

# **COC473 - Lista 2**

Pedro Maciel Xavier  
116023847

25 de setembro de 2020

## Questão 1.:

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

```
1      function power_method(A, n, x, l) result (ok)
2          implicit none
3          integer :: n
4          integer :: k = 0
5
6          double precision :: A(n, n)
7          double precision :: x(n)
8          double precision :: l, ll
9
10         logical :: ok
11
12         !      Begin with random normal vector and set 1st component to
           zero
13         x(:) = rand_vector(n)
14         x(1) = 1.0D0
15
16         !      Initialize Eigenvalues
17         l = 0.0D0
18
19         !      Checks if error tolerance was reached
20         do while (k < MAX_ITER)
21             ll = l
22
23             x(:) = matmul(A, x)
24
25             !      Retrieve Eigenvalue
26             l = x(1)
27
28             !      Retrieve Eigenvector
29             x(:) = x(:) / l
30
31             if (dabs((l - ll) / l) < TOL) then
32                 ok = .TRUE.
33                 return
34             else
35                 k = k + 1
36                 continue
37             end if
38         end do
39         ok = .FALSE.
40         return
41     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.

```
1      function Jacobi_eigen(A, n, L, X) result (ok)
2          implicit none
3          integer :: n, i, j, u, v
4          integer :: k = 0
5
6          double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
7          double precision :: y, z
8
9          logical :: ok
10
11         X(:, :) = id_matrix(n)
12         L(:, :) = A(:, :)
13
14         do while (k < MAX_ITER)
15             z = 0.0D0
16             do i = 1, n
17                 do j = 1, i - 1
18                     y = DABS(L(i, j))
19
20                     ! Found new maximum absolute value
21                     if (y > z) then
22                         u = i
23                         v = j
24                         z = y
25                     end if
26                 end do
27             end do
28
29             if (z >= TOL) then
30                 P(:, :) = given_matrix(L, n, u, v)
31                 L(:, :) = matmul(matmul(transpose(P), L), P)
32                 X(:, :) = matmul(X, P)
33                 k = k + 1
34             else
35                 ok = .TRUE.
36                 return
37             end if
38         end do
39         ok = .FALSE.
40         return
41     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  $\lambda_i$  deste polinômio, podemos afirmar que  $\lambda = 3$  é um dos autovalores. Usando a fórmula de Bhaskara, encontramos os demais.

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

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

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

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

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

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:

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

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

$$\mathbf{x}^{(k)} = \frac{\mathbf{y}^{(k-1)}}{y_1^{(k-1)}}$$

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  $k \rightarrow \infty$ , sobre cada componente

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

Como para toda sequência convergente  $a_k \in \mathbb{R}$ ,  $\lim_{k \rightarrow \infty} a_k = L \implies \lim_{k \rightarrow \infty} a_{k-1} = L$ , 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  $|a_{i,j}| \leq 10^{-3}$ .

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

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

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

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

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

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

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

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 fatoraçoão  $\mathbf{A} = \mathbf{LL}^T$ :

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

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  $\mathbf{L}^T \mathbf{x} = \mathbf{y}$ :

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

e assim temos

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

2.: *Jacobi*

Vamos analisar este algoritmo sob a perspectiva de sua matriz de transformação  $\mathbf{J} = \mathbf{S}^{-1}\mathbf{T}$ , onde  $\mathbf{A} = \mathbf{S} - \mathbf{T}$  e  $\mathbf{S}$  é a matriz diagonal cujos elementos são idênticos aos da diagonal principal de  $\mathbf{A}$ . Vejamos:

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

$$\mathbf{J} = \mathbf{S}^{-1}\mathbf{T} = \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 0 & -2 & 0 \\ -2 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -\frac{2}{3} & 0 \\ -\frac{2}{3} & 0 & \frac{1}{3} \\ 0 & \frac{1}{3} & 0 \end{bmatrix}$$

Calculamos então os autovalores de  $\mathbf{J}$ , que chamaremos  $\Lambda$ . Seja  $\Theta$  a matriz dos autovetores de  $\mathbf{J}$  e  $\Theta^{-1}$  a sua inversa.

$$\Lambda = \begin{bmatrix} -\frac{\sqrt{5}}{3} & 0 & 0 \\ 0 & \frac{\sqrt{5}}{3} & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \Theta = \begin{bmatrix} -2 & -2 & \frac{1}{2} \\ -\sqrt{5} & \sqrt{5} & 0 \\ 1 & 1 & 1 \end{bmatrix} \quad \Theta^{-1} = \begin{bmatrix} -\frac{1}{5} & -\frac{1}{2\sqrt{5}} & \frac{1}{10} \\ -\frac{1}{5} & \frac{1}{2\sqrt{5}} & \frac{1}{10} \\ \frac{2}{5} & 0 & \frac{4}{5} \end{bmatrix}$$

