COC473 - Lista 2

Pedro Maciel Xavier
116023847

22 de setembro de 2019

Questão 1.:

Aqui está o trecho do código que implementa a função para o cálculo do maior autovalor e seu respectivo autovetor través do método das Potências (”*Power Method*”). As funções auxiliares se encontram no código completo, no apêndice.

Algoritmo 1

```fortran
function power_method(A, n, x, l) result (ok)
    implicit none
    integer :: n
    integer :: k = 0

    double precision :: A(n, n)
    double precision :: x(n)
    double precision :: l, ll

    logical :: ok

!   Inicializa vetor aleatório e define primeira componente como '1'
    x(:) = rand_vector(n)
    x(1) = 1.0D0

    l = 0.0D0

    do while (k < MAX_ITER)
        ll = l

        x(:) = matmul(A, x)

!       Obtém autovalor
        l = x(1)

!       Obtém autovetor
        x(:) = x(:) / l

!       Verifica se a tolerância foi atendida
        if (DABS((l - ll) / l) < TOL) then
            ok = .TRUE.
            return
        else
            k = k + 1
            continue
        end if
    end do
    ok = .FALSE.
    return
end function
```

Questão 2.:

O cálculo dos autovalores pelo método de Jacobi está implementado no código abaixo. As funções auxiliares, como a que calcula a matriz de rotação de *Givens* e a que gera uma matriz identidade estão no código completo, nos apêndices.

Algoritmo 2

```fortran
function Jacobi_eigen(A, n, L, X) result (ok)
    implicit none
    integer :: n, i, j, u, v
    integer :: k = 0

    double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
    double precision :: y, z

    logical :: ok

    X(:, :) = id_matrix(n)
    L(:, :) = A(:, :)

    do while (k < MAX_ITER)
        z = 0.0D0
        do i = 1, n
            do j = 1, i - 1
                y = DABS(L(i, j))

!               Encontrado um novo valor absoluto máximo
                if (y > z) then
                    u = i
                    v = j
                    z = y
                end if
            end do
        end do

        if (z >= TOL) then
            P(:, :) = given_matrix(L, n, u, v)
            L(:, :) = matmul(matmul(transpose(P), L), P)
            X(:, :) = matmul(X, P)
            k = k + 1
        else
            ok = .TRUE.
            return
        end if
    end do
    ok = .FALSE.
    return
end function
```

Questão 3.:

Seja o seguinte sistema de equações $\mathbf{Ax} = \mathbf{B}$:

$$\mathbf{A} = \begin{bmatrix} 3 & 2 & 0 \\ 2 & 3 & -1 \\ 0 & -1 & 3 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

a) Para obter o polinômio característico, calculamos

$$\begin{aligned}
\det(\mathbf{A} - \lambda\mathbf{I}) &= \begin{vmatrix} 3-\lambda & 2 & 0 \\ 2 & 3-\lambda & -1 \\ 0 & -1 & 3-\lambda \end{vmatrix} \\
&= (3-\lambda)\begin{vmatrix} 3-\lambda & -1 \\ -1 & 3-\lambda \end{vmatrix} - 2\begin{vmatrix} 2 & 0 \\ -1 & 3-\lambda \end{vmatrix} \\
&= (3-\lambda)[(3-\lambda)^2 - 1] - 4(3-\lambda) \\
&= (3-\lambda)[(3-\lambda)^2 - 5] \\
&= (3-\lambda)(\lambda^2 - 6\lambda + 4)
\end{aligned}$$

Buscando as raízes λ_i deste polinômio, podemos afirmar que $\lambda = 3$ é um dos autovalores. Usando a fórmula de Bhaskara, encontramos os demais.

$$\begin{aligned}
\Delta = (-6)^2 - 4 \cdot 1 \cdot 4 = 20 \\
\implies \lambda_i = \frac{6 \pm \sqrt{20}}{2} = 3 \pm \sqrt{5}
\end{aligned}$$

Assim, $\lambda_i \in \{3 - \sqrt{5}, 3, 3 + \sqrt{5}\}$. Os autovetores $\mathbf{v}_i$, por outro lado, devem satisfazer

$$\mathbf{Av}_i = \lambda_i \mathbf{v}_i$$

Logo, um autovetor $\mathbf{v}$ da forma $[x, y, z]^\mathbf{T}$ obedece

$$\begin{bmatrix} 3 & 2 & 0 \\ 2 & 3 & -1 \\ 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \lambda x \\ \lambda y \\ \lambda z \end{bmatrix}$$

ou seja,

$$\begin{array}{ccccccc}
3x & + & 2y & & & = & \lambda x \\
2x & + & 3y & - & z & = & \lambda y \\
 & - & y & + & 3z & = & \lambda z
\end{array}$$

de onde tiramos que

$$\begin{aligned}
y &= \frac{(\lambda - 3)}{2} x \\
z &= 2x - (\lambda - 3)y = \frac{4 - (\lambda - 3)^2}{2} x
\end{aligned}$$

e com isso dizemos que todo autovetor de $\mathbf{A}$ tem a forma

$$\begin{bmatrix} 1 \\ \frac{(\lambda-3)}{2} \\ \frac{4-(\lambda-3)^2}{2} \end{bmatrix}$$

