COC473 - Lista 3

Pedro Maciel Xavier
116023847

25 de setembro de 2020

Questão 1.:

Começamos com o conjunto de pontos descrito abaixo:

i	x_i	y_i
1	1	1
2	2	2
3	3	9

Dados $n = 3$ pontos, precisaremos de um polinômio interpolador de grau $n-1 = 2$. Assim, supomos $p(x) = ax^2 + bx + c$. Queremos, portanto, satisfazer

$$\forall i \; p(x_i) = ax_i^2 + bx_i + c = y_i$$

que podemos reescrever na forma matricial

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

substituindo:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 9 \end{bmatrix}$$

Resolvendo o sistema linear, encontramos

$$\begin{aligned} a &= 3 \\ b &= -8 \\ c &= 6 \end{aligned}$$

como solução. Temos então o polinômio interpolador $p_{\mathrm{I}}(x) = 3x^2 - 8x + 6$.

Questão 2.:

Acrescentando o ponto $(4, 20)$, temos o seguinte conjunto de pontos:

i	x_i	y_i
1	1	1
2	2	2
3	3	9
4	4	20

Seguindo procedimento análogo ao anterior, só que desta vez com um polinômio de grau $n-1=4$, que chamaremos $p(x) = a_3x^3a_2x^2 + a_1x + a_0$, montamos o sistema

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ 1 & x_4 & x_4^2 & x_4^3 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

substituindo:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 \\ 1 & 3 & 9 & 27 \\ 1 & 4 & 16 & 64 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 9 \\ 20 \end{bmatrix}$$

A solução encontrada nos diz que

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 8 \\ -\frac{35}{3} \\ 5 \\ -\frac{1}{3} \end{bmatrix}$$

Por fim, temos o polinômio interpolador $p_{\text{II}}(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$.

Questão 3.:

Seja $g(x) = b_1 x^{b_2}$. Vamos definir $\Psi(x) = \log g(x) = \log b_1 + b_2 \log x$. Além disso, vamos escrever $\hat{x} = \log x$ e $\hat{b}_1 = \log b_1$. Assim, $\Psi(x) = \hat{b}_1 + b_2\hat{x}$. Por se tratar de um ajuste, não estamos interessados em passar pelos pontos (x_i, y_i) com exatidão, mas sim reduzir a soma das distâncias de cada um destes pontos para a curva ajustada.

Comecemos com a definição do erro quadrático médio:

$$\begin{aligned} E[\Psi(x)]^2 &= \sum_{i=1}^{n} (\Psi(x_i) - y_i)^2 \\ &= \sum_{i=1}^{n} (\hat{b}_1 + b_2\hat{x}_i - y_i)^2 \end{aligned}$$

Em seguida, para minimizar o erro, calculamos o par de derivadas em relação a $\hat{b}_1$ e b_2

$$\begin{aligned} \frac{\partial E[\Psi(x)]^2}{\partial \hat{b}_1} &= 2\sum_{i=1}^{n} (\hat{b}_1 + b_2\hat{x}_i - y_i) \\ \frac{\partial E[\Psi(x)]^2}{\partial b_2} &= 2\sum_{i=1}^{n} (\hat{b}_1 + b_2\hat{x}_i - y_i)\hat{x}_i \end{aligned}$$

e igualamos ambas a 0, a fim de obter os pontos de mínimo global da função quadrática.

$$\begin{aligned} \sum_{i=1}^{n} (\hat{b}_1 + b_2\hat{x}_i - y_i) = 0 \\ \sum_{i=1}^{n} (\hat{b}_1 + b_2\hat{x}_i - y_i)\hat{x}_i = 0 \end{aligned}$$

