARAMETER (N=1000)
      REAL DT, OMEGA, x, v, a, a_novo, t
      DT = 0.01
      OMEGA = 1.0
      x = 1.0
      v = 0.0
      a = -OMEGA*OMEGA*x

      OPEN(10, FILE='massa_mola_verlet.dat')
      DO 20 i = 0, N-1
         t = i*DT
         WRITE(10,*) t, x, v

         ! Atualiza posição
         x = x + v*DT + 0.5*a*DT*DT

         ! Calcula nova aceleração
         a_novo = -OMEGA*OMEGA*x

         ! Atualiza velocidade
         v = v + 0.5*(a + a_novo)*DT

         ! Atualiza aceleração para o próximo passo
         a = a_novo
20    CONTINUE
      CLOSE(10)
      END
```

Este código implementa o método Velocity-Verlet de forma didática e simples. Ele preserva aproximadamente a energia do sistema oscilatório e mantém a estabilidade numérica, sendo especialmente indicado para simulações de dinâmica molecular ou sistemas massa-mola conservativos.

14 Método Leap-Frog

O método *Leap-Frog* é um esquema de integração numérica amplamente utilizado para resolver sistemas oscilatórios, como o oscilador harmônico simples. Sua principal característica é a atualização intercalada de posições e velocidades: as velocidades são calculadas em instantes intermediários (*half-step*) em relação às posições. Essa abordagem proporciona excelente

conservação de energia ao longo do tempo, tornando o Leap-Frog ideal para simulações de dinâmica molecular e sistemas Hamiltonianos.
Matematicamente, o método consiste nos seguintes passos por iteração:

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

onde x, v e a representam, respectivamente, posição, velocidade e aceleração da massa. A integração do sistema segue de forma que cada atualização de posição "salta"sobre as velocidades, e vice-versa, daí o nome *Leap-Frog* (sapo que salta).

Código em Fortran: Leap-Frog

O programa abaixo implementa o método Leap-Frog de forma simples e comentada para o oscilador harmônico sem atrito. As variáveis `X`, `V` e `A` representam posição, velocidade e aceleração, enquanto `V_HALF` armazena a velocidade no meio passo:

```
PROGRAM MASSA_MOLA_LEAPFROG
INTEGER N, I
PARAMETER (N=1000)
REAL DT, OMEGA, X, V, A, V_HALF, T

! Inicializa parâmetros do sistema
DT = 0.01
OMEGA = 1.0
X = 1.0
V = 0.0
A = -OMEGA*OMEGA*X

! Calcula velocidade no meio passo inicial
V_HALF = V + 0.5*DT*A

! Abre arquivo para salvar resultados
OPEN(10, FILE='massa_mola_leapfrog.dat')

! Loop de integração temporal
DO 10 I = 0, N-1
   T = I*DT
   ! Escreve tempo, posição e velocidade estimada
   WRITE(10,*) T, X, V_HALF - 0.5*DT*A

   ! Atualiza posição
   X = X + DT*V_HALF

   ! Calcula nova aceleração
   A = -OMEGA*OMEGA*X

   ! Atualiza velocidade no próximo meio passo
   V_HALF = V_HALF + 0.5*DT*A
```

```
10    CONTINUE

      CLOSE(10)
      END
```

Este código ilustra claramente a lógica do Leap-Frog: cada passo atualiza primeiro a posição com base na velocidade intermediária, depois calcula a nova aceleração e finalmente atualiza a velocidade para o próximo meio passo. A saída do programa, gravada em arquivo, permite analisar a evolução temporal da posição e da velocidade, verificando a conservação aproximada da energia e a precisão do método em longos períodos de integração. Por sua simplicidade e estabilidade, o Leap-Frog é uma excelente ferramenta para introduzir alunos à integração numérica de sistemas oscilatórios e à dinâmica computacional em física.

15 Método de Adams-Bashforth (2ª Ordem)

O método de Adams-Bashforth é um método explícito multi-passo, que utiliza os valores de $f(t, y)$ em passos anteriores para estimar o próximo valor da solução. A fórmula de segunda ordem é derivada aproximando a 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:

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

Para o sistema massa-mola:

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

o método é aplicado separadamente:

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

Como o método utiliza dois passos, precisamos de x_0, x_1 e v_0, v_1, obtidos via 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 Fortran: Massa-Mola com Adams-Bashforth (2ª ordem)

```
PROGRAM MASSA_MOLA_ADAMS
IMPLICIT NONE
INTEGER, PARAMETER :: N = 1000
REAL, PARAMETER :: DT = 0.01, OMEGA = 1.0
REAL X(N), V(N)
INTEGER I
OPEN(10, FILE='massa_mola_adams.dat')

! Condições iniciais
X(1) = 1.0
V(1) = 0.0

! Inicialização com Euler
X(2) = X(1) + DT * V(1)
V(2) = V(1) - DT * OMEGA 2 * X(1)

! Loop principal do Adams-Bashforth
DO I = 2, N-1
   WRITE(10,*) I*DT, 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 2*X(I) + OMEGA 2*X(I-1))
END DO

CLOSE(10)
END
```

O código resolve o oscilador harmônico simples usando Adams–Bashforth de segunda ordem, gravando posição e velocidade em arquivo para análise da precisão e estabilidade do método. A etapa inicial via Euler fornece os valores necessários para iniciar a sequência multi-passo.

16 Método de Taylor de 2ª Ordem

O método de Taylor aproxima a solução de uma EDO usando expansão em série de Taylor. Para ordem 2:

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

Aplicando ao sistema massa-mola:

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\omega^2 x \end{cases} \Rightarrow \begin{cases} x_{n+1} = x_n + \Delta t\, v_n - \frac{\Delta t^2}{2}\omega^2 x_n \\ v_{n+1} = v_n - \Delta t \omega^2 x_n - \frac{\Delta t^2}{2}\omega^2 v_n \end{cases}$$

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

```
PROGRAM MASSA_MOLA_TAYLOR
```

```
IMPLICIT NONE
INTEGER, PARAMETER :: N = 1000
REAL, PARAMETER :: DT = 0.01, OMEGA = 1.0
REAL X, V, X_NOVO, V_NOVO
INTEGER I
OPEN(10, FILE='massa_mola_taylor.dat')

! Condições iniciais
X = 1.0
V = 0.0

! Loop de integração
DO I = 0, N-1
   WRITE(10,*) I*DT, X, V

   X_NOVO = X + DT*V - 0.5*DT*DT*OMEGA 2*X
   V_NOVO = V - DT*OMEGA 2*X - 0.5*DT*DT*OMEGA 2*V

   X = X_NOVO
   V = V_NOVO
END DO

CLOSE(10)
END
```

O código apresenta uma implementação direta do método de Taylor de segunda ordem, permitindo analisar a evolução temporal da posição e velocidade da massa e compará-lo com outros métodos numéricos em termos de precisão e estabilidade.

17 Equações Diferenciais Estocásticas: Método Euler-Maruyama

A equação de Langevin descreve o movimento de uma partícula sujeita simultaneamente a forças determinísticas e a flutuações térmicas aleatórias. Para uma partícula unidimensional de massa m sujeita a fricção viscosa γ e ruído térmico $\xi(t)$, a equação de Langevin pode ser escrita como:

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

onde $v(t)$ é a velocidade da partícula, γ é o coeficiente de atrito viscoso, e $\xi(t)$ representa o ruído aleatório térmico, normalmente modelado como um processo gaussiano branco com

$$\langle \xi(t) \rangle = 0, \qquad \langle \xi(t)\xi(t') \rangle = 2\gamma k_B T \, \delta(t - t').$$

Para integrar numericamente esta equação estocástica, usamos o método de Euler-Maruyama, que é uma extensão do método de Euler para equações diferenciais estocásticas (SDEs). Discretizando o tempo em passos Δt, obtemos:

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

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

onde η_n são variáveis aleatórias independentes, distribuídas normalmente com média zero e variância unitária ($\eta_n \sim \mathcal{N}(0,1)$). O termo $\sqrt{2\gamma k_B T \Delta t}\, \eta_n$ representa o efeito discreto do ruído térmico durante o passo de tempo Δt.

Fisicamente, o método Euler-Maruyama aproxima a evolução da partícula considerando que, em cada passo de tempo, a velocidade sofre uma contribuição determinística de atrito e uma contribuição aleatória do calor do ambiente. A posição é então atualizada com a nova velocidade, garantindo consistência com a dinâmica estocástica da partícula. Este método é particularmente útil para simular o movimento browniano, difusão e outros processos de transporte estocástico.

Código em Fortran : Langevin via Euler-Maruyama

```
PROGRAM LANGEVIN_EULERMARUYAMA
INTEGER N, R, I, J
PARAMETER (N=10000, R=1000)
REAL DT, GAMMA, KB, T, M, X, V, ETA, SQ2GAMMA, X2
DT = 0.01
GAMMA = 1.0
KB = 1.0
T = 1.0
M = 1.0

OPEN(10, FILE='langevin_euler.dat')

DO J = 1, R
   X = 0.0
   V = 0.0
   SQ2GAMMA = SQRT(2.0*GAMMA*KB*T*DT)

   DO I = 1, N
      ! Gera número aleatório gaussian (Box-Muller)
      CALL RANDOM_NUMBER(ETA)
      ETA = SQRT(-2.0*LOG(ETA)) * COS(2.0*3.14159265*ETA)

      ! Atualiza velocidade e posição
      V = V - GAMMA/M*V*DT + SQ2GAMMA/M*ETA
      X = X + V*DT

      ! Acumula x^2 para média
      IF (J .EQ. 1) THEN
         X2(I) = X*X
      ELSE
         X2(I) = X2(I) + X*X
      END IF
   END DO
END DO

DO I = 1, N
   WRITE(10,*) I*DT, X2(I)/R
```

```
      END DO

      CLOSE(10)
      END
```

Este código realiza múltiplas trajetórias da partícula sujeita a ruído térmico e atrito, calculando o deslocamento quadrático médio $\langle x^2(t)\rangle$ ao longo do tempo. Ele exemplifica a adaptação do método Euler-Maruyama para Fortran básico, mantendo simplicidade e clareza para fins didáticos.

18 Equações Diferenciais Estocásticas: Método de Heun Estocástico (2ª Ordem)

O método de Heun estocástico é uma extensão de segunda ordem do método de Euler-Maruyama. Ele permite integrar equações diferenciais estocásticas com maior precisão temporal, usando uma previsão e correção a cada passo.
Considerando novamente a equação de Langevin unidimensional:

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

onde $\xi(t)$ é ruído branco gaussiano com média zero e correlação $\langle \xi(t)\xi(t')\rangle = 2\gamma k_B T\delta(t-t')$, o método de Heun discretiza a equação da seguinte forma:

$$\begin{aligned} v^* &= v_n - \frac{\gamma}{m}v_n\Delta t + \frac{1}{m}\sqrt{2\gamma k_B T\Delta t}\,\eta_n, \\ x^* &= x_n + v_n\Delta t, \\ v_{n+1} &= v_n - \frac{\gamma}{2m}(v_n + v^*)\Delta t + \frac{1}{m}\sqrt{2\gamma k_B T\Delta t}\,\eta_n, \\ x_{n+1} &= x_n + \frac{\Delta t}{2}(v_n + v^*), \end{aligned}$$

onde v^* e x^* representam os valores previstos (previsão), que são então corrigidos para obter v_{n+1} e x_{n+1}. Este esquema reduz o erro de discretização em relação ao Euler-Maruyama, mantendo o tratamento correto do ruído térmico. O código em fortran pode ser encontrado a seguir:

```
      PROGRAM LANGEVIN_HEUN
      INTEGER N, R, I, J
      PARAMETER (N=10000, R=1000)
      REAL DT, GAMMA, KB, T, M
      REAL X, V, XSTAR, VSTAR, ETA, SQ2GAMMA, X2
      REAL, DIMENSION(N) :: X2SUM
```

```
DT = 0.01
GAMMA = 1.0
KB = 1.0
T = 1.0
M = 1.0

OPEN(10, FILE='langevin_heun.dat')

DO J = 1, R
   X = 0.0
   V = 0.0
   SQ2GAMMA = SQRT(2.0*GAMMA*KB*T*DT)

   DO I = 1, N
      ! Gera número aleatório gaussian (Box-Muller)
      CALL RANDOM_NUMBER(ETA)
      ETA = SQRT(-2.0*LOG(ETA)) * COS(2.0*3.14159265*ETA)

      ! Previsão (Euler)
      VSTAR = V - GAMMA/M*V*DT + SQ2GAMMA/M*ETA
      XSTAR = X + V*DT

      ! Correção (Heun)
      V = V - 0.5*GAMMA/M*(V + VSTAR)*DT + SQ2GAMMA/M*ETA
      X = X + 0.5*(V + VSTAR)*DT

      ! Acumula x^2 para média
      IF (J .EQ. 1) THEN
         X2SUM(I) = X*X
      ELSE
         X2SUM(I) = X2SUM(I) + X*X
      END IF
   END DO
END DO

