

COC473 - Lista 2

Pedro Maciel Xavier
116023847

22 de setemebro de 2019

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.

Algoritmo 1

```
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     ! Inicializa vetor aleatório e define primeira componente como '1'
13     x(:) = rand_vector(n)
14     x(1) = 1.0D0
15
16     l = 0.0D0
17
18     do while (k < MAX_ITER)
19         ll = l
20
21         x(:) = matmul(A, x)
22
23         ! Obtém autovalor
24         l = x(1)
25
26         ! Obtém autovetor
27         x(:) = x(:) / l
28
29         ! Verifica se a tolerância foi atendida
30         if (DABS((l - ll) / l) < TOL) then
31             ok = .TRUE.
32             return
33         else
34             k = k + 1
35             continue
36         end if
37     end do
38     ok = .FALSE.
39     return
40 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

```
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                 !           Encontrado um novo valor absoluto máximo
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 λ_i deste polinômio, podemos afirmar que $\lambda = 3$ é um dos autovalores. Usando a fórmula de Bhaskara, encontramos os demais.

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

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 λ_{\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 fatoração $\mathbf{A} = \mathbf{LL}^T$:

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

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*

3.: *Gauss-Seidel*

4.: Autovalores e autovetores

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

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

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

```
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.')
41     end subroutine
42
43     subroutine print_matrix(A, m, n)
44         implicit none
45
46         integer :: m, n
47         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    recursive function det(A, n) result (d)
117        implicit none
118
119        integer :: n
120        double precision :: A(n, n)
121        double precision :: X(n-1, n-1)
122
123        integer :: i
124        double precision :: d, s
125
126        if (n == 1) then
127            d = A(1, 1)
128            return
129        elseif (n == 2) then
130            d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
131            return
132        else
133            d = 0.0D0
134            s = 1.0D0
135            do i = 1, n
136                ! Compute submatrix X
137                X(:, :i-1) = A(2:, :i-1)
138                X(:, i: ) = A(2:, i+1: )
139
140                d = s * det(X, n-1) * A(1, i) + d
141                s = -s
142            end do
143        end if
144        return
145    end function
146
147    function rand_vector(n) result (x)
148        implicit none
149        integer :: n
150        double precision :: x (n)

```

```

151
152         integer :: i
153
154         do i = 1, n
155             x(i) = 2 * ran(0) - 1
156         end do
157         return
158     end function
159
160     function rand_matrix(m, n) result (A)
161         implicit none
162         integer :: m, n
163         double precision :: A(m, n)
164
165         integer :: i
166
167         do i = 1, m
168             A(i, :) = rand_vector(n)
169         end do
170         return
171     end function
172
173     function id_matrix(n) result (A)
174         implicit none
175
176         integer :: n
177         double precision :: A(n, n)
178
179         integer :: j
180
181         A(:, :) = 0.0D0
182
183         do j = 1, n
184             A(j, j) = 1.0D0
185         end do
186         return
187     end function
188
189     function given_matrix(A, n, i, j) result (G)
190         implicit none
191
192         integer :: n, i, j
193         double precision :: A(n, n), G(n, n)
194         double precision :: t, c, s
195
196         G(:, :) = id_matrix(n)
197
198         t = 0.5D0 * DATAN2(2.0D0 * A(i,j), A(i, i) - A(j, j))
199         s = DSIN(t)
200         c = DCOS(t)
201
202         G(i, i) = c
203         G(j, j) = c

```

```

204         G(i, j) = -s
205         G(j, i) = s
206
207         return
208     end function
209
210
211     function diagonally_dominant(A, n) result (ok)
212         implicit none
213
214         integer :: n
215         double precision :: A(n, n)
216
217         logical :: ok
218         integer :: i
219
220         do i = 1, n
221             if (DABS(A(i, i)) < SUM(DABS(A(i, :i-1))) + SUM(DABS(A(
222                 i, i+1:)))) then
223                 ok = .FALSE.
224                 return
225             end if
226         end do
227         ok = .TRUE.
228         return
229     end function
230
231     recursive function positive_definite(A, n) result (ok)
232     ! Checks wether a matrix is positive definite
233     ! according to Sylvester's criterion.
234         implicit none
235
236         integer :: n
237         double precision A(n, n)
238
239         logical :: ok
240
241         if (n == 1) then
242             ok = (A(1, 1) > 0)
243             return
244         else
245             ok = positive_definite(A(:n-1, :n-1), n-1) .AND. (det(A
246                 , n) > 0)
247             return
248         end if
249     end function
250
251     function symmetrical(A, n) result (ok)
252     ! Check if the Matrix is symmetrical
253         integer :: n
254
255         double precision :: A(n, n)