Substituindo os autovalores na relação:

$$\mathbf{v} \in \left\{ \begin{bmatrix} 1 \\ \frac{-\sqrt{5}}{2} \\ \frac{-1}{2} \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 2 \end{bmatrix}, \begin{bmatrix} 1 \\ \frac{\sqrt{5}}{2} \\ \frac{-1}{2} \end{bmatrix} \right\} \quad \text{e} \quad \lambda \in \left\{ 3 - \sqrt{5}, 3, 3 + \sqrt{5} \right\}$$

respectivamente.

b) Como todos os autovalores são positivos ($\lambda_i > 0$), podemos afirmar que $\mathbf{A}$ é positiva definida.

c) O método da potência ("*Power Method*") consiste em, partindo de um vetor $\mathbf{x}^{(0)}$, cuja primeira componente é 1, aplicar sucessivamente a matriz $\mathbf{A}$ sobre o vetor, normalizando suas demais entradas, dividindo-as pelo valor da primeira a cada iteração k. Seja $\mathbf{y}^{(k)} = \mathbf{A}\mathbf{x}^{(k)}$. Assim, podemos dizer que

$$\mathbf{x}^{(k)} = \frac{\mathbf{y}^{(k-1)}}{\mathbf{y}_1^{(k-1)}}$$

Observando as entradas da matriz, afirmamos que

$$\begin{aligned} \mathbf{y}_1^{(k)} &= 3\mathbf{x}_1^{(k)} + 2\mathbf{x}_2^{(k)} \\ \mathbf{y}_2^{(k)} &= 2\mathbf{x}_1^{(k)} + 3\mathbf{x}_2^{(k)} - \mathbf{x}_3^{(k)} \\ \mathbf{y}_3^{(k)} &= -\mathbf{x}_2^{(k)} + 3\mathbf{x}_3^{(k)} \end{aligned}$$

e, portanto

$$\begin{aligned} \mathbf{x}_1^{(k)} &= 1 \\ \mathbf{x}_2^{(k)} &= \frac{2\mathbf{x}_1^{(k-1)} + 3\mathbf{x}_2^{(k-1)} - \mathbf{x}_3^{(k-1)}}{3\mathbf{x}_1^{(k-1)} + 2\mathbf{x}_2^{(k-1)}} \\ \mathbf{x}_3^{(k)} &= \frac{-\mathbf{x}_2^{(k-1)} + 3\mathbf{x}_3^{(k-1)}}{3\mathbf{x}_1^{(k-1)} + 2\mathbf{x}_2^{(k-1)}} \end{aligned}$$

Para calcular o valor de $\mathbf{x}$, tomemos o limite de $\mathbf{x}^{(k)}$ quando $k \to \infty$, sobre cada componente

$$\begin{aligned} \mathbf{x}_1 &= \lim_{k\to\infty} \mathbf{x}_1^{(k)} = 1 \\ \mathbf{x}_2 &= \lim_{k\to\infty} \mathbf{x}_2^{(k)} = \frac{2\lim_{k\to\infty} \mathbf{x}_1^{(k-1)} + 3\lim_{k\to\infty} \mathbf{x}_2^{(k-1)} - \lim_{k\to\infty} \mathbf{x}_3^{(k-1)}}{3\lim_{k\to\infty} \mathbf{x}_1^{(k-1)} + 2\lim_{k\to\infty} \mathbf{x}_2^{(k-1)}} \\ \mathbf{x}_3 &= \lim_{k\to\infty} \mathbf{x}_3^{(k)} = \frac{-\lim_{k\to\infty} \mathbf{x}_2^{(k-1)} + 3\lim_{k\to\infty} \mathbf{x}_3^{(k-1)}}{3\lim_{k\to\infty} \mathbf{x}_1^{(k-1)} + 2\lim_{k\to\infty} \mathbf{x}_2^{(k-1)}} \end{aligned}$$

Como para toda sequência convergente $a_k \in \mathbb{R}$, $\lim_{k\to\infty} a_k = L \implies \lim_{k\to\infty} a_{k-1} = L$, segue que

$$\begin{aligned} \mathbf{x}_1 &= 1 \\ \mathbf{x}_2 &= \frac{2 + 3\mathbf{x}_2 - \mathbf{x}_3}{3 + 2\mathbf{x}_2} \\ \mathbf{x}_3 &= \frac{-\mathbf{x}_2 + 3\mathbf{x}_3}{3 + 2\mathbf{x}_2} \end{aligned}$$

