

# COC473 - Lista 4

Pedro Maciel Xavier  
116023847

27 de outubro de 2020

**Nota:** Na primeira seção da Lista estão os trechos de código dos programas pedidos. Na segunda parte, estão os resultados dos programas assim como a análise destes. Por fim, no apêndice está o código completo. Caso os gráficos estejam pequenos, você pode ampliar sem problemas pois foram renderizados diretamente no formato **.pdf**.

# Programas

## Questão 1.: Bisseção

```
1      function bisection(f, aa, bb, tol) result (x)
2          implicit none
3          double precision, intent(in) :: aa, bb
4          double precision :: a, b, x, t_tol
5          double precision, optional :: tol
6
7          interface
8              function f(x) result (y)
9                  double precision :: x, y
10             end function
11         end interface
12
13         if (.NOT. PRESENT(tol)) then
14             t_tol = D_TOL
15         else
16             t_tol = tol
17         end if
18
19         if (bb < aa) then
20             a = bb
21             b = aa
22         else
23             a = aa
24             b = bb
25         end if
26
27         do while (DABS(a - b) > t_tol)
28             x = (a + b) / 2
29             if (f(a) > f(b)) then
30                 if (f(x) > 0) then
31                     a = x
32                 else
33                     b = x
34                 end if
35             else
36                 if (f(x) < 0) then
37                     a = x
38                 else
39                     b = x
40                 end if
41             end if
42         end do
43         x = (a + b) / 2
44         return
45     end function
```

## Questão 2.: Método de *Newton*

1 .: Original

```
1      function newton(f, df, x0, ok, tol, max_iter) result (x)
2          implicit none
3          integer :: k, t_max_iter
4          integer, optional :: max_iter
5          double precision, intent(in) :: x0
6          double precision :: x, xk, t_tol
7          double precision, optional :: tol
8          logical, intent(out) :: ok
9
10         interface
11             function f(x) result (y)
12                 double precision :: x, y
13             end function
14         end interface
15
16         interface
17             function df(x) result (y)
18                 double precision :: x, y
19             end function
20         end interface
21
22         if (.NOT. PRESENT(max_iter)) then
23             t_max_iter = D_MAX_ITER
24         else
25             t_max_iter = max_iter
26         end if
27
28         if (.NOT. PRESENT(tol)) then
29             t_tol = D_TOL
30         else
31             t_tol = tol
32         end if
33
34         ok = .TRUE.
35         xk = x0
36         do k = 1, t_max_iter
37             x = xk - f(xk) / df(xk)
38             if (DABS(x - xk) > t_tol) then
39                 xk = x
40             else
41                 if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
42                     then
43                     ok = .FALSE.
44                     end if
45                 end if
46                 return
47             end if
48         end do
49         ok = .FALSE.
```

```

48         return
49     end function

```

## 2 .: Secante

```

1      function secant(f, x0, ok, tol, max_iter) result (x)
2          implicit none
3          integer :: k, t_max_iter
4          integer, optional :: max_iter
5          double precision :: xk(3), yk(2)
6          double precision, intent(in) :: x0
7          double precision :: x, t_tol
8          double precision, optional :: tol
9          logical, intent(out) :: ok
10         interface
11             function f(x) result (y)
12                 implicit none
13                 double precision :: x, y
14             end function
15         end interface
16
17         if (.NOT. PRESENT(max_iter)) then
18             t_max_iter = D_MAX_ITER
19         else
20             t_max_iter = max_iter
21         end if
22
23         if (.NOT. PRESENT(tol)) then
24             t_tol = D_TOL
25         else
26             t_tol = tol
27         end if
28
29         ok = .TRUE.
30
31         xk(1) = x0
32         xk(2) = x0 + h
33         yk(1) = f(xk(1))
34         do k = 1, t_max_iter
35             yk(2) = f(xk(2))
36             xk(3) = xk(2) - (yk(2) * (xk(2) - xk(1))) / (yk(2) -
37                 yk(1))
38             if (DABS(xk(3) - xk(2)) > t_tol) then
39                 xk(1:2) = xk(2:3)
40                 yk(1) = yk(2)
41             else
42                 x = xk(3)
43                 if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
44                     then
45                     ok = .FALSE.
46                 end if
47                 return
48             end if
49         end do
50     end function

```

```
47         end do
48         ok = .FALSE.
49         return
50     end function
```

### Questão 3.: Método de Interpolação Inversa

```
1      function inv_interp(f, x00, ok, tol, max_iter) result (x)
2          implicit none
3          logical, intent(out) :: ok
4          integer :: i, j(1), k, t_max_iter
5          integer, optional :: max_iter
6          double precision :: x, xk, t_tol
7          double precision, optional :: tol
8          double precision, intent(in) :: x00(3)
9          double precision :: x0(3), y0(3)
10
11         interface
12             function f(x) result (y)
13                 double precision :: x, y
14             end function
15         end interface
16
17         if (.NOT. PRESENT(max_iter)) then
18             t_max_iter = D_MAX_ITER
19         else
20             t_max_iter = max_iter
21         end if
22
23         if (.NOT. PRESENT(tol)) then
24             t_tol = D_TOL
25         else
26             t_tol = tol
27         end if
28
29         x0(:) = x00(:)
30         xk = 1.0D+308
31
32         ok = .TRUE.
33
34         do k = 1, t_max_iter
35             call cross_sort(x0, y0, 3)
36
37             ! Cálculo de y
38             do i = 1, 3
39                 y0(i) = f(x0(i))
40             end do
41
42             x = lagrange(y0, x0, 3, 0.0D0)
43
44             if (DABS(x - xk) > t_tol) then
45                 j(:) = MAXLOC(DABS(y0))
46                 i = j(1)
47                 x0(i) = x
48                 y0(i) = f(x)
49                 xk = x
50             else
```

```
51         if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
52             then
53                 ok = .FALSE.
54             end if
55             return
56         end if
57     ok = .FALSE.
58     return
59 end function
```

## Questão 4.: Sistemas de equações

### 1 .: Método de *Newton* (Derivadas Parciais Analíticas)

```
1      function sys_newton(ff, dff, x0, n, ok, tol, max_iter)
2          result (x)
3          implicit none
4          logical, intent(out) :: ok
5          integer :: n, k, t_max_iter
6          integer, optional :: max_iter
7          double precision, dimension(n), intent(in) :: x0
8          double precision, dimension(n) :: x, xdx, dx
9          double precision, dimension(n, n) :: J
10         double precision :: t_tol
11         double precision, optional :: tol
12
13         interface
14             function ff(x, n) result (y)
15                 implicit none
16                 integer :: n
17                 double precision :: x(n), y(n)
18             end function
19         end interface
20
21         interface
22             function dff(x, n) result (J)
23                 implicit none
24                 integer :: n
25                 double precision :: x(n), J(n, n)
26             end function
27         end interface
28
29         if (.NOT. PRESENT(max_iter)) then
30             t_max_iter = D_MAX_ITER
31         else
32             t_max_iter = max_iter
33         end if
34
35         if (.NOT. PRESENT(tol)) then
36             t_tol = D_TOL
37         else
38             t_tol = tol
39         end if
40
41         ok = .TRUE.
42
43         x = x0
44
45         do k=1, t_max_iter
46             J = dff(x, n)
47             dx = -MATMUL(inv(J, n, ok), ff(x, n))
48             xdx = x + dx
```



```

48
49         if (.NOT. ok) then
50             exit
51         else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
52             x = xdx
53         else
54             if (VEDGE(x)) then
55                 ok = .FALSE.
56             end if
57             return
58         end if
59     end do
60     ok = .FALSE.
61     return
62 end function

```

## 2 .: Método de *Newton* (Derivadas Parciais Numéricas)

```

1      function sys_newton_num(ff, x0, n, ok, tol, max_iter) result
2      (x)
3      !      Same as previous function, with numerical partial
4      derivatives
5      implicit none
6      logical, intent(out) :: ok
7      integer :: n, i, k, t_max_iter
8      integer, optional :: max_iter
9      double precision, dimension(n), intent(in) :: x0
10     double precision, dimension(n):: x, xdx, xh, dx
11     double precision, dimension(n, n) :: J
12     double precision :: t_tol
13     double precision, optional :: tol
14
15     interface
16         function ff(x, n) result (y)
17             implicit none
18             integer :: n
19             double precision :: x(n), y(n)
20         end function
21     end interface
22
23     if (.NOT. PRESENT(max_iter)) then
24         t_max_iter = D_MAX_ITER
25     else
26         t_max_iter = max_iter
27     end if
28
29     if (.NOT. PRESENT(tol)) then
30         t_tol = D_TOL
31     else
32         t_tol = tol
33     end if
34
35     ok = .TRUE.

```

```

34
35     x = x0
36     xh = 0.0D0
37
38     do k=1, t_max_iter
39         ! Compute Jacobian Matrix
40         do i=1, n
41             ! Partial derivative with respect do the i-th
coordinates
42             xh(i) = h
43             J(:, i) = (ff(x + xh, n) - ff(x - xh, n)) / (2 *
h)
44             xh(i) = 0.0D0
45         end do
46
47         dx = -MATMUL(inv(J, n, ok), ff(x, n))
48         xdx = x + dx
49
50         if (.NOT. ok) then
51             exit
52         else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
53             x = xdx
54         else
55             if (VEDGE(x)) then
56                 ok = .FALSE.
57             end if
58             return
59         end if
60     end do
61     ok = .FALSE.
62     return
63 end function

```

### 3 .: Método de Broyden

```

1     function sys_broyden(ff, x0, B0, n, ok, tol, max_iter)
2         result (x)
3         implicit none
4         logical, intent(out) :: ok
5         integer :: n, k, t_max_iter
6         integer, optional :: max_iter
7         double precision, dimension(n), intent(in) :: x0
8         double precision, dimension(n, n), intent(in) :: B0
9         double precision, dimension(n) :: x, xdx, dx, dff
10        double precision, dimension(n, n) :: J
11        double precision :: t_tol
12        double precision, optional :: tol
13
14        interface
15            function ff(x, n) result (y)
16                implicit none
17                integer :: n
18                double precision :: x(n), y(n)

```

```

18         end function
19     end interface
20
21     if (.NOT. PRESENT(max_iter)) then
22         t_max_iter = D_MAX_ITER
23     else
24         t_max_iter = max_iter
25     end if
26
27     if (.NOT. PRESENT(tol)) then
28         t_tol = D_TOL
29     else
30         t_tol = tol
31     end if
32
33     ok = .TRUE.
34
35     x = x0
36     J = B0
37
38     do k=1, t_max_iter
39         dx = -MATMUL(inv(J, n, ok), ff(x, n))
40         if (.NOT. ok) then
41             exit
42         end if
43         xdx = x + dx
44         dff = ff(xdx, n) - ff(x, n)
45         if ((norm(dx, n) / norm(xdx, n)) > t_tol) then
46             J = J + OUTER_PRODUCT((dff - MATMUL(J, dx)) /
47                                     DOT_PRODUCT(dx, dx), dx, n)
48             x = xdx
49         else
50             if (VEDGE(x) .OR. (NORM(ff(x, n), n) > t_tol))
51                 then
52                 ok = .FALSE.
53             end if
54             return
55         end if
56     end do
57     ok = .FALSE.
58     return
59 end function

```

## Questão 5.: Ajuste de curvas não-lineares

```
1      function sys_least_squares(ff, dff, x, y, b0, m, n, ok, tol,
2          max_iter) result (b)
3          implicit none
4          logical, intent(out) :: ok
5          integer :: m, n, k, t_max_iter
6          integer, optional :: max_iter
7          double precision, dimension(n), intent(in) :: x, y, b0
8          double precision, dimension(n) :: b, bdb, db
9          double precision :: J(n, n)
10         double precision :: t_tol
11         double precision, optional :: tol
12
13         interface
14             function ff(x, b, m, n) result (z)
15                 implicit none
16                 integer :: m, n
17                 double precision, dimension(n), intent(in) :: x
18                 double precision, dimension(m), intent(in) :: b
19                 double precision, dimension(n) :: z
20             end function
21         end interface
22
23         interface
24             function dff(x, b, m, n) result (J)
25                 implicit none
26                 integer :: m, n
27                 double precision, dimension(n), intent(in) :: x
28                 double precision, dimension(m), intent(in) :: b
29                 double precision, dimension(n, m) :: J
30             end function
31         end interface
32
33         if (.NOT. PRESENT(max_iter)) then
34             t_max_iter = D_MAX_ITER
35         else
36             t_max_iter = max_iter
37         end if
38
39         if (.NOT. PRESENT(tol)) then
40             t_tol = D_TOL
41         else
42             t_tol = tol
43         end if
44
45         ok = .TRUE.
46
47         b = b0
48
49         do k=1, t_max_iter
50             J = dff(x, b, m, n)
```

```

50      db = -MATMUL(inv(MATMUL(TRANPOSE(J), J), n, ok),
51      MATMUL(TRANPOSE(J), ff(x, b, m, n) - y))
52      bdb = b + db
53
54      if (.NOT. ok) then
55          exit
56      else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
57          b = bdb
58      else
59          if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
60              t_tol)) then
61              ok = .FALSE.
62          end if
63          return
64      end if
65      ok = .FALSE.
66      return
67  end function

```

## Aplicações

**Questão 1.:** Utilizando os programas desenvolvidos encontre as raízes da seguinte equação por todos os métodos apresentados em sala de aula.

$$f(x) = \log \left( \cosh \left( x \sqrt{gk} \right) \right) - 50$$

onde  $g = 9.806$  e  $k = 0.00341$ .

```
1) f(x) = log(cosh(x * sqrt(g * j))) - 50
: Método da Bisseccção :
[a, b] = [-1000, 1000]
x = 277.22099795937538
y = 2.5661270086629884E-007
: Método de Newton (Zero de função) :
x0 = 347.82615369474877
x = 277.22099655606087
y = 0
: Método da Secante :
x0 = 371.29227463782553
x = 277.22099655606087
y = 0
Interpolação Inversa:
x1 = 446.11733521336538; x2 = 258.37020649446174; x3 = 414.27221497670621;
x = 277.22099655606087
y = 0
```

