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 $\mathbf{M}$.

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

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

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

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

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.

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

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

            logical :: ok

            integer :: i, j, k

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

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

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

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

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

            ok = .TRUE.
            return

        end function

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

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

            logical :: ok
```

```fortran

            integer :: i, j

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

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

            ok = .TRUE.
            return
        end function
```

b) Resolução de um sistema $\mathbf{Ax} = \mathbf{b}$

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

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

            integer :: n

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

            integer :: i

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

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

        end subroutine

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

            integer :: n

```

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

        logical :: ok

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

        if (.NOT. ok) then
            return
        end if

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

        return
    end function
```

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.

```fortran
    recursive function det(A, n) result (d)
        implicit none

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

        integer :: i
        double precision :: d, s

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

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

    function LU_det(A, n) result (d)
```

```fortran
        implicit none

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

        d = 0.0D0

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

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

        return
    end function
```

Questão 3.:

1 .: *Jacobi*

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

```fortran
        function Jacobi_cond(A, n) result (ok)
            implicit none

            integer :: n

            double precision :: A(n, n)

            logical :: ok

            if (.NOT. spectral_radius(A, n) < 1.0D0) then
                ok = .FALSE.
                call ill_cond()
                return
            else
                ok = .TRUE.
                return
            end if
        end function

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

            integer :: n

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

            logical :: ok

            integer :: i, k

            x0 = rand_vector(n)

            ok = Jacobi_cond(A, n)

            if (.NOT. ok) then
                return
            end if

            do k = 1, KMAX
                do i = 1, n
                    x(i) = (b(i) - dot_product(A(i, :), x0)) / A(i, i)
                end do
                x0(:) = x(:)
                e = vector_norm(matmul(A, x) - b, n)
```

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

2 .: *Gauss-Seidel*

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

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

        integer :: n

        double precision :: A(n, n)

        logical :: ok

        integer :: i

        do i = 1, n
            if (A(i, i) == 0.0D0) then
                ok = .FALSE.
                call ill_cond()
                return
            end if
        end do

        if (symmetrical(A, n) .AND. positive_definite(A, n)) then
            ok = .TRUE.
            return
        else
            call warn('Aviso: Esse método pode não convergir.')
            return
        end if
    end function

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

        integer :: n

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

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

```fortran

        ok = Gauss_Seidel_cond(A, n)

        if (.NOT. ok) then
            return
        end if

        do k = 1, KMAX
            do i = 1, n
                s = 0.0D0
                do j = 1, n
                    if (i /= j) then
                        s = s + A(i, j) * x(j)
                    end if
                end do
                x(i) = (b(i) - s) / A(i, i)
            end do
            e = vector_norm(matmul(A, x) - b, n)
            if (e < TOL) then
                return
            end if
        end do
        call error('Erro: Esse método não convergiu.')
        ok = .FALSE.
        return
    end function
```

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.

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

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

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

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

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

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:

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

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

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

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

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 $\mathbf{Ly} = \mathbf{b}$ obtemos:

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

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

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

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

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 $\mathbf{Ly} = \mathbf{b}$ obtemos:

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

Em seguida, para $\mathbf{L^T x} = \mathbf{y}$ 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 $\mathbf{Sx}^{(k+1)} = \mathbf{Tx}^{(k)} + \mathbf{b}$. 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 $\rho\left(\mathbf{S}^{-1}\mathbf{T}\right) < 1$.

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 $\rho\left(\mathbf{S}^{-1}\mathbf{T}\right) \geq 1$.

-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:

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

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

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

Tomando o limite de $\mathbf{x}^{(k)}$ quando $k \to \infty$, amparados pela garantia da convergência, temos que $\mathbf{x}^{(k)}, \mathbf{x}^{(k+1)} \to \mathbf{x}$. 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 A

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

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

2 .: Determinante de 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 $\mathbf{A} = \mathbf{LU}$, $\det(\mathbf{L}) = 1$ 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.