Consequentemente,

$$\begin{cases} 3\mathbf{x}_2 + 2\mathbf{x}_2^2 &= 2 + 3\mathbf{x}_2 - \mathbf{x}_3 \\ 3\mathbf{x}_3 + 2\mathbf{x}_2\mathbf{x}_3 &= -\mathbf{x}_2 + 3\mathbf{x}_3 \end{cases} \implies \begin{cases} 2\mathbf{x}_2^2 &= 2 - \mathbf{x}_3 \\ 2\mathbf{x}_2\mathbf{x}_3 &= -\mathbf{x}_2 \end{cases} \implies \begin{cases} \mathbf{x}_2 &= \frac{\sqrt{5}}{2} \\ \mathbf{x}_3 &= -\frac{1}{2} \end{cases}$$

Portanto, o autovetor $\mathbf{v}_{\max}$ associado ao autovalor de maior módulo é

$$\mathbf{v}_{\max} = \begin{bmatrix} 1 \\ \frac{\sqrt{5}}{2} \\ -\frac{1}{2} \end{bmatrix}$$

e, por construção, o maior autovalor $\lambda_{\max}$ é dado por

$$\lambda_{\max} = 3\mathbf{x}_1 + 2\mathbf{x}_2 = 3 + \sqrt{5}$$

d) Segue o passo-a-passo do algoritmo de Jacobi para autovalores, com uma tolerância $|a_{i,j}| \leq 10^{-3}$.

$$\mathbf{A}^{(0)} = \begin{bmatrix} 3 & 2 & 0 \\ 2 & 3 & -1 \\ 0 & -1 & 3 \end{bmatrix} \; \mathbf{X}^{(0)} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}^{(1)} = \begin{bmatrix} 1 & 0 & 0.707 \\ 0 & 5 & -0.707 \\ 0.707 & -0.707 & 3 \end{bmatrix} \; \mathbf{X}^{(1)} = \begin{bmatrix} 0.707 & 0.707 & 0 \\ -0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{A}^{(2)} = \begin{bmatrix} 1 & 0.674 & 0.214 \\ 0.674 & 2.775 & 0 \\ 0.214 & 0 & 5.225 \end{bmatrix} \; \mathbf{X}^{(2)} = \begin{bmatrix} 0.707 & 0.214 & -0.674 \\ -0.707 & 0.214 & -0.674 \\ 0 & 0.953 & 0.303 \end{bmatrix}$$

$$\mathbf{A}^{(3)} = \begin{bmatrix} 0.773 & 0 & 0.203 \\ 0 & 3.002 & 0.068 \\ 0.203 & 0.068 & 5.225 \end{bmatrix} \; \mathbf{X}^{(3)} = \begin{bmatrix} 0.602 & 0.429 & -0.674 \\ -0.738 & -0.023 & -0.674 \\ -0.304 & 0.903 & 0.303 \end{bmatrix}$$

$$\mathbf{A}^{(4)} = \begin{bmatrix} 0.764 & -0.003 & 0 \\ -0.003 & 3.002 & 0.068 \\ 0 & 0.068 & 5.234 \end{bmatrix} \; \mathbf{X}^{(4)} = \begin{bmatrix} 0.632 & 0.429 & -0.646 \\ -0.707 & -0.023 & -0.707 \\ -0.317 & 0.903 & 0.289 \end{bmatrix}$$

$$\mathbf{A}^{(5)} = \begin{bmatrix} 0.764 & 0 & 0 \\ 0 & 3.002 & 0.068 \\ 0 & 0.068 & 5.234 \end{bmatrix} \; \mathbf{X}^{(5)} = \begin{bmatrix} 0.632 & 0.428 & -0.646 \\ -0.707 & -0.022 & -0.707 \\ -0.316 & 0.904 & 0.289 \end{bmatrix}$$

$$\mathbf{A}^{(6)} = \begin{bmatrix} 0.764 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5.236 \end{bmatrix} \; \mathbf{X}^{(6)} = \begin{bmatrix} 0.632 & 0.447 & -0.632 \\ -0.707 & 0 & -0.707 \\ -0.316 & 0.894 & 0.316 \end{bmatrix}$$

Após 6 iterações, temos os autovalores aproximados de $\mathbf{A}$ nas entradas da diagonal principal de $\mathbf{A}^{(6)}$, e seus respectivos autovetores nas respectivas colunas de $\mathbf{X}^{(6)}$.

e) Vamos resolver agora o sistema $\mathbf{Ax} = \mathbf{b}$ de quatro maneiras distintas.

1.: *Cholesky*

O método de *Cholesky* nos dá uma fórmula direta para a fatoração $\mathbf{A} = \mathbf{LL^T}$:

$$\mathbf{L} = \begin{bmatrix} \sqrt{\mathbf{A}_{1,1}} & 0 & 0 \\ \frac{\mathbf{A}_{2,1}}{\mathbf{L}_{1,1}} & \sqrt{\mathbf{A}_{2,2} - \mathbf{L}_{2,1}^2} & 0 \\ \frac{\mathbf{A}_{3,1}}{\mathbf{L}_{1,1}} & \frac{(\mathbf{A}_{3,2} - \mathbf{L}_{3,1}\mathbf{L}_{2,1})}{L_{2,2}} & \sqrt{\mathbf{A}_{3,3} - \mathbf{L}_{3,1}^2 - \mathbf{L}_{3,2}^2} \end{bmatrix}$$