```

```

255     integer :: i, j
256     logical :: ok
257
258     do i = 1, n
259         do j = 1, i-1
260             if (A(i, j) /= A(j, i)) then
261                 ok = .FALSE.
262                 return
263             end if
264         end do
265     end do
266     ok = .TRUE.
267     return
268 end function
269
270 subroutine swap_rows(A, i, j, n)
271     implicit none
272
273     integer :: n
274     integer :: i, j
275     double precision A(n, n)
276     double precision temp(n)
277
278     temp(:) = A(i, :)
279     A(i, :) = A(j, :)
280     A(j, :) = temp(:)
281 end subroutine
282
283 function row_max(A, j, n) result(k)
284     implicit none
285
286     integer :: n
287     double precision A(n, n)
288
289     integer :: i, j, k
290     double precision :: s
291
292     s = 0.0D0
293     do i = j, n
294         if (A(i, j) > s) then
295             s = A(i, j)
296             k = i
297         end if
298     end do
299     return
300 end function
301
302 function pivot_matrix(A, n) result (P)
303     implicit none
304
305     integer :: n
306     double precision :: A(n, n)
307

```



```

308         double precision :: P(n, n)
309
310         integer :: j, k
311
312         P = id_matrix(n)
313
314         do j = 1, n
315             k = row_max(A, j, n)
316             if (j /= k) then
317                 call swap_rows(P, j, k, n)
318             end if
319         end do
320         return
321     end function
322
323     function vector_norm(x, n) result (s)
324         implicit none
325
326         integer :: n
327         double precision :: x(n)
328
329         double precision :: s
330
331         s = sqrt(dot_product(x, x))
332         return
333     end function
334
335     function matrix_norm(A, n) result (s)
336         ! Frobenius norm
337         implicit none
338         integer :: n
339         double precision :: A(n, n)
340         double precision :: s
341
342         s = DSQRT(SUM(A * A))
343         return
344     end function
345
346     function spectral_radius(A, n) result (r)
347         implicit none
348
349         integer :: n
350         double precision :: A(n, n), M(n, n)
351         double precision :: r
352
353         integer :: i, j, k
354
355         M(:, :) = A(:, :)
356
357         r = 1.0D0
358
359         do k = 1, KMAX
360             M = MATMUL(M, M)

```

```

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

```

```

414         L(i, i) = 1.0D0
415         L(i, :i-1) = A(i, :i-1)
416         U(i, i: ) = A(i, i: )
417     end do
418 end subroutine
419
420 ! === Matrix Factorization Conditions ===
421 function Cholesky_cond(A, n) result (ok)
422     implicit none
423
424     integer :: n
425     double precision :: A(n, n)
426
427     logical :: ok
428
429     ok = symmetrical(A, n) .AND. positive_definite(A, n)
430     return
431
432 end function
433
434 function PLU_cond(A, n) result (ok)
435     implicit none
436
437     integer :: n
438     double precision A(n, n)
439
440     integer :: i, j
441     double precision :: s
442
443     logical :: ok
444
445     do j = 1, n
446         s = 0.0D0
447         do i = 1, j
448             if (A(i, j) > s) then
449                 s = A(i, j)
450             end if
451         end do
452     end do
453
454     ok = (s < 0.01D0)
455
456     return
457 end function
458
459 function LU_cond(A, n) result (ok)
460     implicit none
461
462     integer :: n
463     double precision A(n, n)
464
465     logical :: ok
466

