

# **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  $p_I(x) = 3x^2 - 8x + 6$ .

## 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  $p(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ , 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  $p_{\Pi}(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$ .

### 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  $g_{\text{III}}(x) = 0.766x^{2.186}$ .

#### Questão 4.:

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

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

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

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

## 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

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

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

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

cujas 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  $p_V(x) = 2.5x^2 - 6.1x + 4.5$ .

## Questão 6.:

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

$f(x)$	$f(3.5)$
$p_I(x) = 3x^2 - 8x + 6$	14.750
$p_{II}(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$	14.125
$g_{III}(x) = 0.766x^{2.186}$	11.845
$p_{IV}(x) = -\frac{1}{3}x^3 + 5x^2 - \frac{35}{3}x + 8$	14.125
$p_V(x) = 2.5x^2 - 6.1x + 4.5$	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  $g(x) = a \log x + \frac{b}{x^2+1}$ . Começemos com algumas substituições para auxiliar a notação:

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

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

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

Passando para a forma de matriz

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

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.

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

Isso nos permite escrever

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

que na forma matricial nos dá

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

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

```
1 function least_squares(x, y, s, n) result (ok)
2 !   Dados x(n), y(n), n. O Resultado é armazenado em s(2) = [b, a]
3   implicit none
4   integer :: n
5
6   logical :: ok
7
8   double precision :: A(2,2), b(2), s(2), r(2), x(n), y(n)
9
10  A(1, 1) = n
11  A(1, 2) = SUM(x)
12  A(2, 1) = SUM(x)
13  A(2, 2) = DOT_PRODUCT(x, x)
14
15  b(1) = SUM(y)
16  b(2) = DOT_PRODUCT(x, y)
17
18  ok = Cholesky_solve(A, s, r, b, n)
19  return
20 end function
```

# Appendices

## Código

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

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

```

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

    integer :: n

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

730         return
731     end function
732
733     function Cholesky_solve(A, x, y, b, n) result (ok)
734         implicit none
735
736         integer :: n
737
738         double precision :: A(n, n), L(n, n), U(n, n)
739         double precision :: b(n), x(n), y(n)
740
741         logical :: ok
742
743         ok = Cholesky_decomp(A, L, n)
744
745         if (.NOT. ok) then
746             return
747         end if
748
749         U = transpose(L)
750
751         call LU_backsub(L, U, x, y, b, n)
752
753         return
754     end function
755
756     !
757     !
758     !
759     !
760     !
761     !
762     !
763
764     !
765     ! ===== Power Method =====
766     function power_method(A, n, x, l) result (ok)
767         implicit none
768         integer :: n
769         integer :: k = 0
770
771         double precision :: A(n, n)
772         double precision :: x(n)
773         double precision :: l, ll
774
775         logical :: ok
776
777         !
778         ! Begin with random normal vector and set 1st component to
779         ! zero
780         !
781         x(:) = rand_vector(n)
782         x(1) = 1.0D0
783
784         !
785         ! Initialize Eigenvalues
786         !
787         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

```