Portanto,

$$\mathbf{L} = \begin{bmatrix} \sqrt{3} & 0 & 0 \\ \frac{2}{\sqrt{3}} & \sqrt{\frac{5}{3}} & 0 \\ 0 & -\sqrt{\frac{3}{5}} & 2\sqrt{\frac{3}{5}} \end{bmatrix}$$

Resolvemos primeiro o sistema $\mathbf{Ly} = \mathbf{b}$:

$$\begin{bmatrix} \sqrt{3} & 0 & 0 \\ \frac{2}{\sqrt{3}} & \sqrt{\frac{5}{3}} & 0 \\ 0 & -\sqrt{\frac{3}{5}} & 2\sqrt{\frac{3}{5}} \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

de onde tiramos que

$$\mathbf{y} = \begin{bmatrix} \frac{1}{\sqrt{3}} \\ -\sqrt{\frac{5}{3}} \\ 0 \end{bmatrix}$$

Por fim, resolvemos $\mathbf{L^T x} = \mathbf{y}$:

$$\begin{bmatrix} \sqrt{3} & \frac{2}{\sqrt{3}} & 0 \\ 0 & \sqrt{\frac{5}{3}} & -\sqrt{\frac{3}{5}} \\ 0 & 0 & 2\sqrt{\frac{3}{5}} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{3}} \\ -\sqrt{\frac{5}{3}} \\ 0 \end{bmatrix}$$

e assim temos

$$\mathbf{x} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

2.: *Jacobi*

3.: *Gauss-Seidel*

4.: Autovalores e autovetores

Como $\mathbf{A}$ é simétrica, vale que $\mathbf{x} = \Theta\lambda^{-1}\Theta^{\mathbf{T}}\mathbf{b}$, onde λ é a matriz diagonal dos autovalores e Θ é a matriz dos autovetores. Logo,

$$\mathbf{x} = \begin{bmatrix} 1 & 1 & 1 \\ \frac{-\sqrt{5}}{2} & 0 & \frac{\sqrt{5}}{2} \\ \frac{-1}{2} & 2 & \frac{-1}{2} \end{bmatrix} \begin{bmatrix} \frac{1}{3-\sqrt{5}} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{3+\sqrt{5}} \end{bmatrix} \begin{bmatrix} 1 & \frac{-\sqrt{5}}{2} & \frac{-1}{2} \\ 1 & 0 & 2 \\ 1 & \frac{\sqrt{5}}{2} & \frac{-1}{2} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix}$$

f) Sabemos que, para uma matriz $\mathbf{A} \in \mathbb{C}^n$ temos

$$\det(\mathbf{A}) = \prod_{i=1}^{n} \lambda_i$$

Portanto, $\det(\mathbf{A}) = 3 \cdot (3 - \sqrt{5}) \cdot (3 + \sqrt{5}) = 12$.

Questão 4.:

Algoritmo 3.: Saída do Programa

```
 A:
|   3.00000    2.00000    0.00000|
|   2.00000    3.00000   -1.00000|
|   0.00000   -1.00000    3.00000|
b:
|   1.00000|
|  -1.00000|
|   1.00000|
DET =   12.000000000000000
RAIO ESPECTRAL =   5.2360679804753349
:: Decomposição LU (sem pivoteamento) ::
L:
|   1.00000    0.00000    0.00000|
|   0.66667    1.00000    0.00000|
|   0.00000   -0.60000    1.00000|
U:
|   3.00000    2.00000    0.00000|
|   0.00000    1.66667   -1.00000|
|   0.00000    0.00000    2.40000|
y:
|   1.00000|
|  -1.66667|
|   0.00000|
x:
|   1.00000|
|  -1.00000|
|   0.00000|
:: Decomposição PLU (com pivoteamento) ::
P:
|   1.00000    0.00000    0.00000|
|   0.00000    1.00000    0.00000|
|   0.00000    0.00000    1.00000|
L:
|   1.00000    0.00000    0.00000|
|   0.66667    1.00000    0.00000|
|   0.00000   -0.60000    1.00000|
U:
|   3.00000    2.00000    0.00000|
|   0.00000    1.66667   -1.00000|
|   0.00000    0.00000    2.40000|
y:
|   1.00000|
|  -1.66667|
|   0.00000|
x:
|   1.00000|
|  -1.00000|
|   0.00000|
DET
```

```
12.000000000000000
x:
|    1.00000|
|   -1.00000|
|    0.00000|
:: Decomposição de Cholesky ::
L:
|    1.73205     0.00000     0.00000|
|    1.15470     1.29099     0.00000|
|    0.00000    -0.77460     1.54919|
y:
|    0.57735|
|   -1.29099|
|   -0.00000|
x:
|    1.00000|
|   -1.00000|
|   -0.00000|
:: Método de Jacobi ::
Matriz mal-condicionada.
:: Método de Gauss-Seidel ::
A:
|    3.00000     2.00000     0.00000|
|    2.00000     3.00000    -1.00000|
|    0.00000    -1.00000     3.00000|
x:
|    1.00000|
|   -1.00000|
|   -0.00000|
b:
|    1.00000|
|   -1.00000|
|    1.00000|
e =     6.2803698347351007E-016
:: Método das Potências (Power Method) ::
x:
|    1.00000|
|    1.11803|
|   -0.50000|
lambda:
5.2360680332272143
:: Método de autovalores de Jacobi ::
L:
|    0.76393     0.00000     0.00000|
|    0.00000     3.00000    -0.00000|
|    0.00000    -0.00000     5.23607|
X:
|    0.63246     0.44721    -0.63246|
|   -0.70711     0.00000    -0.70711|
|   -0.31623     0.89443     0.31623|
```