**Questão 2.:** Repita o exercício anterior para a função:

$$f(x) = 4 \cos(x) - e^{2x}$$

```
2) f(x) = 4 * cos(x) - exp(2 * x)
: Método da Bisseção :
[a, b] = [-1000, 1000]
x = 0.59788301587104797
y = 2.7044870004822030E-005
: Método de Newton (Zero de função) :
x0 = 7.6420922705605632
x = 0.59788606703853453
y = -7.0552452768879448E-012
: Método da Secante :
x0 = 163.90726965395132
x = 0.59788606740625483
y = -3.2664564386664097E-009
Interpolação Inversa:
x1 = 28.927984970856357; x2 = 76.336291041363637; x3 = 36.932311592273614;
x = 0.59788606706575032
y = -2.4829072131637986E-01
```

**Questão 3.:** Encontre uma solução para o seguinte sistema de equações não-lineares pelos métodos de *Newton* e *Broyden* utilizando os programas desenvolvidos.

$$16x^4 + 16y^4 + z^4 = 16$$

$$x^2 + y^2 + z^2 = 3$$

$$x^3 - y + z = 1$$

```

3) f(x, y, z) :=
16x4 + 16y4 + z4 = 16
x2 + y2 + z2 = 3
x3 - y + z = 1
: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Analíticas] :
x0 =
|          0.050410167632|
|          0.023212815098|
|          0.240226339953|

x =
|          -0.923983781872|
|          -0.371802309048|
|           1.417045172347|

y =
|           0.000000172017|
|           0.000000006648|
|          -0.000000003388|

: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Numéricas] :
x0 =
|          0.050410167632|
|          0.023212815098|
|          0.240226339953|

x =
|          -0.923984076771|
|          -0.371802906667|
|           1.417045041669|

y =
|           0.000015538859|
|           0.000000625653|
|          -0.000000291755|

: Método de Broyden :
x0 =
|          0.196251502228|
|          0.872439852720|
|          0.225504543157|

x =
|          -0.923983841025|

```



	−0.371802578023
	1.417045058754
y =	
	0.000002750302
	−0.000000005962
	0.000000000488

**Questão 4.:** Resolva, utilizando os programas desenvolvidos, o seguinte sistema de equações não-lineares (usando os Métodos de *Newton* e *Broyden*):

$$2c_3^2 + c_2^2 + 6c_4^2 = 1$$

$$8c_3^3 + 6c_3c_3^3 + 36c_3c_2c_4 + 108c_3c_4^2 = \theta_1$$

$$60c_3^4 + 60c_3^2c_2^2 + 576c_3^2c_2c_4 + 2232c_3^2c_4^2 + 252c_4^2c_2^2 + 1296c_4^3c_2 + 3348c_4^4 + 24c_2^3c_4 + 3c_2 = \theta_2$$

considerando os seguintes casos:

- a)  $\theta_1 = 0.00$  e  $\theta_2 = 3.0$ ;
- b)  $\theta_1 = 0.75$  e  $\theta_2 = 6.5$ ;
- c)  $\theta_1 = 0.00$  e  $\theta_2 = 11.667$ ;

```

4) f(c2, c3, c4) :=
c22 + 2 c32 + 6 c42 = 1
8 c33 + 6 c3 c22 + 36 c3 c2 c4 + 108 c3 c44 = θ1
60 * c34 + 60 * c32 * c22 + 576 * c32 * c2 * c4 + 2232 * c32 * c42 + 252 * c42 * c22 + 1296 * c43 c2 + 3348 c44 + 24 c23 c4 + 3 c2 = θ2
θ1 = 0
θ2 = 3
: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Analíticas] :
x0 =
|          0.912455180936|
|          0.447677569585|
|          0.053445584592|

x =
|          1.000000001460|
|         -0.000000000007|
|          0.000000005261|

y =
|          0.000000002920|
|         -0.000000000040|
|          0.000000130636|

: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Numéricas] :
x0 =
|          0.912455180936|
|          0.447677569585|
|          0.053445584592|

x =
|          1.000000001460|
|         -0.000000000007|
|          0.000000005261|

y =
|          0.000000002920|

```

```

|               -0.000000000040|
|               0.000000130639|
|
: Método de Broyden :
x0 =
|               0.880082976906|
|               -0.058624346719|
|               0.005610397254|
|
x =
|               0.890883790964|
|               0.000003664269|
|               -0.185439148967|
|
y =
|               -0.000000003153|
|               -0.000003875388|
|               0.000005190628|
|
θ1 = 0.75
θ2 = 6.5
: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Analíticas] :
x0 =
|               0.865531749434|
|               0.398829434805|
|               0.570209275020|
|
x =
|               -0.603879130914|
|               0.514676666662|
|               0.132630974981|
|
y =
|               0.000000000313|
|               0.000000001513|
|               0.000000013395|
|
: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Numéricas] :
x0 =
|               0.865531749434|
|               0.398829434805|
|               0.570209275020|
|
x =
|               -0.603879130914|
|               0.514676666662|
|               0.132630974981|
|
y =
|               0.000000000313|
|               0.000000001513|
|               0.000000013395|
|
: Método de Broyden :
x0 =
|               -0.652099171724|
|               -0.241675093684|
|               0.251509996236|

```

```

x =
|          0.746624790210|
|          0.426941431646|
|         -0.114012764821|

y =
|          0.000000012722|
|         -0.000000075283|
|          0.000007678415|

θ1 = 0
θ2 = 11.667
: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Analíticas] :
x0 =
|          0.056922190661|
|          0.554289744902|
|          0.330292900824|

x =
|         -0.568564967649|
|          0.478224922129|
|          0.191197156808|

y =
|          0.000002391357|
|          0.000013764321|
|          0.000072180010|

: Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Numéricas] :
x0 =
|          0.056922190661|
|          0.554289744902|
|          0.330292900824|

x =
|         -0.568564967649|
|          0.478224922129|
|          0.191197156808|

y =
|          0.000002391357|
|          0.000013764321|
|          0.000072179994|

: Método de Broyden :
x0 =
|          0.515962354338|
|         -0.410912574397|
|         -0.179154391741|

x =
|          0.630727845414|
|         -0.449694267405|
|         -0.181536277774|

y =
|          0.000000004143|

```

	−0.000000062961
	0.000005543062

**Questão 5.:** Utilizando o programa desenvolvido, ajuste uma função do tipo  $f(x) = b_0 + b_1x^{b_2}$  ao conjunto de dados abaixo:

$x$	1	2	3
$y$	1	2	9

```

5) f(x) = b1 + b2 x^b3
: Método não-linear de Mínimos Quadrados :
b0 =
|          0.987428050748|
|          0.872923325909|
|          0.900921471035|

b =
|          0.969165494015|
|          0.030834505985|
|          5.063123183571|

x =
|          1.000000000000|
|          2.000000000000|
|          3.000000000000|

y =
|          1.000000000000|
|          1.999999993230|
|          9.000000084507|

: Método não-linear de Mínimos Quadrados [Derivadas Numéricas]:
b0 =
|          0.970546768871|
|          0.995823245139|
|          0.841969959536|

b =
|          0.969165497331|
|          0.030834502669|
|          5.063123237344|

x =
|          1.000000000000|
|          2.000000000000|
|          3.000000000000|

y =
|          1.000000000000|
|          1.999999924089|
|          8.999999698430|

```

# Appendices

## Código - Programa Principal

```
1  program main4
2      use Func
3      use Calc
4      use Util
5      implicit none
6
7      !   Command-line Args
8      integer :: argc
9
10     !   Random seed definition
11     call init_random_seed()
12
13     !   Get Command-Line Args
14     argc = iargc()
15
16     if (argc == 0) then
17         goto 100
18     else
19         goto 11
20     end if
21
22     !   ===== Success =====
23 10    call info(':: Sucesso ::')
24     goto 1
25     !   ===== Errors =====
26 11    call error('Este programa não aceita parâmetros.')
27     goto 1
28     !   ===== Finish =====
29 1     stop
30     !   =====
31
32 100   call Q1
33     goto 200
34
35 200   call Q2
36     goto 300
37
38 300   call Q3
39     goto 400
40
41 400   call Q4
42     goto 500
43
44 500   call Q5
45     goto 10
46     !   =====
47     contains
```

```

48
49  subroutine Q1
50      implicit none
51      logical :: ok
52      integer :: k
53      double precision :: a, b, x, y, x0
54      double precision :: xx(3)
55
56      call blue("1) "//F1_NAME)
57      call info(": Método da Bisseccção :")
58      ! == Bounds definition ==
59      a = -1000.0D0
60      b = 1000.0D0
61      call blue("[a, b] = [("//DSTR(a)//", "//DSTR(b)//"]")
62      ! == Algorithm run =====
63      x = bisection(f1, a, b)
64      y = f1(x)
65      ! == Results =====
66      call show('x', x)
67      call show('y', y)
68      ! =====
69
70      call info(": Método de Newton (Zero de função) :")
71      ! == Bounds definition ==
72      ok = .FALSE.
73      do while(.NOT. ok)
74          x0 = DRAND(0.0D0, b)
75      ! == Algorithm run =====
76          x = newton(f1, df1, x0, ok)
77          y = f1(x)
78      end do
79      ! == Results =====
80      call show('x0', x0)
81      call show('x', x)
82      call show('y', y)
83      ! =====
84
85      call info(": Método da Secante :")
86      ! == Bounds definition ==
87      ok = .FALSE.
88      do while(.NOT. ok)
89          x0 = DRAND(0.0D0, b)
90      ! == Algorithm run =====
91          x = secant(f1, x0, ok)
92          y = f1(x)
93      end do
94      ! == Results =====
95      call show('x0', x0)
96      call show('x', x)
97      call show('y', y)
98      ! =====
99
100     call info("Interpolação Inversa:")

```



```

101      !      == Bounds definition ==
102      ok = .FALSE.
103      do while(.NOT. ok)
104          xx = (/ (DRAND(0.0D0, b), k=1,3)/)
105      !      == Algorithm run =====
106          x = inv_interp(f1, xx, ok)
107          y = f1(x)
108      end do
109      !      == Results =====
110      call blue('x1 = '//DSTR(xx(1))//'; x2 = '//DSTR(xx(2))//';
111              x3 = '//DSTR(xx(3))//';')
111      call show('x', x)
112      call show('y', y)
113      !      =====
114  end subroutine
115
116  subroutine Q2
117      implicit none
118      logical :: ok
119      integer :: k
120      double precision :: a, b, x, y, x0
121      double precision :: xx(3)
122
123      call info(ENL// "2) " //F2_NAME)
124      call info(": Método da Bisseccão :")
125      !      == Bounds definition ==
126      a = -1000.0D0
127      b = 1000.0D0
128      call blue("[a, b] = [ "//DSTR(a)//", "//DSTR(b)//"]")
129      !      == Algorithm run =====
130      x = bisection(f2, a, b)
131      y = f2(x)
132      !      == Results =====
133      call show('x', x)
134      call show('y', y)
135      !      =====
136
137      call info(": Método de Newton (Zero de função) :")
138      !      == Bounds definition ==
139      ok = .FALSE.
140      do while(.NOT. ok)
141          x0 = DRAND(0.0D0, b)
142      !      == Algorithm run =====
143          x = newton(f2, df2, x0, ok)
144          y = f2(x)
145      end do
146      !      == Results =====
147      call show('x0', x0)
148      call show('x', x)
149      call show('y', y)
150      !      =====
151
152      call info(": Método da Secante :")

```

```

153      !      == Bounds definition ==
154      ok = .FALSE.
155      do while(.NOT. ok)
156          x0 = DRAND(0.0D0, b)
157      !      == Algorithm run =====
158          x = secant(f2, x0, ok)
159          y = f2(x)
160      end do
161      !      == Results =====
162      call show('x0', x0)
163      call show('x', x)
164      call show('y', y)
165      !      =====
166
167      call info("Interpolação Inversa:")
168      !      == Bounds definition ==
169      !      allocate(x123(3))
170      ok = .FALSE.
171      do while(.NOT. ok)
172          xx = (/ (DRAND(0.0D0, b), k=1,3)/)
173      !      == Algorithm run =====
174          x = inv_interp(f2, xx, ok)
175          y = f2(x)
176      end do
177      !      == Results =====
178      call blue('x1 = '//DSTR(xx(1))//'; x2 = '//DSTR(xx(2))//';
179              x3 = '//DSTR(xx(3))//';')
180      call show('x', x)
181      call show('y', y)
182      !      =====
183      end subroutine
184
185      subroutine Q3
186      implicit none
187      logical :: ok
188      integer :: k
189      double precision, dimension(F3_N) :: x, y, x0
190      double precision, dimension(F3_N, F3_N) :: J0
191
192      call blue(ENDL// '3) '//F3_NAME)
193      x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
194      call info(": Método de Newton (Sistemas Não-Lineares) [
195              Derivadas Parciais Analíticas] :")
196      !      == Bounds definition ==
197      do k=1, D_MAX_ITER
198      !      == Algorithm run =====
199          x = sys_newton(f3, df3, x0, F3_N, ok)
200          if (.NOT. ok) then
201              x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
202          else
203              exit
204          end if
205      end do

```

```

204     if (.NOT. ok) then
205         call error('Este método não convergiu.')
206     else
207         y = f3(x, F3_N)
208         !      == Results =====
209         call show_vector('x0', x0, F3_N)
210         call show_vector('x', x, F3_N)
211         call show_vector('y', y, F3_N)
212         !      =====
213     end if
214
215     call info(": Método de Newton (Sistemas Não-Lineares) [
216         Derivadas Parciais Numéricas] :")
217     !      == Bounds definition ==
218     do k=1, D_MAX_ITER
219         !      == Algorithm run =====
220         x = sys_newton_num(f3, x0, F3_N, ok)
221         if (.NOT. ok) then
222             x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
223         else
224             exit
225         end if
226     end do
227     if (.NOT. ok) then
228         call error('Este método não convergiu.')
229     else
230         y = f3(x, F3_N)
231         !      == Results =====
232         call show_vector('x0', x0, F3_N)
233         call show_vector('x', x, F3_N)
234         call show_vector('y', y, F3_N)
235         !      =====
236     end if
237
238     call info(": Método de Broyden :")
239     !      == Bounds definition ==
240     J0 = id_matrix(F3_N)
241     do k=1, D_MAX_ITER
242         !      == Algorithm run =====
243         x = sys_broyden(f3, x0, J0, F3_N, ok)
244         if (.NOT. ok) then
245             x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
246         else
247             exit
248         end if
249     end do
250     if (.NOT. ok) then
251         call error('Este método não convergiu.')
252     else
253         y = f3(x, F3_N)
254         !      == Results =====
255         call show_vector('x0', x0, F3_N)
256         call show_vector('x', x, F3_N)

