

COC473 - Lista 1

Pedro Maciel Xavier
116023847

25 de setembro de 2020

Questão 1.:

Abaixo, o passo-a-passo da resolução do sistema $\mathbf{Ax} = \mathbf{b}$. Ao lado de cada etapa, a matriz de combinação de linhas \mathbf{M} .

$$[\mathbf{A}|\mathbf{b}]^{(0)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ -4 & 6 & -4 & 1 & 2 \\ 1 & -4 & 6 & -4 & 1 \\ 0 & 1 & -4 & 5 & 3 \end{array} \right] \mathbf{M}^{(1)} = \left[\begin{array}{cccc} 1 & & & \\ \frac{4}{5} & 1 & & \\ -\frac{1}{5} & & 1 & \\ & & & 1 \end{array} \right]$$

$$[\mathbf{A}|\mathbf{b}]^{(1)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & \frac{6}{5} \\ 0 & -\frac{16}{5} & \frac{29}{5} & -4 & \frac{6}{5} \\ 0 & 1 & -4 & 5 & 3 \end{array} \right] \mathbf{M}^{(2)} = \left[\begin{array}{cccc} 1 & & & \\ & 1 & & \\ & \frac{8}{7} & 1 & \\ & -\frac{5}{14} & & 1 \end{array} \right]$$

$$[\mathbf{A}|\mathbf{b}]^{(2)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & \frac{6}{5} \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{18}{7} \\ 0 & 0 & -\frac{20}{7} & \frac{65}{14} & \frac{18}{7} \end{array} \right] \mathbf{M}^{(3)} = \left[\begin{array}{cccc} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & \frac{4}{3} & 1 \end{array} \right]$$

$$[\mathbf{A}|\mathbf{b}]^{(3)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & \frac{6}{5} \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{18}{7} \\ 0 & 0 & 0 & \frac{5}{6} & 6 \end{array} \right]$$

Por substituição, chegamos ao resultado

$$\mathbf{x} = \frac{1}{5} \begin{bmatrix} 29 \\ 51 \\ 54 \\ 36 \end{bmatrix}$$

Questão 2.:

a) Decomposição LU e de *Cholesky*

Segue a baixo a definição das funções que realizam, respectivamente, a decomposição LU e a de *Cholesky*. Funções auxiliares se encontram no código completo, disponível no apêndice.

```
1      function LU_decomp(A, L, U, n) result (ok)
2          implicit none
3
4          integer :: n
5          double precision :: A(n, n), L(n, n), U(n,n), M(n, n)
6
7          logical :: ok
8
9          integer :: i, j, k
10
11      !      Results Matrix
12      M(:, :) = A(:, :)
13
14      if (.NOT. LU_cond(A, n)) then
15          call ill_cond()
16          ok = .FALSE.
17          return
18      end if
19
20      do k = 1, n-1
21          do i = k+1, n
22              M(i, k) = M(i, k) / M(k, k)
23          end do
24
25          do j = k+1, n
26              do i = k+1, n
27                  M(i, j) = M(i, j) - M(i, k) * M(k, j)
28              end do
29          end do
30      end do
31
32      !      Splits M into L & U
33      call LU_matrix(M, L, U, n)
34
35      ok = .TRUE.
36      return
37
38  end function
39
40  function Cholesky_decomp(A, L, n) result (ok)
41      implicit none
42
43      integer :: n
44      double precision :: A(n, n), L(n, n)
45
46      logical :: ok
```

```

47
48     integer :: i, j
49
50     if (.NOT. Cholesky_cond(A, n)) then
51         call ill_cond()
52         ok = .FALSE.
53         return
54     end if
55
56     do i = 1, n
57         L(i, i) = sqrt(A(i, i) - sum(L(i, :i-1) * L(i, :i-1)))
58         do j = 1 + 1, n
59             L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)))
60                 / L(i, i)
61         end do
62     end do
63
64     ok = .TRUE.
65     return
66 end function

```

b) Resolução de um sistema $Ax = b$

A partir do resultado da decomposição LU temos um par de rotinas para resolver o sistema linear relacionado:

```

1     subroutine LU_backsub(L, U, x, y, b, n)
2         implicit none
3
4         integer :: n
5
6         double precision :: L(n, n), U(n, n)
7         double precision :: b(n), x(n), y(n)
8
9         integer :: i
10
11     ! Ly = b (Forward Substitution)
12     do i = 1, n
13         y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i)
14     end do
15
16     ! Ux = y (Backsubstitution)
17     do i = n, 1, -1
18         x(i) = (y(i) - SUM(U(i, i+1:n) * x(i+1:n))) / U(i, i)
19     end do
20
21 end subroutine
22
23 function LU_solve(A, x, y, b, n) result (ok)
24     implicit none
25
26     integer :: n
27