DO I = 1, N
   WRITE(10,*) I*DT, X2SUM(I)/R
END DO

CLOSE(10)
END
```

O programa acima implementa o método de Heun estocástico para a equação de Langevin. Ele realiza múltiplas trajetórias da partícula sujeita a ruído térmico e atrito, calculando o deslocamento quadrático médio $\langle x^2(t) \rangle$ ao longo do tempo. A abordagem de previsão-correção melhora a precisão temporal e mantém simplicidade para implementação em Fortran básico, sendo ideal para fins didáticos e simulações de movimento browniano.

19 Introdução ao Método das Diferenças Finitas para a Equação de Calor 1D

A Equação da Difusão, ou Equação de Calor unidimensional, é um problema clássico da Física Matemática, descrevendo como a temperatura $u(x, t)$ se distribui ao longo de uma

barra fina (dimensão x) em função do tempo t. A equação é dada por:

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}, \tag{1}$$

onde D é o coeficiente de difusão térmica. Para resolvê-la numericamente, empregamos o Método das Diferenças Finitas, que, como já debatemos anteriormente, consiste em substituir as derivadas (aqui derivadas parciais) por aproximações de diferenças finitas.

Discretização do Domínio e das Derivadas

O domínio contínuo é discretizado em uma grade. A posição espacial x é dividida em N pontos com espaçamento Δx, e o tempo t é dividido em passos Δt. Denotamos a temperatura no ponto de grade i e no passo de tempo k como $u_i^k = u(x_i, t_k)$.

- **Derivada Temporal ($\frac{\partial u}{\partial t}$):** Aproximação progressiva de primeira ordem (forward difference):

$$\frac{\partial u}{\partial t} \approx \frac{u_i^{k+1} - u_i^k}{\Delta t} + O(\Delta t) \tag{2}$$

- **Derivada Espacial ($\frac{\partial^2 u}{\partial x^2}$):** Aproximação centrada de segunda ordem:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{(\Delta x)^2} + O((\Delta x)^2) \tag{3}$$

Formulação do Esquema Explícito

Substituindo estas aproximações na Equação (1), obtemos o Esquema Explícito de Diferenças Finitas:

$$\frac{u_i^{k+1} - u_i^k}{\Delta t} = D\frac{u_{i+1}^k - 2u_i^k + u_{i-1}^k}{(\Delta x)^2}. \tag{4}$$

Reorganizando a equação para isolar a temperatura no próximo passo de tempo u_i^{k+1}, definimos o parâmetro $\lambda = D\frac{\Delta t}{(\Delta x)^2}$. A fórmula de atualização torna-se:

$$u_i^{k+1} = u_i^k + \lambda\left(u_{i+1}^k - 2u_i^k + u_{i-1}^k\right) = \lambda u_{i-1}^k + (1-2\lambda)u_i^k + \lambda u_{i+1}^k. \tag{5}$$

Este é um esquema **explícito**, pois a temperatura futura u_i^{k+1} é calculada diretamente a partir dos valores conhecidos no tempo atual k.

Critério de Estabilidade de Von Neumann

O método explícito é **condicionalmente estável**. Para evitar que as oscilações numéricas cresçam indefinidamente, o parâmetro λ deve satisfazer:

$$\lambda = D\frac{\Delta t}{(\Delta x)^2} \leq \frac{1}{2}. \tag{6}$$

Se Δt for muito grande (ou seja, $\lambda > 1/2$), o erro numérico explode, tornando a solução inútil. Por essa razão, em muitos casos práticos, o **Método Implícito** — que é incondicionalmente estável — é preferido, embora exija a solução de um sistema linear a cada passo de tempo.

Exemplo Básico em Fortran

A seguir, um código simples em Fortran que implementa o esquema explícito para a equação de calor 1D:

```
program calor_1d
  implicit none
  integer, parameter :: N = 10, Nt = 100
  real, parameter :: L = 1.0, D = 0.1, dx = L/(N-1), dt = 0.005
  real :: lambda, u(N), u_new(N)
  integer :: i, k

  ! Calcula lambda e verifica estabilidade
  lambda = D * dt / (dx*dx)
  if (lambda > 0.5) then
     print *, 'Atenção: esquema instável! Reduza dt ou aumente dx.'
     stop
  end if

  ! Condição inicial
  u = 0.0
  u(N/2) = 1.0  ! pulso inicial no centro

  ! Laço de evolução temporal
  do k = 1, Nt
     ! Esquema explícito
     do i = 2, N-1
        u_new(i) = u(i) + lambda * (u(i+1) - 2*u(i) + u(i-1))
     end do

     ! Condições de contorno (Dirichlet: temperatura fixa)
     u_new(1) = 0.0
     u_new(N) = 0.0

     ! Atualiza temperatura
     u = u_new
  end do

  ! Imprime resultado final
  print *, 'Temperaturas finais:'
  print *, u

end program calor_1d
```

O código acima implementa de forma simples o esquema explícito de diferenças finitas para a equação de calor unidimensional. Observações importantes:

- A variável `lambda` controla a estabilidade do método; seu valor deve satisfazer $\lambda \leq 0.5$. Caso contrário, o programa interrompe a execução avisando que o esquema se tornaria instável.
- O vetor `u` armazena a distribuição de temperatura no tempo atual, enquanto `u_new` armazena a distribuição no próximo passo de tempo.
- O laço `do k = 1, Nt` realiza a evolução temporal da temperatura, aplicando a fórmula explícita (5) em cada ponto interno da barra.
- As condições de contorno são fixas (*Dirichlet*), mantendo as extremidades da barra em temperatura zero.
- Ao final da simulação, o programa imprime a distribuição de temperatura final ao longo da barra, permitindo verificar a difusão do pulso inicial.

Este exemplo é didático e facilmente modificável: você pode alterar o número de pontos, o passo de tempo, o coeficiente de difusão ou a condição inicial para explorar diferentes cenários de difusão térmica. Além disso, o mesmo esquema pode ser estendido para problemas em 2D ou 3D, mantendo a lógica do método explícito.

20 Integração Numérica via Monte Carlo

O **método de Monte Carlo** é uma técnica numérica que utiliza amostragem aleatória para estimar integrais, especialmente útil quando a dimensão do problema é alta ou quando integrais analíticas são complicadas. A ideia central é que a média de uma função avaliada em pontos aleatórios do domínio converge para o valor da integral quando o número de pontos cresce.

Exemplo: Estimativa de π

Uma aplicação clássica do método de Monte Carlo é a estimativa de π usando um *quarto de círculo* inscrito em um quadrado de lado unitário. Se lançarmos N pontos aleatórios (x, y) no quadrado e contarmos quantos caem dentro do círculo ($x^2 + y^2 \leq 1$), temos:

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

Código em Fortran

O programa abaixo implementa essa ideia de forma simples:

```
PROGRAM MONTECARLO_PI
INTEGER N, DENTRO, I
PARAMETER (N=1000000)
```

```
REAL X, Y
DENTRO = 0

! Inicializa o gerador de números aleatórios
CALL RANDOM_SEED()

! Laço principal: gera pontos aleatórios e conta os que caem dentro do círculo
DO I = 1, N
   CALL RANDOM_NUMBER(X)
   CALL RANDOM_NUMBER(Y)

   IF (X*X + Y*Y .LE. 1.0) DENTRO = DENTRO + 1
END DO

! Calcula e imprime a estimativa de pi
PRINT *, 'Estimativa de pi = ', 4.0*DENTRO/N
END
```

Discussão

- O programa gera $N = 1.000.000$ pontos aleatórios uniformemente distribuídos no quadrado unitário.
- A variável `DENTRO` conta quantos pontos caem dentro do quarto de círculo.
- A estimativa de π é obtida multiplicando a fração de pontos dentro do círculo por 4.
- Quanto maior o número de pontos N, maior a precisão da estimativa, devido à Lei dos Grandes Números.
- Esta abordagem pode ser facilmente generalizada para integrais em dimensões superiores ou regiões de forma arbitrária.

O método de Monte Carlo é particularmente poderoso em problemas de alta dimensão, onde métodos tradicionais de quadratura se tornam inviáveis. Além disso, ele fornece uma maneira intuitiva de relacionar probabilidades e integrais, como exemplificado neste caso de cálculo de π.

Monte Carlo em Alta Dimensão

O método de Monte Carlo se torna especialmente vantajoso em integrais multidimensionais, onde métodos tradicionais de quadratura se tornam inviáveis devido à explosão combinatória de pontos. Consideremos a integral

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

onde d é a dimensão do problema. A ideia é gerar N vetores aleatórios $\mathbf{x}^{(j)} = (x_1^{(j)}, \ldots, x_d^{(j)}) \in [0,1]^d$ e calcular a média da função avaliando cada vetor:

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

Código em Fortran: Monte Carlo Multidimensional

O programa abaixo implementa este cálculo em Fortran:

```
PROGRAM MONTECARLO_D
INTEGER N, D, I, J
PARAMETER (N=1000000, D=10)
REAL X, SOMA, PROD

! Inicializa soma da integral
SOMA = 0.0

! Inicializa gerador de números aleatórios
CALL RANDOM_SEED()

! Laço principal: gera N vetores aleatórios e calcula f(x)
DO I = 1, N
   PROD = 0.0
   DO J = 1, D
      CALL RANDOM_NUMBER(X)
      PROD = PROD + X*X
   END DO
   SOMA = SOMA + EXP(-PROD)
END DO

! Imprime resultado da integral estimada
PRINT *, 'Integral estimada em ', D, ' dimensoes = ', SOMA/N
END
```

Discussão

- O programa gera $N = 10^6$ vetores aleatórios de dimensão $D = 10$ no hipercubo unitário $[0, 1]^D$.
- Para cada vetor, calcula-se a função $\exp(-\sum_i x_i^2)$ e acumula-se em `SOMA`.
- A estimativa final da integral é obtida dividindo a soma pelo número de amostras N.
- A precisão do método aumenta com o número de amostras N, independentemente da dimensão D, o que é a grande vantagem do método Monte Carlo sobre métodos tradicionais de quadratura.
- Este esquema é facilmente generalizável para outras funções ou domínios de integração multidimensional.

Este exemplo ilustra como Monte Carlo permite avaliar integrais de alta dimensão de forma eficiente e direta, sem a necessidade de criar grades complexas de pontos como nos métodos determinísticos tradicionais.

21 Transformada Discreta de Fourier (DFT)

A **Transformada Discreta de Fourier (DFT)** é uma ferramenta fundamental na análise de sinais, permitindo decompor um vetor de N pontos x_n em suas componentes de frequência complexas X_k:

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

Para fins didáticos em Fortran básico, é comum separar explicitamente as partes real e imaginária da DFT, evitando o uso de tipos complexos.

Código em Fortran: DFT simples para dados reais

O programa abaixo implementa a DFT direta, calculando explicitamente as componentes real (`REX`) e imaginária (`IMX`):

```
PROGRAM DFT_REAL
INTEGER N, K, I
PARAMETER (N=256)
REAL PI, X(N), REX(N), IMX(N), ANGLE
PI = 3.141592653589793

! Sinal de entrada: seno simples
DO I = 1, N
   X(I) = SIN(2.0*PI*5*(I-1)/N)
END DO

! Inicializa arrays de resultado
DO K = 1, N
   REX(K) = 0.0
   IMX(K) = 0.0
END DO

! Calcula DFT
DO K = 1, N
   DO I = 1, N
      ANGLE = 2.0*PI*(K-1)*(I-1)/N
      REX(K) = REX(K) + X(I)*COS(ANGLE)
      IMX(K) = IMX(K) - X(I)*SIN(ANGLE)
   END DO
END DO

! Salva módulo da DFT em arquivo
OPEN(10, FILE='dft_fortran.dat')
DO K = 1, N
   WRITE(10,*) K-1, SQRT(REX(K) 2 + IMX(K) 2)
END DO
CLOSE(10)
END
```

O vetor `X` armazena o sinal de entrada, que neste exemplo é um seno simples com frequência 5. Para cada ponto do vetor, o programa calcula a Transformada Discreta de

Fourier (DFT) separando explicitamente as partes real e imaginária, armazenadas nos arrays `REX` e `IMX`, respectivamente. Essa separação permite compreender melhor a mecânica da DFT, uma vez que cada componente de frequência é representada por sua amplitude e fase. O módulo da DFT, obtido como $\sqrt{\texttt{REX(K)}^2 + \texttt{IMX(K)}^2}$, indica a amplitude de cada frequência e é salvo em arquivo, facilitando a análise posterior ou a visualização gráfica. O método direto utilizado neste código tem complexidade $O(N^2)$, o que o torna adequado para sinais de tamanho moderado e para fins didáticos, permitindo aos alunos observar claramente como cada ponto contribui para a transformada. Para sinais maiores, entretanto, é recomendado utilizar algoritmos *Fast Fourier Transform (FFT)*, que reduzem a complexidade para $O(N \log N)$, mantendo a eficiência computacional. Separar explicitamente as partes real e imaginária ajuda ainda a visualizar como as componentes senoidais se combinam para reproduzir o sinal original. Além disso, esse procedimento fornece uma base sólida para compreender conceitos fundamentais de análise de frequência e processamento de sinais discretos. Ao final, a DFT direta apresentada ilustra como é possível decompor um sinal discreto em suas frequências constituintes, permitindo identificar amplitudes dominantes e compreender a estrutura espectral do sinal. Este exemplo também prepara o terreno para implementações mais avançadas, incluindo FFT e processamento de sinais multidimensionais, mostrando que a manipulação explícita de cada componente ajuda a consolidar a intuição sobre transformadas de Fourier. Em resumo, o código demonstra de forma clara a relação entre sinal temporal e sua representação espectral, destacando a utilidade didática da DFT direta em Fortran antes de avançar para métodos mais eficientes e sofisticados.

Transformada Rápida de Fourier (FFT)

A **Transformada Rápida de Fourier (FFT)** é um algoritmo eficiente para o cálculo da DFT, reduzindo a complexidade computacional de $O(N^2)$ para $O(N \log N)$. A ideia central consiste em explorar a simetria e a periodicidade dos fatores exponenciais complexos da DFT, dividindo o problema original em subproblemas menores. O algoritmo de *Cooley–Tukey*, em particular, separa o vetor de entrada em componentes de índices pares e ímpares, computa suas DFTs parciais e as combina de forma recursiva.

A forma geral da DFT pode ser escrita como:

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi kn/N},$$

que, para N par, pode ser decomposta em:

$$X_k = E_k + e^{-i2\pi k/N} O_k, \quad X_{k+N/2} = E_k - e^{-i2\pi k/N} O_k,$$

onde E_k e O_k representam as DFTs dos elementos de índice par e ímpar de x_n, respectivamente. Essa decomposição recursiva é a essência do algoritmo FFT.

A seguir apresentamos uma implementação simples e didática da FFT para vetores de tamanho $N = 2^m$, separando explicitamente as partes real e imaginária:

```
PROGRAM FFT_SIMPLE
IMPLICIT NONE
INTEGER, PARAMETER :: N = 8
INTEGER :: K, N2, M
REAL :: PI, ANGLE, TEMP_R, TEMP_I
REAL :: XR(N), XI(N), WR, WI, TR, TI

PI = 3.141592653589793

! Sinal de entrada: seno simples
DO K = 1, N
   XR(K) = SIN(2.0*PI*2*(K-1)/N)
   XI(K) = 0.0
END DO

! Bit-reversal (reorganiza índices)
CALL BITREV(XR, XI, N)

! Loop principal (Danielson-Lanczos)
N2 = 1
DO WHILE (N2 < N)
   M = 2 * N2
   DO K = 1, N2
      ANGLE = -2.0*PI*(K-1)/M
      WR = COS(ANGLE)
      WI = SIN(ANGLE)
      DO I = K, N, M
         TR = WR*XR(I+N2) - WI*XI(I+N2)
         TI = WR*XI(I+N2) + WI*XR(I+N2)
         XR(I+N2) = XR(I) - TR
         XI(I+N2) = XI(I) - TI
         XR(I) = XR(I) + TR
         XI(I) = XI(I) + TI
      END DO
   END DO
   N2 = M
END DO

! Salva módulo da FFT em arquivo
OPEN(10, FILE='fft_fortran.dat')
DO K = 1, N
   WRITE(10,*) K-1, SQRT(XR(K) 2 + XI(K) 2)
END DO
CLOSE(10)
END

SUBROUTINE BITREV(XR, XI, N)
REAL XR(N), XI(N), TEMP
INTEGER I, J, M, N2
J = 1
DO I = 1, N-1
   IF (I < J) THEN
      TEMP = XR(J); XR(J) = XR(I); XR(I) = TEMP
      TEMP = XI(J); XI(J) = XI(I); XI(I) = TEMP
   END IF
   M = N / 2
   DO WHILE (J > M .AND. M >= 2)
```

```
      J = J - M
      M = M / 2
    END DO
    J = J + M
  END DO
  END
```

Neste código, o vetor de entrada `XR` armazena o sinal real e `XI` inicia com valores nulos. A subrotina `BITREV` executa a *reordenação bit-reversal*, essencial para que o algoritmo de Danielson–Lanczos combine corretamente as partes pares e ímpares da DFT. A parte principal do código aplica sucessivamente as combinações de frequência (*butterflies*), reduzindo a operação total para $O(N \log N)$. O resultado — o módulo da FFT — é gravado em arquivo (`fft_fortran.dat`), permitindo visualização direta do espectro de frequências. Comparado à DFT direta, o ganho de desempenho é expressivo para N grandes, embora o princípio matemático seja o mesmo. Implementações modernas (por exemplo, FFTW, MKL) seguem o mesmo formalismo, mas otimizam o acesso à memória e a vetorização.

22 Solução Numérica de Sistemas Lineares

Sistemas de equações lineares da forma

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

onde A é uma matriz quadrada de dimensão $N \times N$, $\mathbf{x}$ é o vetor desconhecido e $\mathbf{b}$ é o vetor de termos independentes, são extremamente comuns em problemas de física computacional, engenharia e matemática aplicada. Resolver esses sistemas de forma eficiente e estável é essencial para muitas simulações numéricas, incluindo métodos de elementos finitos, soluções de equações diferenciais discretizadas e análise de circuitos elétricos. Uma das abordagens mais didáticas e intuitivas para resolver esses sistemas é a eliminação de Gauss. Esse método consiste em transformar a matriz A em uma forma triangular superior, eliminando sucessivamente as incógnitas das equações inferiores. Uma vez que a matriz está triangular, realiza-se a substituição regressiva (*back substitution*) para calcular os valores de $\mathbf{x}$ de maneira direta, iniciando pela última incógnita e progredindo até a primeira. O método de Gauss é bastante transparente para fins pedagógicos, pois mostra claramente como as combinações lineares das linhas da matriz reduzem o sistema a uma forma simples de resolver. Além disso, pode ser facilmente implementado em Fortran usando laços aninhados, permitindo que os alunos observem passo a passo a manipulação dos elementos da matriz e a atualização do vetor $\mathbf{b}$. Apesar de não ser o método mais eficiente para matrizes muito grandes ou esparsas, a eliminação de Gauss é uma ferramenta valiosa para entender os princípios fundamentais de resolução de sistemas lineares antes de avançar para métodos mais sofisticados, como decomposição LU, Cholesky ou algoritmos iterativos. Esta abordagem também permite introduzir conceitos importantes, como pivotamento parcial, estabilidade

numérica e propagação de erros, que são cruciais para garantir que a solução obtida seja precisa e confiável em aplicações de física computacional.

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

```
PROGRAM GAUSS_3x3
INTEGER N, I, J, K
PARAMETER (N=3)
REAL A(3,3), B(3), X(3), FATOR, SOMA

DATA A /2,-1,1, 3,3,9, 3,3,5/
DATA B /8,0,-6/

! Eliminação de Gauss
DO K = 1, N-1
   DO I = K+1, N
      FATOR = A(I,K)/A(K,K)
      DO J = K, N
         A(I,J) = A(I,J) - FATOR*A(K,J)
      END DO
      B(I) = B(I) - FATOR*B(K)
   END DO
END DO

! Substituição regressiva
DO I = N, 1, -1
   SOMA = 0.0
   DO J = I+1, N
      SOMA = SOMA + A(I,J)*X(J)
   END DO
   X(I) = (B(I) - SOMA)/A(I,I)
END DO

! Imprime solução
PRINT *, 'Solução:'
DO I = 1, N
   PRINT *, 'x(', I, ') = ', X(I)
END DO
END
```

O programa acima implementa a eliminação de Gauss para um sistema 3x3 de forma direta. Ele primeiro transforma a matriz em triangular superior, eliminando as incógnitas de forma sistemática, e em seguida realiza a substituição regressiva para calcular o vetor solução $\mathbf{x}$. A simplicidade do código permite acompanhar passo a passo como cada elemento da matriz e do vetor $\mathbf{b}$ é atualizado. Este exemplo serve como base didática para sistemas maiores ou para a introdução de técnicas mais avançadas, como pivotamento parcial e decomposição LU. Ao executar o programa, o usuário obtém os valores de $\mathbf{x}$ de forma clara e imediata.

Sistemas Tridiagonais: Método de Thomas

Sistemas lineares tridiagonais aparecem com frequência em problemas de discretização de equações diferenciais, especialmente em esquemas de diferenças finitas e métodos implícitos, como o de Crank–Nicolson. Tais sistemas possuem elementos diferentes de zero apenas na

diagonal principal e nas diagonais imediatamente acima e abaixo, podendo ser escritos na forma:

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

com as condições de contorno $a_1 = 0$ e $c_N = 0$.

O método de Thomas é uma forma simplificada e eficiente de eliminação de Gauss, especialmente adaptada à estrutura tridiagonal da matriz. Diferentemente da eliminação geral, que possui custo computacional cúbico $\mathcal{O}(N^3)$, o método de Thomas apresenta complexidade linear $\mathcal{O}(N)$, tornando-o ideal para grandes sistemas unidimensionais.

O procedimento consiste em duas etapas principais:

1. **Eliminação direta (forward sweep):** A partir da primeira equação, elimina-se progressivamente o termo subdiagonal a_i, modificando os coeficientes efetivos:

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

   com as condições iniciais $c'_1 = c_1/b_1$ e $d'_1 = d_1/b_1$.

2. **Substituição retroativa (backward substitution):** Uma vez obtidos os coeficientes modificados, a solução é calculada de trás para frente:

$$x_N = d'_N, \qquad x_i = d'_i - c'_i x_{i+1}, \quad i = N-1, N-2, \ldots, 1.$$

A estrutura tridiagonal garante que cada etapa envolva apenas um número constante de operações por linha, o que assegura a eficiência do algoritmo. Além disso, o método é estável para matrizes estritamente diagonalmente dominantes, o que é típico em discretizações de operadores diferenciais como o laplaciano. Na prática, o método de Thomas é amplamente utilizado na solução de equações de difusão, propagação de ondas, e na evolução temporal de equações de Schrödinger discretizadas, onde o operador cinético leva naturalmente à forma tridiagonal.

Código em Fortran : Método de Thomas para 4×4

```
PROGRAM THOMAS_4x4
INTEGER N, I
PARAMETER (N=4)
REAL A(4), B(4), C(4), D(4)
REAL C_PRIME(4), D_PRIME(4), X(4), M

DATA A /0,1,1,1/
DATA B /4,4,4,3/
DATA C /1,1,1,0/
DATA D /15,15,15,10/
```

```
! Forward sweep
C_PRIME(1) = C(1)/B(1)
D_PRIME(1) = D(1)/B(1)

DO I = 2, N
   M = B(I) - A(I)*C_PRIME(I-1)
   C_PRIME(I) = (I<N) ? C(I)/M : 0
   D_PRIME(I) = (D(I) - A(I)*D_PRIME(I-1))/M
END DO

! Back substitution
X(N) = D_PRIME(N)
DO I = N-1, 1, -1
   X(I) = D_PRIME(I) - C_PRIME(I)*X(I+1)
END DO

! Imprime solução
PRINT *, 'Solução (Thomas):'
DO I = 1, N
   PRINT *, 'x(', I, ') = ', X(I)
END DO
END
```

Esse código aproveita a estrutura tridiagonal para reduzir operações e memória, sendo ideal para grandes sistemas lineares com diagonais dominantes.

23 Evolução de uma Onda Gaussiana no Oscilador Harmônico Quântico: Método Suzuki–Trotter com Crank–Nicolson

Neste exemplo, consideramos a equação de Schrödinger dependente do tempo unidimensional:

$$i\hbar\frac{\partial\psi(x,t)}{\partial t} = \hat{H}\psi(x,t), \quad \hat{H} = \frac{\hat{p}^2}{2m} + \frac{1}{2}m\omega^2x^2,$$

onde $\hat{p} = -i\hbar\frac{\partial}{\partial x}$ e x é a coordenada espacial. O objetivo é evoluir numericamente uma função de onda inicial gaussiana:

$$\psi(x,0) = \frac{1}{(\pi\sigma^2)^{1/4}}\exp\Big[-\frac{(x-x_0)^2}{2\sigma^2}\Big].$$

O método Suzuki–Trotter de segunda ordem permite aproximar o propagador unitário para um passo de tempo Δt como:

$$e^{-i\hat{H}\Delta t/\hbar} \approx e^{-i\hat{T}\Delta t/2\hbar}\, e^{-i\hat{V}\Delta t/\hbar}\, e^{-i\hat{T}\Delta t/2\hbar},$$

onde $\hat{T} = \hat{p}^2/2m$ e $\hat{V} = \frac{1}{2}m\omega^2x^2$. No método de evolução temporal de Crank–Nicolson, a aplicação do operador cinético $\hat{T}$ é realizada de forma implícita no espaço real, o que requer a resolução de um sistema linear tridiagonal a cada passo temporal. O método baseia-se

em uma discretização simétrica da equação de Schrödinger, de modo que o operador de evolução é aproximado pela média entre os passos *explícito* e *implícito*. Essa simetria assegura estabilidade numérica incondicional e preservação da norma da função de onda.

Dentro do esquema de fatoração de Suzuki–Trotter (ou *split-step*), a evolução durante um intervalo Δt é dividida nas seguintes etapas:

1. **Aplicação de metade do passo de potencial:**

$$\psi(x) \longrightarrow e^{-iV(x)\,\Delta t/(2\hbar)}\,\psi(x)$$

2. **Aplicação do passo completo do operador cinético via Crank–Nicolson:**

$$\left(1+\frac{i\,\hat{T}\,\Delta t}{2\hbar}\right)\psi^{(t+\Delta t)} = \left(1-\frac{i\,\hat{T}\,\Delta t}{2\hbar}\right)\psi^{(t)}$$

   ou, de forma equivalente,

$$\psi^{(t+\Delta t)} = \left(1+\frac{i\,\hat{T}\,\Delta t}{2\hbar}\right)^{-1}\left(1-\frac{i\,\hat{T}\,\Delta t}{2\hbar}\right)\psi^{(t)}.$$

   A resolução dessa equação envolve um sistema tridiagonal, uma vez que o operador $\hat{T}$ contém apenas derivadas de segunda ordem (ou vizinhos imediatos na malha discreta).

3. **Aplicação da segunda metade do passo de potencial:**

$$\psi(x) \longrightarrow e^{-iV(x)\,\Delta t/(2\hbar)}\,\psi(x).$$

Este procedimento é unitário, garantindo a conservação exata da norma da função de onda, e apresenta erro global de ordem $\mathcal{O}(\Delta t^3)$, característico do esquema simétrico de Suzuki–Trotter.

Código em Fortran: Suzuki–Trotter com Crank–Nicolson

```
PROGRAM OSC_HARM_CN
  IMPLICIT NONE
  INTEGER, PARAMETER :: N = 256, NSTEPS = 10000
  REAL(8), PARAMETER :: XMAX = 5.0D0, DT = 0.01D0, HBAR = 1.0D0, MASS = 1.0D0
  REAL(8) :: dx, x(N), V(N)
  COMPLEX(8) :: psi(N)
  REAL(8) :: norm
  INTEGER :: i, istep, step_norm

  ! --- Discretização espacial ---
  dx = 2.0D0*XMAX / (N-1)
  DO i=1,N
     x(i) = -XMAX + (i-1)*dx
     psi(i) = CMPLX(EXP(-x(i) 2),0.0D0)  ! psi inicial gaussiana
```

```
     V(i) = 0.5D0*x(i) 2                ! potencial harmonico
  END DO

  ! --- Normalizar psi inicial ---
  norm = compute_norm(psi, dx, N)
  psi = psi / SQRT(norm)

  step_norm = INT(2.0D0/DT)

  ! --- Loop de tempo ---
  DO istep = 1, NSTEPS
     CALL apply_phase(psi, V, DT/2.0D0/HBAR, N)      ! meia fase potencial
     CALL crank_nicolson(psi, dx, DT, MASS, HBAR, N) ! passo cinético
     CALL apply_phase(psi, V, DT/2.0D0/HBAR, N)      ! meia fase potencial