```

```

256         call show_vector('y', y, F3_N)
257     !     =====
258     end if
259 end subroutine
260
261 subroutine Q4
262     implicit none
263     logical :: ok
264     integer :: i, k
265     double precision, dimension(3) :: x, y, x0
266     double precision, dimension(3, 3) :: J0
267
268     call info('4) '//F4_NAME)
269
270     do i=1, F4_N
271         F4_T1 = F4_TT1(i)
272         F4_T2 = F4_TT2(i)
273
274         call show('01', F4_T1)
275         call show('02', F4_T2)
276
277         call info(": Método de Newton (Sistemas Não-Lineares) [
278             Derivadas Parciais Analíticas] :")
279         x0 = rand_vector(F4_N, 0.0D0, 1.0D0)
280         do k=1, 5 * D_MAX_ITER
281             ! == Algorithm run =====
282             x = sys_newton(f4, df4, x0, F4_N, ok)
283             if (.NOT. ok) then
284                 x0 = rand_vector(F4_N, 0.0D0, 1.0D0)
285             else
286                 exit
287             end if
288         end do
289         if (.NOT. ok) then
290             call error('Este método não convergiu.')
291         else
292             y = f4(x, F4_N)
293             ! == Results =====
294             call show_vector('x0', x0, F4_N)
295             call show_vector('x', x, F4_N)
296             call show_vector('y', y, F4_N)
297             !     =====
298         end if
299
300         call info(": Método de Newton (Sistemas Não-Lineares) [
301             Derivadas Parciais Numéricas] :")
302         do k=1, 5 * D_MAX_ITER
303             ! == Algorithm run =====
304             x = sys_newton_num(f4, x0, F4_N, ok)
305             if (.NOT. ok) then
306                 x0 = rand_vector(F4_N, 0.0D0, 1.0D0)
307             else
308                 exit

```

```

307         end if
308     end do
309     if (.NOT. ok) then
310         call error('Este método não convergiu.')
311     else
312         y = f4(x, F4_N)
313         ! == Results =====
314         call show_vector('x0', x0, F4_N)
315         call show_vector('x', x, F4_N)
316         call show_vector('y', y, F4_N)
317         ! =====
318     end if
319
320     call info(": Método de Broyden :")
321     J0 = id_matrix(F4_N)
322     do k=1, 5 * D_MAX_ITER
323         ! == Algorithm run =====
324         x = sys_broyden(f4, x0, J0, F4_N, ok)
325         if (.NOT. ok) then
326             x0 = rand_vector(F4_N, -1.0D0, 1.0D0)
327         else
328             exit
329         end if
330     end do
331     if (.NOT. ok) then
332         call error('Este método não convergiu.')
333     else
334         y = f4(x, F4_N)
335         ! == Results =====
336         call show_vector('x0', x0, F4_N)
337         call show_vector('x', x, F4_N)
338         call show_vector('y', y, F4_N)
339         ! =====
340     end if
341 end do
342 end subroutine
343
344 subroutine Q5
345     implicit none
346     logical :: ok
347     integer :: k
348     double precision, dimension(3) :: x, y, b, b0
349
350     call info("5) "//F5_NAME)
351
352     ! =====
353     x = (/1.0D0, 2.0D0, 3.0D0/)
354     y = (/1.0D0, 2.0D0, 9.0D0/)
355
356     b0 = rand_vector(F5_N, -1.0D0, 1.0D0)
357
358     call info(": Método não-linear de Mínimos Quadrados :")
359     do k=1, D_MAX_ITER

```

```

360      !      == Algorithm run =====
361          b = sys_least_squares(f5, df5, x, y, b0, F5_N, F5_N, ok)
362          if (.NOT. ok) then
363              b0 = rand_vector(F5_N, 0.8D0, 1.0D0)
364          else
365              exit
366          end if
367      end do
368      if (.NOT. ok) then
369          call error('Este método não convergiu.')
370      else
371          y = f5(x, b, F5_N, F5_N)
372          !      == Results =====
373          call show_vector('b0', b0, F5_N)
374          call show_vector('b', b, F5_N)
375          call show_vector('x', x, F5_N)
376          call show_vector('y', y, F5_N)
377          !      =====
378      end if
379
380      call info(": Método não-linear de Mínimos Quadrados [
381          Derivadas Numéricas]:")
382      do k=1, D_MAX_ITER
383          !      == Algorithm run =====
384          b = sys_least_squares(f5, df5, x, y, b0, F5_N, F5_N, ok)
385          if (.NOT. ok) then
386              b0 = rand_vector(F5_N, 0.8D0, 1.0D0)
387          else
388              exit
389          end if
390      end do
391      if (.NOT. ok) then
392          call error('Este método não convergiu.')
393      else
394          y = f5(x, b, F5_N, F5_N)
395          !      == Results =====
396          call show_vector('b0', b0, F5_N)
397          call show_vector('b', b, F5_N)
398          call show_vector('x', x, F5_N)
399          call show_vector('y', y, F5_N)
400          !      =====
401      end if
402  end subroutine
403 end program main4

```

## Código - Definição das Funções

```

1  !      Func Module
2
3      module Func
4          use Util

```

```

5      implicit none
6
7      !      >> F1 <<
8      character (len = *), parameter :: F1_NAME = "f(x) = log(cosh
          (x * sqrt(g * j))) - 50"
9      double precision :: F1_G = 9.80600D0
10     double precision :: F1_K = 0.00341D0
11
12     !      >> F2 <<
13     character (len = *), parameter :: F2_NAME = "f(x) = 4 * cos(
          x) - exp(2 * x)"
14
15     !      >> F3 <<
16     character (len = *), parameter :: F3_NAME = "f(x, y, z) :="
          //ENDL// &
17     "16x4 + 16y4 + z4 = 16"//ENDL// &
18     "x2 + y2 + x2 = 3"//ENDL// &
19     "x3 - y + z = 1"
20     integer :: F3_N = 3
21
22     !      >> F4 <<
23     character (len = *), parameter :: F4_NAME = "f(c2, c3, c4)
          :="//ENDL// &
24     "c22 + 2 c32 + 6 c42 = 1"//ENDL// &
25     "8 c33 + 6 c3 c22 + 36 c3 c2 c4 + 108 c3 c44 = theta1"//ENDL// &
26     "60 * c34 + 60 * c32 * c22 + 576 * c32 * c2 * c4 + "// &
27     "2232 * c32 * c42 + 252 * c42 * c22 + "// &
28     "1296 * c43 c2 + 3348 c44 + 24 c23 c4 + 3 c2 = theta2"
29     double precision :: F4_TT1(3) = (/ 0.0D0, 0.75D0, 0.000D0
          /)
30     double precision :: F4_TT2(3) = (/ 3.0D0, 6.50D0, 11.667D0
          /)
31     double precision :: F4_T1 = 0.0D0
32     double precision :: F4_T2 = 0.0D0
33     integer :: F4_N = 3
34
35     !      >> F5 <<
36     character (len = *), parameter :: F5_NAME = "f(x) = b1 + b2
          x~b3"
37     integer :: F5_N = 3
38
39     !      >> F6 <<
40     character (len = *), parameter :: F6_NAME = "f(x) = exp(-x2
          /2) / sqrt(2 pi)"
41
42     !      >> F7 <<
43     character (len = *), parameter :: F7_NAME = "Ssigma(w) = RAO(w)2
          Seta(w)"//ENDL//TAB// &
44     "RAO(w) = 1 / sqrt((1 - (w/wn)2)2 + (2xiw/wn)2)"
45
46     character (len = *), parameter :: F7a_NAME = "Seta(w) = 2"
47     character (len = *), parameter :: F7b_NAME = "Seta(w) = 2"
48

```

```

49 ! >> F8 <<
50 character (len = *), parameter :: F8_NAME = "Sσ(ω) = RAO(ω)2
      Sη(ω)"/ENL//TAB// &
51 "RAO(ω) = 1 / √((1 - (ω/ωn)2)2 + (2ξω/ωn)2)"
52
53 character (len = *), parameter :: F8a_NAME = "Sη(ω) = ((4 π3
      Hs2) / (ω5 Tz4)) exp(-(16 π3) / (ω4 Tz4))"
54 character (len = *), parameter :: F8b_NAME = "Sη(ω) = ((4 π3
      Hs2) / (ω5 Tz4)) exp(-(16 π3) / (ω4 Tz4))"
55
56
57 ! >> F13 <<
58 character (len = *), parameter :: F13_NAME = "y'(t) = -2 t y
      (t)2"/ENL// "y(0) = 1"
59 double precision :: F13_A = 0.0D0
60 double precision :: F13_B = 10.0D0
61 double precision :: F13_Y0 = 1.0D0
62
63 ! >> F14 <<
64 character (len = *), parameter :: F14_NAME = "m y''(t) + c y
      '(t) + k y(t) = F(t)"/ENL// &
65 "m = 1; c = 0.2; k = 1;"/ENL// &
66 "F(t) = 2 sin(ω t) + sin(2 ω t) + cos(3 ω t)"/ENL// &
67 "ω = 0.5;"/ENL// &
68 "y'(0) = 0; y(0) = 0;"
69 double precision :: F14_M = 1.0D0
70 double precision :: F14_C = 0.2D0
71 double precision :: F14_K = 1.0D0
72 double precision :: F14_W = 0.5D0
73 double precision :: F14_Y0 = 0.0D0
74 double precision :: F14_DY0 = 0.0D0
75 double precision :: F14_A = 0.0D0
76 double precision :: F14_B = 100.0D0
77
78 ! >> F15 <<
79 character (len = *), parameter :: F15_NAME = "z''(t) = -g -k
      z'(t) |z'(t)|"/ENL// &
80 "z'(0) = 0; z(0) = 0;"/ENL// &
81 "g = 9.806; k = 1;"
82 double precision :: F15_G = 9.80600D0
83 double precision :: F15_KD = 1.0D0
84 double precision :: F15_BY0 = 100.0D0
85 double precision :: F15_Y0 = 0.0D0
86 double precision :: F15_DY0 = 0.0D0
87 double precision :: F15_A = 0.0D0
88 double precision :: F15_B = 20.0D0
89
90
91
92 double precision :: t1 = 0.0D0
93 double precision :: t2 = 0.0D0
94
95 double precision :: wn = 1.00D0

```



```

96      double precision :: xi = 0.05D0
97      double precision :: Hs = 3.0D0
98      double precision :: Tz = 5.0D0
99
100     character (len = *), parameter :: F9_NAME = "f(x) = 2 + 2x -
           x2 + 3x3"
101
102     character (len = *), parameter :: F10_NAME = "f(x) = 1 / (1
           + x2)"
103
104     character (len = *), parameter :: F11_NAME = "f(x) = exp(- x
           2/2) / √(2 π)"
105     character (len = *), parameter :: F12_NAME = "f(x) = x2 exp
           (- x2/2) / √(2 π)"
106
107 !      >> L5-QE <<
108     character (len = *), parameter :: FL5_QE1_NAME = 'f(x) = x3
           + exp(-x)'
109     character (len = *), parameter :: DFL5_QE1_NAME = "f'(x) = 3
           x2 - exp(-x)"
110     character (len = *), parameter :: FL5_QE2_NAME = 'f(x) = 3√x
           + log(x)'
111     character (len = *), parameter :: DFL5_QE2_NAME = "f'(x) = 1
           / (3 3√x2) + (1 / x)"
112     character (len = *), parameter :: FL5_QE3_NAME = 'f(x) = 1 -
           exp(-x2 / 25)'
113     character (len = *), parameter :: DFL5_QE3_NAME = "f'(x) =
           (2 x / 25) exp(-x2 / 25)"
114
115
116     contains
117
118     function FL5_QE1(x) result (y)
119         implicit none
120         double precision :: x, y
121         y = x ** 3 + DEXP(-x)
122         return
123     end function
124
125     function DFL5_QE1(x) result (y)
126         implicit none
127         double precision :: x, y
128         y = 3 * x ** 2 - DEXP(-x)
129         return
130     end function
131
132     function FL5_QE2(x) result (y)
133         implicit none
134         double precision :: x, y
135         y = x ** (1.0D0/3.0D0) + DLOG(x)
136         return
137     end function
138

```

```

139  function DFL5_QE2(x) result (y)
140      implicit none
141      double precision :: x, y
142      y = 1 / (3 * x ** (2.0D0/3.0D0)) + (1 / x)
143      return
144  end function
145
146  function FL5_QE3(x) result (y)
147      implicit none
148      double precision :: x, y
149      y = 1 - DEXP(-(x ** 2) / 25)
150      return
151  end function
152
153  function DFL5_QE3(x) result (y)
154      implicit none
155      double precision :: x, y
156      y = (2 * X) / (25 * DEXP((x ** 2) / 25))
157      return
158  end function
159
160
161  function f1(x) result (y)
162      implicit none
163      double precision :: x, y
164      y = DLOG(DCOSH(x * DSQRT(F1_G * F1_K))) - 50.0D0
165      return
166  end function
167
168  function df1(x) result (y)
169      implicit none
170      double precision :: x, y
171      y = (DSINH(x * DSQRT(F1_G * F1_K)) * DSQRT(F1_G * F1_K)) /
172          DCOSH(x * DSQRT(F1_G * F1_K))
173      return
174  end function
175
176  function f2(x) result (y)
177      implicit none
178      double precision :: x, y
179      y = 4 * DCOS(x) - DEXP(2 * x)
180      return
181  end function
182
183  function df2(x) result (y)
184      implicit none
185      double precision :: x, y
186      y = - 4 * DSIN(x) - 2 * DEXP(2 * x)
187      return
188  end function
189
190  ! ===== R^n -> R^n functions =====
191  function f3(x, n) result (y)