```

```

28     double precision :: A(n, n), L(n, n), U(n, n)
29     double precision :: b(n), x(n), y(n)
30
31     logical :: ok
32
33     ok = LU_decomp(A, L, U, n)
34
35     if (.NOT. ok) then
36         return
37     end if
38
39     call LU_backsub(L, U, x, y, b, n)
40
41     return
42 end function

```

c) Cálculo do determinante $\det(\mathbf{A})$

Aqui estão apresentadas duas rotinas para o cálculo do determinante. Uma através do algoritmo recursivo usual (Teorema de *Laplace*) e outra a partir da decomposição LU.

```

1     recursive function det(A, n) result (d)
2         implicit none
3
4         integer :: n
5         double precision :: A(n, n)
6         double precision :: X(n-1, n-1)
7
8         integer :: i
9         double precision :: d, s
10
11     if (n == 1) then
12         d = A(1, 1)
13         return
14     elseif (n == 2) then
15         d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
16         return
17     else
18         d = 0.0D0
19         s = 1.0D0
20         do i = 1, n
21             ! Compute submatrix X
22             X(:, :i-1) = A(2:, :i-1)
23             X(:, i:) = A(2:, i+1:)
24
25             d = s * det(X, n-1) * A(1, i) + d
26             s = -s
27         end do
28     end if
29     return
30 end function
31
32 function LU_det(A, n) result (d)

```

```

33      implicit none
34
35      integer :: n
36      integer :: i
37      double precision :: A(n, n), L(n, n), U(n, n)
38      double precision :: d
39
40      d = 0.0D0
41
42      if (.NOT. LU_decomp(A, L, U, n)) then
43          call ill_cond()
44          return
45      end if
46
47      do i = 1, n
48          d = d * L(i, i) * U(i, i)
49      end do
50
51      return
52  end function

```

Questão 3.:

1 .: *Jacobi*

Segue o algoritmo iterativo de *Jacobi* para solução de sistemas lineares, com os respectivos sinais relacionados à convergência do método.

```
1      function Jacobi_cond(A, n) result (ok)
2          implicit none
3
4          integer :: n
5
6          double precision :: A(n, n)
7
8          logical :: ok
9
10         if (.NOT. spectral_radius(A, n) < 1.0D0) then
11             ok = .FALSE.
12             call ill_cond()
13             return
14         else
15             ok = .TRUE.
16             return
17         end if
18     end function
19
20     function Jacobi(A, x, b, e, n) result (ok)
21         implicit none
22
23         integer :: n
24
25         double precision :: A(n, n)
26         double precision :: b(n), x(n), x0(n)
27         double precision :: e
28
29         logical :: ok
30
31         integer :: i, k
32
33         x0 = rand_vector(n)
34
35         ok = Jacobi_cond(A, n)
36
37         if (.NOT. ok) then
38             return
39         end if
40
41         do k = 1, KMAX
42             do i = 1, n
43                 x(i) = (b(i) - dot_product(A(i, :), x0)) / A(i, i)
44             end do
45             x0(:) = x(:)
46             e = vector_norm(matmul(A, x) - b, n)
```

```

47         if (e < TOL) then
48             return
49         end if
50     end do
51     call error('Erro: Esse método não convergiu.')
52     ok = .FALSE.
53     return
54 end function

```

2 :: Gauss-Seidel

Agora, a implementação da variante de *Gauss-Seidel*, assim como os respectivos avisos quanto à convergência do método.

```

1     function Gauss_Seidel_cond(A, n) result (ok)
2         implicit none
3
4         integer :: n
5
6         double precision :: A(n, n)
7
8         logical :: ok
9
10        integer :: i
11
12        do i = 1, n
13            if (A(i, i) == 0.0D0) then
14                ok = .FALSE.
15                call ill_cond()
16                return
17            end if
18        end do
19
20        if (symmetrical(A, n) .AND. positive_definite(A, n)) then
21            ok = .TRUE.
22            return
23        else
24            call warn('Aviso: Esse método pode não convergir.')
25            return
26        end if
27    end function
28
29    function Gauss_Seidel(A, x, b, e, n) result (ok)
30        implicit none
31
32        integer :: n
33
34        double precision :: A(n, n)
35        double precision :: b(n), x(n)
36        double precision :: e, s
37
38        logical :: ok
39        integer :: i, j, k