```

```

467         ok = positive_definite(A, n)
468
469         return
470     end function
471 !
472 !      /- - - - - \ / - - - - - \ / - - - - - \ / - - - - - \ / - - - - - \
473 !      / /      / /      / /      / /      / /      / /      / /      / /
474 !      / /      / /      / /      / /      / /      / /      / /      / /
475 !      / /      / /      / /      / /      / /      / /      / /      / /
476 !      / /      / /      / /      / /      / /      / /      / /      / /
477 !      / /      / /      / /      / /      / /      / /      / /      / /
478 !      / /      / /      / /      / /      / /      / /      / /      / /
479 !      / /      / /      / /      / /      / /      / /      / /      / /
480 !      / /      / /      / /      / /      / /      / /      / /      / /
481 !      / /      / /      / /      / /      / /      / /      / /      / /
482 !      / /      / /      / /      / /      / /      / /      / /      / /
483 !      / /      / /      / /      / /      / /      / /      / /      / /
484 !      / /      / /      / /      / /      / /      / /      / /      / /
485 !      / /      / /      / /      / /      / /      / /      / /      / /
486 !      / /      / /      / /      / /      / /      / /      / /      / /
487 !      / /      / /      / /      / /      / /      / /      / /      / /
488 !      / /      / /      / /      / /      / /      / /      / /      / /
489 !      / /      / /      / /      / /      / /      / /      / /      / /
490 !      / /      / /      / /      / /      / /      / /      / /      / /
491 !      / /      / /      / /      / /      / /      / /      / /      / /
492 !      / /      / /      / /      / /      / /      / /      / /      / /
493 !      / /      / /      / /      / /      / /      / /      / /      / /
494 !      / /      / /      / /      / /      / /      / /      / /      / /
495 !      / /      / /      / /      / /      / /      / /      / /      / /
496 !      / /      / /      / /      / /      / /      / /      / /      / /
497 !      / /      / /      / /      / /      / /      / /      / /      / /
498 !      / /      / /      / /      / /      / /      / /      / /      / /
499 !      / /      / /      / /      / /      / /      / /      / /      / /
500 !      / /      / /      / /      / /      / /      / /      / /      / /
501 !      / /      / /      / /      / /      / /      / /      / /      / /
502 !      / /      / /      / /      / /      / /      / /      / /      / /
503 !      / /      / /      / /      / /      / /      / /      / /      / /
504 !      / /      / /      / /      / /      / /      / /      / /      / /
505 !      / /      / /      / /      / /      / /      / /      / /      / /
506 !      / /      / /      / /      / /      / /      / /      / /      / /
507 !      / /      / /      / /      / /      / /      / /      / /      / /
508 !      / /      / /      / /      / /      / /      / /      / /      / /
509 !      / /      / /      / /      / /      / /      / /      / /      / /
510 !      / /      / /      / /      / /      / /      / /      / /      / /
511 !      / /      / /      / /      / /      / /      / /      / /      / /
512 !      / /      / /      / /      / /      / /      / /      / /      / /
513 !      / /      / /      / /      / /      / /      / /      / /      / /
514 !      / /      / /      / /      / /      / /      / /      / /      / /
515 !      / /      / /      / /      / /      / /      / /      / /      / /
516 !      / /      / /      / /      / /      / /      / /      / /      / /
517 !      / /      / /      / /      / /      / /      / /      / /      / /
518 !      / /      / /      / /      / /      / /      / /      / /      / /
519 !      / /      / /      / /      / /      / /      / /      / /      / /

```

```

479 !      ===== Matrix Factorization Methods =====
480 function PLU_decomp(A, P, L, U, n) result (ok)
481     implicit none
482
483     integer :: n
484     double precision :: A(n,n), P(n,n), L(n,n), U(n,n)
485
486     logical :: ok
487
488     !      Permutation Matrix
489     P = pivot_matrix(A, n)
490
491     !      Decomposition over Row-Swapped Matrix
492     ok = LU_decomp(matmul(P, A), L, U, n)
493     return
494 end function
495
496 function LU_decomp(A, L, U, n) result (ok)
497     implicit none
498
499     integer :: n
500     double precision :: A(n, n), L(n, n), U(n,n), M(n, n)
501
502     logical :: ok
503
504     integer :: i, j, k
505
506     !      Results Matrix
507     M(:, :) = A(:, :)
508
509     if (.NOT. LU_cond(A, n)) then
510         call ill_cond()
511         ok = .FALSE.
512         return
513     end if
514
515     do k = 1, n-1
516         do i = k+1, n
517             M(i, k) = M(i, k) / M(k, k)
518         end do
519

```

```

520         do j = k+1, n
521             do i = k+1, n
522                 M(i, j) = M(i, j) - M(i, k) * M(k, j)
523             end do
524         end do
525     end do
526
527 !     Splits M into L & U
528     call LU_matrix(M, L, U, n)
529
530     ok = .TRUE.
531     return
532
533 end function
534
535 function Cholesky_decomp(A, L, n) result (ok)
536     implicit none
537
538     integer :: n
539     double precision :: A(n, n), L(n, n)
540
541     logical :: ok
542
543     integer :: i, j
544
545     if (.NOT. Cholesky_cond(A, n)) then
546         call ill_cond()
547         ok = .FALSE.
548         return
549     end if
550
551     do i = 1, n
552         L(i, i) = sqrt(A(i, i) - sum(L(i, :i-1) * L(i, :i-1)))
553         do j = 1 + 1, n
554             L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)))
555                 / L(i, i)
556         end do
557     end do
558
559     ok = .TRUE.
560     return
561 end function
562 !
563 === Linear System Solving Conditions ===
564 function Jacobi_cond(A, n) result (ok)
565     implicit none
566
567     integer :: n
568
569     double precision :: A(n, n)
570
571     logical :: ok