```

```

191 |       $R^3 \rightarrow R^3$  (n == 3)
192 |      implicit none
193 |      integer :: n
194 |      double precision, dimension(n) :: x, y
195 |
196 |      y = (/ &
197 |          (16 * x(1) ** 4 + 16 * x(2) ** 4 + x(3) ** 4) - 16.0D0,
198 |          &
199 |          x(1) ** 2 + x(2) ** 2 + x(3) ** 2 - 3.0D0, &
200 |          x(1) ** 3 - x(2) + x(3) - 1.0D0 &
201 |          /)
202 |      return
203 | end function
204 | ! ===== Derivative =====
205 | function df3(x, n) result (J)
206 |     implicit none
207 |     integer :: n
208 |     double precision, dimension(n) :: x
209 |     double precision, dimension(n, n) :: J
210 |
211 |     J(1, :) = (/ 64 * x(1) ** 3, 64 * x(2) ** 3, 4 * x(3) ** 3
212 |         /)
213 |     J(2, :) = (/ 2 * x(1) , 2 * x(2) , 2 * x(3)
214 |         /)
215 |     J(3, :) = (/ 3 * x(1) ** 2, -1.0D0, 1.0D0
216 |         /)
217 |     return
218 | end function
219 | ! ===== Another function =====
220 | function f4(x, n) result (y)
221 |     implicit none
222 |     integer :: n
223 |     double precision, dimension(n) :: x, y
224 |     y = (/ &
225 |         x(1)**2+2*x(2)**2+6*x(3)**2, &
226 |         2*x(2)*(3*x(1)**2+4*x(2)**2+18*x(1)*x(3)+54*x(3)**4), &
227 |         3*(x(1)+20*x(1)**2*x(2)**2+20*x(2)**4+8*x(1)*(x(1)
228 |             **2+24*x(2)**2)*x(3)+&
229 |             12*(7*x(1)**2+62*x(2)**2)*x(3)**2+432*x(1)*x(3)**3+1116*
230 |             x(3)**4)&
231 |         /) - (/ 1.0D0, F4_T1, F4_T2 /)
232 |     return
233 | end function
234 | ! ===== Derivatives =====
235 | function df4(x, n) result (J)
236 | !       $R^3 \rightarrow R^3 \otimes \mathbb{R}$  (n == 3)
237 |     implicit none
238 |     integer :: n
239 |     double precision :: x(n), J(n, n)

```

```

238      J(1, :) = (/ &
239          2*x(1), &
240          4*x(2), &
241          12*x(3) &
242          /)
243      J(2, :) = (/ &
244          12*x(1)*x(2)+36*x(2)*x(3), &
245          6*x(1)**2+24*x(2)**2+36*x(1)*x(3)+108*x(3)**4, &
246          36*x(1)*x(2)+432*x(2)*x(3)**3 &
247          /)
248      J(3, :) = (/ &
249          3+120*x(1)*x(2)**2+72*x(1)**2*x(3)+576*x(2)**2*x(3)+504*
250              x(1)*x(3)**2+1296*x(3)**3, &
251          120*x(1)**2*x(2)+240*x(2)**3+1152*x(1)*x(2)*x(3)+4464*x
252              (2)*x(3)**2, &
253          24*x(1)**3+576*x(1)*x(2)**2+504*x(1)**2*x(3)+4464*x(2)
254              **2*x(3)+3888*x(1)*x(3)**2+13392*x(3)**3 &
255          /)
256      return
257 end function
258
259 ! ===== One more function =====
260 function f5(x, b, m, n) result (z)
261     implicit none
262     integer :: m, n
263     double precision, dimension(m), intent(in) :: b
264     double precision, dimension(n), intent(in) :: x
265     double precision, dimension(n) :: z
266
267     z = b(1) + (b(2) * (x ** b(3)))
268     return
269 end function
270
271 ! ===== Derivatives =====
272 function df5(x, b, m, n) result (J)
273     implicit none
274     integer :: m, n
275     double precision, dimension(m), intent(in) :: b
276     double precision, dimension(n), intent(in) :: x
277     double precision, dimension(n, m) :: J
278
279     m = 3
280     J(:, 1) = 1.0D0
281     J(:, 2) = x ** b(3)
282     J(:, 3) = b(2) * DLOG(x) * (x ** b(3))
283     return
284 end function
285
286 ! ===== Function 6 =====
287 function f6(x) result (y)
288     implicit none
289     double precision :: x, y
290
291     y = DEXP(-(x*x)/2) / DSQRT(2 * PI)

```

```

288         return
289     end function
290
291 ! ===== Functions 7 & 8 =====
292 function RAO(w) result (z)
293     implicit none
294     double precision :: w, z
295
296     z = 1.0 / DSQRT((1.0D0 - (w/wn) ** 2) ** 2 + (2 * xi * (w/wn
297         )) ** 2)
298 end function
299
300 function Sn1(w) result (z)
301     implicit none
302     double precision :: w, z
303     z = 2.0D0
304     return
305 end function
306
307 function Sn2(w) result (z)
308     implicit none
309     double precision :: w, z
310     z = (4 * (Hs**2) * (PI**3)) / ( DEXP( (16 * (PI**3))/((Tz*w)
311         **4) ) * (Tz**4) * (w**5) )
312     return
313 end function
314
315 function Ss(w, Sn) result (z)
316     implicit none
317     double precision :: w, z
318     interface
319         function Sn(w) result (z)
320             implicit none
321             double precision :: w, z
322         end function
323     end interface
324     z = (RAO(w) ** 2) * Sn(w)
325     return
326 end function
327
328 function f7a(w) result (z)
329     implicit none
330     double precision :: w, z
331     z = Ss(w, Sn1)
332     return
333 end function
334
335 function f7b(w) result (z)
336     implicit none
337     double precision :: w, z
338     z = (w ** 2) * Ss(w, Sn1)
339     return
340 end function

```

```

339
340     function f8a(w) result (z)
341         implicit none
342         double precision :: w, z
343         z = Ss(w, Sn2)
344         return
345     end function
346
347     function f8b(w) result (z)
348         implicit none
349         double precision :: w, z
350         z = (w ** 2) * Ss(w, Sn2)
351         return
352     end function
353
354 !     ===== Function 9 =====
355     function f9(x) result (y)
356         implicit none
357         double precision :: x, y
358         y = 2.0D0 + 2.0D0 * x - x ** 2 + 3.0D0 * x ** 3
359         return
360     end function
361
362 !     ===== Function 10 =====
363     function f10(x) result (y)
364         implicit none
365         double precision :: x, y
366         y = 1.0D0 / (1.0D0 + x ** 2)
367         return
368     end function
369
370 !     ===== Function 11 =====
371     function f11a(x) result (y)
372         implicit none
373         double precision :: x, y
374         y = DEXP((x ** 2) / 2) / DSQRT(8 * PI)
375         return
376     end function
377
378     function f11b(x) result (y)
379         implicit none
380         double precision :: x, y
381         y = DEXP(-(x ** 2) / 2) / DSQRT(8 * PI)
382         return
383     end function
384
385 !     ===== Function 12 =====
386     function f12(x) result (y)
387         implicit none
388         double precision :: x, y
389         y = (x ** 2) * DEXP((x ** 2) / 2) / DSQRT(2 * PI)
390         return
391     end function

```

```

392
393 ! ===== Function 13 =====
394 function df13(t, y) result (u)
395     implicit none
396     double precision :: t, y, u
397     u = - 2 * t * (y ** 2)
398     return
399 end function
400
401 function f13(t) result (y)
402     implicit none
403     double precision :: t, y
404     y = 1 / (1 + (t**2))
405     return
406 end function
407
408 ! ===== Function 14 =====
409 function F14_F(t) result (y)
410     implicit none
411     double precision :: t, y
412     y = 2 * DSIN(F14_W * t) + DSIN(2 * F14_W * t) + DCOS(3 *
413         F14_W * t)
414     return
415 end function
416
417 function d2f14(t, y, dy) result (u)
418     implicit none
419     double precision :: t, y, dy, u
420     u = (F14_F(t) - F14_K * y - F14_C * dy) / F14_M
421     return
422 end function
423
424 ! ===== Function 15 =====
425 function d2f15(t, y, dy) result (u)
426     implicit none
427     double precision :: t, y, dy, u
428     if (y >= 0) then
429         u = - F15_G
430     else
431         u = - F15_G - F15_KD * dy * DABS(dy)
432     end if
433     return
434 end function
435 end module Func

```

## Código - Métodos Numéricos

```

1 ! Calc Module
2
3 module Calc

```

```

4      use Util
5      use Matrix
6      implicit none
7      integer :: INT_N = 128
8      double precision :: h = 1.0D-5
9      !double precision :: D_TOL = 1.0D-5
10
11      character (len=*), parameter :: GAUSS_LEGENDRE_QUAD = "
12      quadratures/gauss-legendre/ gauss-legendre"
13      character (len=*), parameter :: GAUSS_HERMITE_QUAD = "
14      quadratures/gauss-hermite/ gauss-hermite"
15
16      contains
17      ! ===== Numerical Methods =====
18      function d(f, x, dx, kind) result (y)
19      implicit none
20      character (len=*), optional :: kind
21      double precision, optional :: dx
22      character (len=:), allocatable :: t_kind
23      double precision :: x, y, t_dx
24
25      interface
26      function f(x) result (y)
27      implicit none
28      double precision :: x, y
29      end function
30      end interface
31
32      if (.NOT. PRESENT(dx)) then
33      t_dx = h
34      else
35      t_dx = dx
36      end if
37
38      if (.NOT. PRESENT(kind)) then
39      t_kind = "central"
40      else
41      t_kind = kind
42      end if
43
44      if (t_kind == "central") then
45      y = (f(x + t_dx) - f(x - t_dx)) / (2 * t_dx)
46      else if (t_kind == "forward") then
47      y = (f(x + t_dx) - f(x)) / t_dx
48      else if (t_kind == "backward") then
49      y = (f(x) - f(x - t_dx)) / t_dx
50      else
51      call error("Unexpected value '//t_kind//' for
52      derivative kind."// &
53      "Options are: 'central', 'forward' and 'backward'.")
54      end if
55      return
56      end function
57
58

```



```

54 function dp(f, x, i, n) result (y)
55     implicit none
56     integer :: i, n
57     double precision :: f
58     double precision :: x(n), xh(n)
59     double precision :: y
60
61     xh(:) = 0.0D0
62     xh(i) = h
63
64     y = (f(x + xh) - f(x - xh)) / (2 * h)
65     return
66 end function
67
68 function grad(f, x, n) result (y)
69     implicit none
70     integer :: i, n
71     double precision :: f
72     double precision :: xh(n), x(n), y(n)
73
74     xh(:) = 0.0D0
75     do i=1, n
76         ! Compute partial derivative with respect to x_i
77         xh(i) = h
78         y(i) = (f(x + xh) - f(x - xh)) / (2 * h)
79         xh(i) = 0.0D0
80     end do
81     return
82 end function
83
84 ! =====
85
86 function lagrange(x0, y0, n, x) result (y)
87     implicit none
88     integer :: n
89     double precision :: x0(n), y0(n)
90     double precision :: x, y, yi
91     integer :: i, j
92
93     y = 0.0D0
94     do i = 1, n
95         yi = y0(i)
96         do j = 1, n
97             if (i /= j) then
98                 yi = yi * (x - x0(j)) / (x0(i) - x0(j))
99             end if
100         end do
101         y = y + yi
102     end do
103
104     return
105 end function
106

```

```

107  function bisection(f, aa, bb, tol) result (x)
108      implicit none
109      double precision, intent(in) :: aa, bb
110      double precision :: a, b, x, t_tol
111      double precision, optional :: tol
112
113      interface
114          function f(x) result (y)
115              double precision :: x, y
116          end function
117      end interface
118
119      if (.NOT. PRESENT(tol)) then
120          t_tol = D_TOL
121      else
122          t_tol = tol
123      end if
124
125      if (bb < aa) then
126          a = bb
127          b = aa
128      else
129          a = aa
130          b = bb
131      end if
132
133      do while (DABS(a - b) > t_tol)
134          x = (a + b) / 2
135          if (f(a) > f(b)) then
136              if (f(x) > 0) then
137                  a = x
138              else
139                  b = x
140              end if
141          else
142              if (f(x) < 0) then
143                  a = x
144              else
145                  b = x
146              end if
147          end if
148      end do
149      x = (a + b) / 2
150      return
151  end function
152
153  function newton(f, df, x0, ok, tol, max_iter) result (x)
154      implicit none
155      integer :: k, t_max_iter
156      integer, optional :: max_iter
157      double precision, intent(in) :: x0
158      double precision :: x, xk, t_tol
159      double precision, optional :: tol

```

```

160      logical, intent(out) :: ok
161
162      interface
163          function f(x) result (y)
164              double precision :: x, y
165          end function
166      end interface
167
168      interface
169          function df(x) result (y)
170              double precision :: x, y
171          end function
172      end interface
173
174      if (.NOT. PRESENT(max_iter)) then
175          t_max_iter = D_MAX_ITER
176      else
177          t_max_iter = max_iter
178      end if
179
180      if (.NOT. PRESENT(tol)) then
181          t_tol = D_TOL
182      else
183          t_tol = tol
184      end if
185
186      ok = .TRUE.
187      xk = x0
188      do k = 1, t_max_iter
189          x = xk - f(xk) / df(xk)
190          if (DABS(x - xk) > t_tol) then
191              xk = x
192          else
193              if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
194                  then
195                      ok = .FALSE.
196                  end if
197              return
198          end if
199      end do
200      ok = .FALSE.
201      return
202  end function
203
204  function secant(f, x0, ok, tol, max_iter) result (x)
205      implicit none
206      integer :: k, t_max_iter
207      integer, optional :: max_iter
208      double precision :: xk(3), yk(2)
209      double precision, intent(in) :: x0
210      double precision :: x, t_tol
211      double precision, optional :: tol
212      logical, intent(out) :: ok

```

```

212         interface
213             function f(x) result (y)
214                 implicit none
215                 double precision :: x, y
216             end function
217         end interface
218
219         if (.NOT. PRESENT(max_iter)) then
220             t_max_iter = D_MAX_ITER
221         else
222             t_max_iter = max_iter
223         end if
224
225         if (.NOT. PRESENT(tol)) then
226             t_tol = D_TOL
227         else
228             t_tol = tol
229         end if
230
231         ok = .TRUE.
232
233         xk(1) = x0
234         xk(2) = x0 + h
235         yk(1) = f(xk(1))
236         do k = 1, t_max_iter
237             yk(2) = f(xk(2))
238             xk(3) = xk(2) - (yk(2) * (xk(2) - xk(1))) / (yk(2) -
239                 yk(1))
240             if (DABS(xk(3) - xk(2)) > t_tol) then
241                 xk(1:2) = xk(2:3)
242                 yk(1) = yk(2)
243             else
244                 x = xk(3)
245                 if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
246                     then
247                     ok = .FALSE.
248                 end if
249                 return
250             end if
251         end do
252         ok = .FALSE.
253         return
254     end function
255
256     function inv_interp(f, x00, ok, tol, max_iter) result (x)
257         implicit none
258         logical, intent(out) :: ok
259         integer :: i, j(1), k, t_max_iter
260         integer, optional :: max_iter
261         double precision :: x, xk, t_tol
262         double precision, optional :: tol
263         double precision, intent(in) :: x00(3)
264         double precision :: x0(3), y0(3)