```



```

40
41      ok = Gauss_Seidel_cond(A, n)
42
43      if (.NOT. ok) then
44          return
45      end if
46
47      do k = 1, KMAX
48          do i = 1, n
49              s = 0.0D0
50              do j = 1, n
51                  if (i /= j) then
52                      s = s + A(i, j) * x(j)
53                  end if
54              end do
55              x(i) = (b(i) - s) / A(i, i)
56          end do
57          e = vector_norm(matmul(A, x) - b, n)
58          if (e < TOL) then
59              return
60          end if
61      end do
62      call error('Erro: Esse método não convergiu.')
63      ok = .FALSE.
64      return
65  end function

```

Questão 4.:

a) Resolveremos agora o sistema linear $\mathbf{Ax} = \mathbf{b}$ dado por:

$$\mathbf{A} = \begin{bmatrix} 5 & -4 & 1 & 0 \\ -4 & 6 & -4 & 1 \\ 1 & -4 & 6 & -4 \\ 0 & 1 & -4 & 5 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} -1 \\ 2 \\ 1 \\ 3 \end{bmatrix}$$

-Eliminação Gaussiana

Vamos fazer de maneira semelhante a questão 1, mas dessa vez queremos que os coeficientes da diagonal principal sejam todos iguais a 1.

$$[\mathbf{A}|\mathbf{b}]^{(0)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ -4 & 6 & -4 & 1 & 2 \\ 1 & -4 & 6 & -4 & 1 \\ 0 & 1 & -4 & 5 & 3 \end{array} \right] \quad \mathbf{M}^{(1)} = \begin{bmatrix} 1 & & & & \\ \frac{4}{5} & 1 & & & \\ -\frac{1}{5} & & 1 & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix}$$

$$[\mathbf{A}|\mathbf{b}]^{(1)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & \frac{6}{5} \\ 0 & -\frac{16}{5} & \frac{29}{5} & -4 & \frac{6}{5} \\ 0 & 1 & -4 & 5 & 3 \end{array} \right] \quad \mathbf{M}^{(2)} = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & \frac{8}{7} & 1 & & \\ & -\frac{5}{14} & & 1 & \end{bmatrix}$$

$$[\mathbf{A}|\mathbf{b}]^{(2)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & \frac{6}{5} \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{18}{7} \\ 0 & 0 & -\frac{20}{7} & \frac{65}{14} & \frac{18}{7} \end{array} \right] \quad \mathbf{M}^{(3)} = \begin{bmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & \frac{4}{3} & 1 & \end{bmatrix}$$

$$[\mathbf{A}|\mathbf{b}]^{(3)} = \left[\begin{array}{cccc|c} 5 & -4 & 1 & 0 & -1 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 & \frac{6}{5} \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} & \frac{18}{7} \\ 0 & 0 & 0 & \frac{5}{6} & 6 \end{array} \right] \quad \mathbf{M}^{(4)} = \begin{bmatrix} \frac{1}{5} & & & & \\ & \frac{5}{14} & & & \\ & & \frac{7}{15} & & \\ & & & \frac{6}{5} & \end{bmatrix}$$

$$[\mathbf{A}|\mathbf{b}]^{(4)} = \left[\begin{array}{cccc|c} 1 & -\frac{4}{5} & \frac{1}{5} & 0 & -\frac{1}{5} \\ 0 & 1 & -\frac{8}{7} & \frac{5}{14} & \frac{3}{7} \\ 0 & 0 & 1 & -\frac{4}{3} & \frac{6}{5} \\ 0 & 0 & 0 & 1 & \frac{36}{5} \end{array} \right]$$

Substituindo sucessivamente os valores para \mathbf{x}_i obtemos:

$$\mathbf{x} = \frac{1}{5} \begin{bmatrix} 29 \\ 51 \\ 54 \\ 36 \end{bmatrix}$$

-Eliminação de *Gauss-Jordan*

Continuando de onde parou a eliminação Gaussiana seguimos com:

$$[\mathbf{A}|\mathbf{b}]^{(4)} = \left[\begin{array}{cccc|c} 1 & -\frac{4}{5} & \frac{1}{5} & 0 & -\frac{1}{5} \\ 0 & 1 & -\frac{8}{7} & \frac{5}{14} & \frac{3}{7} \\ 0 & 0 & 1 & -\frac{4}{3} & \frac{6}{5} \\ 0 & 0 & 0 & 1 & \frac{36}{5} \end{array} \right] \mathbf{M}^{(5)} = \left[\begin{array}{cccc} 1 & & & \\ & 1 & & -\frac{5}{14} \\ & & 1 & \frac{4}{3} \\ & & & 1 \end{array} \right]$$