Dizemos que uma iteração do método de *Jacobi* é representada pela relação  $\mathbf{x}^{(k+1)} = \mathbf{S}^{-1}(\mathbf{T}\mathbf{x}^{(k)} + \mathbf{b})$ . Seja  $\hat{\mathbf{b}} = \mathbf{S}^{-1}\mathbf{b}$ . A partir disso, reescrevemos a iteração como  $\mathbf{x}^{(k+1)} = \mathbf{J}\mathbf{x}^{(k)} + \hat{\mathbf{b}}$ . Aplicando sucessivamente esta operação temos:

$$\begin{aligned} \mathbf{x}^{(k+1)} &= \mathbf{J}\mathbf{x}^{(k)} + \hat{\mathbf{b}} \\ \mathbf{x}^{(k+2)} &= \mathbf{J}^2\mathbf{x}^{(k)} + \mathbf{J}\hat{\mathbf{b}} + \hat{\mathbf{b}} \\ \mathbf{x}^{(k+3)} &= \mathbf{J}^3\mathbf{x}^{(k)} + \mathbf{J}^2\hat{\mathbf{b}} + \mathbf{J}\hat{\mathbf{b}} + \hat{\mathbf{b}} \\ &\vdots \\ \mathbf{x}^{(k+m)} &= \mathbf{J}^m\mathbf{x}^{(k)} + \sum_{j=0}^{m-1} \mathbf{J}^j\hat{\mathbf{b}} \end{aligned}$$

O autovalor de maior módulo,  $\lambda_{\max} = \frac{\sqrt{5}}{3}$  nos indica que o raio espectral da matriz de iteração é  $\rho(\mathbf{J}) = |\frac{\sqrt{5}}{3}| \approx 0.7453 < 1$ . Portanto, temos a garantia da convergência do método, vamos adiante.

Outra informação de grande valor que o raio espectral nos dá é a taxa de convergência. Sabemos que o comportamento do erro é limitado pela aplicação da matriz  $\mathbf{J}$ , segundo o qual vale

$$\begin{aligned} \mathbf{e}^{(k)} &\approx \mathbf{J}^k \mathbf{e}^{(0)} \\ \therefore e^{(k)} &\approx \rho(\mathbf{J})^k e^{(0)} \end{aligned}$$

onde temos o erro  $e^{(k)} = \|\mathbf{e}^{(k)}\| = \|\mathbf{x}^{(k)} - \mathbf{x}\|$ .

Assim, podemos obter uma estimativa do número  $k$  de passos necessários para a convergência, supondo uma tolerância  $\text{tol} = 10^{-5}$ . Logo,

$$10^{-5} > e^{(k)} = \rho(\mathbf{J})^k e^{(0)} = \left(\frac{\sqrt{5}}{3}\right)^k e^{(0)}$$



$$\begin{aligned}
\log_{10}(10^{-5}) &> \log_{10} \left[ \left( \frac{\sqrt{5}}{3} \right)^k e^{(0)} \right] \\
-5 &> k \log_{10} \left( \frac{\sqrt{5}}{3} \right) + \log_{10}(e^{(0)}) \\
\therefore k &> \frac{5 + \log_{10}(e^{(0)})}{-\log_{10}(\sqrt{5}/3)}
\end{aligned}$$

Vamos supor que  $e^{(0)} = O(1)$ , e assim temos que  $k > 39$ .

Calcularemos, portanto,  $\mathbf{x}^{(k)}$  da seguinte forma:

$$\begin{aligned}
\mathbf{x}^{(k)} &= \mathbf{J}^k \mathbf{x}^{(0)} + \sum_{j=1}^k \mathbf{J}^{j-1} \hat{\mathbf{b}} \\
&= \Theta \Lambda^k \Theta^{-1} \mathbf{x}^{(0)} + \left[ \sum_{j=1}^k \Theta \Lambda^{j-1} \Theta^{-1} \right] \hat{\mathbf{b}} \\
&= \Theta \Lambda^k \Theta^{-1} \mathbf{x}^{(0)} + \Theta \left[ \sum_{j=0}^{k-1} \Lambda^j \right] \Theta^{-1} \hat{\mathbf{b}}
\end{aligned}$$

Aqui trazemos o interessante fato de que

$$\sum_{j=0}^{k-1} \Lambda^j = \begin{bmatrix} \sum_{j=0}^{k-1} \lambda_1^j & & \\ & \sum_{j=0}^{k-1} \lambda_2^j & \\ & & \sum_{j=0}^{k-1} \lambda_3^j \end{bmatrix} = \begin{bmatrix} \frac{1 - \lambda_1^k}{1 - \lambda_1} & & \\ & \frac{1 - \lambda_2^k}{1 - \lambda_2} & \\ & & \frac{1 - \lambda_3^k}{1 - \lambda_3} \end{bmatrix}$$

uma vez que  $\forall i \lambda_i \neq 1$ .