```

```

263
264     interface
265         function f(x) result (y)
266             double precision :: x, y
267         end function
268     end interface
269
270     if (.NOT. PRESENT(max_iter)) then
271         t_max_iter = D_MAX_ITER
272     else
273         t_max_iter = max_iter
274     end if
275
276     if (.NOT. PRESENT(tol)) then
277         t_tol = D_TOL
278     else
279         t_tol = tol
280     end if
281
282     x0(:) = x00(:)
283     xk = 1.0D+308
284
285     ok = .TRUE.
286
287     do k = 1, t_max_iter
288         call cross_sort(x0, y0, 3)
289
290         ! Cálculo de y
291         do i = 1, 3
292             y0(i) = f(x0(i))
293         end do
294
295         x = lagrange(y0, x0, 3, 0.0D0)
296
297         if (DABS(x - xk) > t_tol) then
298             j(:) = MAXLOC(DABS(y0))
299             i = j(1)
300             x0(i) = x
301             y0(i) = f(x)
302             xk = x
303         else
304             if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
305                 then
306                     ok = .FALSE.
307                 end if
308             return
309         end if
310     end do
311     ok = .FALSE.
312     return
313 end function

```

```

314  function sys_newton(ff, dff, x0, n, ok, tol, max_iter)
      result (x)
315      implicit none
316      logical, intent(out) :: ok
317      integer :: n, k, t_max_iter
318      integer, optional :: max_iter
319      double precision, dimension(n), intent(in) :: x0
320      double precision, dimension(n) :: x, xdx, dx
321      double precision, dimension(n, n) :: J
322      double precision :: t_tol
323      double precision, optional :: tol
324
325      interface
326          function ff(x, n) result (y)
327              implicit none
328              integer :: n
329              double precision :: x(n), y(n)
330          end function
331      end interface
332
333      interface
334          function dff(x, n) result (J)
335              implicit none
336              integer :: n
337              double precision :: x(n), J(n, n)
338          end function
339      end interface
340
341      if (.NOT. PRESENT(max_iter)) then
342          t_max_iter = D_MAX_ITER
343      else
344          t_max_iter = max_iter
345      end if
346
347      if (.NOT. PRESENT(tol)) then
348          t_tol = D_TOL
349      else
350          t_tol = tol
351      end if
352
353      ok = .TRUE.
354
355      x = x0
356
357      do k=1, t_max_iter
358          J = dff(x, n)
359          dx = -MATMUL(inv(J, n, ok), ff(x, n))
360          xdx = x + dx
361
362          if (.NOT. ok) then
363              exit
364          else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
365              x = xdx

```

```

366         else
367             if (VEDGE(x)) then
368                 ok = .FALSE.
369             end if
370             return
371         end if
372     end do
373     ok = .FALSE.
374     return
375 end function
376
377 function sys_newton_num(ff, x0, n, ok, tol, max_iter) result
378 (x)
379 ! Same as previous function, with numerical partial
    derivatives
379     implicit none
380     logical, intent(out) :: ok
381     integer :: n, i, k, t_max_iter
382     integer, optional :: max_iter
383     double precision, dimension(n), intent(in) :: x0
384     double precision, dimension(n):: x, xdx, xh, dx
385     double precision, dimension(n, n) :: J
386     double precision :: t_tol
387     double precision, optional :: tol
388
389     interface
390         function ff(x, n) result (y)
391             implicit none
392             integer :: n
393             double precision :: x(n), y(n)
394         end function
395     end interface
396
397     if (.NOT. PRESENT(max_iter)) then
398         t_max_iter = D_MAX_ITER
399     else
400         t_max_iter = max_iter
401     end if
402
403     if (.NOT. PRESENT(tol)) then
404         t_tol = D_TOL
405     else
406         t_tol = tol
407     end if
408
409     ok = .TRUE.
410
411     x = x0
412     xh = 0.0D0
413
414     do k=1, t_max_iter
415 ! Compute Jacobian Matrix
416         do i=1, n

```

```

417 | ! Partial derivative with respect do the i-th
      coordinates
418 |         xh(i) = h
419 |         J(:, i) = (ff(x + xh, n) - ff(x - xh, n)) / (2 *
      h)
420 |         xh(i) = 0.0D0
421 |     end do
422 |
423 |     dx = -MATMUL(inv(J, n, ok), ff(x, n))
424 |     xdx = x + dx
425 |
426 |     if (.NOT. ok) then
427 |         exit
428 |     else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
429 |         x = xdx
430 |     else
431 |         if (VEDGE(x)) then
432 |             ok = .FALSE.
433 |         end if
434 |         return
435 |     end if
436 | end do
437 | ok = .FALSE.
438 | return
439 | end function
440 |
441 | function sys_broyden(ff, x0, B0, n, ok, tol, max_iter)
      result (x)
442 |     implicit none
443 |     logical, intent(out) :: ok
444 |     integer :: n, k, t_max_iter
445 |     integer, optional :: max_iter
446 |     double precision, dimension(n), intent(in) :: x0
447 |     double precision, dimension(n, n), intent(in) :: B0
448 |     double precision, dimension(n) :: x, xdx, dx, dff
449 |     double precision, dimension(n, n) :: J
450 |     double precision :: t_tol
451 |     double precision, optional :: tol
452 |
453 |     interface
454 |         function ff(x, n) result (y)
455 |             implicit none
456 |             integer :: n
457 |             double precision :: x(n), y(n)
458 |         end function
459 |     end interface
460 |
461 |     if (.NOT. PRESENT(max_iter)) then
462 |         t_max_iter = D_MAX_ITER
463 |     else
464 |         t_max_iter = max_iter
465 |     end if
466 |

```



```

467         if (.NOT. PRESENT(tol)) then
468             t_tol = D_TOL
469         else
470             t_tol = tol
471         end if
472
473         ok = .TRUE.
474
475         x = x0
476         J = B0
477
478         do k=1, t_max_iter
479             dx = -MATMUL(inv(J, n, ok), ff(x, n))
480             if (.NOT. ok) then
481                 exit
482             end if
483             xdx = x + dx
484             dff = ff(xdx, n) - ff(x, n)
485             if ((norm(dx, n) / norm(xdx, n)) > t_tol) then
486                 J = J + OUTER_PRODUCT((dff - MATMUL(J, dx)) /
487                                         DOT_PRODUCT(dx, dx), dx, n)
488                 x = xdx
489             else
490                 if (VEDGE(x) .OR. (NORM(ff(x, n), n) > t_tol))
491                     then
492                     ok = .FALSE.
493                 end if
494                 return
495             end if
496         end do
497         ok = .FALSE.
498         return
499     end function
500
501     function sys_least_squares(ff, dff, x, y, b0, m, n, ok, tol,
502                                max_iter) result (b)
503         implicit none
504         logical, intent(out) :: ok
505         integer :: m, n, k, t_max_iter
506         integer, optional :: max_iter
507         double precision, dimension(n), intent(in) :: x, y, b0
508         double precision, dimension(n) :: b, bdb, db
509         double precision :: J(n, n)
510         double precision :: t_tol
511         double precision, optional :: tol
512
513         interface
514             function ff(x, b, m, n) result (z)
515                 implicit none
516                 integer :: m, n
517                 double precision, dimension(n), intent(in) :: x
518                 double precision, dimension(m), intent(in) :: b
519                 double precision, dimension(n) :: z

```

```

517         end function
518     end interface
519
520     interface
521         function dff(x, b, m, n) result (J)
522             implicit none
523             integer :: m, n
524             double precision, dimension(n), intent(in) :: x
525             double precision, dimension(m), intent(in) :: b
526             double precision, dimension(n, m) :: J
527         end function
528     end interface
529
530     if (.NOT. PRESENT(max_iter)) then
531         t_max_iter = D_MAX_ITER
532     else
533         t_max_iter = max_iter
534     end if
535
536     if (.NOT. PRESENT(tol)) then
537         t_tol = D_TOL
538     else
539         t_tol = tol
540     end if
541
542     ok = .TRUE.
543
544     b = b0
545
546     do k=1, t_max_iter
547         J = dff(x, b, m, n)
548         db = -MATMUL(inv(MATMUL(TRANPOSE(J), J), n, ok),
549                     MATMUL(TRANPOSE(J), ff(x, b, m, n) - y))
550         bdb = b + db
551
552         if (.NOT. ok) then
553             exit
554         else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
555             b = bdb
556         else
557             if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
558                             t_tol)) then
559                 ok = .FALSE.
560             end if
561             return
562         end if
563     end do
564     ok = .FALSE.
565     return
566 end function
567
568 function sys_least_squares_num(ff, x, y, b0, m, n, ok, tol,
569                               max_iter) result (b)

```

```

567 | ! Same as previous function, with numerical partial
    | derivatives
568 |     implicit none
569 |     integer :: m, n, i, k, t_max_iter
570 |     integer, optional :: max_iter
571 |     double precision, dimension(n), intent(in) :: x, y, b0
572 |     double precision, dimension(n) :: b, bdb, db, bh
573 |     double precision :: J(n, n)
574 |     double precision :: t_tol
575 |     double precision, optional :: tol
576 |
577 |     logical, intent(out) :: ok
578 |     interface
579 |         function ff(x, b, m, n) result (z)
580 |             implicit none
581 |             integer :: m, n
582 |             double precision, dimension(n), intent(in) :: x
583 |             double precision, dimension(m), intent(in) :: b
584 |             double precision, dimension(n) :: z
585 |         end function
586 |     end interface
587 |
588 |     if (.NOT. PRESENT(max_iter)) then
589 |         t_max_iter = D_MAX_ITER
590 |     else
591 |         t_max_iter = max_iter
592 |     end if
593 |
594 |     if (.NOT. PRESENT(tol)) then
595 |         t_tol = D_TOL
596 |     else
597 |         t_tol = tol
598 |     end if
599 |
600 |     ok = .TRUE.
601 |
602 |     bh = 0.0D0
603 |     b = b0
604 |
605 |     do k=1, t_max_iter
606 |         ! Compute Jacobian Matrix
607 |         do i=1, m
608 |             ! Partial derivative with respect to the i-th
    | coordinates
609 |                 bh(i) = h
610 |                 J(:, i) = (ff(x, b + bh, m, n) - ff(x, b - bh, m
    |                 , n)) / (2 * h)
611 |                 bh(i) = 0.0D0
612 |             end do
613 |             db = -MATMUL(inv(MATMUL(TRANPOSE(J), J), n, ok),
    |             MATMUL(TRANPOSE(J), ff(x, b, m, n) - y))
614 |             bdb = b + db
615 |

```

```

616         if (.NOT. ok) then
617             exit
618         else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
619             b = bdb
620         else
621             if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
622                 t_tol)) then
623                 ok = .FALSE.
624             end if
625             return
626         end if
627     end do
628     ok = .FALSE.
629     return
630 end function
631
632 ! ===== Numerical Integration =====
633 !
634 subroutine load_quad(x, w, k, fname)
635     ! Load Quadrature
636     implicit none
637     integer :: k, m, n
638     character (len=*) :: fname
639     double precision, dimension(k) :: x, w
640     double precision, dimension(:, :), allocatable :: xw
641     call read_matrix(fname, xw, m, n)
642     if (n /= 2 .OR. m /= k) then
643         call error("Invalid Matrix dimensions.")
644         stop "ERROR"
645     end if
646     x(:) = xw(:, 1)
647     w(:) = xw(:, 2)
648     deallocate(xw)
649 end subroutine
650
651 function num_int(f, a, b, n, kind) result (s)
652     implicit none
653     integer :: n
654     character (len=*), optional :: kind
655     double precision :: a, b, s
656     interface
657         function f(x) result (y)
658             double precision :: x, y
659         end function
660     end interface
661
662     if (.NOT. PRESENT(kind)) then
663         kind = "polynomial"
664     end if
665
666     if (kind == "polynomial") then
667         s = polynomial_int(f, a, b, n)
668     else if (kind == "gauss-legendre") then
669         s = gauss_legendre_int(f, a, b, n)
670     end if
671 end function

```

```

668     else if (kind == "gauss-hermite") then
669         s = gauss_hermite_int(f, a, b, n)
670     else if (kind == "romberg") then
671         s = romberg_int(f, a, b, n)
672     else
673         call error("Unknown integration kind '//kind//'. '//
        &
674         "Available options are: 'polynomial', 'gauss-
        legendre', 'gauss-hermite' and 'romberg'."))
675     end if
676
677 end function
678
679 function polynomial_int(f, a, b, n) result (s)
680     implicit none
681     integer :: n, i
682     double precision :: a, b, s
683     double precision, dimension(n) :: x, y, w
684     double precision, dimension(n, n) :: V
685     interface
686         function f(x) result (y)
687             double precision :: x, y
688         end function
689     end interface
690
691     x(:) = ((b-a)/(n-1)) * (/ (i, i=0,n-1) /) + a
692     y(:) = (/ ((b**i - a**i)/i, i=1, n) /)
693     V(:, :) = vandermond_matrix(x, n)
694     w(:) = solve(V, y, n)
695     s = 0.0D0
696     do i=1, n
697         s = s + (w(i) * f(x(i)))
698     end do
699     return
700 end function
701
702 function gauss_legendre_int(f, a, b, n) result (s)
703     implicit none
704     integer, intent(in) :: n
705     double precision, intent(in) :: a, b
706     double precision :: s
707     double precision, dimension(n) :: xx, ww
708     integer :: k
709     character(len=*), parameter :: fname =
        GAUSS_LEGENDRE_QUAD
710     interface
711         function f(x) result (y)
712             double precision :: x, y
713         end function
714     end interface
715
716     call load_quad(xx, ww, n, fname//STR(n)//".txt")
717