$$[\mathbf{A}|\mathbf{b}]^{(5)} = \left[\begin{array}{cccc|c} 1 & -\frac{4}{5} & \frac{1}{5} & 0 & -\frac{1}{5} \\ 0 & 1 & -\frac{8}{7} & 0 & -\frac{15}{7} \\ 0 & 0 & 1 & 0 & \frac{54}{5} \\ 0 & 0 & 0 & 1 & \frac{36}{5} \end{array} \right] \mathbf{M}^{(6)} = \left[\begin{array}{ccc} 1 & & -\frac{1}{5} \\ & 1 & \frac{8}{7} \\ & & 1 \\ & & & 1 \end{array} \right]$$

$$[\mathbf{A}|\mathbf{b}]^{(6)} = \left[\begin{array}{cccc|c} 1 & -\frac{4}{5} & 0 & 0 & -\frac{59}{25} \\ 0 & 1 & 0 & 0 & \frac{51}{5} \\ 0 & 0 & 1 & 0 & \frac{54}{5} \\ 0 & 0 & 0 & 1 & \frac{36}{5} \end{array} \right] \mathbf{M}^{(7)} = \left[\begin{array}{ccc} 1 & \frac{4}{5} & \\ & 1 & \\ & & 1 \\ & & & 1 \end{array} \right]$$

$$[\mathbf{A}|\mathbf{b}]^{(7)} = \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & \frac{29}{5} \\ 0 & 1 & 0 & 0 & \frac{51}{5} \\ 0 & 0 & 1 & 0 & \frac{54}{5} \\ 0 & 0 & 0 & 1 & \frac{36}{5} \end{array} \right]$$

Daqui, obtemos o resultado imediatamente:

$$\mathbf{x} = \frac{1}{5} \begin{bmatrix} 29 \\ 51 \\ 54 \\ 36 \end{bmatrix}$$

-Decomposição $\mathbf{A} = \mathbf{LU}$

O Resultado da decomposição LU da matriz \mathbf{A} é:

$$L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -\frac{4}{5} & 1 & 0 & 0 \\ \frac{1}{5} & -\frac{8}{7} & 1 & 0 \\ 0 & \frac{5}{14} & -\frac{4}{3} & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} 5 & -4 & 1 & 0 \\ 0 & \frac{14}{5} & -\frac{16}{5} & 1 \\ 0 & 0 & \frac{15}{7} & -\frac{20}{7} \\ 0 & 0 & 0 & \frac{5}{6} \end{bmatrix}$$

Resolvendo primeiro $\mathbf{L}\mathbf{y} = \mathbf{b}$ obtemos:

$$\mathbf{y} = \begin{bmatrix} -1 \\ \frac{6}{5} \\ \frac{18}{7} \\ 6 \end{bmatrix}$$

Por fim, resolvendo $\mathbf{U}\mathbf{x} = \mathbf{y}$:

$$\mathbf{x} = \frac{1}{5} \begin{bmatrix} 29 \\ 51 \\ 54 \\ 36 \end{bmatrix}$$

-Decomposição de *Cholesky* $\mathbf{A} = \mathbf{L}\mathbf{L}^T$

Pela fórmula temos:

$$\mathbf{L} = \begin{bmatrix} \sqrt{5} & 0 & 0 & 0 \\ \frac{-4}{\sqrt{5}} & \sqrt{\frac{14}{5}} & 0 & 0 \\ \frac{1}{\sqrt{5}} & -\frac{16}{\sqrt{70}} & \sqrt{\frac{15}{7}} & 0 \\ 0 & \sqrt{\frac{5}{14}} & -\frac{20}{\sqrt{105}} & \sqrt{\frac{5}{6}} \end{bmatrix}$$

Resolvendo $\mathbf{L}\mathbf{y} = \mathbf{b}$ obtemos:

$$\mathbf{y} = \begin{bmatrix} -\frac{1}{\sqrt{5}} \\ \frac{18}{35} \\ \frac{108}{35} \\ \frac{216}{5} \end{bmatrix}$$

Em seguida, para $\mathbf{L}^T\mathbf{x} = \mathbf{y}$ encontramos:

$$\mathbf{x} = \frac{1}{5} \begin{bmatrix} 29 \\ 51 \\ 54 \\ 36 \end{bmatrix}$$

-Método Iterativo *Jacobi*

Podemos compreender os métodos iterativos e o seu comportamento de convergência separando a matriz \mathbf{A} em duas matrizes \mathbf{S} e \mathbf{T} tais que $\mathbf{A} = \mathbf{S} - \mathbf{T}$. Assim, construímos o processo iterativo de modo que $\mathbf{S}\mathbf{x}^{(k+1)} = \mathbf{T}\mathbf{x}^{(k)} + \mathbf{b}$. No caso do método de *Jacobi*, \mathbf{S} é a matriz composta pela diagonal principal de \mathbf{A} .