Escolhendo  $k = 40$  e  $\mathbf{x}^{(0)} = [0, 0, 0]$ , vamos às contas:

$$\mathbf{x}^{(40)} = 0 + \Theta \left[ \sum_{j=0}^{39} \Lambda^j \right] \Theta^{-1} \mathbf{S}^{-1} \mathbf{b}$$

Na forma matricial temos:

$$\begin{aligned}
\mathbf{x}^{(40)} &= \begin{bmatrix} -2 & -2 & \frac{1}{2} \\ -\sqrt{5} & \sqrt{5} & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \frac{1 - (-\sqrt{5}/3)^{40}}{1 + \sqrt{5}/3} & & \\ & \frac{1 + (\sqrt{5}/3)^{40}}{1 - \sqrt{5}/3} & \\ & & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{5} & -\frac{1}{2\sqrt{5}} & \frac{1}{10} \\ -\frac{1}{5} & \frac{1}{2\sqrt{5}} & \frac{1}{10} \\ \frac{2}{5} & 0 & \frac{4}{5} \end{bmatrix} \begin{bmatrix} \frac{1}{3} & 0 & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \\
&\approx \begin{bmatrix} -2 & -2 & \frac{1}{2} \\ -\sqrt{5} & \sqrt{5} & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0.57294 & & \\ & 3.92702 & \\ & & 1.00000 \end{bmatrix} \begin{bmatrix} -\frac{1}{5} & -\frac{1}{2\sqrt{5}} & \frac{1}{10} \\ -\frac{1}{5} & \frac{1}{2\sqrt{5}} & \frac{1}{10} \\ \frac{2}{5} & 0 & \frac{4}{5} \end{bmatrix} \begin{bmatrix} \frac{1}{3} \\ -\frac{1}{3} \\ \frac{1}{3} \end{bmatrix}
\end{aligned}$$

$$\mathbf{x}^{(40)} \approx \begin{bmatrix} 0.999993 \\ -0.999992 \\ 0.000003 \end{bmatrix}$$

### 3.: Gauss-Seidel

Neste caso, temos a garantia da convergência, uma vez que a matriz é *positiva* (pois seus autovalores são todos positivos), além de ser *simétrica*.

No item anterior, a diagonalização da matriz de iteração foi possível devido ao fato de que  $\mathbf{A}$  e  $\mathbf{S}$  eram *simétricas*. Isto propagou esta propriedade às matrizes  $\mathbf{T}$  e  $\mathbf{J}$ . Neste caso, como a matriz de iteração relacionada ao algoritmo de *Gauss-Seidel* não é simétrica, não podemos contar com isso. Outras alternativas podem surgir nesse sentido, como a forma normal de *Jordan*. No entanto, ela é conhecida por não ser numericamente estável e não costuma ser utilizada.

Vamos portanto, nos contentar em seguir o algoritmo anterior, inicializando o processo de *Gauss-Seidel* com o vetor  $\mathbf{x}^{(0)} = [0.999993, -0.999992, 0.000003]$  resultante do algoritmo anterior. Segue que:

$$\begin{aligned} \mathbf{x}_1^{(1)} &= \frac{1}{3}(1 + 2 \cdot 0.999993) \approx 0.999995 \\ \mathbf{x}_2^{(1)} &= \frac{1}{3}(-1 - 2 \cdot 0.999995 + 0.000003) \approx -0.999996 \\ \mathbf{x}_3^{(1)} &= \frac{1}{3}(1 - 0.999995) \approx 0.000001 \end{aligned}$$

Calculando o resíduo temos:

$$R = \frac{\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\|}{\|\mathbf{x}^{(1)}\|} \approx 3.1 \times 10^{-6}$$

e o algoritmo termina com

$$\mathbf{x} \approx \begin{bmatrix} 0.999995 \\ -0.999996 \\ 0.000001 \end{bmatrix}$$