     IF (MOD(istep, step_norm) == 0) THEN
        norm = compute_norm(psi, dx, N)
        WRITE(*,'(A,I5,A,F12.8)') 'Step ', istep, ', Norm = ', norm
     END IF
  END DO

CONTAINS

!-------------------------------------------------------
! Aplicar fase devido ao potencial
!-------------------------------------------------------
SUBROUTINE apply_phase(psi, V, dt, N)
  INTEGER, INTENT(IN) :: N
  REAL(8), INTENT(IN) :: V(N), dt
  COMPLEX(8), INTENT(INOUT) :: psi(N)
  INTEGER :: i
  DO i=1,N
     psi(i) = psi(i) * CMPLX(COS(-V(i)*dt), SIN(-V(i)*dt))
  END DO
END SUBROUTINE apply_phase

!-------------------------------------------------------
! Passo cinético via Crank-Nicolson
!-------------------------------------------------------
SUBROUTINE crank_nicolson(psi, dx, dt, mass, hbar, N)
  INTEGER, INTENT(IN) :: N
  REAL(8), INTENT(IN) :: dx, dt, mass, hbar
  COMPLEX(8), INTENT(INOUT) :: psi(N)
  COMPLEX(8) :: a(N-1), b(N), c(N-1), rhs(N), psi_new(N)
  COMPLEX(8) :: r
  INTEGER :: i

r = CMPLX(0.0D0, dt*hbar/(4.0D0*mass*dx*dx), 8)! Crank-Nicolson coeficiente
  ! Matriz tridiagonal A*psi_new = B*psi_old
  ! Bordas fixas
  b(1) = 1.0D0 + 2.0D0*r
  c(1) = -r
  DO i=2,N-1
     a(i-1) = -r
     b(i)   = 1.0D0 + 2.0D0*r
     c(i)   = -r
  END DO
  a(N-1) = -r
  b(N) = 1.0D0 + 2.0D0*r
```

```
  ! RHS: B*psi_old
  rhs(1) = (1.0D0 - 2.0D0*r)*psi(1) + r*psi(2)
  DO i=2,N-1
     rhs(i) = r*psi(i-1) + (1.0D0 - 2.0D0*r)*psi(i) + r*psi(i+1)
  END DO
  rhs(N) = r*psi(N-1) + (1.0D0 - 2.0D0*r)*psi(N)

  ! Resolver sistema tridiagonal (Thomas algorithm)
  CALL thomas_solver(a, b, c, rhs, psi_new, N)

  psi = psi_new
END SUBROUTINE crank_nicolson

!-------------------------------------------------------
! Solver tridiagonal (Thomas algorithm)
!-------------------------------------------------------
SUBROUTINE thomas_solver(a, b, c, d, x, N)
  INTEGER, INTENT(IN) :: N
  COMPLEX(8), INTENT(IN) :: a(N-1), b(N), c(N-1), d(N)
  COMPLEX(8), INTENT(OUT) :: x(N)
  COMPLEX(8) :: c_star(N-1), d_star(N)
  INTEGER :: i

  c_star(1) = c(1)/b(1)
  d_star(1) = d(1)/b(1)

  DO i=2,N-1
     c_star(i) = c(i)/(b(i) - a(i-1)*c_star(i-1))
     d_star(i) = (d(i) - a(i-1)*d_star(i-1))/(b(i) - a(i-1)*c_star(i-1))
  END DO
  d_star(N) = (d(N) - a(N-1)*d_star(N-1))/(b(N) - a(N-1)*c_star(N-1))

  x(N) = d_star(N)
  DO i=N-1,1,-1
     x(i) = d_star(i) - c_star(i)*x(i+1)
  END DO
END SUBROUTINE thomas_solver

!-------------------------------------------------------
! Calcular norma da função de onda
!-------------------------------------------------------
FUNCTION compute_norm(psi, dx, N)
  INTEGER, INTENT(IN) :: N
  REAL(8), INTENT(IN) :: dx
  COMPLEX(8), INTENT(IN) :: psi(N)
  INTEGER :: i
  REAL(8) :: compute_norm

  compute_norm = 0.0D0
  DO i=1,N
     compute_norm = compute_norm + ABS(psi(i)) 2 * dx
  END DO
END FUNCTION compute_norm

END PROGRAM OSC_HARM_CN
```

Neste exemplo, o operador cinético é aplicado de forma implícita no espaço real utilizando o método de Crank–Nicolson, o que garante simultaneamente a estabilidade numérica e a preservação da unitariedade da evolução temporal. O operador potencial, por sua vez, é tratado de maneira explícita, atuando diretamente sobre a função de onda no espaço real. A combinação desses dois passos é realizada dentro do esquema simétrico de Suzuki–Trotter de segunda ordem, que mantém o erro global de truncamento na ordem $\mathcal{O}(\Delta t^3)$ e assegura a conservação da norma da função de onda ao longo do tempo. Tal estrutura numérica constitui uma implementação eficiente e precisa para a evolução de sistemas quânticos conservativos, como o oscilador harmônico ou partículas em poços de potencial arbitrários. Além disso, o método é facilmente generalizável para dimensões superiores, bastando estender o operador cinético para incluir derivadas espaciais adicionais. A modularidade do esquema também permite a inclusão de potenciais mais complexos, dependentes do tempo ou do espaço, preservando as propriedades fundamentais da evolução unitária. Dessa forma, o procedimento apresentado representa um ponto de partida sólido para simulações numéricas em mecânica quântica e para o estudo de fenômenos de propagação e confinamento de ondas em diferentes contextos físicos.

24 Evolução de Onda Acústica 1D: Método Crank–Nicolson

Neste exemplo, consideramos a equação de onda acústica unidimensional linear com densidade e elasticidade constantes:

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

onde $u(x,t)$ representa o deslocamento acústico, c é a velocidade do som no meio, $x \in [0, L]$ e $t \geq 0$. O objetivo é evoluir numericamente uma perturbação inicial gaussiana:

$$u(x,0) = \exp\Big[-\frac{(x-x_0)^2}{2\sigma^2}\Big], \quad \frac{\partial u}{\partial t}(x,0) = 0.$$

Para aplicar o método de Crank–Nicolson, reescrevemos a equação de onda de segunda ordem como um sistema para u e $v = \partial u/\partial t$. No entanto, existe uma forma alternativa popular que mantém o método implícito em tempo central:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2} = c^2 \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2},$$

que pode ser rearranjada de forma que a solução de u^{n+1} envolva um sistema tridiagonal , típico do esquema Crank–Nicolson aplicado a equações de onda 1D.

O esquema discreto Crank–Nicolson para $i = 2, \ldots, N-1$ é dado por:

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

onde $r = \frac{c^2 \Delta t^2}{2\Delta x^2}$, e as condições de contorno são normalmente de Dirichlet (bordas fixas) ou

Neumann (bordas livres).

A solução numérica envolve:

1. Inicializar $u(x,0)$ e $u(x,\Delta t)$ usando a condição inicial e uma aproximação explícita do primeiro passo.
2. Em cada passo temporal, montar o sistema tridiagonal $A\,u^{n+1} = \text{RHS}$.
3. Resolver o sistema tridiagonal utilizando o algoritmo de Thomas (*tridiagonal matrix solver*).
4. Atualizar os vetores $u^{n-1} \leftarrow u^n$ e $u^n \leftarrow u^{n+1}$.

Código em Fortran 90: Crank–Nicolson para Onda Acústica 1D

```
PROGRAM wave_1d_cn
  IMPLICIT NONE
  INTEGER, PARAMETER :: N = 200, NSTEPS = 1000
  REAL(8), PARAMETER :: L = 1.0D0, DX = L/(N-1), DT = 0.001D0, C = 1.0D0
  REAL(8) :: r
  REAL(8), DIMENSION(N) :: u, u_prev, u_next, x
  INTEGER :: i, istep

  ! --- Malha espacial ---
  DO i = 1, N
     x(i) = (i-1)*DX
     u(i) = EXP(-100.0D0*(x(i)-0.5D0) 2)  ! Pulso gaussiano inicial
  END DO
  u_prev = u  ! Primeira aproximação explícita (passo t=0)

  ! Coeficiente Crank-Nicolson
  r = C 2 * DT 2 / (2.0D0*DX 2)

  ! --- Loop temporal ---


 OPEN(10, FILE='wave.dat', STATUS='REPLACE')

DO istep = 1, NSTEPS
   CALL build_rhs(u, u_prev, r, N, u_next)
   CALL thomas_solver(r, u_next, N)
   u_prev = u
   u = u_next

   IF (MOD(istep,2) == 0) THEN
      DO i=1,N
         WRITE(10,'(3F12.6)') istep*DT, x(i), u(i)
      END DO
   END IF
END DO

CLOSE(10)



CLOSE(10)
```

```
CONTAINS

!-------------------------------------------------------
! Monta o lado direito da equação tridiagonal
!-------------------------------------------------------
SUBROUTINE build_rhs(u, u_prev, r, N, rhs)
  INTEGER, INTENT(IN) :: N
  REAL(8), INTENT(IN) :: u(N), u_prev(N), r
  REAL(8), INTENT(OUT) :: rhs(N)
  INTEGER :: i

  ! Condições de contorno Dirichlet
  rhs(1) = 0.0D0
  rhs(N) = 0.0D0

  ! Construção do RHS
  DO i = 2, N-1
     rhs(i) = r*u(i-1) + (2.0D0-2.0D0*r)*u(i) + r*u(i+1) - u_prev(i)
  END DO
END SUBROUTINE build_rhs

!-------------------------------------------------------
! Solver tridiagonal (Thomas algorithm)
!-------------------------------------------------------
SUBROUTINE thomas_solver(r, u, N)
  INTEGER, INTENT(IN) :: N
  REAL(8), INTENT(IN) :: r
  REAL(8), INTENT(INOUT) :: u(N)
  REAL(8), DIMENSION(N-1) :: a, c, c_star
  REAL(8), DIMENSION(N)   :: b, d_star
  INTEGER :: i

  ! Montagem das diagonais
  b = 1.0D0 + 2.0D0*r
  a = -r
  c = -r
  d_star = u

  ! Thomas algorithm forward sweep
  c_star(1) = c(1)/b(1)
  d_star(1) = d_star(1)/b(1)
  DO i = 2, N-1
     c_star(i) = c(i)/(b(i)-a(i-1)*c_star(i-1))
     d_star(i) = (d_star(i)-a(i-1)*d_star(i-1))/(b(i)-a(i-1)*c_star(i-1))
  END DO
  d_star(N) = (d_star(N)-a(N-1)*d_star(N-1))/(b(N)-a(N-1)*c_star(N-1))

  ! Back substitution
  u(N) = d_star(N)
  DO i = N-1,1,-1
     u(i) = d_star(i) - c_star(i)*u(i+1)
  END DO
END SUBROUTINE thomas_solver

END PROGRAM wave_1d_cn
```

O código apresentado implementa uma solução numérica para a equação de onda acústica

unidimensional utilizando o método de Crank–Nicolson, que combina estabilidade incondicional com um esquema implícito simples de resolver. A abordagem tridiagonal, resolvida aqui pelo algoritmo de Thomas, garante eficiência computacional, mesmo para malhas relativamente finas. As condições de contorno Dirichlet fixas mantêm o pulso confinado, permitindo observar claramente a propagação e reflexão da onda dentro do domínio. A estrutura modular do programa, com sub-rotinas para montar o lado direito da equação e resolver sistemas tridiagonais, facilita ajustes, como a inclusão de diferentes perfis iniciais ou outros tipos de condições de contorno. O arquivo de saída gerado permite a análise detalhada da evolução temporal da onda e a construção de gráficos ou animações para visualização. Apesar de ser uma implementação básica, este código serve como ponto de partida sólido para extensões, como malhas não uniformes, condições periódicas, fontes externas ou até a generalização para duas ou três dimensões. Além disso, o uso de Fortran 90 garante que o código seja eficiente e facilmente legível, com a possibilidade de adaptação a simulações mais complexas em física computacional. A experiência com este programa oferece uma compreensão prática de como métodos implícitos podem ser aplicados à propagação de ondas e estabelece uma base para estudos avançados em acústica, elasticidade e outros sistemas dinâmicos lineares.

25 Métodos Iterativos para Busca de Raízes

Encontrar raízes de funções contínuas $f(x)$ é uma tarefa central em diversos problemas de física, engenharia e matemática aplicada. Nesta seção, apresentamos três métodos clássicos: bisseção, Newton-Raphson e secante. Cada método possui vantagens e limitações, e a escolha depende da função, da disponibilidade da derivada e da precisão desejada.

25.1 Método da Bisseção

O método da bisseção é baseado no Teorema do Valor Intermediário: se $f(a) \cdot f(b) < 0$, existe pelo menos uma raiz em $[a, b]$. O algoritmo é iterativo:

1. Calcule o ponto médio $c = (a + b)/2$.
2. Avalie $f(c)$.
3. Se $|f(c)| < \epsilon$ ou $(b - a)/2 < \epsilon$, a raiz foi encontrada.
4. Caso contrário, substitua o intervalo por $[a, c]$ ou $[c, b]$ dependendo do sinal de $f(c)$ e repita.

O método é robusto e garantido, mas converge linearmente, podendo ser lento.

Código em Fortran: Bisseção para Polinômio de 4º Grau

```
      PROGRAM BISSECAO
      REAL A, B, C, EPSILON, F
      INTEGER ITER, MAX_IT

      PARAMETER (EPSILON=1.0E-6, MAX_IT=100)
      DATA A /0.0/, B /1.0/
      ITER = 0

      OPEN(1, FILE='bissecao.dat', STATUS='UNKNOWN')