```
 :: Decomposição PLU (com pivoteamento) ::
P:
|   1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.00000    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.00000    0.00000    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.00000    0.00000    0.00000    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.00000    0.00000    0.00000    0.00000    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.00000    0.00000    0.00000    0.00000    0.00000    1.00000    0.00000    0.00000    0.00000    0.00000|
|   0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    1.00000    0.00000    0.00000    0.00000|
|   0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    1.00000    0.00000    0.00000|
|   0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    1.00000    0.00000|
|   0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    1.00000|
L:
|   1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.56250    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.50000    0.37696    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.43750    0.34031    0.32255    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.37500    0.30366    0.29532    0.29901    1.00000    0.00000    0.00000    0.00000    0.00000    0.00000|
|   0.31250    0.26702    0.26809    0.27600    0.32992    1.00000    0.00000    0.00000    0.00000    0.00000|
|   0.25000    0.23037    0.24085    0.25298    0.30556    0.35667    1.00000    0.00000    0.00000    0.00000|
|   0.18750    0.19372    0.21362    0.22997    0.28119    0.33049    0.38325    1.00000    0.00000    0.00000|
|   0.12500    0.15707    0.18638    0.20695    0.25683    0.30431    0.35453    0.41274    1.00000    0.00000|
|   0.06250    0.12042    0.15915    0.18393    0.23247    0.27813    0.32580    0.38049    0.44836    1.00000|
U:
|  16.00000    9.00000    8.00000    7.00000    6.00000    5.00000    4.00000    3.00000    2.00000    1.00000|
|   0.00000   11.93750    4.50000    4.06250    3.62500    3.18750    2.75000    2.31250    1.87500    1.43750|
|   0.00000    0.00000   12.30366    3.96859    3.63351    3.29843
```

```
    2.96335    2.62827    2.29319    1.95812|
|    0.00000    0.00000    0.00000   13.27489    3.96936    3.66383
    3.35830    3.05277    2.74723    2.44170|
|    0.00000    0.00000    0.00000    0.00000   12.38928    4.08745
    3.78561    3.48378    3.18195    2.88011|
|    0.00000    0.00000    0.00000    0.00000    0.00000   11.34240
    4.04545    3.74851    3.45156    3.15462|
|    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000
    10.20359    3.91051    3.61743    3.32435|
|    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000
    0.00000    9.00891    3.71834    3.42776|
|    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000
    0.00000    0.00000    7.77482    3.48594|
|    0.00000    0.00000    0.00000    0.00000    0.00000    0.00000
    0.00000    0.00000    0.00000    6.50648|
y:
|    4.00000|
|   -2.25000|
|    6.84817|
|   -3.19319|
|   10.11566|
|   -4.94113|
|    5.34820|
|   -4.30386|
|    2.02376|
|   -2.47118|
x:
|    0.16240|
|   -0.43991|
|    0.49837|
|   -0.43889|
|    0.90442|
|   -0.53865|
|    0.69105|
|   -0.51094|
|    0.43059|
|   -0.37980|
DET
20385044096.000000
```

Appendices

Código

```
!   Matrix Module

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

        integer :: MAX_ITER = 1000

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

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

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

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

        subroutine print_matrix(A, m, n)
            implicit none

            integer :: m, n
```

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

            integer :: i, j

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

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

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

            integer :: i

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

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

            close(33)
        end subroutine

        subroutine print_vector(x, n)
            implicit none

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

            integer :: i

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

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

```
            end do
        end subroutine

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

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

            read(33,*) b(:)

            close(33)
        end subroutine

!       =========== Matrix Methods ============
        function rand_vector(n) result (x)
            implicit none
            integer :: n
            double precision :: x (n)

            integer :: i

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

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

            integer :: i

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

        function id_matrix(n) result (A)
            implicit none

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

            integer :: j

```

```fortran
            A(:, :) = 0.0D0

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

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

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

            G(:, :) = id_matrix(n)

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

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

            return
        end function


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

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

            logical :: ok
            integer :: i

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

        recursive function positive_definite(A, n) result (ok)
!       Checks wether a matrix is positive definite
!       according to Sylvester's criterion.
```