### 4.: Autovalores e autovetores

Como  $\mathbf{A}$  é simétrica, vale que  $\mathbf{x} = \Theta \lambda^{-1} \Theta^T \mathbf{b}$ , onde  $\lambda$  é a matriz diagonal dos autovalores e  $\Theta$  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 1.: Saída do Programa

```
1  A:
2  | 3.00000 2.00000 0.00000|
3  | 2.00000 3.00000 -1.00000|
4  | 0.00000 -1.00000 3.00000|
5  b:
6  | 1.00000|
7  | -1.00000|
8  | 1.00000|
9  DET = 12.000000000000000
10 RAO ESPECTRAL = 5.2360679804753349
11 :: Decomposição LU (sem pivoteamento) ::
12 L:
13 | 1.00000 0.00000 0.00000|
14 | 0.66667 1.00000 0.00000|
15 | 0.00000 -0.60000 1.00000|
16 U:
17 | 3.00000 2.00000 0.00000|
18 | 0.00000 1.66667 -1.00000|
19 | 0.00000 0.00000 2.40000|
20 y:
21 | 1.00000|
22 | -1.66667|
23 | 0.00000|
24 x:
25 | 1.00000|
26 | -1.00000|
27 | 0.00000|
28 :: Decomposição PLU (com pivoteamento) ::
29 P:
30 | 1.00000 0.00000 0.00000|
31 | 0.00000 1.00000 0.00000|
32 | 0.00000 0.00000 1.00000|
33 L:
34 | 1.00000 0.00000 0.00000|
35 | 0.66667 1.00000 0.00000|
36 | 0.00000 -0.60000 1.00000|
37 U:
38 | 3.00000 2.00000 0.00000|
39 | 0.00000 1.66667 -1.00000|
40 | 0.00000 0.00000 2.40000|
41 y:
42 | 1.00000|
43 | -1.66667|
44 | 0.00000|
45 x:
46 | 1.00000|
47 | -1.00000|
48 | 0.00000|
49 DET
```

```

50 12.000000000000000
51 x:
52 | 1.00000|
53 | -1.00000|
54 | 0.00000|
55 :: Decomposição de Cholesky ::
56 L:
57 | 1.73205 0.00000 0.00000|
58 | 1.15470 1.29099 0.00000|
59 | 0.00000 -0.77460 1.54919|
60 y:
61 | 0.57735|
62 | -1.29099|
63 | -0.00000|
64 x:
65 | 1.00000|
66 | -1.00000|
67 | -0.00000|
68 :: Método de Jacobi ::
69 Matriz mal-condicionada.
70 :: Método de Gauss-Seidel ::
71 A:
72 | 3.00000 2.00000 0.00000|
73 | 2.00000 3.00000 -1.00000|
74 | 0.00000 -1.00000 3.00000|
75 x:
76 | 1.00000|
77 | -1.00000|
78 | -0.00000|
79 b:
80 | 1.00000|
81 | -1.00000|
82 | 1.00000|
83 e = 6.2803698347351007E-016
84 :: Método das Potências (Power Method) ::
85 x:
86 | 1.00000|
87 | 1.11803|
88 | -0.50000|
89 lambda:
90 5.2360680332272143
91 :: Método de autovalores de Jacobi ::
92 L:
93 | 0.76393 0.00000 0.00000|
94 | 0.00000 3.00000 -0.00000|
95 | 0.00000 -0.00000 5.23607|
96 X:
97 | 0.63246 0.44721 -0.63246|
98 | -0.70711 0.00000 -0.70711|
99 | -0.31623 0.89443 0.31623|

```

# Appendices

## Código

```
1  !   Matrix Module
2
3  module Matrix
4      implicit none
5      integer :: NMAX = 1000
6      integer :: KMAX = 1000
7
8      integer :: MAX_ITER = 1000
9
10     double precision :: TOL = 1.0D-8
11 contains
12 !     ===== I/O Methods =====
13     subroutine error(text)
14 !         Red Text
15         implicit none
16         character(len=*) :: text
17         write (*, *) '//a'char(27)//'[31m'//text//''//a'char(27)//'
18             [0m'
19     end subroutine
20
21     subroutine warn(text)
22 !         Yellow Text
23         implicit none
24         character(len=*) :: text
25         write (*, *) '//a'char(27)//'[93m'//text//''//a'char(27)//'
26             [0m'
27     end subroutine
28
29     subroutine info(text)
30 !         Green Text
31         implicit none
32         character(len=*) :: text
33         write (*, *) '//a'char(27)//'[32m'//text//''//a'char(27)//'
34             [0m'
35     end subroutine
36
37     subroutine ill_cond()
38 !         Prompts the user with an ill-conditioning warning.
39         implicit none
40         call error('Matriz mal-condicionada: este método não irá
41             convergir.')
42     end subroutine
43
44     subroutine print_matrix(A, m, n)
45         implicit none
46
47         integer :: m, n
```

```

44         double precision :: A(m, n)
45
46         integer :: i, j
47
48 20         format(' /', F10.5, ' ')
49 21         format(F10.5, '/')
50 22         format(F10.5, ' ')
51
52         do i = 1, m
53             do j = 1, n
54                 if (j == 1) then
55                     write(*, 20, advance='no') A(i, j)
56                 elseif (j == n) then
57                     write(*, 21, advance='yes') A(i, j)
58                 else
59                     write(*, 22, advance='no') A(i, j)
60                 end if
61             end do
62         end do
63     end subroutine
64
65     subroutine read_matrix(fname, A, m, n)
66         implicit none
67         character(len=*) :: fname
68         integer :: m, n
69         double precision, allocatable :: A(:, :)
70
71         integer :: i
72
73         open(unit=33, file=fname, status='old', action='read')
74         read(33, *) m
75         read(33, *) n
76         allocate(A(m, n))
77
78         do i = 1, m
79             read(33,*) A(i,:)
80         end do
81
82         close(33)
83     end subroutine
84
85     subroutine print_vector(x, n)
86         implicit none
87
88         integer :: n
89         double precision :: x(n)
90
91         integer :: i
92
93 30         format(' /', F10.5, '/')
94
95         do i = 1, n
96             write(*, 30) x(i)