 10   CONTINUE
      C = (A + B)/2.0
      F = C  4 - 3.0*C  3 - 7.0*C  2 + 27.0*C - 18.0
      WRITE(1,*) ITER, C, F

      IF (ABS(F) < EPSILON .OR. (B-A)/2.0 < EPSILON) GO TO 20
      IF ((A  4 - 3.0*A  3 - 7.0*A  2 + 27.0*A - 18.0)*F < 0.0) THEN
         B = C
      ELSE
         A = C
      END IF

      ITER = ITER + 1
      IF (ITER < MAX_IT) GO TO 10

 20   CONTINUE
      PRINT *, 'Raiz aproximada: ', C
      PRINT *, 'f(C) = ', F

      CLOSE(1)
      END
```

O arquivo `bissecao.dat` contém cada iteração, permitindo análise da convergência do método.

25.2 Método de Newton-Raphson

O método de Newton-Raphson utiliza a derivada $f'(x)$ para acelerar a convergência. A fórmula iterativa é:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Quando a aproximação inicial está próxima da raiz, a convergência é quadrática, muito mais rápida que a bisseção. Entretanto, exige o cálculo explícito de $f'(x)$ e pode divergir se o chute inicial for ruim.

Código em Fortran: Newton-Raphson para Polinômio de 4º Grau

```
      PROGRAM NEWTON
      REAL X, X_NEW, F, DF, EPSILON
      INTEGER ITER, MAX_IT
```

```
      PARAMETER (EPSILON=1.0E-6, MAX_IT=100)
      X = 0.5
      ITER = 0

      OPEN(1, FILE='newton.dat', STATUS='UNKNOWN')

10    CONTINUE
      F = X  4 - 3.0*X  3 - 7.0*X  2 + 27.0*X - 18.0
      DF = 4.0*X  3 - 9.0*X  2 - 14.0*X + 27.0
      X_NEW = X - F/DF
      WRITE(1,*) ITER, X_NEW, X_NEW  4 - 3.0*X_NEW  3 - 7.0*X_NEW  2 + 27.0*X_NEW - 18.0

      IF (ABS(X_NEW - X) < EPSILON .OR. ABS(F) < EPSILON) GO TO 20
      X = X_NEW
      ITER = ITER + 1
      IF (ITER < MAX_IT) GO TO 10

20    CONTINUE
      PRINT *, 'Raiz aproximada: ', X_NEW
      PRINT *, 'f(X) = ', X_NEW  4 - 3.0*X_NEW  3 - 7.0*X_NEW  2 + 27.0*X_NEW - 18.0

      CLOSE(1)
      END
```

O arquivo `newton.dat` mostra a convergência rápida do método, evidenciando a eficiência da abordagem quando a derivada é conhecida.

25.3 Método da Secante

O método da secante é uma variação do Newton-Raphson que não exige a derivada. Ele utiliza duas aproximações anteriores x_{n-1} e x_n para estimar a derivada:

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

A convergência é superlinear, mais rápida que a bisseção, porém ligeiramente inferior ao Newton-Raphson. É útil quando $f'(x)$ não está disponível.

Código em Fortran: Método da Secante para Polinômio de 4º Grau

```
      PROGRAM SECANTE
      REAL X0, X1, X_NEW, F0, F1, EPSILON
      INTEGER ITER, MAX_IT

      PARAMETER (EPSILON=1.0E-6, MAX_IT=100)
      X0 = 0.0
      X1 = 1.0
      ITER = 0

      OPEN(1, FILE='secante.dat', STATUS='UNKNOWN')

10    CONTINUE
      F0 = X0  4 - 3.0*X0  3 - 7.0*X0  2 + 27.0*X0 - 18.0
      F1 = X1  4 - 3.0*X1  3 - 7.0*X1  2 + 27.0*X1 - 18.0
      X_NEW = X1 - F1*(X1 - X0)/(F1 - F0)
```

```
      WRITE(1,*) ITER, X_NEW, X_NEW  4 - 3.0*X_NEW  3 - 7.0*X_NEW  2 + 27.0*X_NEW - 18.0

      IF (ABS(X_NEW - X1) < EPSILON .OR. ABS(F1) < EPSILON) GO TO 20
      X0 = X1
      X1 = X_NEW
      ITER = ITER + 1
      IF (ITER < MAX_IT) GO TO 10

 20   CONTINUE
      PRINT *, 'Raiz aproximada: ', X_NEW
      PRINT *, 'f(X) = ', X_NEW  4 - 3.0*X_NEW  3 - 7.0*X_NEW  2 + 27.0*X_NEW - 18.0

      CLOSE(1)
      END
```

O arquivo `secante.dat` permite acompanhar a evolução das aproximações. Este método é especialmente útil quando não se deseja calcular derivadas, mantendo boa convergência e simplicidade.

Resumo Comparativo:

- **Bisseção:** convergência linear, robusto, não exige derivada, garante raiz.
- **Newton-Raphson:** convergência quadrática, rápida, exige derivada, sensível à escolha inicial.
- **Secante:** convergência superlinear, rápida, não exige derivada, eficiente quando derivada é difícil.

25.4 Exemplo : Potencial Químico de um Gás de Bose Ideal 3D

Em um gás de Bose ideal tridimensional, o número de partículas N está relacionado ao potencial químico μ e à temperatura T pela seguinte integral sobre a densidade de estados $g(\epsilon)$:

$$N = \int_0^\infty d\epsilon\, g(\epsilon) \frac{1}{\exp\left[(\epsilon - \mu)/(k_B T)\right] - 1}.$$

Para partículas livres em 3D, a densidade de estados é proporcional a $\epsilon^{1/2}$:

$$g(\epsilon) = A\,\epsilon^{1/2}, \quad A = \frac{V(2m)^{3/2}}{4\pi^2\hbar^3}.$$

O problema consiste em encontrar o valor de μ que satisfaz $N = N(\mu)$ para uma dada temperatura T. Este é um problema de raiz não-linear , adequado para métodos numéricos como bisseção ou Newton-Raphson. Definimos:

$$F(\mu) = N - \int_0^\infty d\epsilon\, g(\epsilon) \frac{1}{\exp[(\epsilon - \mu)/(k_B T)] - 1} = 0.$$

A integral pode ser discretizada ou avaliada numericamente. Em seguida, aplicamos Newton-Raphson:

$$\mu_{n+1} = \mu_n - \frac{F(\mu_n)}{F'(\mu_n)}, \quad F'(\mu) = \int_0^\infty d\epsilon\, g(\epsilon) \frac{e^{(\epsilon-\mu)/k_BT}}{\left(e^{(\epsilon-\mu)/k_BT}-1\right)^2} \frac{1}{k_BT}.$$

Este método converge rapidamente se a aproximação inicial μ_0 estiver próxima da solução.

Código em Fortran: Newton-Raphson para $\mu(T, N)$

```
      PROGRAM MU_BOSE
      IMPLICIT NONE
      INTEGER, PARAMETER :: NPOINTS=1000, MAX_IT=100
      REAL*8 :: MU, MU_NEW, T, KB, N_PART, EPS
      REAL*8 :: F, DF, EPSILON
      INTEGER :: I, ITER
      REAL*8 :: EPS_INTEGRAL, EMAX, DE, E, SUM_F, SUM_DF

      ! Parâmetros físicos
      T = 1.0D0
      KB = 1.0D0
      N_PART = 1000.0D0
      MU = 0.5D0      ! Chute inicial para mu
      EPSILON = 1.0D-6
      EMAX = 20.0D0
      DE = EMAX/NPOINTS
      ITER = 0

      OPEN(10, FILE='mu_bose.dat')

10    CONTINUE
      ! Avaliação da função F(mu) e derivada F'(mu) por soma discreta
      SUM_F = 0.0D0
      SUM_DF = 0.0D0
      DO I = 1, NPOINTS
         E = I*DE
         SUM_F = SUM_F + E  0.5D0 / (EXP((E - MU)/ (KB*T)) - 1.0D0)
         SUM_DF = SUM_DF + E  0.5D0 * EXP((E - MU)/(KB*T)) / &
                  ((EXP((E - MU)/(KB*T)) - 1.0D0)  2 * KB*T)
      END DO

      F = N_PART - SUM_F * DE
      DF = -SUM_DF * DE

      ! Newton-Raphson update
      MU_NEW = MU - F/DF
      WRITE(10,*) ITER, MU, F

      IF (ABS(MU_NEW - MU) < EPSILON .OR. ITER >= MAX_IT) GO TO 20
      MU = MU_NEW
      ITER = ITER + 1
      GO TO 10

20    CONTINUE
      PRINT *, 'Potencial químico aproximado: ', MU_NEW
      CLOSE(10)
      END
```

O programa discretiza a integral de número de partículas, calcula a função $F(\mu)$ e sua derivada $F'(\mu)$, e aplica Newton-Raphson para encontrar o potencial químico. O arquivo `mu_bose.dat` registra as iterações e o valor de μ, permitindo análise da convergência e verificação da solução. Este método é eficiente e didático para o estudo de gases de Bose ideais e suas propriedades termodinâmicas.

Exemplo: busca de raízes complexas por método da secante - modos ressonantes

Considere a equação transcendente

$$f(z) = z\tan z - a = 0,$$

onde $z \in \mathbb{C}$ é a incógnita complexa e $a \in \mathbb{R}$ é um parâmetro adimensional que depende da geometria e das propriedades materiais do sistema físico. Essa equação surge naturalmente em diversos contextos de física matemática e engenharia, especialmente na determinação de modos normais e frequências próprias de cavidades, guias de onda e poços de potencial quânticos. Por exemplo, ao impor condições de contorno do tipo $\psi(0) = 0$ e $\psi'(L) = 0$ em um problema de partícula em uma caixa ou em uma camada dielétrica, a relação de quantização entre k e L conduz a uma condição do tipo $k\tan(kL) = \alpha$, onde α está associada ao contraste de impedâncias ou à barreira de potencial. Ao reescrever a equação em termos da variável adimensional $z = kL$ e do parâmetro $a = \alpha L$, obtém-se exatamente a forma acima. Quando a é real e positivo, as soluções reais z_n correspondem aos modos estacionários do sistema (frequências próprias reais), enquanto as soluções complexas, com $\operatorname{Im} z \neq 0$, descrevem modos com atenuação ou amplificação temporal, caracterizando estados quase ligados ou ressonâncias. Localizar essas raízes complexas é, portanto, essencial para determinar as frequências ressonantes e os pólos da função de espalhamento ou do determinante da matriz de contorno, permitindo uma análise completa do espectro físico do sistema.

O **método da secante** estende-se naturalmente a funções complexas: dadas duas aproximações iniciais z_0, z_1, a iteração é

$$z_{n+1} = z_n - f(z_n)\,\frac{z_n - z_{n-1}}{f(z_n) - f(z_{n-1})},$$

até que $|z_{n+1} - z_n|$ (ou $|f(z_{n+1})|$) fique abaixo de uma tolerância desejada. A secante não exige cálculo explícito da derivada e costuma convergir superlinearly quando as aproximações iniciais estão próximas da raiz.

Abaixo está um código Fortran simples que implementa a secante para funções complexas e aplica ao exemplo $f(z) = z\tan z - a$. O programa grava as iterações em arquivo para análise da convergência.

```
PROGRAM SECANT_COMPLEX
```

```
IMPLICIT NONE
INTEGER, PARAMETER :: MAX_IT = 200
INTEGER :: IT
REAL*8 :: TOL
COMPLEX*16 :: Z0, Z1, Z2, F0, F1, F2
REAL*8 :: A
OPEN(10, FILE='secant_complex.dat')

! Parâmetros do problema
A = 1.0D0           ! parâmetro adimensional 'a'
TOL = 1.0D-10
Z0 = (2.5D0, -0.5D0)   ! chute inicial 1 (complexo)
Z1 = (3.0D0, -0.2D0)   ! chute inicial 2 (complexo)

! Função f(z) = z * tan(z) - a
F0 = Z0 * CTAN(Z0) - CMPLX(A,0.0D0)
F1 = Z1 * CTAN(Z1) - CMPLX(A,0.0D0)

WRITE(10,*) 'it', 'real(z)', 'imag(z)', 'abs(f)'
WRITE(10,*) 0, REAL(Z0), AIMAG(Z0), ABS(F0)
WRITE(10,*) 1, REAL(Z1), AIMAG(Z1), ABS(F1)

DO IT = 2, MAX_IT
   IF (ABS(F1 - F0) == 0.0D0) THEN
      WRITE(*,*) 'Divisão por zero na secante. Abortando.'
      EXIT
   END IF

   ! Secant update (complex)
   Z2 = Z1 - F1 * (Z1 - Z0) / (F1 - F0)

   ! Avalia função em Z2
   F2 = Z2 * CTAN(Z2) - CMPLX(A,0.0D0)

   WRITE(10,*) IT, REAL(Z2), AIMAG(Z2), ABS(F2)

   ! Critério de convergência (distância e residuo)
   IF (ABS(Z2 - Z1) < TOL .OR. ABS(F2) < TOL) THEN
      WRITE(*,*) 'Convergencia atingida em it =', IT
      WRITE(*,*) 'raiz aproximada z =', Z2
      EXIT
   END IF

   ! Desliza as janelas
   Z0 = Z1; F0 = F1
   Z1 = Z2; F1 = F2
END DO

IF (IT >= MAX_IT) THEN
   WRITE(*,*) 'Max it reached; ultima aproximacao z =', Z2
END IF