Um critério de convergência que surge dessa perspectiva depende do raio espectral da matriz $\mathbf{S}^{-1}\mathbf{T}$. O raio espectral é dado pelo maior autovalor de uma matriz, ou seja,

$$\rho(\mathbf{A}) = \max_i |\lambda_i|$$

Em geral, um método iterativo é dito convergente se e somente se $\rho(\mathbf{S}^{-1}\mathbf{T}) < 1$.

Neste caso, temos

$$\mathbf{S}^{-1}\mathbf{T} = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ 0 & \frac{1}{6} & 0 & 0 \\ 0 & 0 & \frac{1}{6} & 0 \\ 0 & 0 & 0 & \frac{1}{5} \end{bmatrix} \begin{bmatrix} 0 & -4 & 1 & 0 \\ -4 & 0 & -4 & 1 \\ 1 & -4 & 0 & -4 \\ 0 & 1 & -4 & 0 \end{bmatrix} = \begin{bmatrix} 0 & \frac{-4}{5} & \frac{1}{5} & 0 \\ \frac{-4}{6} & 0 & \frac{-4}{6} & \frac{1}{6} \\ \frac{1}{6} & \frac{-4}{6} & 0 & \frac{-4}{6} \\ 0 & \frac{1}{5} & \frac{-4}{5} & 0 \end{bmatrix}$$

O autovalor de maior módulo é $\lambda_{\max} = \frac{2+\sqrt{34}}{6} \approx 1.30516$, portanto o método de *Jacobi* não irá convergir neste caso, já que $\rho(\mathbf{S}^{-1}\mathbf{T}) \geq 1$.

-Método Iterativo *Gauss-Seidel*

Já no método de *Gauss-Seidel*, o simples fato da matriz ser **positiva definida** e **simétrica** nos garante a convergência do método.

De maneira similar ao que foi feito no exercício anterior, vamos representar o algoritmo através de duas matrizes, \mathbf{S} e \mathbf{T} tais que $\mathbf{A} = \mathbf{S} - \mathbf{T}$. No entanto, para o algoritmo de *Gauss-Seidel*, a matriz \mathbf{S} é dada pela porção triangular inferior de \mathbf{A} , ou seja:

$$\mathbf{S} = \begin{bmatrix} 5 & 0 & 0 & 0 \\ -4 & 6 & 0 & 0 \\ 1 & -4 & 6 & 0 \\ 0 & 1 & -4 & 5 \end{bmatrix} \quad \mathbf{T} = \begin{bmatrix} 0 & 4 & -1 & 0 \\ 0 & 0 & 4 & -1 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Uma iteração do algoritmo pode, portanto, ser representada pela expressão $\mathbf{x}^{(k+1)} = \mathbf{S}^{-1}(\mathbf{T}\mathbf{x}^{(k)} + \mathbf{b})$:

$$\begin{bmatrix} \mathbf{x}_1^{(k+1)} \\ \mathbf{x}_2^{(k+1)} \\ \mathbf{x}_3^{(k+1)} \\ \mathbf{x}_4^{(k+1)} \end{bmatrix} = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ \frac{2}{15} & \frac{1}{6} & 0 & 0 \\ \frac{1}{18} & \frac{1}{9} & \frac{1}{6} & 0 \\ \frac{4}{225} & \frac{1}{18} & \frac{2}{15} & \frac{1}{5} \end{bmatrix} \left(\begin{bmatrix} 0 & 4 & -1 & 0 \\ 0 & 0 & 4 & -1 \\ 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1^{(k)} \\ \mathbf{x}_2^{(k)} \\ \mathbf{x}_3^{(k)} \\ \mathbf{x}_4^{(k)} \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \\ 1 \\ 3 \end{bmatrix} \right)$$