```fortran
        implicit none

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

        logical :: ok

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

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

        double precision :: A(n, n)

        integer :: i, j
        logical :: ok

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

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

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

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

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

```fortran

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

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

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

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

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

        double precision :: P(n, n)

        integer :: j, k

        P = id_matrix(n)

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

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

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

        double precision :: s

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

    function matrix_norm(A, n) result (s)
!       Frobenius norm
        implicit none
```

```fortran
            integer :: n
            double precision :: A(n, n)
            double precision :: s

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

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

            integer :: n
            double precision :: A(n, n), x(n)
            double precision :: r, l

            logical :: ok

            ok = power_method(A, n, x, l)

            r = DABS(l)
            return
        end function

        recursive function det(A, n) result (d)
            implicit none

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

            integer :: i
            double precision :: d, s

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

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

```fortran

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

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

            d = 0.0D0

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

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

            return
        end function

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

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

            integer :: i

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

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

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

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

            logical :: ok

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

```fortran

        end function

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

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

            integer :: i, j
            double precision :: s

            logical :: ok

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

            ok = (s < 0.01D0)

            return
        end function

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

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

            logical :: ok

            ok = positive_definite(A, n)

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

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

            integer :: n
```

```fortran
            double precision :: A(n,n), P(n,n), L(n,n), U(n,n)

            logical :: ok

!           Permutation Matrix
            P = pivot_matrix(A, n)

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

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

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

            logical :: ok

            integer :: i, j, k

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

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

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

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

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

            ok = .TRUE.
            return

        end function

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

```fortran

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

        logical :: ok

        integer :: i, j

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

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

        ok = .TRUE.
        return
    end function

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

        integer :: n

        double precision :: A(n, n)

        logical :: ok

        if (.NOT. spectral_radius(A, n) < 1.0D0) then
            ok = .FALSE.
            call ill_cond()
            return
        else
            ok = .TRUE.
            return
        end if
    end function

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

        integer :: n

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

```fortran

            logical :: ok

            integer :: i, k

            x0 = rand_vector(n)

            ok = Jacobi_cond(A, n)

            if (.NOT. ok) then
                return
            end if

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

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

            integer :: n

            double precision :: A(n, n)

            logical :: ok

            integer :: i

            do i = 1, n
                if (A(i, i) == 0.0D0) then
                    ok = .FALSE.
                    call ill_cond()
                    return
                end if
            end do

            if (symmetrical(A, n) .AND. positive_definite(A, n)) then
                ok = .TRUE.
                return
            else
                call warn('Aviso: Esse método pode não convergir.')
                return
```

```fortran
            end if
        end function

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

            integer :: n

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

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

            ok = Gauss_Seidel_cond(A, n)

            if (.NOT. ok) then
                return
            end if

            do k = 1, KMAX
                do i = 1, n
                    s = 0.0D0
                    do j = 1, n
                        if (i /= j) then
                            s = s + A(i, j) * x(j)
                        end if
                    end do
                    x(i) = (b(i) - s) / A(i, i)
                end do
                e = vector_norm(matmul(A, x) - b, n)
                if (e < TOL) then
                    return
                end if
            end do
            call error('Erro: Esse método não convergiu.')
            ok = .FALSE.
            return
        end function

!       Decomposição LU e afins

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

            integer :: n

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

            integer :: i

```

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

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

        end subroutine

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

            integer :: n

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

            logical :: ok

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

            if (.NOT. ok) then
                return
            end if

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

            return
        end function

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

            integer :: n

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

            logical :: ok

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

            if (.NOT. ok) then
                return
            end if

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

            x(:) = matmul(P, x)

```

```
            return
        end function

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

            integer :: n

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

            logical :: ok

            ok = Cholesky_decomp(A, L, n)

            if (.NOT. ok) then
                return
            end if

            U = transpose(L)

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

            return
        end function

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

!       ============ Power Method ============
        function power_method(A, n, x, l) result (ok)
            implicit none
            integer :: n
            integer :: k = 0

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

            logical :: ok

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

!           Initialize Eigenvalues
            l = 0.0D0
```

```fortran

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

                x(:) = matmul(A, x)

!               Retrieve Eigenvalue
                l = x(1)

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

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

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

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

            logical :: ok

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

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

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

```

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

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

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

            logical :: ok

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

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

            b(1) = SUM(y)
            b(2) = dot_product(x, y)

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

    end module Matrix
```