Escrevendo na forma matricial

$$\begin{bmatrix} n & \sum_{i=1}^{n} \hat{x}_i \\ \sum_{i=1}^{n} \hat{x}_i & \sum_{i=1}^{n} \hat{x}_i^2 \end{bmatrix} \begin{bmatrix} \hat{b}_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i\hat{x}_i \end{bmatrix}$$

e substituindo os valores

$$\begin{bmatrix} 4 & 3.178 \\ 3.178 & 3.609 \end{bmatrix} \begin{bmatrix} \hat{b}_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 5.886 \\ 7.047 \end{bmatrix}$$

e, por fim, resolvendo temos:

$$\begin{bmatrix} \hat{b}_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} -0.265 \\ 2.186 \end{bmatrix}$$

desfazendo a substituição, concluímos que $b_1 = e^{\hat{b}_1} = 0.766$. Obtivemos assim, um ajuste através da função $g_{\mathrm{III}}(x) = 0.766x^{2.186}$.

Questão 4.:

Para o cálculo do polinômio interpolador de Lagrange, analisamos primeiro o produto $\Phi_i(x)$ correspondente a cada ponto x_i.

$$\Phi_1(x) = \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)}$$
$$\Phi_2(x) = \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)}$$
$$\Phi_3(x) = \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)}$$
$$\Phi_4(x) = \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)}$$

Calculando o somatório das expressões, ponderados por cada y_i, chegamos a

$$p_{\mathrm{IV}}(x) = \sum_{i=1}^{n} y_i \Phi_i(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$$

Questão 5.:

Partindo de um polinômio na forma $f(x) = ax^2 + bx + c$, vamos construir um ajuste da maneira semelhante àquela feita anteriormente, no item 3). Partindo das derivadas parciais em relação aos coeficientes temos

$$\frac{\partial E[f(x)]^2}{\partial a} = 2\sum_{i=1}^{n}(ax_i^2 + bx_i + c - y_i)x_i^2 = 0$$
$$\frac{\partial E[f(x)]^2}{\partial b} = 2\sum_{i=1}^{n}(ax_i^2 + bx_i + c - y_i)x_i = 0$$
$$\frac{\partial E[f(x)]^2}{\partial c} = 2\sum_{i=1}^{n}(ax_i^2 + bx_i + c - y_i) = 0$$

de onde segue, na forma matricial, que

$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^4 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i x_i \\ \sum_{i=1}^{n} y_i x_i^2 \end{bmatrix}$$

substituindo, obtemos

$$\begin{bmatrix} 4 & 10 & 30 \\ 10 & 30 & 100 \\ 30 & 100 & 354 \end{bmatrix} \begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 32 \\ 112 \\ 410 \end{bmatrix}$$

cuja solução é

$$\begin{bmatrix} c \\ b \\ a \end{bmatrix} = \begin{bmatrix} 4.5 \\ -6.1 \\ 2.5 \end{bmatrix}$$

Assim, calculamos os coeficientes do polinômio $p_{\mathrm{V}}(x) = 2.5x^2 - 6.1x + 4.5$.

Questão 6.:

Organizados na tabela seguinte estão as funções calculadas nas questões anteriores assim como o valor das mesmas para $x = 3.5$.

$f(x)$	$f(3.5)$
$p_{\mathrm{I}}(x) = 3x^2 - 8x + 6$	14.750
$p_{\mathrm{II}}(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$	14.125
$g_{\mathrm{III}}(x) = 0.766x^{2.186}$	11.845
$p_{\mathrm{IV}}(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$	14.125
$p_{\mathrm{V}}(x) = 2.5x^2 - 6.1x + 4.5$	13.775

Questão 7.:

Temos agora novos dados, organizados na tabela abaixo:

i	x_i	y_i
1	1	1
2	2	2.5
3	3	3.5
4	4	4.3

Seja $g(x) = a \log x + \frac{b}{x^2+1}$. Comecemos com algumas substituições para auxiliar a notação:

$$\hat{x} = \log x$$
$$\check{x} = \frac{1}{x^2+1}$$

Reescrevemos $g(x) = a\hat{x}+b\check{x}$. Seguimos com o procedimento usual de ajuste por mínimos quadrados. Partimos das derivadas do erro quadrático médio em relação aos coeficientes a e b:

$$\frac{\partial E[g(x)]^2}{\partial a} = 2\sum_{i=1}^{n}(a\hat{x} + b\check{x} - y_i)\hat{x} = 0$$
$$\frac{\partial E[g(x)]^2}{\partial b} = 2\sum_{i=1}^{n}(a\hat{x} + b\check{x} - y_i)\check{x} = 0$$

Passando para a forma de matriz

$$\begin{bmatrix} \sum_{i=1}^n \hat{x}^2 & \sum_{i=1}^n \hat{x}\check{x} \\ \sum_{i=1}^n \hat{x}\check{x} & \sum_{i=1}^n \check{x}^2 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^n y_i\hat{x} \\ \sum_{i=1}^n y_i\check{x} \end{bmatrix}$$

e substituindo pelos valores da tabela:

$$\begin{bmatrix} 3.609 & 0.330 \\ 0.330 & 3.303 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 11.539 \\ 1.602 \end{bmatrix}$$

o que nos dá

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 3.013 \\ 2.004 \end{bmatrix}$$

e assim construímos $g(x) = 3.013 \log x + \frac{2.004}{x^2+1}$.

Questão 8.:

Dados dois vetores $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ contendo em cada i-ésima coordenada uma observação de um dado fenômeno, queremos ajustar a reta $y = f(x) = ax + b$ que melhor aproxima os valores quando $x = \mathbf{x}_i$ e $y = \mathbf{y}_i$. Como de costume, calculamos o erro quadrático médio proporcionado por $f(x)$:

$$E[f(x)]^2 = \sum_{i=1}^{n}(f(\mathbf{x}_i) - \mathbf{y}_i)^2 = \sum_{i=1}^{n}(a\mathbf{x}_i + b - \mathbf{y}_i)^2$$

Para minimizar o erro, encontramos primeiro as derivadas parciais com respeito aos parâmetros a e b e dizemos que esta se anula, pois assim obtemos seu ponto crítico de mínimo.

$$\frac{\partial E[f(x)]^2}{\partial a} = 2\sum_{i=1}^{n}(a\mathbf{x}_i + b - \mathbf{y}_i)x_i = 0$$
$$\frac{\partial E[f(x)]^2}{\partial b} = 2\sum_{i=1}^{n}(a\mathbf{x}_i + b - \mathbf{y}_i) = 0$$

Isso nos permite escrever

$$a\sum_{i=1}^{n}\mathbf{x}_i^2 + b\sum_{i=1}^{n}x_i = \sum_{i=1}^{n}\mathbf{x}_i\mathbf{y}_i$$
$$a\sum_{i=1}^{n}\mathbf{x}_i + bn = \sum_{i=1}^{n}\mathbf{y}_i$$

que na forma matricial nos dá

$$\begin{bmatrix} n & \sum_{i=1}^{n}\mathbf{x}_i \\ \sum_{i=1}^{n}\mathbf{x}_i & \mathbf{x}^{\mathrm{T}}\mathbf{x} \end{bmatrix} \begin{bmatrix} b \\ a \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n}\mathbf{y}_i \\ \mathbf{x}^{\mathrm{T}}\mathbf{y} \end{bmatrix}$$

Por fim, a resolução do sistema nos entrega o resultado. Como a matriz é simétrica, podemos utilizar a decomposição de Cholesky para resolver o sistema.

Algoritmo 1.: Mínimos quadrados

```
function least_squares(x, y, s, n) result (ok)
!   Dados x(n), y(n), n. O Resultado é armazenado em s(2) = [b, a]
    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
```

Appendices

Código

```fortran
!   Matrix Module

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

        integer :: MAX_ITER = 1000

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

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

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

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

```
            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
```

```

        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
```

```
            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
```

```

        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

```

```
!           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
```