Tomando o limite de $\mathbf{x}^{(k)}$ quando $k \rightarrow \infty$, amparados pela garantia da convergência, temos que $\mathbf{x}^{(k)}, \mathbf{x}^{(k+1)} \rightarrow \mathbf{x}$. Assim, obtemos as relações:

$$\mathbf{x}_1 = \frac{1}{5}(4\mathbf{x}_2 - \mathbf{x}_3 - 1)$$

$$\begin{aligned}
\mathbf{x}_2 &= \frac{2}{15}(4\mathbf{x}_2 - \mathbf{x}_3 - 1) + \frac{1}{6}(4\mathbf{x}_3 - \mathbf{x}_4 + 2) \\
\mathbf{x}_3 &= \frac{1}{18}(4\mathbf{x}_2 - \mathbf{x}_3 - 1) + \frac{1}{9}(4\mathbf{x}_3 - \mathbf{x}_4 + 2) + \frac{1}{6}(4\mathbf{x}_4 + 1) \\
\mathbf{x}_4 &= \frac{3}{5} + \frac{4}{225}(-1 + 4\mathbf{x}_2 - \mathbf{x}_3) + \frac{1}{18}(2 + 4\mathbf{x}_3 - \mathbf{x}_4) + \frac{2}{15}(1 + 4\mathbf{x}_4)
\end{aligned}$$

Resolvendo temos

$$\begin{aligned}
\mathbf{x}_1 &= \frac{1}{185}(120\mathbf{x}_2 - 151) \\
\mathbf{x}_2 &= \frac{51}{5} \\
\mathbf{x}_3 &= \frac{2}{37}(14\mathbf{x}_2 + 57) \\
\mathbf{x}_4 &= \frac{4}{235}(8\mathbf{x}_2 + 23\mathbf{x}_3 + 93)
\end{aligned}$$

e, por fim, obtemos

$$\mathbf{x} = \begin{bmatrix} \frac{29}{5} \\ \frac{51}{5} \\ \frac{54}{5} \\ \frac{36}{5} \end{bmatrix}$$

De fato, se tomamos $\mathbf{x}_0 = \mathbf{x}$ e inicializamos o algoritmo, temos:

$$\begin{aligned}
\mathbf{x}_1^{(1)} &= \frac{1}{5}(-1 + 4\mathbf{x}_2^{(0)} - \mathbf{x}_3^{(0)}) = \frac{1}{5}(-1 + \frac{204}{5} - \frac{54}{5}) = \frac{29}{5} \\
\mathbf{x}_2^{(1)} &= \frac{1}{6}(2 + 4\mathbf{x}_1^{(1)} + 4\mathbf{x}_3^{(0)} - \mathbf{x}_4^{(0)}) = \frac{1}{6}(2 + \frac{116}{5} + \frac{216}{5} - \frac{36}{5}) = \frac{51}{5} \\
\mathbf{x}_3^{(1)} &= \frac{1}{6}(1 - \mathbf{x}_1^{(1)} + 4\mathbf{x}_2^{(1)} + 4\mathbf{x}_4^{(0)}) = \frac{1}{6}(1 - \frac{29}{5} + \frac{204}{5} + \frac{144}{5}) = \frac{54}{5} \\
\mathbf{x}_4^{(1)} &= \frac{1}{5}(3 - \mathbf{x}_2^{(1)} + 4\mathbf{x}_3^{(1)}) = \frac{1}{5}(3 - \frac{51}{5} + \frac{162}{5}) = \frac{36}{5}
\end{aligned}$$

e, portanto,

$$R = \frac{\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\|}{\|\mathbf{x}^{(1)}\|} = 0$$

e o algoritmo termina.

1 .: Inversa de \mathbf{A}

Multiplicando todas as matrizes de combinação de linhas $\mathbf{M}^{(i)}$ obtidas durante a eliminação de *Gauss-Jordan* obtemos

$$\mathbf{A}^{-1} = \prod_i \mathbf{M}^{(i)} = \frac{1}{5} \begin{bmatrix} 6 & 8 & 7 & 4 \\ 8 & 13 & 12 & 7 \\ 7 & 12 & 13 & 8 \\ 4 & 7 & 8 & 6 \end{bmatrix}$$

2 .: Determinante de \mathbf{A}

Uma vez que $\det(\mathbf{A} \cdot \mathbf{B}) = \det(\mathbf{A}) \cdot \det(\mathbf{B})$ para quaisquer matrizes \mathbf{A}, \mathbf{B} , podemos calcular o determinante de \mathbf{A} a partir de sua fatoração LU. Além disso, matrizes triangulares tem a propriedade

de que seu determinante é o produto dos elementos na diagonal principal. Assim, sendo $\mathbf{A} = \mathbf{L}\mathbf{U}$, $\det(\mathbf{L}) = 1$ e

$$\det(\mathbf{A}) = \prod_{i=1}^4 \mathbf{U}_{i,i} = 5 \cdot \frac{14}{5} \cdot \frac{15}{7} \cdot \frac{5}{6} = 25$$

Questão 5.: Questão 6.:

Algoritmo 1.: Saída do programa.