Appendices

Código

```fortran
!   Matrix Module

    module Matrix
        implicit none
        integer :: NMAX = 1000
        integer :: KMAX = 1000

        integer :: MAX_ITER = 1000

        double precision :: TOL = 1.0D-8
    contains
!       ===== I/O Metods =====
        subroutine error(text)
!           Red Text
            implicit none
            character(len=*) :: text
            write (*, *) ''//achar(27)//'[31m'//text//''//achar(27)//'[0m'
        end subroutine

        subroutine warn(text)
!           Yellow Text
            implicit none
            character(len=*) :: text
            write (*, *) ''//achar(27)//'[93m'//text//''//achar(27)//'[0m'
        end subroutine

        subroutine info(text)
!           Green Text
            implicit none
            character(len=*) :: text
            write (*, *) ''//achar(27)//'[32m'//text//''//achar(27)//'[0m'
        end subroutine

        subroutine ill_cond()
!           Prompts the user with an ill-conditioning warning.
            implicit none
            call error('Matriz mal-condicionada.')
        end subroutine

        subroutine print_matrix(A, m, n)
            implicit none

            integer :: m, n
            double precision :: A(m, n)
```

```fortran

        integer :: i, j

20      format(' |', F10.5, ' ')
21      format(F10.5, '|')
22      format(F10.5, ' ')

        do i = 1, m
            do j = 1, n
                if (j == 1) then
                    write(*, 20, advance='no') A(i, j)
                elseif (j == n) then
                    write(*, 21, advance='yes') A(i, j)
                else
                    write(*, 22, advance='no') A(i, j)
                end if
            end do
        end do
    end subroutine

    subroutine read_matrix(fname, A, m, n)
        implicit none
        character(len=*) :: fname
        integer :: m, n
        double precision, allocatable :: A(:, :)

        integer :: i

        open(unit=33, file=fname, status='old', action='read')
        read(33, *) m
        read(33, *) n
        allocate(A(m, n))

        do i = 1, m
            read(33,*) A(i,:)
        end do

        close(33)
    end subroutine

    subroutine print_vector(x, n)
        implicit none

        integer :: n
        double precision :: x(n)

        integer :: i

30      format(' |', F10.5, '|')

        do i = 1, n
            write(*, 30) x(i)
        end do
```

```fortran
        end subroutine

        subroutine read_vector(fname, b, t)
            implicit none
            character(len=*) :: fname
            integer :: t
            double precision, allocatable :: b(:)

            open(unit=33, file=fname, status='old', action='read')
            read(33, *) t
            allocate(b(t))

            read(33,*) b(:)

            close(33)
        end subroutine

!       =========== Matrix Methods ============
        recursive function det(A, n) result (d)
            implicit none

            integer :: n
            double precision :: A(n, n)
            double precision :: X(n-1, n-1)

            integer :: i
            double precision :: d, s

            if (n == 1) then
                d = A(1, 1)
                return
            elseif (n == 2) then
                d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
                return
            else
                d = 0.0D0
                s = 1.0D0
                do i = 1, n
!                   Compute submatrix X
                    X(:,  :i-1) = A(2:,    :i-1)
                    X(:, i:   ) = A(2:, i+1:   )

                    d = s * det(X, n-1) * A(1, i) + d
                    s = -s
                end do
            end if
            return
        end function

        function rand_vector(n) result (x)
            implicit none
            integer :: n
            double precision :: x (n)
```

```fortran

        integer :: i

        do i = 1, n
            x(i) = 2 * ran(0) - 1
        end do
        return
    end function

    function rand_matrix(m, n) result (A)
        implicit none
        integer :: m, n
        double precision :: A(m, n)

        integer :: i

        do i = 1, m
            A(i, :) = rand_vector(n)
        end do
        return
    end function

    function id_matrix(n) result (A)
        implicit none

        integer :: n
        double precision :: A(n, n)

        integer :: j

        A(:, :) = 0.0D0

        do j = 1, n
            A(j, j) = 1.0D0
        end do
        return
    end function

    function given_matrix(A, n, i, j) result (G)
        implicit none

        integer :: n, i, j
        double precision :: A(n, n), G(n, n)
        double precision :: t, c, s

        G(:, :) = id_matrix(n)

        t = 0.5D0 * DATAN2(2.0D0 * A(i,j), A(i, i) - A(j, j))
        s = DSIN(t)
        c = DCOS(t)

        G(i, i) = c
        G(j, j) = c
```

```
            G(i, j) = -s
            G(j, i) = s

            return
        end function


        function diagonally_dominant(A, n) result (ok)
            implicit none

            integer :: n
            double precision :: A(n, n)

            logical :: ok
            integer :: i

            do i = 1, n
                if (DABS(A(i, i)) < SUM(DABS(A(i, :i-1))) + SUM(DABS(A(i, i+1:)))) then
                    ok = .FALSE.
                    return
                end if
            end do
            ok = .TRUE.
            return
        end function

        recursive function positive_definite(A, n) result (ok)
!       Checks wether a matrix is positive definite
!       according to Sylvester's criterion.
            implicit none

            integer :: n
            double precision A(n, n)

            logical :: ok

            if (n == 1) then
                ok = (A(1, 1) > 0)
                return
            else
                ok = positive_definite(A(:n-1, :n-1), n-1) .AND. (det(A, n) > 0)
                return
            end if
        end function

        function symmetrical(A, n) result (ok)
!           Check if the Matrix is symmetrical
            integer :: n

            double precision :: A(n, n)