```

```

97         end do
98     end subroutine
99
100    subroutine read_vector(fname, b, t)
101        implicit none
102        character(len=*) :: fname
103        integer :: t
104        double precision, allocatable :: b(:)
105
106        open(unit=33, file=fname, status='old', action='read')
107        read(33, *) t
108        allocate(b(t))
109
110        read(33,*) b(:)
111
112        close(33)
113    end subroutine
114
115    ! ===== Matrix Methods =====
116    function rand_vector(n) result (x)
117        implicit none
118        integer :: n
119        double precision :: x (n)
120
121        integer :: i
122
123        do i = 1, n
124            x(i) = 2 * ran(0) - 1
125        end do
126        return
127    end function
128
129    function rand_matrix(m, n) result (A)
130        implicit none
131        integer :: m, n
132        double precision :: A(m, n)
133
134        integer :: i
135
136        do i = 1, m
137            A(i, :) = rand_vector(n)
138        end do
139        return
140    end function
141
142    function id_matrix(n) result (A)
143        implicit none
144
145        integer :: n
146        double precision :: A(n, n)
147
148        integer :: j
149

```

```

150         A(:, :) = 0.0D0
151
152         do j = 1, n
153             A(j, j) = 1.0D0
154         end do
155         return
156     end function
157
158     function given_matrix(A, n, i, j) result (G)
159         implicit none
160
161         integer :: n, i, j
162         double precision :: A(n, n), G(n, n)
163         double precision :: t, c, s
164
165         G(:, :) = id_matrix(n)
166
167         t = 0.5D0 * DATAN2(2.0D0 * A(i, j), A(i, i) - A(j, j))
168         s = DSIN(t)
169         c = DCOS(t)
170
171         G(i, i) = c
172         G(j, j) = c
173         G(i, j) = -s
174         G(j, i) = s
175
176         return
177     end function
178
179
180     function diagonally_dominant(A, n) result (ok)
181         implicit none
182
183         integer :: n
184         double precision :: A(n, n)
185
186         logical :: ok
187         integer :: i
188
189         do i = 1, n
190             if (DABS(A(i, i)) < SUM(DABS(A(i, :i-1))) + SUM(DABS(A(
191                 i, i+1:)))) then
192                 ok = .FALSE.
193                 return
194             end if
195         end do
196         ok = .TRUE.
197         return
198     end function
199
200     recursive function positive_definite(A, n) result (ok)
201     ! Checks wether a matrix is positive definite
202     ! according to Sylvester's criterion.

```



```

202         implicit none
203
204         integer :: n
205         double precision A(n, n)
206
207         logical :: ok
208
209         if (n == 1) then
210             ok = (A(1, 1) > 0)
211             return
212         else
213             ok = positive_definite(A(:n-1, :n-1), n-1) .AND. (det(A
214                 , n) > 0)
215             return
216         end if
217     end function
218
219 ! function symmetrical(A, n) result (ok)
220     ! Check if the Matrix is symmetrical
221     integer :: n
222
223     double precision :: A(n, n)
224
225     integer :: i, j
226     logical :: ok
227
228     do i = 1, n
229         do j = 1, i-1
230             if (A(i, j) /= A(j, i)) then
231                 ok = .FALSE.
232                 return
233             end if
234         end do
235     end do
236     ok = .TRUE.
237     return
238 end function
239
240 subroutine swap_rows(A, i, j, n)
241     implicit none
242
243     integer :: n
244     integer :: i, j
245     double precision A(n, n)
246     double precision temp(n)
247
248     temp(:) = A(i, :)
249     A(i, :) = A(j, :)
250     A(j, :) = temp(:)
251 end subroutine
252
253 function row_max(A, j, n) result(k)
254     implicit none