```

```

718         xx(:) = ((b - a) * xx(:) + (b + a)) / 2
719         s = 0.0D0
720         do k=1, n
721             s = s + (ww(k) * f(xx(k)))
722         end do
723         s = s * ((b - a) / 2)
724         return
725     end function
726
727     function gauss_hermite_int(f, a, b, n) result (s)
728         implicit none
729         integer, intent(in) :: n
730         double precision, intent(in) :: a, b
731         double precision :: s
732         double precision, dimension(n) :: xx, ww
733         integer :: k
734         character(len=*), parameter :: fname =
735             GAUSS_HERMITE_QUAD
736         interface
737             function f(x) result (y)
738                 double precision :: x, y
739             end function
740         end interface
741
742         call load_quad(xx, ww, n, fname//STR(n)//".txt")
743
744         if (a /= DNINF .OR. b /= DINP) then
745             call error("O Método de Gauss-Hermite deve ser usado
746                 no intervalo dos reais.")
747             stop
748         end if
749
750         s = 0.0D0
751         do k=1, n
752             s = s + (ww(k) * f(xx(k)))
753         end do
754
755         return
756     end function
757
758     recursive function adapt_int(f, a, b, n, tol, kind) result (
759         s)
760         implicit none
761         integer :: n
762         character (len=*), optional :: kind
763         double precision, intent(in) :: a, b
764         double precision :: p, q, e, r, s, t_tol
765         double precision, optional :: tol
766         interface
767             function f(x) result (y)
768                 double precision :: x, y
769             end function
770         end interface

```

```

768
769         if (.NOT. PRESENT(tol)) then
770             t_tol = D_TOL
771         else
772             t_tol = tol
773         end if
774
775         if (n > 1) then
776             p = num_int(f, a, b, n / 2, kind = kind)
777             q = num_int(f, a, b, n, kind = kind)
778             e = DABS(p - q)
779             if (e <= t_tol) then
780                 s = q
781             else
782                 r = (b + a) / 2
783                 s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
784                     + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
785             end if
786             return
787         else
788             s = 0.0D0
789             return
790         end if
791     end function
792
793     function romberg_int(f, a, b, n, tol) result (s)
794         implicit none
795         integer, intent(in) :: n
796         double precision, intent(in) :: a, b
797         double precision, optional :: tol
798         interface
799             function f(x) result (y)
800                 double precision :: x, y
801             end function
802         end interface
803         integer :: i, j, k, t_n
804         double precision :: s, dx, t_tol
805         ! Previous row, Current row and Temporary row
806         double precision, dimension(:, :), allocatable :: R
807
808         if (.NOT. PRESENT(tol)) then
809             t_tol = D_TOL
810         else
811             t_tol = tol
812         end if
813
814         t_n = IL0G2(n)
815
816         dx = (b - a)
817
818         allocate(R(t_n + 1, t_n + 1))
819
820         R(1, 1) = (f(a) + f(b)) * dx / 2

```

```

820
821      do i = 1, t_n
822          dx = dx / 2
823
824          R(i + 1, 1) = (f(a) + 2 * SUM((/ (f(a + k*dx), k=1,
            (2**i)-1) /)) + f(b)) * dx / 2;
825
826          do j = 1, i
827              k = 4 ** j
828              R(i + 1, j + 1) = (k*R(i + 1, j) - R(i, j)) / (k
            - 1)
829          end do
830
831          if (DABS(R(i + 1, i + 1) - R(i, i)) > t_tol) then
832              continue
833          else
834              exit
835          end if
836      end do
837      s = R(i, i)
838
839      deallocate(R)
840  end function
841
842  function richard(f, x, p, q, dx, kind) result (y)
843  !      Richard Extrapolation
844      implicit none
845      double precision, optional :: dx, p, q
846      character(len=*), optional :: kind
847      double precision :: x, y, t_p, t_q, t_dx, dx1, dx2, d1,
            d2
848      interface
849          function f(x) result (y)
850              implicit none
851              double precision :: x, y
852          end function
853      end interface
854
855      if (.NOT. PRESENT(dx)) then
856          t_dx = h
857      else
858          t_dx = dx
859      end if
860
861      if (.NOT. PRESENT(p)) then
862          t_p = 1.0D0
863      else
864          t_p = p
865      end if
866
867      if (.NOT. PRESENT(q)) then
868          t_q = 2.0D0
869      else

```



```

870         t_q = q
871     end if
872
873     dx1 = t_dx
874     d1 = d(f, x, dx1, kind = kind)
875     dx2 = dx1 / t_q
876     d2 = d(f, x, dx2, kind = kind)
877
878     y = d1 + (d1 - d2) / ((t_q ** (-t_p)) - 1.0D0)
879     return
880 end function
881
882 ! ===== Ordinary Differential Equations =====
883 function ode_solve(df, y0, t, n, kind) result (y)
884     implicit none
885     integer :: n
886     double precision, intent(in) :: y0
887     double precision, dimension(n), intent(in) :: t
888     double precision, dimension(n) :: y
889     character(len=*), optional :: kind
890     character(len=:), allocatable :: t_kind
891     interface
892         function df(t, y) result (u)
893             implicit none
894             double precision :: t, y, u
895         end function
896     end interface
897
898     if (.NOT. PRESENT(kind)) then
899         t_kind = 'euler'
900     else
901         t_kind = kind
902     end if
903
904     if (t_kind == 'euler') then
905         y = euler(df, y0, t, n)
906     else if (t_kind == 'runge-kutta2') then
907         y = runge_kutta2(df, y0, t, n)
908     else if (t_kind == 'runge-kutta4') then
909         y = runge_kutta4(df, y0, t, n)
910     else
911         call error("As opções são: 'euler', 'runge-kutta2' e '
912             runge-kutta4'.")
913         stop
914     end if
915     return
916 end function
917
918 function euler(df, y0, t, n) result (y)
919     implicit none
920     integer :: k, n
921     double precision, intent(in) :: y0

```

```

922     double precision :: dt
923     double precision, dimension(n), intent(in) :: t
924     double precision, dimension(n) :: y
925     interface
926         function df(t, y) result (u)
927             implicit none
928             double precision :: t, y, u
929         end function
930     end interface
931
932     y(1) = y0
933     do k=2, n
934         dt = t(k) - t(k - 1)
935         y(k) = y(k - 1) + df(t(k - 1), y(k - 1)) * dt
936     end do
937     return
938 end function
939
940 function runge_kutta2(df, y0, t, n) result (y)
941     implicit none
942     integer :: k, n
943     double precision, intent(in) :: y0
944     double precision :: k1, k2, dt
945     double precision, dimension(n), intent(in) :: t
946     double precision, dimension(n) :: y
947     interface
948         function df(t, y) result (u)
949             implicit none
950             double precision :: t, y, u
951         end function
952     end interface
953
954     y(1) = y0
955     do k=2, n
956         dt = t(k) - t(k - 1)
957         k1 = df(t(k - 1), y(k - 1))
958         k2 = df(t(k - 1) + dt, y(k - 1) + k1 * dt)
959         y(k) = y(k - 1) + dt * (k1 + k2) / 2
960     end do
961     return
962 end function
963
964 function runge_kutta4(df, y0, t, n) result (y)
965     implicit none
966     integer :: k, n
967     double precision, intent(in) :: y0
968     double precision :: k1, k2, k3, k4, dt
969     double precision, dimension(n), intent(in) :: t
970     double precision, dimension(n) :: y
971     interface
972         function df(t, y) result (u)
973             implicit none
974             double precision :: t, y, u

```

```

975         end function
976     end interface
977
978     y(1) = y0
979     do k=2, n
980         dt = t(k) - t(k - 1)
981         k1 = df(t(k - 1), y(k - 1))
982         k2 = df(t(k - 1) + dt / 2, y(k - 1) + k1 * dt / 2)
983         k3 = df(t(k - 1) + dt / 2, y(k - 1) + k2 * dt / 2)
984         k4 = df(t(k - 1) + dt, y(k - 1) + dt * k3)
985         y(k) = y(k - 1) + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6
986     end do
987     return
988 end function
989
990 function ode2_solve(d2f, y0, dy0, t, n, kind) result (y)
991     implicit none
992     integer :: n
993     double precision, intent(in) :: y0, dy0
994     double precision, dimension(n), intent(in) :: t
995     double precision, dimension(n) :: y
996     character(len=*), optional :: kind
997     character(len=:), allocatable :: t_kind
998     interface
999         function d2f(t, y, dy) result (u)
1000             implicit none
1001             double precision :: t, y, dy, u
1002         end function
1003     end interface
1004
1005     if (.NOT. PRESENT(kind)) then
1006         t_kind = 'taylor'
1007     else
1008         t_kind = kind
1009     end if
1010
1011     if (t_kind == 'taylor') then
1012         y = taylor(d2f, y0, dy0, t, n)
1013     else if (t_kind == 'runge-kutta-nystrom') then
1014         y = runge_kutta_nystrom(d2f, y0, dy0, t, n)
1015     else
1016         call error("As opções são: 'taylor', 'runge-kutta-  
nystrom'.")
1017         stop
1018     end if
1019     return
1020 end function
1021
1022 function taylor(d2f, y0, dy0, t, n) result (y)
1023     implicit none
1024     integer :: k, n
1025     double precision, intent(in) :: y0, dy0
1026     double precision :: dt, dy, d2y

```

```

1027     double precision, dimension(n), intent(in) :: t
1028     double precision, dimension(n) :: y
1029     interface
1030         function d2f(t, y, dy) result (d2y)
1031             implicit none
1032             double precision :: t, y, dy, d2y
1033         end function
1034     end interface
1035     ! Solution
1036     y(1) = y0
1037     ! 1st derivative
1038     dy = dy0
1039     do k=2, n
1040         dt = t(k) - t(k - 1)
1041         d2y = d2f(t(k - 1), y(k - 1), dy)
1042         y(k) = y(k - 1) + (dy * dt) + (d2y * dt ** 2) / 2
1043         dy = dy + d2y * dt
1044     end do
1045     return
1046 end function
1047
1048 function runge_kutta_nystrom(d2f, y0, dy0, t, n) result (y)
1049     implicit none
1050     integer :: k, n
1051     double precision, intent(in) :: y0, dy0
1052     double precision :: k1, k2, k3, k4, dt, dy, l, q
1053     double precision, dimension(n), intent(in) :: t
1054     double precision, dimension(n) :: y
1055     interface
1056         function d2f(t, y, dy) result (u)
1057             implicit none
1058             double precision :: t, y, dy, u
1059         end function
1060     end interface
1061
1062     y(1) = y0
1063     dy = dy0
1064     do k=2, n
1065         dt = t(k) - t(k - 1)
1066         k1 = (d2f(t(k - 1), y(k - 1), dy) * dt) / 2
1067         q = ((dy + k1 / 2) * dt) / 2
1068         k2 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k1) * dt
1069             ) / 2
1070         k3 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k2) * dt
1071             ) / 2
1072         l = (dy + k3) * dt
1073         k4 = (d2f(t(k - 1) + dt, y(k - 1) + l, dy + 2* k3) * dt)
1074             / 2
1075
1076         y(k) = y(k - 1) + (dy + (k1 + k2 + k3) / 3) * dt
1077         dy = dy + (k1 + 2 * k2 + 2 * k3 + k4) / 3
1078     end do
1079     return

```

```

1077     end function
1078     end module Calc

```

## Código - Métodos com Matrizes

```

1  !   Matrix Module
2
3  module Matrix
4      use Util
5      implicit none
6      integer :: D_MAX_ITER = 1000
7      double precision :: D_TOL = 1.0D-5
8  contains
9      subroutine ill_cond()
10 !           Prompts the user with an ill-conditioning warning.
11         implicit none
12         call error('Matriz mal-condicionada: este método não irá
13             convergir.')
14     end subroutine
15
16     subroutine show_matrix(var, A, m, n)
17         implicit none
18         integer :: m, n
19         character(len=*) :: var
20         double precision, dimension(m, n), intent(in) :: A
21         write (*, *) ' '//achar(27)//'[36m'//var//' = '
22         call print_matrix(A, m, n)
23         write (*, *) ' '//achar(27)//'[0m'
24     end subroutine
25
26     subroutine print_matrix(A, m, n)
27         implicit none
28         integer :: m, n
29         double precision :: A(m, n)
30         integer :: i, j
31         format(' /', F32.12, ' ')
32         format(F30.12, '/')
33         format(F30.12, ' ')
34         do i = 1, m
35             do j = 1, n
36                 if (j == 1) then
37                     write(*, 20, advance='no') A(i, j)
38                 elseif (j == n) then
39                     write(*, 21, advance='yes') A(i, j)
40                 else
41                     write(*, 22, advance='no') A(i, j)
42                 end if
43             end do
44         end do
45     end subroutine

```

```

46      subroutine read_matrix(fname, A, m, n)
47          implicit none
48          character(len=*) :: fname
49          integer :: m, n
50          double precision, dimension(:, :), allocatable :: A
51          integer :: i
52          open(unit=33, file=fname, status='old', action='read')
53          read(33, *) m
54          read(33, *) n
55          allocate(A(m, n))
56          do i = 1, m
57              read(33,*) A(i,:)
58          end do
59          close(33)
60      end subroutine
61
62      subroutine print_vector(x, n)
63          implicit none
64          integer :: n
65          double precision :: x(n)
66          integer :: i
67          format(' /', F30.12, '/')
68          do i = 1, n
69              write(*, 30) x(i)
70          end do
71      end subroutine
72
73      subroutine read_vector(fname, b, n)
74          implicit none
75          character(len=*) :: fname
76          integer :: n
77          double precision, allocatable :: b(:)
78
79          open(unit=33, file=fname, status='old', action='read')
80          read(33, *) n
81          allocate(b(n))
82          read(33, *) b(:)
83          close(33)
84      end subroutine
85
86      subroutine show_vector(var, x, n)
87          implicit none
88          integer :: n
89          character(len=*) :: var
90          double precision :: x(n)
91          write (*, *) '//a'//achar(27)//'[36m'//var//' = '
92          call print_vector(x, n)
93          write (*, *) '//a'//achar(27)//'[0m'
94      end subroutine
95
96
97      ! ===== Matrix Methods =====
98