```

```fortran
            integer :: i, j
            logical :: ok

            do i = 1, n
                do j = 1, i-1
                    if (A(i, j) /= A(j, i)) then
                        ok = .FALSE.
                        return
                    end if
                end do
            end do
            ok = .TRUE.
            return
        end function

        subroutine swap_rows(A, i, j, n)
            implicit none

            integer :: n
            integer :: i, j
            double precision A(n, n)
            double precision temp(n)

            temp(:) = A(i, :)
            A(i, :) = A(j, :)
            A(j, :) = temp(:)
        end subroutine

        function row_max(A, j, n) result(k)
            implicit none

            integer :: n
            double precision A(n, n)

            integer :: i, j, k
            double precision :: s

            s = 0.0D0
            do i = j, n
                if (A(i, j) > s) then
                    s = A(i, j)
                    k = i
                end if
            end do
            return
        end function

        function pivot_matrix(A, n) result (P)
            implicit none

            integer :: n
            double precision :: A(n, n)

```

```fortran
            double precision :: P(n, n)

            integer :: j, k

            P = id_matrix(n)

            do j = 1, n
                k = row_max(A, j, n)
                if (j /= k) then
                    call swap_rows(P, j, k, n)
                end if
            end do
            return
        end function

        function vector_norm(x, n) result (s)
            implicit none

            integer :: n
            double precision :: x(n)

            double precision :: s

            s = sqrt(dot_product(x, x))
            return
        end function

        function matrix_norm(A, n) result (s)
!           Frobenius norm
            implicit none
            integer :: n
            double precision :: A(n, n)
            double precision :: s

            s = DSQRT(SUM(A * A))
            return
        end function

        function spectral_radius(A, n) result (r)
            implicit none

            integer :: n
            double precision :: A(n, n), M(n, n)
            double precision :: r

            integer :: i, j, k

            M(:, :) = A(:, :)

            r = 1.0D0

            do k = 1, KMAX
                M = MATMUL(M, M)
```

```
                do i = 1, n
                    do j = 1, n
!                       Algum valor infinito
                        if (M(i, j) - 1 == M(i, j)) then
                            return
                        end if
                    end do
                end do
                r = matrix_norm(M, n)
                do j = 1, i
                    r = DSQRT(r)
                end do
            end do
            write(*, *) "r: "
            write(*, *) r
            return
        end function

        function LU_det(A, n) result (d)
            implicit none

            integer :: n
            integer :: i
            double precision :: A(n, n), L(n, n), U(n, n)
            double precision :: d

            d = 0.0D0

            if (.NOT. LU_decomp(A, L, U, n)) then
                call ill_cond()
                return
            end if

            do i = 1, n
                d = d * L(i, i) * U(i, i)
            end do

            return
        end function

        subroutine LU_matrix(A, L, U, n)
!           Splits Matrix in Lower and Upper-Triangular
            implicit none

            integer :: n
            double precision :: A(n, n), L(n, n), U(n, n)

            integer :: i

            L(:, :) = 0.0D0
            U(:, :) = 0.0D0

            do i = 1, n
```

```fortran
                L(i, i) = 1.0D0
                L(i,  :i-1) = A(i,  :i-1)
                U(i, i:   ) = A(i, i:   )
            end do
        end subroutine

!       === Matrix Factorization Conditions ===
        function Cholesky_cond(A, n) result (ok)
            implicit none

            integer :: n
            double precision :: A(n, n)

            logical :: ok

            ok = symmetrical(A, n) .AND. positive_definite(A, n)
            return

        end function

        function PLU_cond(A, n) result (ok)
            implicit none

            integer :: n
            double precision A(n, n)

            integer :: i, j
            double precision :: s

            logical :: ok

            do j = 1, n
                s = 0.0D0
                do i = 1, j
                    if (A(i, j) > s) then
                        s = A(i, j)
                    end if
                end do
            end do

            ok = (s < 0.01D0)

            return
        end function

        function LU_cond(A, n) result (ok)
            implicit none

            integer :: n
            double precision A(n, n)

            logical :: ok