1	:: Decomposição PLU (com pivoteamento) ::						
2	P:						
3		1.00000	0.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
4		0.00000	1.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
5		0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
6		0.00000	0.00000	0.00000	1.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
7		0.00000	0.00000	0.00000	0.00000	1.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
8		0.00000	0.00000	0.00000	0.00000	0.00000	1.00000
		0.00000	0.00000	0.00000	0.00000		
9		0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
		1.00000	0.00000	0.00000	0.00000		
10		0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	1.00000	0.00000	0.00000		
11		0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	1.00000	0.00000		
12		0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	1.00000		
13	L:						
14		1.00000	0.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
15		0.56250	1.00000	0.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
16		0.50000	0.37696	1.00000	0.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
17		0.43750	0.34031	0.32255	1.00000	0.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
18		0.37500	0.30366	0.29532	0.29901	1.00000	0.00000
		0.00000	0.00000	0.00000	0.00000		
19		0.31250	0.26702	0.26809	0.27600	0.32992	1.00000
		0.00000	0.00000	0.00000	0.00000		
20		0.25000	0.23037	0.24085	0.25298	0.30556	0.35667
		1.00000	0.00000	0.00000	0.00000		
21		0.18750	0.19372	0.21362	0.22997	0.28119	0.33049
		0.38325	1.00000	0.00000	0.00000		
22		0.12500	0.15707	0.18638	0.20695	0.25683	0.30431
		0.35453	0.41274	1.00000	0.00000		
23		0.06250	0.12042	0.15915	0.18393	0.23247	0.27813
		0.32580	0.38049	0.44836	1.00000		
24	U:						
25		16.00000	9.00000	8.00000	7.00000	6.00000	5.00000
		4.00000	3.00000	2.00000	1.00000		
26		0.00000	11.93750	4.50000	4.06250	3.62500	3.18750
		2.75000	2.31250	1.87500	1.43750		
27		0.00000	0.00000	12.30366	3.96859	3.63351	3.29843

		2.96335	2.62827	2.29319	1.95812	
28		0.00000	0.00000	0.00000	13.27489	3.96936 3.66383
		3.35830	3.05277	2.74723	2.44170	
29		0.00000	0.00000	0.00000	0.00000	12.38928 4.08745
		3.78561	3.48378	3.18195	2.88011	
30		0.00000	0.00000	0.00000	0.00000	0.00000 11.34240
		4.04545	3.74851	3.45156	3.15462	
31		0.00000	0.00000	0.00000	0.00000	0.00000 0.00000
		10.20359	3.91051	3.61743	3.32435	
32		0.00000	0.00000	0.00000	0.00000	0.00000 0.00000
		0.00000	9.00891	3.71834	3.42776	
33		0.00000	0.00000	0.00000	0.00000	0.00000 0.00000
		0.00000	0.00000	7.77482	3.48594	
34		0.00000	0.00000	0.00000	0.00000	0.00000 0.00000
		0.00000	0.00000	0.00000	6.50648	
35	y:					
36		4.00000				
37		-2.25000				
38		6.84817				
39		-3.19319				
40		10.11566				
41		-4.94113				
42		5.34820				
43		-4.30386				
44		2.02376				
45		-2.47118				
46	x:					
47		0.16240				
48		-0.43991				
49		0.49837				
50		-0.43889				
51		0.90442				
52		-0.53865				
53		0.69105				
54		-0.51094				
55		0.43059				
56		-0.37980				
57	DET					
58		20385044096.000000				