```

```

99      function clip(x, n, a, b) result (y)
100         integer, intent(in) :: n
101         integer :: k
102         double precision, intent(in) :: a, b
103         double precision, dimension(n), intent(in) :: x
104         double precision, dimension(n) :: y
105
106         do k=1, n
107             if ((a <= x(k)) .AND. (x(k) <= b)) then
108                 y(k) = x(k)
109             else
110                 y(k) = DNAN
111             end if
112         end do
113         return
114     end function
115
116     function rand_vector(n, a, b) result (r)
117         implicit none
118         integer :: n, i
119         double precision, dimension(n) :: r
120         double precision, optional :: a, b
121         double precision :: t_a, t_b
122
123         if (.NOT. PRESENT(a)) then
124             t_a = -1.0D0
125         else
126             t_a = a
127         end if
128
129         if (.NOT. PRESENT(b)) then
130             t_b = 1.0D0
131         else
132             t_b = b
133         end if
134
135         do i = 1, n
136             r(i) = DRAND(t_a, t_b)
137         end do
138         return
139     end function
140
141     function rand_matrix(m, n, a, b) result (R)
142         implicit none
143         integer :: m, n, i
144         double precision, dimension(m, n) :: R
145         double precision, optional :: a, b
146
147         do i = 1, m
148             R(i, :) = rand_vector(n, a=a, b=b)
149         end do
150         return
151     end function

```

```

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

```



```

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

```

```

256
257      subroutine swap_rows(A, i, j, n)
258          implicit none
259
260          integer :: n
261          integer :: i, j
262          double precision A(n, n)
263          double precision temp(n)
264
265          temp(:) = A(i, :)
266          A(i, :) = A(j, :)
267          A(j, :) = temp(:)
268      end subroutine
269
270      function outer_product(x, y, n) result (A)
271          implicit none
272          integer :: n
273          double precision, dimension(n), intent(in) :: x, y
274          double precision, dimension(n, n) :: A
275          integer :: i, j
276          do i=1,n
277              do j=1,n
278                  A(i, j) = x(i) * y(j)
279              end do
280          end do
281          return
282      end function
283
284      ! ===== Matrix Method =====
285      function inv(A, n, ok) result (Ainv)
286          integer :: n
287          double precision :: A(n, n), Ainv(n, n)
288          double precision :: work(n)
289          integer :: ipiv(n)      ! pivot indices
290          integer :: info
291
292          logical :: ok
293
294          ! External procedures defined in LAPACK
295          external DGETRF
296          external DGETRI
297
298          ! Store A in Ainv to prevent it from being overwritten
299          !   by LAPACK
300          Ainv(:, :) = A(:, :)
301
302          ! DGETRF computes an LU factorization of a general M-by-
303          !   N matrix A
304          ! using partial pivoting with row interchanges.
305          call DGETRF(n, n, Ainv, n, ipiv, info)
306
307          if (info /= 0) then
308              ok = .FALSE.
309          else
310              ok = .TRUE.
311          end if
312      end function

```

```

307         return
308     end if
309
310     ! DGETRI computes the inverse of a matrix using the LU
311     ! factorization
312     ! computed by DGETRF.
313     call DGETRI(n, Ainv, n, ipiv, work, n, info)
314
315     if (info /= 0) then
316         ok = .FALSE.
317         return
318     end if
319
320     return
321 end function
322
323 function row_max(A, j, n) result(k)
324     implicit none
325
326     integer :: n
327     double precision A(n, n)
328
329     integer :: i, j, k
330     double precision :: s
331
332     s = 0.0D0
333     do i = j, n
334         if (A(i, j) > s) then
335             s = A(i, j)
336             k = i
337         end if
338     end do
339     return
340 end function
341
342 function pivot_matrix(A, n) result (P)
343     implicit none
344
345     integer :: n
346     double precision :: A(n, n)
347
348     double precision :: P(n, n)
349
350     integer :: j, k
351
352     P = id_matrix(n)
353
354     do j = 1, n
355         k = row_max(A, j, n)
356         if (j /= k) then
357             call swap_rows(P, j, k, n)
358         end if
359     end do

```

```

359         return
360     end function
361
362     function vector_norm(x, n) result (s)
363         implicit none
364         integer :: n
365         double precision :: x(n)
366         double precision :: s
367         s = sqrt(dot_product(x, x))
368         return
369     end function
370
371     function NORM(x, n) result (s)
372         implicit none
373         integer :: n
374         double precision :: x(n)
375         double precision :: s
376         s = SQRT(DOT_PRODUCT(x, x))
377         return
378     end function
379
380     function matrix_norm(A, n) result (s)
381         ! Frobenius norm
382         implicit none
383         integer :: n
384         double precision :: A(n, n)
385         double precision :: s
386
387         s = DSQRT(SUM(A * A))
388         return
389     end function
390
391     function spectral_radius(A, n) result (r)
392         implicit none
393
394         integer :: n
395         double precision :: A(n, n), x(n)
396         double precision :: r, l
397         logical :: ok
398         ok = power_method(A, n, x, 1)
399         r = DABS(l)
400         return
401     end function
402
403     recursive function det(A, n) result (d)
404         implicit none
405         integer :: n
406         double precision, dimension(n, n) :: A
407         double precision, dimension(n-1, n-1) :: X
408         integer :: i
409         double precision :: d, s
410
411         if (n == 1) then

```

```

412         d = A(1, 1)
413         return
414     elseif (n == 2) then
415         d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
416         return
417     else
418         d = 0.0D0
419         s = 1.0D0
420         do i = 1, n
421             ! Compute submatrix X
422             X(:, :i-1) = A(2:, :i-1)
423             X(:, i:) = A(2:, i+1:)
424             d = s * det(X, n-1) * A(1, i) + d
425             s = -s
426         end do
427     end if
428     return
429 end function
430
431 function LU_det(A, n) result (d)
432     implicit none
433
434     integer :: n
435     integer :: i
436     double precision :: A(n, n), L(n, n), U(n, n)
437     double precision :: d
438
439     d = 0.0D0
440
441     if (.NOT. LU_decomp(A, L, U, n)) then
442         call ill_cond()
443         return
444     end if
445
446     do i = 1, n
447         d = d * L(i, i) * U(i, i)
448     end do
449
450     return
451 end function
452
453 subroutine LU_matrix(A, L, U, n)
454     ! Splits Matrix in Lower and Upper-Triangular
455     implicit none
456
457     integer :: n
458     double precision :: A(n, n), L(n, n), U(n, n)
459
460     integer :: i
461
462     L(:, :) = 0.0D0
463     U(:, :) = 0.0D0
464

```

```

465         do i = 1, n
466             L(i, i) = 1.0D0
467             L(i, :i-1) = A(i, :i-1)
468             U(i, i:) = A(i, i:)
469         end do
470     end subroutine
471
472 !     === Matrix Factorization Conditions ===
473     function Cholesky_cond(A, n) result (ok)
474         implicit none
475         integer :: n
476         double precision :: A(n, n)
477         logical :: ok
478         ok = symmetrical(A, n) .AND. positive_definite(A, n)
479         return
480     end function
481
482     function PLU_cond(A, n) result (ok)
483         implicit none
484         integer :: n
485         double precision A(n, n)
486         integer :: i, j
487         double precision :: s
488         logical :: ok
489         do j = 1, n
490             s = 0.0D0
491             do i = 1, j
492                 if (A(i, j) > s) then
493                     s = A(i, j)
494                 end if
495             end do
496         end do
497         ok = (s < 0.01D0)
498         return
499     end function
500
501     function LU_cond(A, n) result (ok)
502         implicit none
503         integer :: n
504         double precision A(n, n)
505         logical :: ok
506         ok = positive_definite(A, n)
507         return
508     end function
509
510 !
511 !
512 !
513 !
514 !
515 !
516
517 !     ===== Matrix Factorization Methods =====

```

```

518     function PLU_decomp(A, P, L, U, n) result (ok)
519         implicit none
520         integer :: n
521         double precision :: A(n,n), P(n,n), L(n,n), U(n,n)
522         logical :: ok
523         ! Permutation Matrix
524         P = pivot_matrix(A, n)
525         ! Decomposition over Row-Swapped Matrix
526         ok = LU_decomp(matmul(P, A), L, U, n)
527         return
528     end function
529
530     function LU_decomp(A, L, U, n) result (ok)
531         implicit none
532         integer :: n
533         double precision :: A(n, n), L(n, n), U(n,n), M(n, n)
534         logical :: ok
535         integer :: i, j, k
536         ! Results Matrix
537         M(:, :) = A(:, :)
538         if (.NOT. LU_cond(A, n)) then
539             call ill_cond()
540             ok = .FALSE.
541             return
542         end if
543         do k = 1, n-1
544             do i = k+1, n
545                 M(i, k) = M(i, k) / M(k, k)
546             end do
547             do j = k+1, n
548                 do i = k+1, n
549                     M(i, j) = M(i, j) - M(i, k) * M(k, j)
550                 end do
551             end do
552         end do
553
554         ! Splits M into L & U
555         call LU_matrix(M, L, U, n)
556
557         ok = .TRUE.
558         return
559     end function
560
561     function Cholesky_decomp(A, L, n) result (ok)
562         implicit none
563
564         integer :: n
565         double precision :: A(n, n), L(n, n)
566
567         logical :: ok
568
569         integer :: i, j
570

```

```

571
572         if (.NOT. Cholesky_cond(A, n)) then
573             call ill_cond()
574             ok = .FALSE.
575             return
576         end if
577
578         do i = 1, n
579             L(i, i) = sqrt(A(i, i) - sum(L(i, :i-1) * L(i, :i-1)
580                                     ))
581             do j = 1 + 1, n
582                 L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)
583                                         )) / L(i, i)
584             end do
585         end do
586
587         ok = .TRUE.
588         return
589     end function
590
591 function Jacobi_cond(A, n) result (ok)
592     implicit none
593
594     integer :: n
595
596     double precision :: A(n, n)
597
598     logical :: ok
599
600     if (.NOT. spectral_radius(A, n) < 1.0D0) then
601         ok = .FALSE.
602         call ill_cond()
603         return
604     else
605         ok = .TRUE.
606         return
607     end if
608 end function
609
610 function Jacobi(A, x, b, e, n, tol, max_iter) result (ok)
611     implicit none
612
613     logical :: ok
614
615     integer :: n, i, k, t_max_iter
616     integer, optional :: max_iter
617
618     double precision :: A(n, n)
619     double precision :: b(n), x(n), x0(n)
620     double precision :: e, t_tol
621     double precision, optional :: tol
622
623     if (.NOT. PRESENT(tol)) then

```



```

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

```

```

674
675         if (symmetrical(A, n) .AND. positive_definite(A, n))
676             then
677                 ok = .TRUE.
678                 return
679             else
680                 call warn('Aviso: Esse método pode não convergir.')
681                 return
682             end if
683     end function
684
685 function Gauss_Seidel(A, x, b, e, n, tol, max_iter) result (
686     ok)
687     implicit none
688     logical :: ok
689     integer :: n, i, j, k, t_max_iter
690     integer, optional :: max_iter
691     double precision :: A(n, n)
692     double precision :: b(n), x(n)
693     double precision :: e, s, t_tol
694     double precision, optional :: tol
695
696     if (.NOT. PRESENT(tol)) then
697         t_tol = D_TOL
698     else
699         t_tol = tol
700     end if
701
702     if (.NOT. PRESENT(max_iter)) then
703         t_max_iter = D_MAX_ITER
704     else
705         t_max_iter = max_iter
706     end if
707
708     ok = Gauss_Seidel_cond(A, n)
709
710     if (.NOT. ok) then
711         return
712     end if
713
714     do k = 1, t_max_iter
715         do i = 1, n
716             s = 0.0D0
717             do j = 1, n
718                 if (i /= j) then
719                     s = s + A(i, j) * x(j)
720                 end if
721             end do
722             x(i) = (b(i) - s) / A(i, i)
723         end do
724         e = vector_norm(matmul(A, x) - b, n)
725         if (e < t_tol) then
726             return
727         end if
728     end do

```

```

725         end if
726     end do
727     call error('Erro: Esse método não convergiu.')
728     ok = .FALSE.
729     return
730 end function
731
732 ! Decomposição LU e afins
733 subroutine LU_backsub(L, U, x, y, b, n)
734     implicit none
735     integer :: n
736     double precision :: L(n, n), U(n, n)
737     double precision :: b(n), x(n), y(n)
738     integer :: i
739 ! Ly = b (Forward Substitution)
740     do i = 1, n
741         y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i)
742     end do
743 ! Ux = y (Backsubstitution)
744     do i = n, 1, -1
745         x(i) = (y(i) - SUM(U(i, i+1:n) * x(i+1:n))) / U(i, i)
746     end do
747 end subroutine
748
749 function LU_solve(A, x, y, b, n) result (ok)
750     implicit none
751
752     integer :: n
753
754     double precision :: A(n, n), L(n, n), U(n, n)
755     double precision :: b(n), x(n), y(n)
756
757     logical :: ok
758
759     ok = LU_decomp(A, L, U, n)
760
761     if (.NOT. ok) then
762         return
763     end if
764
765     call LU_backsub(L, U, x, y, b, n)
766
767     return
768 end function
769
770 function PLU_solve(A, x, y, b, n) result (ok)
771     implicit none
772
773     integer :: n
774
775     double precision :: A(n, n), P(n,n), L(n, n), U(n, n)
776     double precision :: b(n), x(n), y(n)

```

```

777
778         logical :: ok
779
780         ok = PLU_decomp(A, P, L, U, n)
781
782         if (.NOT. ok) then
783             return
784         end if
785
786         call LU_backsub(L, U, x, y, matmul(P, b), n)
787
788         x(:) = matmul(P, x)
789
790         return
791     end function
792
793     function Cholesky_solve(A, x, y, b, n) result (ok)
794         implicit none
795
796         integer :: n
797
798         double precision :: A(n, n), L(n, n), U(n, n)
799         double precision :: b(n), x(n), y(n)
800
801         logical :: ok
802
803         ok = Cholesky_decomp(A, L, n)
804
805         if (.NOT. ok) then
806             return
807         end if
808
809         U = transpose(L)
810
811         call LU_backsub(L, U, x, y, b, n)
812
813         return
814     end function
815
816     !
817     !
818     !
819     !
820     !
821     !
822     !
823
824     !
825     !
826     !
827     !
828     !
829     !