CLOSE(10)
END
```

Ao aplicar o método da secante a funções complexas, é fundamental escolher aproximações iniciais z_0 e z_1 que estejam suficientemente próximas da raiz desejada, pois isso aumenta significativamente as chances de convergência. Em muitos casos, é útil realizar um mapeamento prévio do módulo de $f(z)$ no plano complexo, de modo a identificar regiões onde

as raízes podem estar localizadas. Deve-se ter atenção especial ao fato de que a função tangente complexa, presente neste exemplo, possui polos em $z = (\frac{\pi}{2} + k\pi)$, os quais devem ser evitados ou tratados com cuidado para prevenir divergências numéricas. Em aplicações físicas mais realistas, a função $f(z)$ pode representar determinantes de matrizes de contorno ou equações de dispersão, e o mesmo algoritmo da secante continua aplicável sem modificações conceituais. Quando a derivada analítica $f'(z)$ é de difícil obtenção ou computacionalmente cara, a secante é especialmente vantajosa por dispensar o seu cálculo. No entanto, caso $f'(z)$ seja conhecido e de avaliação simples, o método de Newton para números complexos tende a convergir mais rapidamente. Em todos os casos, é importante monitorar a diferença entre iterações sucessivas e o valor de $|f(z)|$, adotando critérios de parada adequados para garantir tanto precisão quanto estabilidade numérica.

26 Diagonalização de Matrizes pelo Método de Jacobi

O método de Jacobi é um algoritmo iterativo para diagonalizar matrizes simétricas. Ele é especialmente útil para obter autovalores e autovetores de pequenas a médias dimensões, e funciona aplicando sucessivas rotações ortogonais para eliminar os elementos fora da diagonal principal.

Consideremos a matriz simétrica $H \in \mathbb{R}^{4\times4}$, definida por:

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

Esta matriz possui a forma característica do *modelo de Anderson unidimensional*, amplamente utilizado no estudo da localização eletrônica em sistemas desordenados. O termo diagonal H_{ii} representa a energia de sítio (ou potencial local) associada ao sítio i, enquanto os elementos fora da diagonal ($H_{i,i\pm1} = -1$) descrevem o acoplamento entre sítios vizinhos — tipicamente correspondente ao termo de salto (*hopping*) entre estados localizados adjacentes. A estrutura tridiagonal reflete a conectividade linear de uma cadeia unidimensional, na qual cada sítio interage apenas com seus vizinhos imediatos.

O algoritmo de Jacobi procede da seguinte forma:

1. Identificar o maior elemento fora da diagonal, H_{pq}, em módulo.

2. Calcular o ângulo de rotação θ que zera este elemento:

$$\tan(2\theta) = \frac{2H_{pq}}{H_{qq} - H_{pp}}$$

3. Construir a matriz de rotação $R(p, q, \theta)$ e atualizar a matriz:

$$H' = R^T H R$$

4. Repetir os passos até que todos os elementos fora da diagonal sejam menores que uma tolerância ϵ.

Ao final, a diagonal de H contém os autovalores, e a composição das matrizes de rotação fornece os autovetores.

Código em Fortran: Método de Jacobi para a matriz 4×4

```
PROGRAM JACOBI_4x4
IMPLICIT NONE
INTEGER, PARAMETER :: N=4, MAXIT=100
REAL*8 :: H(N,N), A(N,N), V(N,N)
REAL*8 :: eps, t, c, s, tau, temp
INTEGER :: i, j, p, q, iter, k

eps = 1.0D-8

! Inicializa matriz H
DATA H /0.5D0, -1D0, 0D0, 0D0, &
          -1D0, -1.2D0, -1D0, 0D0, &
           0D0, -1D0, 0.7D0, -1D0, &
           0D0, 0D0, -1D0, -0.3D0/
A = H

! Inicializa autovetores como matriz identidade
V = 0.0D0
DO i = 1, N
   V(i,i) = 1.0D0
END DO

DO iter = 1, MAXIT
   ! Encontra elemento máximo fora da diagonal
   t = 0.0D0
   DO i = 1, N-1
      DO j = i+1, N
         IF (ABS(A(i,j)) > t) THEN
            t = ABS(A(i,j))
            p = i
            q = j
         END IF
      END DO
   END DO

   ! Testa convergência
   IF (t < eps) EXIT

   ! Calcula ângulo de rotação
   tau = (A(q,q)-A(p,p))/(2.0D0*A(p,q))
   IF (tau >= 0.0D0) THEN
      t = 1.0D0/(tau + SQRT(1.0D0+tau*tau))
   ELSE
      t = -1.0D0/(-tau + SQRT(1.0D0+tau*tau))
```

```
      END IF
      c = 1.0D0/SQRT(1.0D0 + t*t)
      s = t*c

      ! Atualiza matriz A
      DO k = 1, N
         IF (k /= p .AND. k /= q) THEN
            temp = A(k,p)
            A(k,p) = c*temp - s*A(k,q)
            A(p,k) = A(k,p)
            A(k,q) = s*temp + c*A(k,q)
            A(q,k) = A(k,q)
         END IF
      END DO
      temp = A(p,p)
      A(p,p) = c*c*temp - 2.0D0*s*c*A(p,q) + s*s*A(q,q)
      A(q,q) = s*s*temp + 2.0D0*s*c*A(p,q) + c*c*A(q,q)
      A(p,q) = 0.0D0
      A(q,p) = 0.0D0

      ! Atualiza autovetores
      DO k = 1, N
         temp = V(k,p)
         V(k,p) = c*temp - s*V(k,q)
         V(k,q) = s*temp + c*V(k,q)
      END DO
   END DO

   ! Imprime autovalores e autovetores
   PRINT *, 'Autovalores:'
   DO i = 1, N
      PRINT *, A(i,i)
   END DO

   PRINT *, 'Autovetores (colunas):'
   DO i = 1, N
      PRINT *, V(i,1:N)
   END DO

   END
```

O programa executa iterações até que todos os elementos fora da diagonal sejam menores que $\epsilon = 10^{-8}$, garantindo convergência. A diagonal final de A contém os autovalores da matriz H, enquanto a matriz V armazena os autovetores correspondentes nas colunas. Este código ilustra de forma clara e didática a aplicação do método de Jacobi em Fortran para matrizes pequenas, sendo facilmente adaptável a dimensões maiores.

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

Nesta seção, aplicamos dois métodos numéricos em sequência para determinar um autovalor e o autovetor correspondente de uma matriz tridiagonal simétrica H, representativa do Modelo de Anderson unidimensional:

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

O problema de autovalores consiste em resolver:

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

A estratégia adotada é:

1. Aplicar o **método da bisseção** para encontrar λ tal que $\det(H - \lambda I) = 0$, obtendo um autovalor.
2. Resolver o sistema linear $(H - \lambda I)\vec{v} = 0$ pelo **método de Thomas** para encontrar o autovetor correspondente.

Como o sistema é homogêneo, fixamos arbitrariamente $v_1 = 1$ para evitar a solução trivial.

Código em Fortran : Bisseção + Thomas para Matriz Tridiagonal

```
PROGRAM AUTOVALOR_THOMAS
  IMPLICIT NONE
  INTEGER, PARAMETER :: N=4, MAX_IT=100
  INTEGER :: ITER, I
  REAL :: EPSILON, A, B, C, F_A, F_B, F_C, LAMBDA
  REAL :: D(N), SUB, SUP
  REAL :: ALPHA(N), V(N), NORM

  EPSILON = 1.0E-8
  A = 1.5
  B = 2.0
  ITER = 0
  SUB = -1.0
  SUP = -1.0
  DATA D /0.5, -1.2, 0.7, -0.3/

  !--- Determinantes nos extremos do intervalo
  CALL DET_TRIDIAG(N, D, SUB, SUP, A, F_A)
  CALL DET_TRIDIAG(N, D, SUB, SUP, B, F_B)

  !--- Bisseção para encontrar o autovalor
10 CONTINUE
  C = 0.5*(A + B)
  CALL DET_TRIDIAG(N, D, SUB, SUP, C, F_C)

  IF (F_A*F_C < 0.0) THEN
     B = C
     F_B = F_C
  ELSE
     A = C
     F_A = F_C
  ENDIF
```

```
  ITER = ITER + 1
  IF ((ABS(B-A) > EPSILON) .AND. (ITER < MAX_IT)) GOTO 10

  LAMBDA = 0.5*(A + B)
  PRINT *, 'Autovalor aproximado: ', LAMBDA

  !--- Monta a diagonal de (H - lambda*I)
  DO I = 1, N
     ALPHA(I) = D(I) - LAMBDA
  END DO

  !--- Resolve o sistema homogêneo (H - lambda I) v = 0
  !     Fixamos v(1) = 1 e calculamos v(2..N) por recorrência:
  !       a1*v1 + sup*v2 = 0        => v2 = -a1/sup * v1
  !       sub*v_{i-1} + a_i*v_i + sup*v_{i+1} = 0
  !       => v_{i+1} = -(sub*v_{i-1} + a_i*v_i)/sup
  CALL SOLVE_NULL_TRIDIAG(N, ALPHA, SUB, SUP, V)

  !--- Normaliza o autovetor
  NORM = SQRT(SUM(V 2))
  IF (NORM /= 0.0) V = V / NORM

  PRINT *, 'Autovetor normalizado:'
  DO I = 1, N
     PRINT '(I2,2X,F12.8)', I, V(I)
  END DO

CONTAINS

  SUBROUTINE DET_TRIDIAG(N, D, SUB, SUP, LAMBDA, DET)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: N
    REAL, INTENT(IN) :: D(N), SUB, SUP, LAMBDA
    REAL, INTENT(OUT) :: DET
    REAL :: B(N)
    INTEGER :: I

    ! Eliminação recursiva para o determinante de tridiagonal (H - lambda I).
    ! B(1) = a1
    B(1) = D(1) - LAMBDA
    DO I = 2, N
       ! B(i) = a_i - sub*sup/B(i-1)
       B(I) = (D(I) - LAMBDA) - (SUB*SUP)/B(I-1)
    END DO

    ! Determinante é produto dos pivôs B(i)
    DET = 1.0
    DO I = 1, N
       DET = DET * B(I)
    END DO
  END SUBROUTINE DET_TRIDIAG

  SUBROUTINE SOLVE_NULL_TRIDIAG(N, A, SUB, SUP, V)
    IMPLICIT NONE
    INTEGER, INTENT(IN) :: N
    REAL, INTENT(IN) :: A(N)
    REAL, INTENT(IN) :: SUB, SUP
    REAL, INTENT(OUT) :: V(N)
    INTEGER :: i
```

```
    ! Trata casos degenerados
    IF (ABS(SUP) < 1.0E-14) THEN
       ! Não podemos propagar se SUP ~ 0
       V = 0.0
       V(1) = 1.0
       RETURN
    END IF

    ! Fixamos v(1) = 1.0 (ou qualquer não-nulo)
    V(1) = 1.0

    ! Calcula v(2) a partir da primeira equação: a1*v1 + sup*v2 = 0
    V(2) = -A(1)/SUP * V(1)

    ! Recorrência para obter v(3..N)
    DO i = 2, N-1
       V(i+1) = -(SUB*V(i-1) + A(i)*V(i)) / SUP
    END DO

    ! É possível que os valores cresçam; normalização é feita fora
  END SUBROUTINE SOLVE_NULL_TRIDIAG

END PROGRAM AUTOVALOR_THOMAS
```

Este programa encontra o autovalor $\lambda \approx 1.69$ via bisseção e resolve o sistema linear tridiagonal com o método de Thomas para obter o autovetor correspondente. Ao final, o autovetor é normalizado, e pequenas diferenças podem ocorrer devido a erros numéricos da bisseção e do método de Thomas.

28 Autovetor e Autovalor: Método da Potência

O *método da potência* é um procedimento iterativo para estimar o maior autovalor (em módulo) e seu autovetor correspondente de uma matriz simétrica ou Hermitiana $H \in \mathbb{R}^{N\times N}$.

Dada uma matriz H simétrica com autovalores ordenados por módulo $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_N|$, e um vetor inicial $\mathbf{v}^{(0)}$ com projeção não nula no autovetor dominante $\mathbf{u}_1$, as iterações

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

convergem para $\mathbf{u}_1$. O autovalor dominante é estimado pelo quociente de Rayleigh:

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

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

Exemplo: Matriz Simétrica 3×3

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

O maior autovalor é $\lambda_{\max} = 4$ com autovetor dominante proporcional a $(1, 2, 1)^T$.

Código em Fortran : Método da Potência

```
      PROGRAM POTENCIA
      IMPLICIT NONE
      INTEGER N, I, J, MAX_IT
      PARAMETER (N=3, MAX_IT=1000)
      REAL H(N,N), V(N), W(N), LAMBDA, LAMBDA_OLD, TOL, NORM, SUM

      TOL = 1.0E-10
C     --- Matriz H ---
      DATA H /2.0,1.0,0.0, 1.0,3.0,1.0, 0.0,1.0,2.0/

C     --- Vetor inicial normalizado ---
      DO I=1,N
         V(I) = 1.0
      END DO
      NORM = 0.0
      DO I=1,N
         NORM = NORM + V(I) 2
      END DO
      NORM = SQRT(NORM)
      DO I=1,N
         V(I) = V(I)/NORM
      END DO

      LAMBDA_OLD = 0.0

      DO I=1,MAX_IT
C        --- Multiplicação matriz-vetor W = H*V ---
         DO J=1,N
            SUM = 0.0
            DO K=1,N
               SUM = SUM + H(J,K)*V(K)
            END DO
            W(J) = SUM
         END DO

C        --- Quociente de Rayleigh ---
         LAMBDA = 0.0
         DO J=1,N
            LAMBDA = LAMBDA + V(J)*W(J)
         END DO

C        --- Normaliza W para próxima iteração ---
         NORM = 0.0
         DO J=1,N
            NORM = NORM + W(J) 2
         END DO
         NORM = SQRT(NORM)
         DO J=1,N
```

```
            V(J) = W(J)/NORM
         END DO

C        --- Critério de parada ---
         IF (ABS(LAMBDA - LAMBDA_OLD) .LT. TOL) EXIT
         LAMBDA_OLD = LAMBDA
      END DO

      PRINT *, 'Autovalor dominante ~ ', LAMBDA
      PRINT *, 'Autovetor correspondente ~ '
      DO I=1,N
         PRINT *, V(I)
      END DO

C     --- Verificação Hv ~ lambda*v ---
      PRINT *, 'Teste Hv e lambda*v:'
      DO I=1,N
         SUM = 0.0
         DO J=1,N
            SUM = SUM + H(I,J)*V(J)
         END DO
         PRINT *, SUM, LAMBDA*V(I)
      END DO