```

```

254
255     integer :: n
256     double precision A(n, n)
257
258     integer :: i, j, k
259     double precision :: s
260
261     s = 0.0D0
262     do i = j, n
263         if (A(i, j) > s) then
264             s = A(i, j)
265             k = i
266         end if
267     end do
268     return
269 end function
270
271 function pivot_matrix(A, n) result (P)
272     implicit none
273
274     integer :: n
275     double precision :: A(n, n)
276
277     double precision :: P(n, n)
278
279     integer :: j, k
280
281     P = id_matrix(n)
282
283     do j = 1, n
284         k = row_max(A, j, n)
285         if (j /= k) then
286             call swap_rows(P, j, k, n)
287         end if
288     end do
289     return
290 end function
291
292 function vector_norm(x, n) result (s)
293     implicit none
294
295     integer :: n
296     double precision :: x(n)
297
298     double precision :: s
299
300     s = sqrt(dot_product(x, x))
301     return
302 end function
303
304 function matrix_norm(A, n) result (s)
305     ! Frobenius norm
306     implicit none

```

```

307         integer :: n
308         double precision :: A(n, n)
309         double precision :: s
310
311         s = DSQRT(SUM(A * A))
312         return
313     end function
314
315     function spectral_radius(A, n) result (r)
316         implicit none
317
318         integer :: n
319         double precision :: A(n, n), x(n)
320         double precision :: r, l
321
322         logical :: ok
323
324         ok = power_method(A, n, x, l)
325
326         r = DABS(l)
327         return
328     end function
329
330     recursive function det(A, n) result (d)
331         implicit none
332
333         integer :: n
334         double precision :: A(n, n)
335         double precision :: X(n-1, n-1)
336
337         integer :: i
338         double precision :: d, s
339
340         if (n == 1) then
341             d = A(1, 1)
342             return
343         elseif (n == 2) then
344             d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
345             return
346         else
347             d = 0.0D0
348             s = 1.0D0
349             do i = 1, n
350                 ! Compute submatrix X
351                 X(:, :i-1) = A(2:, :i-1)
352                 X(:, i: ) = A(2:, i+1: )
353
354                 d = s * det(X, n-1) * A(1, i) + d
355                 s = -s
356             end do
357         end if
358         return
359     end function

```

```

360
361     function LU_det(A, n) result (d)
362         implicit none
363
364         integer :: n
365         integer :: i
366         double precision :: A(n, n), L(n, n), U(n, n)
367         double precision :: d
368
369         d = 0.0D0
370
371         if (.NOT. LU_decomp(A, L, U, n)) then
372             call ill_cond()
373             return
374         end if
375
376         do i = 1, n
377             d = d * L(i, i) * U(i, i)
378         end do
379
380         return
381     end function
382
383     subroutine LU_matrix(A, L, U, n)
384         ! Splits Matrix in Lower and Upper-Triangular
385         implicit none
386
387         integer :: n
388         double precision :: A(n, n), L(n, n), U(n, n)
389
390         integer :: i
391
392         L(:, :) = 0.0D0
393         U(:, :) = 0.0D0
394
395         do i = 1, n
396             L(i, i) = 1.0D0
397             L(i, :i-1) = A(i, :i-1)
398             U(i, i:) = A(i, i:)
399         end do
400     end subroutine
401
402     ! === Matrix Factorization Conditions ===
403     function Cholesky_cond(A, n) result (ok)
404         implicit none
405
406         integer :: n
407         double precision :: A(n, n)
408
409         logical :: ok
410
411         ok = symmetrical(A, n) .AND. positive_definite(A, n)
412         return

```

```

413
414     end function
415
416     function PLU_cond(A, n) result (ok)
417         implicit none
418
419         integer :: n
420         double precision A(n, n)
421
422         integer :: i, j
423         double precision :: s
424
425         logical :: ok
426
427         do j = 1, n
428             s = 0.0D0
429             do i = 1, j
430                 if (A(i, j) > s) then
431                     s = A(i, j)
432                 end if
433             end do
434         end do
435
436         ok = (s < 0.01D0)
437
438         return
439     end function
440
441     function LU_cond(A, n) result (ok)
442         implicit none
443
444         integer :: n
445         double precision A(n, n)
446
447         logical :: ok
448
449         ok = positive_definite(A, n)
450
451         return
452     end function
453
454     !
455     !
456     !
457     !
458     !
459     !
460
461     !
462     !
463     !
464     !
465     !