```

```fortran
            ok = positive_definite(A, n)

            return
        end function
!        _      _____  _____ _______          __
!       | |    |_   _|/ ____|__   __|/\      /_ |
!       | |      | | | (___    | |  /  \      | |
!       | |      | |  \___ \   | | / /\ \     | |
!       | |____ _| |_ ____) |  | |/ ____ \    | |
!       |______|_____|_____/   |_/_/    \_\   |_|
!       ========================================

!       ======= Matrix Factorization Methods ========
        function PLU_decomp(A, P, L, U, n) result (ok)
            implicit none

            integer :: n
            double precision :: A(n,n), P(n,n), L(n,n), U(n,n)

            logical :: ok

!           Permutation Matrix
            P = pivot_matrix(A, n)

!           Decomposition over Row-Swapped Matrix
            ok = LU_decomp(matmul(P, A), L, U, n)
            return
        end function

        function LU_decomp(A, L, U, n) result (ok)
            implicit none

            integer :: n
            double precision :: A(n, n), L(n, n), U(n,n), M(n, n)

            logical :: ok

            integer :: i, j, k

!           Results Matrix
            M(:, :) = A(:, :)

            if (.NOT. LU_cond(A, n)) then
                call ill_cond()
                ok = .FALSE.
                return
            end if

            do k = 1, n-1
                do i = k+1, n
                    M(i, k) = M(i, k) / M(k, k)
                end do

```

```fortran
                do j = k+1, n
                    do i = k+1, n
                        M(i, j) = M(i, j) - M(i, k) * M(k, j)
                    end do
                end do
            end do

!           Splits M into L & U
            call LU_matrix(M, L, U, n)

            ok = .TRUE.
            return

        end function

        function Cholesky_decomp(A, L, n) result (ok)
            implicit none

            integer :: n
            double precision :: A(n, n), L(n, n)

            logical :: ok

            integer :: i, j

            if (.NOT. Cholesky_cond(A, n)) then
                call ill_cond()
                ok = .FALSE.
                return
            end if

            do i = 1, n
                L(i, i) = sqrt(A(i, i) - sum(L(i, :i-1) * L(i, :i-1)))
                do j = 1 + 1, n
                    L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)))
                        / L(i, i)
                end do
            end do

            ok = .TRUE.
            return
        end function

!       === Linear System Solving Conditions ===
        function Jacobi_cond(A, n) result (ok)
            implicit none

            integer :: n

            double precision :: A(n, n)

            logical :: ok

```

```fortran
            if (.NOT. spectral_radius(A, n) < 1) then
                ok = .FALSE.
                call ill_cond()
                return
            else
                ok = .TRUE.
                return
            end if
        end function

        function Gauss_Seidel_cond(A, n) result (ok)
            implicit none

            integer :: n

            double precision :: A(n, n)

            logical :: ok

            integer :: i

            do i = 1, n
                if (A(i, i) == 0.0D0) then
                    ok = .FALSE.
                    call error('Erro: Esse método não irá convergir.')
                    return
                end if
            end do

            if (.NOT. (diagonally_dominant(A, n) .OR. (symmetrical(A, n
               ) .AND. positive_definite(A, n)))) then
                call warn('Aviso: Esse método pode não convergir.')
            end if

            ok = .TRUE.
            return
        end function

!       == Linear System Solving Methods ==
        function Jacobi(A, x, b, e, n) result (ok)
            implicit none

            integer :: n

            double precision :: A(n, n)
            double precision :: b(n), x(n), x0(n)
            double precision :: e

            logical :: ok

            integer :: i, k

            x0 = rand_vector(n)
```

```

            ok = Jacobi_cond(A, n)

            if (.NOT. ok) then
                return
            end if

            do k = 1, KMAX
                do i = 1, n
                    x(i) = (b(i) - dot_product(A(i, :), x0)) / A(i, i)
                end do
                x0(:) = x(:)
                e = vector_norm(matmul(A, x) - b, n)
                if (e < TOL) then
                    return
                end if
            end do
            call error('Erro: Esse método não convergiu.')
            ok = .FALSE.
            return
        end function

        function Gauss_Seidel(A, x, b, e, n) result (ok)
            implicit none

            integer :: n

            double precision :: A(n, n)
            double precision :: b(n), x(n)
            double precision :: e, s

            logical :: ok
            integer :: i, j, k

            ok = Gauss_Seidel_cond(A, n)

            if (.NOT. ok) then
                return
            end if

            do k = 1, KMAX
                do i = 1, n
                    s = 0.0D0
                    do j = 1, n
                        if (i /= j) then
                            s = s + A(i, j) * x(j)
                        end if
                    end do
                    x(i) = (b(i) - s) / A(i, i)
                end do
                e = vector_norm(matmul(A, x) - b, n)
                if (e < TOL) then
                    return
```

```
            end if
        end do
        call error('Erro: Esse método não convergiu.')
        ok = .FALSE.
        return
    end function

    subroutine LU_backsub(L, U, x, y, b, n)
        implicit none

        integer :: n

        double precision :: L(n, n), U(n, n)
        double precision :: b(n), x(n), y(n)

        integer :: i

!       Ly = b (Forward Substitution)
        do i = 1, n
            y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i)
        end do

!       Ux = y (Backsubstitution)
        do i = n, 1, -1
            x(i) = (y(i) - SUM(U(i,i+1:n) * x(i+1:n))) / U(i, i)
        end do

    end subroutine

    function LU_solve(A, x, y, b, n) result (ok)
        implicit none

        integer :: n

        double precision :: A(n, n), L(n, n), U(n, n)
        double precision :: b(n), x(n), y(n)

        logical :: ok

        ok = LU_decomp(A, L, U, n)

        if (.NOT. ok) then
            return
        end if

        call LU_backsub(L, U, x, y, b, n)

        return
    end function

    function PLU_solve(A, x, y, b, n) result (ok)
        implicit none