```

```

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

```

```

624
625         ok = Jacobi_cond(A, n)
626
627         if (.NOT. ok) then
628             return
629         end if
630
631         do k = 1, KMAX
632             do i = 1, n
633                 x(i) = (b(i) - dot_product(A(i, :), x0)) / A(i, i)
634             end do
635             x0(:) = x(:)
636             e = vector_norm(matmul(A, x) - b, n)
637             if (e < TOL) then
638                 return
639             end if
640         end do
641         call error('Erro: Esse método não convergiu.')
642         ok = .FALSE.
643         return
644     end function
645
646     function Gauss_Seidel(A, x, b, e, n) result (ok)
647         implicit none
648
649         integer :: n
650
651         double precision :: A(n, n)
652         double precision :: b(n), x(n)
653         double precision :: e, s
654
655         logical :: ok
656         integer :: i, j, k
657
658         ok = Gauss_Seidel_cond(A, n)
659
660         if (.NOT. ok) then
661             return
662         end if
663
664         do k = 1, KMAX
665             do i = 1, n
666                 s = 0.0D0
667                 do j = 1, n
668                     if (i /= j) then
669                         s = s + A(i, j) * x(j)
670                     end if
671                 end do
672                 x(i) = (b(i) - s) / A(i, i)
673             end do
674             e = vector_norm(matmul(A, x) - b, n)
675             if (e < TOL) then
676                 return

```

```

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

```



```

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

```

```

783         implicit none
784         integer :: n
785         integer :: k = 0
786
787         double precision :: A(n, n)
788         double precision :: x(n)
789         double precision :: l, ll
790
791         logical :: ok
792
793         ! Begin with random normal vector and set 1st component to
       zero
794         x(:) = rand_vector(n)
795         x(1) = 1.0D0
796
797         ! Initialize Eigenvalues
798         l = 0.0D0
799
800         ! Checks if error tolerance was reached
801         do while (k < MAX_ITER)
802             ll = l
803
804             x(:) = matmul(A, x)
805
806             ! Retrieve Eigenvalue
807             l = x(1)
808
809             ! Retrieve Eigenvector
810             x(:) = x(:) / l
811
812             if (dabs((l - ll) / l) < TOL) then
813                 ok = .TRUE.
814                 return
815             else
816                 k = k + 1
817                 continue
818             end if
819         end do
820         ok = .FALSE.
821         return
822     end function
823
824     function Jacobi_eigen(A, n, L, X) result (ok)
825         implicit none
826         integer :: n, i, j, u, v
827         integer :: k = 0
828
829         double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
830         double precision :: y, z
831
832         logical :: ok
833
834         X(:, :) = id_matrix(n)

```

```

835     L(:, :) = A(:, :)
836
837     do while (k < MAX_ITER)
838         z = 0.0D0
839         do i = 1, n
840             do j = 1, i - 1
841                 y = DABS(L(i, j))
842
843             !
844                 Found new maximum absolute value
845                 if (y > z) then
846                     u = i
847                     v = j
848                     z = y
849                 end if
850             end do
851         end do
852
853         if (z >= TOL) then
854             P(:, :) = given_matrix(L, n, u, v)
855             L(:, :) = matmul(matmul(transpose(P), L), P)
856             X(:, :) = matmul(X, P)
857             k = k + 1
858         else
859             ok = .TRUE.
860             return
861         end if
862     end do
863     ok = .FALSE.
864     return
865 end function
866
867 !
868 !
869 !
870 !
871 !
872 !
873
874 function least_squares(x, y, s, n) result (ok)
875     implicit none
876     integer :: n
877
878     logical :: ok
879
880     double precision :: A(2,2), b(2), s(2), r(2), x(n), y(n)
881
882     A(1, 1) = n
883     A(1, 2) = SUM(x)
884     A(2, 1) = SUM(x)
885     A(2, 2) = dot_product(x, x)
886
887     b(1) = SUM(y)

```

```
888         b(2) = dot_product(x, y)
889
890         ok = Cholesky_solve(A, s, r, b, n)
891         return
892     end function
893
894 end module Matrix
```