

COC473 - Lista 1

Pedro Maciel Xavier
116023847

22 de setemebro de 2019

Questão 1.:

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

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

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

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

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

Por substituição, chegamos ao resultado

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

Questão 2.:

a) Decomposição LU e de *Cholesky*

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

Algoritmo 1

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

```

45
46     logical :: ok
47
48     integer :: i, j
49
50     if (.NOT. Cholesky_cond(A, n)) then
51         call ill_cond()
52         ok = .FALSE.
53         return
54     end if
55
56     do i = 1, n
57         L(i, i) = sqrt(A(i, i) - sum(L(i,:i-1) * L(i,:i-1)))
58         do j = 1 + 1, n
59             L(j, i) = (A(i, j) - sum(L(i,:i-1) * L(j,:i-1))) / L(i, i)
60         end do
61     end do
62
63     ok = .TRUE.
64     return
65 end function

```

b) Resolução de um sistema $Ax = b$

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

Algoritmo 2

```

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

```

```

26     integer :: n
27
28     double precision :: A(n, n), L(n, n), U(n, n)
29     double precision :: b(n), x(n)
30
31     logical :: ok
32
33     ok = LU_decomp(A, L, U, n)
34
35     if (.NOT. ok) then
36         return
37     end if
38
39     call LU_backsub(L, U, x, b, n)
40
41     return
42 end function

```

c) Cálculo do determinante $\det(A)$

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.

Algoritmo 3

```

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

```

```

30 end function
31
32 function LU_det(A, n) result (d)
33     implicit none
34
35     integer :: n
36     integer :: i
37     double precision :: A(n, n), L(n, n), U(n, n)
38     double precision :: d
39
40     d = 0.0D0
41
42     if (.NOT. LU_decomp(A, L, U, n)) then
43         call ill_cond()
44         return
45     end if
46
47     do i = 1, n
48         d = d * L(i, i) * U(i, i)
49     end do
50
51     return
52 end function

```

Questão 3.:

1 .: *Jacobi*

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

Algoritmo 4

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

```

45     x0(:) = x(:)
46     e = vector_norm(matmul(A, x) - b, n)
47     if (e < TOL) then
48         return
49     end if
50 end do
51 call error('Erro: Esse método não convergiu.')
52 ok = .FALSE.
53 return
54 end function

```

2 :: Gauss-Seidel

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

Algoritmo 5

```

1  function Gauss_Seidel_cond(A, n) result (ok)
2      implicit none
3
4      integer :: n
5
6      double precision :: A(n, n)
7
8      logical :: ok
9
10     integer :: i
11
12     do i = 1, n
13         if (A(i, i) == 0.0D0) then
14             ok = .FALSE.
15             call error('Erro: Esse método não irá convergir.')
16             return
17         end if
18     end do
19
20     if (.NOT. (diagonally_dominant(A, n) .OR. (symmetrical(A, n) .AND.
21         positive_definite(A, n)))) then
22         call warn('Aviso: Esse método pode não convergir.')
23     end if
24
25     ok = .TRUE.
26     return
27 end function
28
29 function Gauss_Seidel(A, x, b, e, n) result (ok)
30     implicit none
31
32     integer :: n
33
34     double precision :: A(n, n)
35     double precision :: b(n), x(n)
36     double precision :: e, s

```



```

36
37     logical :: ok
38     integer :: i, j, k
39
40     ok = Gauss_Seidel_cond(A, n)
41
42     if (.NOT. ok) then
43         return
44     end if
45
46     do k = 1, KMAX
47         do i = 1, n
48             s = 0.0D0
49             do j = 1, n
50                 if (i /= j) then
51                     s = s + A(i, j) * x(j)
52                 end if
53             end do
54             x(i) = (b(i) - s) / A(i, i)
55         end do
56         e = vector_norm(matmul(A, x) - b, n)
57         if (e < TOL) then
58             return
59         end if
60     end do
61     call error('Erro: Esse método não convergiu.')
62     ok = .FALSE.
63     return
64 end function

```

Questão 4.:

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

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

-Eliminação Gaussiana

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

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

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

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

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

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

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

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

-Eliminação de *Gauss-Jordan*

Continuando de onde parou a eliminação Gaussiana seguimos com:

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

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

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

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

Daqui, obtemos o resultado imediatamente:

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

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

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

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

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

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

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

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

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

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

Pela fórmula temos:

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

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

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

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

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

-Método Iterativo *Jacobi*

-Método Iterativo *Gauss-Seidel*

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 **A**, **B**, podemos calcular o determinante de **A** a partir de sua fatoração LU. Além disso, matrizes triangulares tem a propriedade de que seu determinante é o produto dos elementos na diagonal principal. Assim, sendo $\mathbf{A} = \mathbf{L}\mathbf{U}$, $\det(\mathbf{L}) = 1$ e

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

Questão 5.: Questão 6.:

Algoritmo 6.: Saída do programa.

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

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

Appendices

Código

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



```

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

```

```

98     end subroutine
99
100    subroutine read_vector(fname, b, t)
101        implicit none
102        character(len=*) :: fname
103        integer :: t
104        double precision, allocatable :: b(:)
105
106        open(unit=33, file=fname, status='old', action='read')
107        read(33, *) t
108        allocate(b(t))
109
110        read(33,*) b(:)
111
112        close(33)
113    end subroutine
114
115    ! ===== Matrix Methods =====
116    recursive function det(A, n) result (d)
117        implicit none
118
119        integer :: n
120        double precision :: A(n, n)
121        double precision :: X(n-1, n-1)
122
123        integer :: i
124        double precision :: d, s
125
126        if (n == 1) then
127            d = A(1, 1)
128            return
129        elseif (n == 2) then
130            d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
131            return
132        else
133            d = 0.0D0
134            s = 1.0D0
135            do i = 1, n
136                ! Compute submatrix X
137                X(:, :i-1) = A(2:, :i-1)
138                X(:, i: ) = A(2:, i+1: )
139
140                d = s * det(X, n-1) * A(1, i) + d
141                s = -s
142            end do
143        end if
144        return
145    end function
146
147    function rand_vector(n) result (x)
148        implicit none
149        integer :: n
150        double precision :: x (n)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



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