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     !
466     !
467     !
468     !
469     !
470     !
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     !
520     !
521     !
522     !
523     !
524     !
525     !
526     !
527     !
528     !
529     !
530     !
531     !
532     !
533     !
534     !
535     !
536     !
537     !
538     !
539     !
540     !
541     !
542     !
543     !
544     !
545     !
546     !
547     !
548     !
549     !
550     !
551     !
552     !
553     !
554     !
555     !
556     !
557     !
558     !
559     !
560     !
561     !
562     !
563     !
564     !
565     !
566     !
567     !
568     !
569     !
570     !
571     !
572     !
573     !
574     !
575     !
576     !
577     !
578     !
579     !
580     !
581     !
582     !
583     !
584     !
585     !
586     !
587     !
588     !
589     !
590     !
591     !
592     !
593     !
594     !
595     !
596     !
597     !
598     !
599     !
600     !
601     !
602     !
603     !
604     !
605     !
606     !
607     !
608     !
609     !
610     !
611     !
612     !
613     !
614     !
615     !
616     !
617     !
618     !
619     !
620     !
621     !
622     !
623     !
624     !
625     !
626     !
627     !
628     !
629     !
630     !
631     !
632     !
633     !
634     !
635     !
636     !
637     !
638     !
639     !
640     !
641     !
642     !
643     !
644     !
645     !
646     !
647     !
648     !
649     !
650     !
651     !
652     !
653     !
654     !
655     !
656     !
657     !
658     !
659     !
660     !
661     !
662     !
663     !
664     !
665     !
666     !
667     !
668     !
669     !
670     !
671     !
672     !
673     !
674     !
675     !
676     !
677     !
678     !
679     !
680     !
681     !
682     !
683     !
684     !
685     !
686     !
687     !
688     !
689     !
690     !
691     !
692     !
693     !
694     !
695     !
696     !
697     !
698     !
699     !
700     !
701     !
702     !
703     !
704     !
705     !
706     !
707     !
708     !
709     !
710     !
711     !
712     !
713     !
714     !
715     !
716     !
717     !
718     !
719     !
720     !
721     !
722     !
723     !
724     !
725     !
726     !
727     !
728     !
729     !
730     !
731     !
732     !
733     !
734     !
735     !
736     !
737     !
738     !
739     !
740     !
741     !
742     !
743     !
744     !
745     !
746     !
747     !
748     !
749     !
750     !
751     !
752     !
753     !
754     !
755     !
756     !
757     !
758     !
759     !
760     !
761     !
762     !
763     !
764     !
765     !
766     !
767     !
768     !
769     !
770     !
771     !
772     !
773     !
774     !
775     !
776     !
777     !
778     !
779     !
780     !
781     !
782     !
783     !
784     !
785     !
786     !
787     !
788     !
789     !
790     !
791     !
792     !
793     !
794     !
795     !
796     !
797     !
798     !
799     !
800     !
801     !
802     !
803     !
804     !
805     !
806     !
807     !
808     !
809     !
810     !
811     !
812     !
813     !
814     !
815     !
816     !
817     !
818     !
819     !
820     !
821     !
822     !
823     !
824     !
825     !
826     !
827     !
828     !
829     !
830     !
831     !
832     !
833     !
834     !
835     !
836     !
837     !
838     !
839     !
840     !
841     !
842     !
843     !
844     !
845     !
846     !
847     !
848     !
849     !
850     !
851     !
852     !
853     !
854     !
855     !
856     !
857     !
858     !
859     !
860     !
861     !
862     !
863     !
864     !
865     !
866     !
867     !
868     !
869     !
870     !
871     !
872     !
873     !
874     !
875     !
876     !
877     !
878     !
879     !
880     !
881     !
882     !
883     !
884     !
885     !
886     !
887     !
888     !
889     !
890     !
891     !
892     !
893     !
894     !
895     !
896     !
897     !
898     !
899     !
900     !
901     !
902     !
903     !
904     !
905     !
906     !
907     !
908     !
909     !
910     !
911     !
912     !
913     !
914     !
915     !
916     !
917     !
918     !
919     !
920     !
921     !
922     !
923     !
924     !
925     !
926     !
927     !
928     !
929     !
930     !
931     !
932     !
933     !
934     !
935     !
936     !
937     !
938     !
939     !
940     !
941     !
942     !
943     !
944     !
945     !
946     !
947     !
948     !
949     !
950     !
951     !
952     !
953     !
954     !
955     !
956     !
957     !
958     !
959     !
960     !
961     !
962     !
963     !
964     !
965     !
966     !
967     !
968     !
969     !
970     !
971     !
972     !
973     !
974     !
975     !
976     !
977     !
978     !
979     !
980     !
981     !
982     !
983     !
984     !
985     !
986     !
987     !
988     !
989     !
990     !
991     !
992     !
993     !
994     !
995     !
996     !
997     !
998     !
999     !
1000    !

```

```

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     ! Begin with random normal vector and set 1st component to
       zero
777         x(:) = rand_vector(n)
778         x(1) = 1.0D0
779
780     ! Initialize Eigenvalues
781         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

```