```

```
            integer :: n

            double precision :: A(n, n), P(n,n), L(n, n), U(n, n)
            double precision :: b(n), x(n), y(n)

            logical :: ok

            ok = PLU_decomp(A, P, L, U, n)

            if (.NOT. ok) then
                return
            end if

            call LU_backsub(L, U, x, y, matmul(P, b), n)

            x(:) = matmul(P, x)

            return
        end function

        function Cholesky_solve(A, x, y, b, n) result (ok)
            implicit none

            integer :: n

            double precision :: A(n, n), L(n, n), U(n, n)
            double precision :: b(n), x(n), y(n)

            logical :: ok

            ok = Cholesky_decomp(A, L, n)

            if (.NOT. ok) then
                return
            end if

            U = transpose(L)

            call LU_backsub(L, U, x, y, b, n)

            return
        end function

!        _      _____  _____ _______         ___
!       | |    |_   _|/ ____|__   __|/\     |__ \
!       | |      | | | (___    | |  /  \       ) |
!       | |      | |  \___ \   | | / /\ \     / /
!       | |____ _| |_ ____) |  | |/ ____ \   / /_
!       |______|_____|_____/   |_/_/    \_\ |____|
!       ==========================================

!       ============ Power Method ============
        function power_method(A, n, x, l) result (ok)
```

```fortran
            implicit none
            integer :: n
            integer :: k = 0

            double precision :: A(n, n)
            double precision :: x(n)
            double precision :: l, ll

            logical :: ok

!           Begin with random normal vector and set 1st component to zero
            x(:) = rand_vector(n)
            x(1) = 1.0D0

!           Initialize Eigenvalues
            l = 0.0D0

!           Checks if error tolerance was reached
            do while (k < MAX_ITER)
                ll = l

                x(:) = matmul(A, x)

!               Retrieve Eigenvalue
                l = x(1)

!               Retrieve Eigenvector
                x(:) = x(:) / l

                if (dabs((l - ll) / l) < TOL) then
                    ok = .TRUE.
                    return
                else
                    k = k + 1
                    continue
                end if
            end do
            ok = .FALSE.
            return
        end function

        function Jacobi_eigen(A, n, L, X) result (ok)
            implicit none
            integer :: n, i, j, u, v
            integer :: k = 0

            double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
            double precision :: y, z

            logical :: ok

            X(:, :) = id_matrix(n)
```

```
            L(:, :) = A(:, :)

            do while (k < MAX_ITER)
                z = 0.0D0
                do i = 1, n
                    do j = 1, i - 1
                        y = DABS(L(i, j))

!                       Found new maximum absolute value
                        if (y > z) then
                            u = i
                            v = j
                            z = y
                        end if
                    end do
                end do

                if (z >= TOL) then
                    P(:, :) = given_matrix(L, n, u, v)
                    L(:, :) = matmul(matmul(transpose(P), L), P)
                    X(:, :) = matmul(X, P)
                    k = k + 1
                else
                    ok = .TRUE.
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

!        _      _____  _____ _______         _____
!       | |    |_   _|/ ____|__   __|/\     |__   |
!       | |      | | | (___    | |  /  \       )  /
!       | |      | |  \___ \   | | / /\ \     |_  \
!       | |____ _| |_ ____) |  | |/ ____ \   ___)  |
!       |______|_____|_____/   |_/_/    \_\ |_____/
!       ==========================================

        function least_squares(x, y, s, n) result (ok)
            implicit none
            integer :: n

            logical :: ok

            double precision :: A(2,2), b(2), s(2), r(2), x(n), y(n)

            A(1, 1) = n
            A(1, 2) = SUM(x)
            A(2, 1) = SUM(x)
            A(2, 2) = dot_product(x, x)

            b(1) = SUM(y)
```

```
            b(2) = dot_product(x, y)

            ok = Cholesky_solve(A, s, r, b, n)
            return
        end function

    end module Matrix
```