```

===== Matrix Factorization Methods =====

```

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

    integer :: n

```

```

466         double precision :: A(n,n), P(n,n), L(n,n), U(n,n)
467
468         logical :: ok
469
470         !
471         Permutation Matrix
472         P = pivot_matrix(A, n)
473
474         !
475         Decomposition over Row-Swapped Matrix
476         ok = LU_decomp(matmul(P, A), L, U, n)
477         return
478     end function
479
480     function LU_decomp(A, L, U, n) result (ok)
481         implicit none
482
483         integer :: n
484         double precision :: A(n, n), L(n, n), U(n,n), M(n, n)
485
486         logical :: ok
487
488         integer :: i, j, k
489
490         !
491         Results Matrix
492         M(:, :) = A(:, :)
493
494         if (.NOT. LU_cond(A, n)) then
495             call ill_cond()
496             ok = .FALSE.
497             return
498         end if
499
500         do k = 1, n-1
501             do i = k+1, n
502                 M(i, k) = M(i, k) / M(k, k)
503             end do
504
505             do j = k+1, n
506                 do i = k+1, n
507                     M(i, j) = M(i, j) - M(i, k) * M(k, j)
508                 end do
509             end do
510         end do
511
512         !
513         Splits M into L & U
514         call LU_matrix(M, L, U, n)
515
516         ok = .TRUE.
517         return
518     end function
519
520     function Cholesky_decomp(A, L, n) result (ok)
521         implicit none

```

```

519
520     integer :: n
521     double precision :: A(n, n), L(n, n)
522
523     logical :: ok
524
525     integer :: i, j
526
527     if (.NOT. Cholesky_cond(A, n)) then
528         call ill_cond()
529         ok = .FALSE.
530         return
531     end if
532
533     do i = 1, n
534         L(i, i) = sqrt(A(i, i) - sum(L(i, :i-1) * L(i, :i-1)))
535         do j = 1 + 1, n
536             L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)))
537                 / L(i, i)
538         end do
539     end do
540
541     ok = .TRUE.
542     return
543 end function
544
545 function Jacobi_cond(A, n) result (ok)
546     implicit none
547
548     integer :: n
549
550     double precision :: A(n, n)
551
552     logical :: ok
553
554     if (.NOT. spectral_radius(A, n) < 1.0D0) then
555         ok = .FALSE.
556         call ill_cond()
557         return
558     else
559         ok = .TRUE.
560         return
561     end if
562 end function
563
564 function Jacobi(A, x, b, e, n) result (ok)
565     implicit none
566
567     integer :: n
568
569     double precision :: A(n, n)
570     double precision :: b(n), x(n), x0(n)
571     double precision :: e

```

```

571
572     logical :: ok
573
574     integer :: i, k
575
576     x0 = rand_vector(n)
577
578     ok = Jacobi_cond(A, n)
579
580     if (.NOT. ok) then
581         return
582     end if
583
584     do k = 1, KMAX
585         do i = 1, n
586             x(i) = (b(i) - dot_product(A(i, :), x0)) / A(i, i)
587         end do
588         x0(:) = x(:)
589         e = vector_norm(matmul(A, x) - b, n)
590         if (e < TOL) then
591             return
592         end if
593     end do
594     call error('Erro: Esse método não convergiu.')
595     ok = .FALSE.
596     return
597 end function
598
599 function Gauss_Seidel_cond(A, n) result (ok)
600     implicit none
601
602     integer :: n
603
604     double precision :: A(n, n)
605
606     logical :: ok
607
608     integer :: i
609
610     do i = 1, n
611         if (A(i, i) == 0.0D0) then
612             ok = .FALSE.
613             call ill_cond()
614             return
615         end if
616     end do
617
618     if (symmetrical(A, n) .AND. positive_definite(A, n)) then
619         ok = .TRUE.
620         return
621     else
622         call warn('Aviso: Esse método pode não convergir.')
623         return

```



```

624         end if
625     end function
626
627     function Gauss_Seidel(A, x, b, e, n) result (ok)
628         implicit none
629
630         integer :: n
631
632         double precision :: A(n, n)
633         double precision :: b(n), x(n)
634         double precision :: e, s
635
636         logical :: ok
637         integer :: i, j, k
638
639         ok = Gauss_Seidel_cond(A, n)
640
641         if (.NOT. ok) then
642             return
643         end if
644
645         do k = 1, KMAX
646             do i = 1, n
647                 s = 0.0D0
648                 do j = 1, n
649                     if (i /= j) then
650                         s = s + A(i, j) * x(j)
651                     end if
652                 end do
653                 x(i) = (b(i) - s) / A(i, i)
654             end do
655             e = vector_norm(matmul(A, x) - b, n)
656             if (e < TOL) then
657                 return
658             end if
659         end do
660         call error('Erro: Esse método não convergiu.')
661         ok = .FALSE.
662         return
663     end function
664
665 ! Decomposição LU e afins
666
667     subroutine LU_backsub(L, U, x, y, b, n)
668         implicit none
669
670         integer :: n
671
672         double precision :: L(n, n), U(n, n)
673         double precision :: b(n), x(n), y(n)
674
675         integer :: i
676