      END
```

O programa implementa o método da potência em Fortran básico:

- Inicializa o vetor $\mathbf{v}^{(0)}$ e normaliza.
- Itera $w = Hv$, estima o autovalor pelo quociente de Rayleigh e normaliza w para gerar a próxima iteração.
- Para quando a diferença entre autovalores consecutivos é menor que a tolerância.
- Ao final, imprime o autovalor dominante e o autovetor correspondente.
- Realiza verificação numérica comparando Hv com λv.

Exemplo: Matriz Simétrica 4×4

$$H_4 = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 1 & 3 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}, \quad \mathbf{v}^{(0)} = \frac{1}{2}(1,1,1,1)^T$$

O maior autovalor é $\lambda_{\max} \approx 4.41421356$. O código acima funciona para $N = 4$ com pequenas adaptações no parâmetro `N` e nos dados da matriz.

Em resumo, o método da potência é uma ferramenta simples e eficiente para estimar o maior autovalor (em módulo) de uma matriz simétrica ou Hermitiana, assim como o autovetor correspondente. Apesar de sua simplicidade, ele apresenta convergência rápida quando o autovalor dominante é bem separado dos demais. Para matrizes de ordem moderada e com

estrutura simples (como tridiagonais), o método fornece resultados precisos com poucas iterações. Além disso, a implementação em Fortran , mesmo básica, ilustra claramente os passos fundamentais: multiplicação matriz-vetor, normalização, cálculo do quociente de Rayleigh e verificação numérica da consistência da solução. Esse formalismo serve de base para métodos mais avançados de autovalores, incluindo variantes para autovalores múltiplos e métodos iterativos refinados em mecânica computacional e física quântica.

29 Método da Potência Inversa com Deslocamento — implementação com Thomas

O método da **potência inversa com deslocamento** (inverse power method with shift) é uma técnica eficiente para obter o autovalor mais próximo de um valor σ (o *shift*) e seu autovetor associado. Em cada iteração resolvemos um sistema linear

$$(H - \sigma I)\, y_k = x_{k-1},$$

normalizamos y_k para obter x_k e (opcionalmente) extraímos o autovalor por quociente de Rayleigh $\lambda_k = \dfrac{x_k^T H x_k}{x_k^T x_k}$. Quando σ está próximo de um autovalor real, a iteração converge rapidamente para o autovetor correspondente.

A vantagem para matrizes tridiagonais é que o sistema $(H - \sigma I)y = x$ pode ser resolvido eficientemente com o **método de Thomas** (eliminação otimizada para tridiagonais, $O(N)$). A seguir apresentamos um código Fortran que implementa exatamente isso para a matriz 4×4 utilizada anteriormente:

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

O programa usa deslocamento $\sigma = 1.7$, resolve o sistema tridiagonal em cada passo com Thomas, executa a iteração inversa e imprime o autovalor estimado (por quociente de Rayleigh) e o autovetor normalizado ao final.

```
program inverse_power_shift_thomas
  implicit none
  integer, parameter :: N = 4, MAX_IT = 1000
  integer :: k, i, iter
  real(8) :: sigma, tol, lambda_old, lambda, diff
  real(8) :: D(N), sub, sup
  real(8) :: diag(N), a_sub(N-1), c_sup(N-1)
  real(8) :: x(N), y(N), Hx(N)
  real(8) :: normx, normy, dot1, dot2

  ! --- matriz H (diagonal D e sub/sup diagonais)
```

```
  D = (/ 0.5d0, -1.2d0, 0.7d0, -0.3d0 /)
  sub = -1.0d0
  sup = -1.0d0

  ! --- parâmetros do método
  sigma = 1.7d0         ! deslocamento
  tol = 1.0d-10         ! tolerância para convergência
  lambda_old = 0.0d0

  ! --- inicializa vetor x (chute inicial)
  do i = 1, N
     x(i) = 1.0d0
  end do
  ! normaliza x
  normx = sqrt(sum(x 2))
  x = x / normx

  ! --- monta a matriz tridiagonal (H - sigma I) componentes que serão usadas por Thomas
  do i = 1, N
     diag(i) = D(i) - sigma    ! diagonal de (H - sigma I)
  end do
  do i = 1, N-1
     a_sub(i) = sub            ! subdiagonal (i+1,i)
     c_sup(i) = sup            ! superdiagonal (i,i+1)
  end do

  ! --- iteração da potência inversa com deslocamento
  do iter = 1, MAX_IT
     ! resolve (H - sigma I) y = x  usando Thomas (tridiagonal)
     call thomas_solve(N, a_sub, diag, c_sup, x, y)

     ! normaliza y para obter o próximo x
     normy = sqrt(sum(y 2))
     if (normy == 0.0d0) then
        print *, 'Erro: vetor resolvido nulo (possível singularidade com sigma).'
        stop
     end if
     x = y / normy

     ! estimativa do autovalor via quociente de Rayleigh
     ! Hx = H * x  (produto tridiagonal)
     call tridiag_matvec(N, D, sub, sup, x, Hx)
     lambda = dot_product(x, Hx) / dot_product(x, x)

     diff = abs(lambda - lambda_old)
     if (diff < tol) exit
     lambda_old = lambda
  end do

  ! --- resultado final
  print *, 'Shift (sigma) = ', sigma
  print *, 'Autovalor estimado (Rayleigh) = ', lambda
  print *, 'Iteracoes = ', iter
  print *, 'Autovetor normalizado (x):'
  do i = 1, N
     print '(I2,2X,F12.8)', i, x(i)
  end do

contains
```

```
subroutine thomas_solve(n, a, b, c, rhs, sol)
  ! Resolve sistema tridiagonal A sol = rhs, com A = diag(b) + sub(a) + sup(c)
  implicit none
  integer, intent(in) :: n
  real(8), intent(in) :: a(n-1), b(n), c(n-1), rhs(n)
  real(8), intent(out) :: sol(n)
  real(8) :: cp(n-1), dp(n)
  integer :: i
  real(8) :: denom

  ! Forward elimination
  if (abs(b(1)) < 1.0d-14) then
     print *, 'Pivot muito pequeno em Thomas (b1).'
     stop
  end if
  cp(1) = c(1)/b(1)
  dp(1) = rhs(1)/b(1)

  do i = 2, n-1
     denom = b(i) - a(i-1)*cp(i-1)
     if (abs(denom) < 1.0d-14) then
        print *, 'Denominador muito pequeno em Thomas, i=', i
        stop
     end if
     cp(i) = c(i)/denom
     dp(i) = (rhs(i) - a(i-1)*dp(i-1))/denom
  end do

  denom = b(n) - a(n-1)*cp(n-1)
  if (abs(denom) < 1.0d-14) then
     print *, 'Denominador muito pequeno em Thomas na última equação.'
     stop
  end if
  dp(n) = (rhs(n) - a(n-1)*dp(n-1))/denom

  ! Back substitution
  sol(n) = dp(n)
  do i = n-1, 1, -1
     sol(i) = dp(i) - cp(i)*sol(i+1)
  end do
end subroutine thomas_solve

subroutine tridiag_matvec(n, Ddiag, subd, supd, vec, out)
  implicit none
  integer, intent(in) :: n
  real(8), intent(in) :: Ddiag(n), subd, supd, vec(n)
  real(8), intent(out) :: out(n)
  integer :: i

  ! Produto tridiagonal H * vec, H has diag Ddiag, sub = subd, sup = supd
  if (n == 1) then
     out(1) = Ddiag(1) * vec(1)
     return
  end if
  out(1) = Ddiag(1)*vec(1) + supd*vec(2)
  do i = 2, n-1
     out(i) = subd*vec(i-1) + Ddiag(i)*vec(i) + supd*vec(i+1)
  end do
  out(n) = subd*vec(n-1) + Ddiag(n)*vec(n)
end subroutine tridiag_matvec
```

```
end program inverse_power_shift_thomas
```

Comentários e observações:

- O `thomas_solve` implementa a versão clássica do algoritmo de Thomas (eliminação progressiva com armazenamento de coeficientes modificados).
- O sistema resolvido em cada iteração é $(H - \sigma I)y = x$. Se σ estiver extremamente próximo de um autovalor verdadeiro, a matriz pode ficar quase singular e os pivôs na eliminação podem ser muito pequenos — o código detecta isso e termina.
- A estimativa do autovalor é feita através do *quociente de Rayleigh* $\lambda \approx (x^T Hx)/(x^T x)$, que converge ao autovalor correspondente ao autovetor x.
- Você pode ajustar σ (shift), a tolerância `tol` e o chute inicial para controlar convergência e robustez.

30 Autovalor e Autovetor: método da potência inversa com deslocamento aplicado ao Modelo de Anderson em 2D

O cálculo de autovalores e autovetores continua sendo central em sistemas quânticos discretos. Em Hamiltonianos 2D, como o do Modelo de Anderson, as matrizes não são tridiagonais, tornando métodos diretos de diagonalização custosos. O *método da potência inversa com deslocamento* permite encontrar autovalores próximos de um chute inicial μ mesmo em matrizes densas, transformando o problema $Hv = \lambda v$ em $(H - \mu I)^{-1}v = \tilde{\lambda}v$.
Para exemplificar, consideramos uma rede $L \times L$ de sítios, com energia aleatória $\varepsilon_{i,j}$ e hopping t entre vizinhos. O Hamiltoniano H pode ser representado matricialmente como:

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

onde $\varepsilon_{i,j} \in [-W/2, W/2]$ e t é o termo de hopping. Para ilustrar explicitamente a estrutura do Hamiltoniano em duas dimensões, consideremos $L = 3$, de modo que temos $N = L^2 = 9$ sítios. O Hamiltoniano H pode ser escrito como uma matriz 9×9, onde os elementos diagonais contêm as energias aleatórias $\varepsilon_{i,j}$ e os elementos fora da diagonal representam o hopping t entre vizinhos imediatos:

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

Esta forma mostra claramente como cada sítio está conectado apenas aos vizinhos imediatos e como a desordem local é representada pelos elementos diagonais da matriz. A implementação completa do método da potência inversa deslocado e o método de Gauss para esta matriz H em Fortran básico pode ser feita da seguinte forma:

```
program Anderson2D_PotenciaInversa
  implicit none

  !==== Declarações de variáveis ====
  integer :: L, N, i, j, k, it, max_it
   integer, allocatable :: seed_vals(:)
  real(8), allocatable :: H(:,:), A_shifted(:,:)
  real(8), allocatable :: v(:), wvec(:), Hv(:)
  real(8) :: mu, lambda_val, lambda_old_val, W_val, hopping_val, tol_val
  real(8) :: rand_num_val, dp,xx2
  integer :: seed_size_val
    integer :: n22, i22

  !==== Parâmetros ====
  W_val = 1.0d0
  hopping_val = 1.0d0
  tol_val = 1.0d-10
  max_it = 1000

  !==== Leitura tamanho rede ====
  print *, 'Digite L (numero de sitios na aresta):'
  read(*,*) L
  if (L <= 0) then
     print *, 'L invalido. Encerrando.'
     stop
  end if
  N = L*L

  !==== Alocação ====
  allocate(H(N,N), A_shifted(N,N))
  allocate(v(N), wvec(N), Hv(N))

  ! -------------------------
  ! Inicializa RNG com semente fixa
  ! -------------------------
  call random_seed(size = n22)
  allocate(seed_vals(n22))
```

```
do i22 = 1, n22
   seed_vals(i22) = 12345
end do
call random_seed(put=seed_vals)

!==== Inicializa Hamiltoniano ====
H = 0.0d0
do i = 1, N
   call random_number(xx2)
   H(i,i) = W_val*(xx2 - 0.5d0)
end do

!==== Termos hopping ====
do i = 0, L-1
   do j = 0, L-1
      k = i*L + j + 1
      if (j+1 < L) then
         H(k,k+1) = hopping_val
         H(k+1,k) = hopping_val
      end if
      if (i+1 < L) then
         H(k,k+L) = hopping_val
         H(k+L,k) = hopping_val
      end if
   end do
end do

!==== Vetor inicial e normaliza ====
do i = 1, N
   v(i) = 1.0d0
end do
call normalize_vec(N, v)

!==== Deslocamento ====
mu = 1.42d0
A_shifted = H
do i = 1, N
   A_shifted(i,i) = A_shifted(i,i) - mu
end do

lambda_val = 0.0d0
lambda_old_val = 0.0d0

!==== Iteração potência inversa ====
do it = 1, max_it
   call gauss_solve(N, A_shifted, v, wvec)
   call normalize_vec(N, wvec)
   call matvec_mul(N, H, wvec, Hv)
   dp = sum(wvec*Hv)/sum(wvec*wvec)
   lambda_val = dp

   if (abs(lambda_val - lambda_old_val) < tol_val) exit
   lambda_old_val = lambda_val
   v = wvec
end do

!==== Resultados ====
print *, 'Shift (mu) = ', mu
print *, 'Autovalor estimado = ', lambda_val
print *, 'Iteracoes = ', it
```

```
  print *, 'Autovetor normalizado:'
  do i = 1, N
     write(*,'(I4,2X,F14.8)') i, v(i)
  end do

  !==== Teste Hv ~ lambda*v ====
  call matvec_mul(N,H,v,Hv)
  print *, 'Teste Hv  vs  lambda*v:'
  do i = 1, N
     write(*,'(2(F14.8))') Hv(i), lambda_val*v(i)
  end do

  !==== Libera memória ====
  deallocate(H,A_shifted,v,wvec,Hv,seed_vals)

contains

  subroutine normalize_vec(n,x)
    integer, intent(in) :: n
    real(8), intent(inout) :: x(n)
    x = x / sqrt(sum(x 2))
  end subroutine normalize_vec

  subroutine matvec_mul(n,A,x,b)
    integer, intent(in) :: n
    real(8), intent(in) :: A(n,n), x(n)
    real(8), intent(out) :: b(n)
    integer :: i,j
    b = 0.0d0
    do i = 1, n
       do j = 1, n
          b(i) = b(i) + A(i,j)*x(j)
       end do
    end do
  end subroutine matvec_mul

  subroutine gauss_solve(n,A_in,rhs_in,sol)
    integer, intent(in) :: n
    real(8), intent(in) :: A_in(n,n), rhs_in(n)
    real(8), intent(out) :: sol(n)
    real(8), allocatable :: M(:,:)
    integer :: i,j,k,piv
    real(8) :: maxval,factor,pivotval

    allocate(M(n,n+1))
    do i = 1, n
       do j = 1, n
          M(i,j) = A_in(i,j)
       end do
       M(i,n+1) = rhs_in(i)
    end do