```

```

===== Power Method =====
function power_method(A, n, x, l, tol, max_iter) result (ok)
    implicit none
    logical :: ok
    integer :: n, k, t_max_iter
    integer, optional :: max_iter

```

```

830     double precision :: A(n, n)
831     double precision :: x(n)
832     double precision :: l, ll, t_tol
833     double precision, optional :: tol
834
835     if (.NOT. PRESENT(tol)) then
836         t_tol = D_TOL
837     else
838         t_tol = tol
839     end if
840
841     if (.NOT. PRESENT(max_iter)) then
842         t_max_iter = D_MAX_ITER
843     else
844         t_max_iter = max_iter
845     end if
846
847     ! Begin with random normal vector and set 1st component to
      zero
848     x(:) = rand_vector(n)
849     x(1) = 1.0D0
850
851     ! Initialize Eigenvalues
852     l = 0.0D0
853
854     ! Checks if error tolerance was reached
855     do k=1, t_max_iter
856         ll = l
857
858         x(:) = matmul(A, x)
859
860         ! Retrieve Eigenvalue
861         l = x(1)
862
863         ! Retrieve Eigenvector
864         x(:) = x(:) / l
865
866         if (dabs((l - ll) / l) < t_tol) then
867             ok = .TRUE.
868             return
869         end if
870     end do
871     ok = .FALSE.
872     return
873 end function
874
875 function Jacobi_eigen(A, n, L, X, tol, max_iter) result (ok)
876     implicit none
877     logical :: ok
878     integer :: n, i, j, k, u, v, t_max_iter
879     integer, optional :: max_iter
880     double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
881     double precision :: y, z, t_tol

```

```

882      double precision, optional :: tol
883
884      if (.NOT. PRESENT(tol)) then
885          t_tol = D_TOL
886      else
887          t_tol = tol
888      end if
889
890      if (.NOT. PRESENT(max_iter)) then
891          t_max_iter = D_MAX_ITER
892      else
893          t_max_iter = max_iter
894      end if
895
896      X(:, :) = id_matrix(n)
897      L(:, :) = A(:, :)
898
899      do k=1, t_max_iter
900          z = 0.0D0
901          do i = 1, n
902              do j = 1, i - 1
903                  y = DABS(L(i, j))
904
905                  ! Found new maximum absolute value
906                  if (y > z) then
907                      u = i
908                      v = j
909                      z = y
910                  end if
911              end do
912          end do
913
914          if (z >= t_tol) then
915              P(:, :) = given_matrix(L, n, u, v)
916              L(:, :) = matmul(matmul(transpose(P), L), P)
917              X(:, :) = matmul(X, P)
918          else
919              ok = .TRUE.
920              return
921          end if
922      end do
923      ok = .FALSE.
924      return
925  end function
926
927  !
928  ! / | _ _ _ _ _ / _ _ _ _ _ / _ _ _ _ _ / _ _ _ _ _ /
929  ! / | _ _ _ _ _ / | | | ( _ _ _ _ _ / | | | \ _ _ _ ) /
930  ! / | _ _ _ _ _ / | | \ _ _ _ \ | | | / \ / \ _ _ _ \
931  ! / | _ _ _ _ _ / | | _ _ _ _ _ ) / | | | / _ _ _ _ _ \ _ _ _ ) /
932  ! / _ _ _ _ _ / _ _ _ _ _ / _ _ _ _ _ / | _ / _ / \ _ \ / _ _ _ _ _ /
933  ! =====

```

```

935 function least_squares(x, y, s, n) result (ok)
936     implicit none
937     integer :: n
938
939     logical :: ok
940
941     double precision :: A(2,2), b(2), s(2), r(2), x(n), y(n)
942
943     A(1, 1) = n
944     A(1, 2) = SUM(x)
945     A(2, 1) = SUM(x)
946     A(2, 2) = dot_product(x, x)
947
948     b(1) = SUM(y)
949     b(2) = dot_product(x, y)
950
951     ok = Cholesky_solve(A, s, r, b, n)
952     return
953 end function
954
955 ! ===== Extra Stuff =====
956
957 function Gauss_solve(A0, x, b0, n) result (ok)
958     implicit none
959     integer n
960     double precision, dimension(n, n), intent(in) :: A0
961     double precision, dimension(n, n) :: A
962     double precision, dimension(n), intent(in) :: b0
963     double precision, dimension(n) :: b, x, s
964     double precision :: c, pivot, store
965     integer i, j, k, l
966
967     logical :: ok
968
969     ok = .TRUE.
970
971     A(:, :) = A0(:, :)
972     b(:) = b0(:)
973
974     do k=1, n-1
975         do i=k,n
976             s(i) = 0.0
977             do j=k,n
978                 s(i) = MAX(s(i), DABS(A(i,j)))
979             end do
980         end do
981
982         pivot = DABS(A(k,k) / s(k))
983         l = k
984         do j=k+1,n
985             if(DABS(A(j,k) / s(j)) > pivot) then
986                 pivot = DABS(A(j,k) / s(j))
987                 l = j

```

```

988         end if
989     end do
990
991     if(pivot == 0.0) then
992         ok = .FALSE.
993         return
994     end if
995
996     if (l /= k) then
997         do j=k,n
998             store = A(k,j)
999             A(k,j) = A(l,j)
1000             A(l,j) = store
1001         end do
1002         store = b(k)
1003         b(k) = b(l)
1004         b(l) = store
1005     end if
1006
1007     do i=k+1,n
1008         c = A(i,k) / A(k,k)
1009         A(i,k) = 0.0D0
1010         b(i) = b(i) - c*b(k)
1011         do j=k+1,n
1012             A(i,j) = A(i,j) - c * A(k,j)
1013         end do
1014     end do
1015 end do
1016
1017 x(n) = b(n) / A(n,n)
1018 do i=n-1,1,-1
1019     c = 0.0D0
1020     do j=i+1,n
1021         c = c + A(i,j) * x(j)
1022     end do
1023     x(i) = (b(i) - c) / A(i,i)
1024 end do
1025
1026 return
1027 end function
1028
1029 function solve(A, b, n, kind) result (x)
1030     implicit none
1031     integer :: n
1032     double precision, dimension(n), intent(in) :: b
1033     double precision, dimension(n) :: x, y
1034     double precision, dimension(n, n), intent(in) :: A
1035     character(len=*), optional :: kind
1036     character(len=:), allocatable :: t_kind
1037
1038     logical :: ok = .TRUE.
1039
1040     if (.NOT. PRESENT(kind)) then

```



```

1041         call debug("Indeed, not present.")
1042         t_kind = "gauss"
1043     else
1044         t_kind = kind
1045     end if
1046
1047     call debug("Now it is: "//t_kind)
1048     if (t_kind == "LU") then
1049         ok = LU_solve(A, x, y, b, n)
1050     else if (t_kind == "PLU") then
1051         ok = PLU_solve(A, x, y, b, n)
1052     else if (t_kind == "cholesky") then
1053         ok = Cholesky_solve(A, x, y, b, n)
1054     else if (t_kind == "gauss") then
1055         ok = Gauss_solve(A, x, b, n)
1056     else
1057         ok = .FALSE.
1058     end if
1059
1060     call debug(":: Solved via '//t_kind//"'::")
1061
1062     if (.NOT. ok) then
1063         call error("Failed to solve system Ax = b.")
1064     end if
1065
1066     return
1067 end function
1068
1069 end module Matrix

```

## Código - Biblioteca Auxiliar

```

1  !   Util Module
2  module Util
3      implicit none
4      character, parameter :: ENDL = ACHAR(10)
5      character, parameter :: TAB = ACHAR(9)
6
7      double precision :: DINF, DNINF, DNAN
8      DATA DINF/x'7ff0000000000000'/, DNINF/x'fff0000000000000'/,
9          DNAN/x'7ff8000000000000'/
10
11      double precision :: PI = 4.0D0 * DATAN(1.0D0)
12
13      logical :: DEBUG_MODE = .FALSE.
14      logical :: QUIET_MODE = .FALSE.
15
16      type StringArray
17          character (:), allocatable :: str
18      end type StringArray
19
20      contains

```

```

19
20  function EDGE(x) result (y)
21      double precision, intent(in) :: x
22      logical :: y
23
24      y = ISNAN(x) .OR. (x == DINF) .OR. (x == DNINF)
25      return
26  end function
27
28  function VEDGE(x) result (y)
29      double precision, dimension(:), intent(in) :: x
30      logical :: y
31
32      y = ANY(ISNAN(x)) .OR. ANY(x == DINF) .OR. ANY(x == DNINF)
33      return
34  end function
35
36  function MEDGE(x) result (y)
37      double precision, dimension(:, :), intent(in) :: x
38      logical :: y
39
40      y = ANY(ISNAN(x)) .OR. ANY(x == DINF) .OR. ANY(x == DNINF)
41      return
42  end function
43
44  function quote(s, q) result (r)
45      character(len=*), intent(in) :: s
46      character(len=*), optional, intent(in) :: q
47      character(len=:), allocatable :: t_q
48      character(len=:), allocatable :: r
49
50      if (.NOT. PRESENT(q)) then
51          t_q = ' '
52      else
53          t_q = q
54      end if
55
56      r = t_q//s//t_q
57  end function
58
59  function DLOG2(x) result (y)
60      implicit none
61      double precision, intent(in) :: x
62      double precision :: y
63
64      y = DLOG(x) / DLOG(2.0D0)
65      return
66  end function
67
68  function ILOG2(n) result (k)
69      integer, intent(in) :: n
70      integer :: k
71      double precision :: x

```

```

72         x = n
73         x = DLOG2(x)
74         k = FLOOR(x)
75         return
76     end function
77
78 !     ==== Random seed Initialization ====
79     subroutine init_random_seed()
80         integer :: i, n, clock
81         integer, allocatable :: seed(:)
82
83         call RANDOM_SEED(SIZE=n)
84         allocate(seed(n))
85         call SYSTEM_CLOCK(COUNT=clock)
86         seed = clock + 37 * (/ (i - 1, i = 1, n) /)
87         call RANDOM_SEED(PUT=seed)
88         deallocate(seed)
89     end subroutine
90
91     function DRAND(a, b) result (y)
92         implicit none
93         double precision :: a, b, x, y
94         ! x in [0, 1)
95         call RANDOM_NUMBER(x)
96         y = (x * (b - a)) + a
97         return
98     end function
99
100 !     ===== I/O Metods =====
101     function STR(k) result (t)
102 !     "Convert an integer to string."
103         integer, intent(in) :: k
104         character(len=128) :: s
105         character(len=:), allocatable :: t
106         write(s, *) k
107         t = TRIM(ADJUSTL(s))
108         return
109         return
110     end function
111
112     function DSTR(x) result (q)
113         integer :: j, k
114         double precision, intent(in) :: x
115         character(len=64) :: s
116         character(len=:), allocatable :: p, q
117
118         if (ISNAN(x)) then
119             q = '? '
120             return
121         else if (x == DINF) then
122             q = '∞ '
123             return
124         else if (x == DNINF) then

```

```

125         q = '-∞'
126         return
127     end if
128
129     write(s, *) x
130     p = TRIM(ADJUSTL(s))
131     do j=LEN(p), 1, -1
132         if (p(j:j) == '0') then
133             continue
134         else if (p(j:j) == '.') then
135             k = j - 1
136             exit
137         else
138             k = j
139             exit
140         end if
141     end do
142     q = p(:k)
143     return
144 end function
145
146 subroutine display(text, ansi_code)
147     implicit none
148     character(len=*) :: text
149     character(len=*), optional :: ansi_code
150     if (QUIET_MODE) then
151         return
152     else
153         if (PRESENT(ansi_code)) then
154             write (*, *) '//achar(27)//'[ansi_code//'m'
155                 //text//'//achar(27)//'[0m'
156         else
157             write (*, *) text
158         end if
159     end if
160 end subroutine
161
162 !
163 subroutine error(text)
164     Red Text
165     implicit none
166     character(len=*) :: text
167     call display(text, '31')
168 end subroutine
169
170 !
171 subroutine warn(text)
172     Yellow Text
173     implicit none
174     character(len=*) :: text
175     call display(text, '93')
176 end subroutine
177
178 !
179 subroutine debug(text)
180     Yellow Text

```

```

177         implicit none
178         character(len=*) :: text
179         if (DEBUG_MODE) then
180             call display('[DEBUG] '//text, '93')
181         end if
182     end subroutine
183
184     subroutine info(text)
185         ! Green Text
186         implicit none
187         character(len=*) :: text
188         call display(text, '32')
189     end subroutine
190
191     subroutine blue(text)
192         ! Blue Text
193         implicit none
194         character(len=*) :: text
195         call display(text, '36')
196     end subroutine
197
198     subroutine show(var, val)
199         ! Violet Text
200         implicit none
201         character(len=*) :: var
202         double precision :: val
203         write (*, *) '//achar(27)//'[36m'//var//' = '//DSTR(val
204             )//'//achar(27)//'[0m'
205     end subroutine
206
207     recursive subroutine cross_quick_sort(x, y, u, v, n)
208         integer :: n, i, j, u, v
209         double precision :: p, aux, auy
210         double precision :: x(n), y(n)
211
212         i = u
213         j = v
214
215         p = x((u + v) / 2)
216
217         do while (i <= j)
218             do while (x(i) < p)
219                 i = i + 1
220             end do
221             do while(x(j) > p)
222                 j = j - 1
223             end do
224             if (i <= j) then
225                 aux = x(i)
226                 auy = y(i)
227                 x(i) = x(j)
228                 y(i) = y(j)
229                 x(j) = aux

```

```

229         y(j) = auy
230         i = i + 1
231         j = j - 1
232     end if
233 end do
234
235     if (u < j) then
236         call cross_quick_sort(x, y, u, j, n)
237     end if
238     if (i < v) then
239         call cross_quick_sort(x, y, i, v, n)
240     end if
241     return
242 end subroutine
243
244 subroutine cross_sort(x, y, n)
245     implicit none
246     integer :: n
247     double precision :: x(n), y(n)
248
249     call cross_quick_sort(x, y, 1, n, n)
250 end subroutine
251
252 subroutine linspace(a, b, dt, n, t)
253     implicit none
254     integer :: k, n
255     double precision, intent(in) :: a, b, dt
256     double precision, dimension(:), allocatable :: t
257     n = 1 + FLOOR((b - a) / dt)
258     allocate(t(n))
259     t(:) = dt * (/ (k, k=0, n-1) /)
260 end subroutine
261 end module Util

```