```

```

677 !      Ly = b (Forward Substitution)
678 do i = 1, n
679     y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i)
680 end do
681
682 !      Ux = y (Backsubstitution)
683 do i = n, 1, -1
684     x(i) = (y(i) - SUM(U(i,i+1:n) * x(i+1:n))) / U(i, i)
685 end do
686
687 end subroutine
688
689 function LU_solve(A, x, y, b, n) result (ok)
690     implicit none
691
692     integer :: n
693
694     double precision :: A(n, n), L(n, n), U(n, n)
695     double precision :: b(n), x(n), y(n)
696
697     logical :: ok
698
699     ok = LU_decomp(A, L, U, n)
700
701     if (.NOT. ok) then
702         return
703     end if
704
705     call LU_backsub(L, U, x, y, b, n)
706
707     return
708 end function
709
710 function PLU_solve(A, x, y, b, n) result (ok)
711     implicit none
712
713     integer :: n
714
715     double precision :: A(n, n), P(n,n), L(n, n), U(n, n)
716     double precision :: b(n), x(n), y(n)
717
718     logical :: ok
719
720     ok = PLU_decomp(A, P, L, U, n)
721
722     if (.NOT. ok) then
723         return
724     end if
725
726     call LU_backsub(L, U, x, y, matmul(P, b), n)
727
728     x(:) = matmul(P, x)
729

```

```

730         return
731     end function
732
733     function Cholesky_solve(A, x, y, b, n) result (ok)
734         implicit none
735
736         integer :: n
737
738         double precision :: A(n, n), L(n, n), U(n, n)
739         double precision :: b(n), x(n), y(n)
740
741         logical :: ok
742
743         ok = Cholesky_decomp(A, L, n)
744
745         if (.NOT. ok) then
746             return
747         end if
748
749         U = transpose(L)
750
751         call LU_backsub(L, U, x, y, b, n)
752
753         return
754     end function
755
756     !
757     !
758     !
759     !
760     !
761     !
762     !
763
764     ! ===== Power Method =====
765     function power_method(A, n, x, l) result (ok)
766         implicit none
767         integer :: n
768         integer :: k = 0
769
770         double precision :: A(n, n)
771         double precision :: x(n)
772         double precision :: l, ll
773
774         logical :: ok
775
776     !
777     !
778     !
779     !
780     !
781     !

```

zero

```

x(:) = rand_vector(n)
x(1) = 1.0D0

Initialize Eigenvalues
l = 0.0D0

```

```

782
783 !           Checks if error tolerance was reached
784 do while (k < MAX_ITER)
785     ll = 1
786
787     x(:) = matmul(A, x)
788
789 !           Retrieve Eigenvalue
790     l = x(1)
791
792 !           Retrieve Eigenvector
793     x(:) = x(:) / l
794
795     if (dabs((l - ll) / l) < TOL) then
796         ok = .TRUE.
797         return
798     else
799         k = k + 1
800         continue
801     end if
802 end do
803 ok = .FALSE.
804 return
805 end function
806
807 function Jacobi_eigen(A, n, L, X) result (ok)
808     implicit none
809     integer :: n, i, j, u, v
810     integer :: k = 0
811
812     double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
813     double precision :: y, z
814
815     logical :: ok
816
817     X(:, :) = id_matrix(n)
818     L(:, :) = A(:, :)
819
820     do while (k < MAX_ITER)
821         z = 0.0D0
822         do i = 1, n
823             do j = 1, i - 1
824                 y = DABS(L(i, j))
825
826 !           Found new maximum absolute value
827                 if (y > z) then
828                     u = i
829                     v = j
830                     z = y
831                 end if
832             end do
833         end do
834

```

```

835         if (z >= TOL) then
836             P(:, :) = given_matrix(L, n, u, v)
837             L(:, :) = matmul(matmul(transpose(P), L), P)
838             X(:, :) = matmul(X, P)
839             k = k + 1
840         else
841             ok = .TRUE.
842             return
843         end if
844     end do
845     ok = .FALSE.
846     return
847 end function
848
849 !
850 !
851 !
852 !
853 !
854 !
855 !
856
857 function least_squares(x, y, s, n) result (ok)
858     implicit none
859     integer :: n
860
861     logical :: ok
862
863     double precision :: A(2,2), b(2), s(2), r(2), x(n), y(n)
864
865     A(1, 1) = n
866     A(1, 2) = SUM(x)
867     A(2, 1) = SUM(x)
868     A(2, 2) = dot_product(x, x)
869
870     b(1) = SUM(y)
871     b(2) = dot_product(x, y)
872
873     ok = Cholesky_solve(A, s, r, b, n)
874     return
875 end function
876
877 end module Matrix

```