    ! eliminação com pivot parcial
    do k = 1, n-1
       piv = k
       maxval = abs(M(k,k))
       do i = k+1, n
          if (abs(M(i,k)) > maxval) then
             maxval = abs(M(i,k))
             piv = i
```

```
         end if
      end do
      if (piv /= k) then
         do j = k, n+1
            factor = M(k,j); M(k,j) = M(piv,j); M(piv,j) = factor
         end do
      end if

      pivotval = M(k,k)
      if (abs(pivotval) < 1.0d-14) then
         print *, 'Matriz quase singular!'
         stop
      end if

      do i = k+1, n
         factor = M(i,k)/pivotval
         do j = k, n+1
            M(i,j) = M(i,j) - factor*M(k,j)
         end do
      end do
   end do

   ! substituição regressiva
   sol(n) = M(n,n+1)/M(n,n)
   do i = n-1,1,-1
      sol(i) = M(i,n+1)
      do j = i+1, n
         sol(i) = sol(i) - M(i,j)*sol(j)
      end do
      sol(i) = sol(i)/M(i,i)
   end do

   deallocate(M)
 end subroutine gauss_solve

end program Anderson2D_PotenciaInversa
```

Este programa implementa a iteração da potência inversa com deslocamento para o cálculo do autovalor de um Hamiltoniano 2D de Anderson, incluindo desordem diagonal e termos de hopping entre vizinhos. A matriz de Hamiltoniano é construída para uma rede quadrada de tamanho $L \times L$, com elementos diagonais gerados a partir de uma sequência de números aleatórios fixa, garantindo reprodutibilidade dos resultados. O vetor inicial é uniforme e normalizado, servindo como chute inicial para o método iterativo. Em cada iteração, resolve-se o sistema linear $(H - \mu I)w = v$ por eliminação de Gauss com pivotamento parcial, seguido de normalização do vetor resultante. O autovalor aproximado é calculado pelo quociente de Rayleigh , garantindo convergência eficiente. O código utiliza subrotinas claras para normalização , multiplicação matriz-vetor e solução de sistemas lineares, mantendo modularidade e legibilidade. A implementação da semente fixa do gerador de números aleatórios permite que o Hamiltoniano e, consequentemente, os autovalores e autovetores, sejam idênticos em execuções repetidas. Por fim, o programa imprime o autovalor estimado, o número de iterações necessárias, o autovetor normalizado e um teste de consistência

verificando $Hv \approx \lambda v$, tornando-o uma ferramenta didática e confiável para estudo de sistemas desordenados em duas dimensões.

O Hamiltoniano de Anderson construído para uma rede 2D pode ser diagonalizado utilizando o método de Jacobi, que é um procedimento clássico para matrizes simétricas. Esse método consiste em aplicar rotações sucessivas sobre pares de linhas e colunas da matriz, de forma a eliminar gradualmente os elementos fora da diagonal. Cada rotação é escolhida para zerar o maior elemento fora da diagonal, garantindo estabilidade numérica e preservando os autovetores. Ao final do processo, a matriz (H) torna-se aproximadamente diagonal, com os elementos na diagonal representando os autovalores, enquanto a matriz de autovetores (V) contém as correspondentes direções próprias. Apesar de conceitualmente simples e intuitivo, o método de Jacobi possui limitações importantes: seu custo computacional cresce rapidamente com o tamanho da matriz, tornando-o impraticável para redes grandes. Além disso, é necessário armazenar toda a matriz em memória, o que pode ser proibitivo para sistemas de dimensão elevada. Por essas razões, ele é indicado principalmente para redes pequenas, servindo como uma ferramenta didática e de validação de resultados. Para matrizes maiores, métodos iterativos, como a potência inversa com deslocamento, são mais eficientes, pois permitem focar apenas nos autovalores de interesse sem a necessidade de diagonalizar a matriz completa. Uma implementação do método de Jacobi para diagonalizar a matriz de Anderson 2d pode ser encontrada a seguir:

```
program JacobiAnderson2D
  implicit none

  ! -------------------------
  ! Declaração de variáveis
  ! -------------------------
  integer :: L, N, i, j, k, p, q, iter, max_it
  integer :: n22, i22
  integer, allocatable :: seed_vals(:)
  real(8), allocatable :: H(:,:), V(:,:)
  real(8) :: rand_num, W, hopping
  real(8) :: tol, offmax, a_pp, a_qq, a_pq, tau, t, c, s, temp
  real(8) :: theta,xx2

  ! -------------------------
  ! Parâmetros
  ! -------------------------
  W = 1.0d0
  hopping = 1.0d0
  tol = 1.0d-8
  max_it = 10000

  ! -------------------------
  ! Entrada do tamanho L
  ! -------------------------
  print *, 'Digite L (numero de sitios na aresta, por ex. 6):'
```

```
 read(*,*) L
 if (L < 1) then
    print *, 'L invalido.'
    stop
 end if

 N = L*L

 allocate(H(N,N), V(N,N))

 ! -------------------------
 ! Inicializa RNG com semente fixa
 ! -------------------------
 call random_seed(size = n22)
 allocate(seed_vals(n22))
 do i22 = 1, n22
    seed_vals(i22) = 12345
 end do
 call random_seed(put=seed_vals)


 ! monta Hamiltoniano H (zero inicialmente)
 H = 0.0d0
 do i = 1, N
call random_number(xx2)
    H(i,i) = W*(xx2 - 0.5d0)   ! termo diagonal aleatorio em [-W/2, W/2]
 end do

 ! adiciona hopping entre vizinhos (2D - indexacao row-major)
 do i = 0, L-1
    do j = 0, L-1
       k = i*L + j + 1
       if (j+1 < L) then
          ! vizinho à direita
          H(k, k+1) = hopping
          H(k+1, k) = hopping
       end if
       if (i+1 < L) then
          ! vizinho abaixo
          H(k, k+L) = hopping
          H(k+L, k) = hopping
       end if
    end do
 end do

 ! inicializa V (autovetores) como identidade
 V = 0.0d0
 do i = 1, N
    V(i,i) = 1.0d0
 end do

 ! -------------------------
 ! método de Jacobi (clássico)
 ! -------------------------
 iter = 0
 do
    ! encontra maior elemento fora da diagonal em módulo
    offmax = 0.0d0
    p = 1; q = 2
    do i = 1, N-1
```

```
   do j = i+1, N
      if (abs(H(i,j)) > offmax) then
         offmax = abs(H(i,j))
         p = i
         q = j
      end if
   end do
end do

if (offmax < tol) exit
if (iter >= max_it) then
   print *, 'Atingiu o maximo de iteracoes sem convergencia.'
   exit
end if

! calcula parametros da rotacao para zerar H(p,q)
a_pp = H(p,p)
a_qq = H(q,q)
a_pq = H(p,q)

if (a_pp == a_qq) then
   t = 1.0d0
else
   tau = (a_qq - a_pp) / (2.0d0 * a_pq)
   ! escolha de t de forma numericamente estável
   if (tau >= 0.0d0) then
      t = 1.0d0 / (tau + sqrt(1.0d0 + tau*tau))
   else
      t = -1.0d0 / (-tau + sqrt(1.0d0 + tau*tau))
   end if
end if
c = 1.0d0 / sqrt(1.0d0 + t*t)
s = t * c

! atualiza elementos da matriz H
! diagonal p e q
H(p,p) = c*c*a_pp - 2.0d0*s*c*a_pq + s*s*a_qq
H(q,q) = s*s*a_pp + 2.0d0*s*c*a_pq + c*c*a_qq
H(p,q) = 0.0d0
H(q,p) = 0.0d0

! atualiza linhas/colunas p e q (k != p,q)
do k = 1, N
   if ((k /= p) .and. (k /= q)) then
      temp = H(k,p)
      H(k,p) = c*temp - s*H(k,q)
      H(p,k) = H(k,p)
      H(k,q) = s*temp + c*H(k,q)
      H(q,k) = H(k,q)
   end if
end do

! atualiza a matriz de autovetores V (colunas)
do k = 1, N
   temp = V(k,p)
   V(k,p) = c*temp - s*V(k,q)
   V(k,q) = s*temp + c*V(k,q)
end do

iter = iter + 1
```

```
  end do

  ! imprime autovalores (diagonal de H) e autovetores (colunas de V)
  print *, 'Convergencia Jacobi em iter = ', iter
  print *, 'Autovalores (aproximados):'
  do i = 1, N
     print '(I3,2X,F12.8)', i, H(i,i)
  end do

  print *, 'Alguns autovetores (colunas de V) - cada linha corresponde ao componente i em todas as colunas:'
  ! imprime V linha por linha (componentes)
  do i = 1, N
     write(*,'(1I3,2X,4(1X,F12.8))') i, (V(i,j), j=1,min(N,4))   ! mostra primeiras 4 colunas por legibilidade
  end do

  ! libera memória
  deallocate(H)
  deallocate(V)

end program JacobiAnderson2D
```

O código acima implementa a diagonalização do Hamiltoniano de Anderson em duas dimensões utilizando o método de Jacobi. Esse método clássico atua em matrizes simétricas aplicando rotações sucessivas sobre pares de linhas e colunas, de modo a zerar progressivamente os elementos fora da diagonal. Cada rotação é escolhida para eliminar o maior elemento fora da diagonal, preservando a simetria da matriz e garantindo estabilidade numérica. Ao final do processo, a matriz (H) torna-se aproximadamente diagonal, cujos elementos representam os autovalores, enquanto a matriz (V) armazena os autovetores correspondentes. Apesar de didático e fácil de implementar, o método de Jacobi é limitado a matrizes pequenas, pois seu custo computacional cresce rapidamente com o tamanho da rede. Além disso, exige armazenar toda a matriz em memória, o que pode ser proibitivo para sistemas maiores. Para Hamiltonianos de maior dimensão, métodos iterativos como a potência inversa com deslocamento são mais adequados, pois permitem focar apenas nos autovalores de interesse, reduzindo tanto o tempo de cálculo quanto o consumo de memória. Portanto, o Jacobi é uma excelente ferramenta pedagógica e de verificação, mas não substitui técnicas mais eficientes para sistemas extensos.

31 Resumo do Curso de Física Computacional em Fortran

Este curso de Física Computacional em Fortran cumpriu seu objetivo principal de fornecer uma introdução prática e progressiva às ferramentas numéricas essenciais utilizadas na física moderna.

Na **Parte 1**, o foco foi estabelecer uma base sólida, abrangendo desde a sintaxe fundamental da linguagem Fortran (como estruturas de repetição, manipulação de *arrays* e entrada/saída de dados) até técnicas de cálculo elementar, como derivação, integração e análise estatística (regressão linear e desvio padrão).

A **Parte 2** levou o estudante a domínios mais complexos, como a solução de equações diferenciais (RK4, Velocity-Verlet), simulações estocásticas (Euler-Maruyama), Transformadas de Fourier e, crucialmente, métodos de autovalores e autovetores (Jacobi, Potência Inversa). O sucesso na implementação desses métodos avançados dependia não apenas do domínio técnico, mas da compreensão profunda de seus fundamentos e limitações físicas.

A Dualidade Fortran: F77 para Fundamentos, F90 para Complexidade

Um pilar metodológico central destas notas foi a abordagem pragmática quanto aos padrões da linguagem Fortran. Optou-se, muitas vezes, por uma transição entre o **Fortran 77 (F77)** e o **Fortran 90/95 (F90+)**.

- **Códigos Fundamentais em F77 :** Para algoritmos introdutórios e estruturas de controle básicas, utilizou-se um estilo de codificação que remete ao Fortran 77 (como a sintaxe de *loops* baseada em rótulos, em alguns exemplos, ou estruturas simples de I/O). Esta escolha visa familiarizar o estudante com a vasta base de códigos científicos legados e reforçar a clareza da lógica computacional em sua forma mais concisa.
- **Códigos Complexos Alavancando F90:** Nos problemas mais complexos, como os métodos de diagonalização (e.g., Jacobi com alocação dinâmica), integração Monte Carlo de alta dimensão e manipulação avançada de sistemas lineares, a ênfase foi integralmente nas potencialidades do Fortran 90/95.

O uso do **Fortran 90/95** permitiu empregar ferramentas que são ineficientes ou impossíveis no F77:

1. **Módulos (`MODULE` e `USE`):** Essenciais para estruturar grandes projetos, modularizar sub-rotinas e compartilhar variáveis de forma segura.
2. **Alocação Dinâmica (`ALLOCATABLE` e `ALLOCATE`):** Crucial para trabalhar com matrizes cujo tamanho é determinado durante a execução, como em simulações de redes variáveis ou problemas de autovalores em sistemas grandes.

3. **Tipos de Dados e Precisão Avançados:** O uso de `REAL(8)` e `COMPLEX(8)` garantiu a precisão dupla necessária para a estabilidade e rigor de simulações físicas longas.

4. **Estruturas Modernas de Controle:** O uso de `END DO` e `END IF` melhora a legibilidade e a manutenção dos códigos.

Este material fornece uma base para o desenvolvimento de simulações numéricas em Fortran, mas aplicações avançadas exigem exploração de técnicas de otimização, como vetorização e uso de rotinas intrínsecas, além de paralelismo via OpenMP ou MPI para eficiência em *clusters* e *supercomputadores*. Recomenda-se também o emprego de bibliotecas científicas otimizadas, como LAPACK e FFTW, para substituir implementações didáticas de transformadas e diagonalizações, e a integração com linguagens de *script*, como Python, para pós-processamento e análise de dados, mantendo o Fortran restrito ao cálculo intensivo.