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

```fortran
    function bissection(f, aa, bb, tol) result (x)
        implicit none
        double precision, intent(in) :: aa, bb
        double precision :: a, b, x, t_tol
        double precision, optional :: tol

        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        if (bb < aa) then
            a = bb
            b = aa
        else
            a = aa
            b = bb
        end if

        do while (DABS(a - b) > t_tol)
            x = (a + b) / 2
            if (f(a) > f(b)) then
                if (f(x) > 0) then
                    a = x
                else
                    b = x
                end if
            else
                if (f(x) < 0) then
                    a = x
                else
                    b = x
                end if
            end if
        end do
        x = (a + b) / 2
        return
    end function
```

Questão 2.: Método de *Newton*

1 .: Original

```
function newton(f, df, x0, ok, tol, max_iter) result (x)
    implicit none
    integer :: k, t_max_iter
    integer, optional :: max_iter
    double precision, intent(in) :: x0
    double precision :: x, xk, t_tol
    double precision, optional :: tol
    logical, intent(out) :: ok

    interface
        function f(x) result (y)
            double precision :: x, y
        end function
    end interface

    interface
        function df(x) result (y)
            double precision :: x, y
        end function
    end interface

    if (.NOT. PRESENT(max_iter)) then
        t_max_iter = D_MAX_ITER
    else
        t_max_iter = max_iter
    end if

    if (.NOT. PRESENT(tol)) then
        t_tol = D_TOL
    else
        t_tol = tol
    end if

    ok = .TRUE.
    xk = x0
    do k = 1, t_max_iter
        x = xk - f(xk) / df(xk)
        if (DABS(x - xk) > t_tol) then
            xk = x
        else
            if (ISNAN(x) .OR. x == DINF .OR. x == DNINF) then
                ok = .FALSE.
            end if
            return
        end if
    end do
    ok = .FALSE.
```

```
    return
end function
```

2 .: Secante

```
function secant(f, x0, ok, tol, max_iter) result (x)
    implicit none
    integer :: k, t_max_iter
    integer, optional :: max_iter
    double precision :: xk(3), yk(2)
    double precision, intent(in) :: x0
    double precision :: x, t_tol
    double precision, optional :: tol
    logical, intent(out) :: ok
    interface
        function f(x) result (y)
            implicit none
            double precision :: x, y
        end function
    end interface

    if (.NOT. PRESENT(max_iter)) then
        t_max_iter = D_MAX_ITER
    else
        t_max_iter = max_iter
    end if

    if (.NOT. PRESENT(tol)) then
        t_tol = D_TOL
    else
        t_tol = tol
    end if

    ok = .TRUE.

    xk(1) = x0
    xk(2) = x0 + h
    yk(1) = f(xk(1))
    do k = 1, t_max_iter
        yk(2) = f(xk(2))
        xk(3) = xk(2) - (yk(2) * (xk(2) - xk(1))) / (yk(2) - yk(1))
        if (DABS(xk(3) - xk(2)) > t_tol) then
            xk(1:2) = xk(2:3)
            yk(1) = yk(2)
        else
            x = xk(3)
            if (ISNAN(x) .OR. x == DINF .OR. x == DNINF) then
                ok = .FALSE.
            end if
            return
        end if
```

```
            end do
            ok = .FALSE.
            return
        end function
```

Questão 3.: Método de Interpolação Inversa

```fortran
        function inv_interp(f, x00, ok, tol, max_iter) result (x)
            implicit none
            logical, intent(out) :: ok
            integer :: i, j(1), k, t_max_iter
            integer, optional :: max_iter
            double precision :: x, xk, t_tol
            double precision, optional :: tol
            double precision, intent(in) :: x00(3)
            double precision :: x0(3), y0(3)

            interface
                function f(x) result (y)
                    double precision :: x, y
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            x0(:) = x00(:)
            xk = 1.0D+308

            ok = .TRUE.

            do k = 1, t_max_iter
                call cross_sort(x0, y0, 3)

!               Cálculo de y
                do i = 1, 3
                    y0(i) = f(x0(i))
                end do

                x = lagrange(y0, x0, 3, 0.0D0)

                if (DABS(x - xk) > t_tol) then
                    j(:) = MAXLOC(DABS(y0))
                    i = j(1)
                    x0(i) = x
                    y0(i) = f(x)
                    xk = x
                else
```

```fortran
         if (ISNAN(x) .OR. x == DINF .OR. x == DNINF) then
            ok = .FALSE.
         end if
         return
      end if
   end do
   ok = .FALSE.
   return
end function
```

Questão 4.: Sistemas de equações

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

```fortran
    function sys_newton(ff, dff, x0, n, ok, tol, max_iter)
       result (x)
        implicit none
        logical, intent(out) :: ok
        integer :: n, k, t_max_iter
        integer, optional :: max_iter
        double precision, dimension(n), intent(in) :: x0
        double precision, dimension(n) :: x, xdx, dx
        double precision, dimension(n, n) :: J
        double precision :: t_tol
        double precision, optional :: tol

        interface
            function ff(x, n) result (y)
                implicit none
                integer :: n
                double precision :: x(n), y(n)
            end function
        end interface

        interface
            function dff(x, n) result (J)
                implicit none
                integer :: n
                double precision :: x(n), J(n, n)
            end function
        end interface

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        ok = .TRUE.

        x = x0

        do k=1, t_max_iter
            J = dff(x, n)
            dx = -MATMUL(inv(J, n, ok), ff(x, n))
            xdx = x + dx
```

```

            if (.NOT. ok) then
                exit
            else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
                x = xdx
            else
                if (VEDGE(x)) then
                    ok = .FALSE.
                end if
                return
            end if
        end do
        ok = .FALSE.
        return
    end function
```

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

```
    function sys_newton_num(ff, x0, n, ok, tol, max_iter) result
        (x)
!       Same as previous function, with numerical partial
    derivatives
        implicit none
        logical, intent(out) :: ok
        integer :: n, i, k, t_max_iter
        integer, optional :: max_iter
        double precision, dimension(n), intent(in) :: x0
        double precision, dimension(n):: x, xdx, xh, dx
        double precision, dimension(n, n) :: J
        double precision :: t_tol
        double precision, optional :: tol

        interface
            function ff(x, n) result (y)
                implicit none
                integer :: n
                double precision :: x(n), y(n)
            end function
        end interface

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        ok = .TRUE.
```

```
            x = x0
            xh = 0.0D0

            do k=1, t_max_iter
!               Compute Jacobian Matrix
                do i=1, n
!                   Partial derivative with respect do the i-th coordinates
                    xh(i) = h
                    J(:, i) = (ff(x + xh, n) - ff(x - xh, n)) / (2 * h)
                    xh(i) = 0.0D0
                end do

                dx = -MATMUL(inv(J, n, ok), ff(x, n))
                xdx = x + dx

                if (.NOT. ok) then
                    exit
                else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
                    x = xdx
                else
                    if (VEDGE(x)) then
                        ok = .FALSE.
                    end if
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function
```

3 .: Método de *Broyden*

```
        function sys_broyden(ff, x0, B0, n, ok, tol, max_iter) result (x)
            implicit none
            logical, intent(out) :: ok
            integer :: n, k, t_max_iter
            integer, optional :: max_iter
            double precision, dimension(n), intent(in) :: x0
            double precision, dimension(n, n), intent(in) :: B0
            double precision, dimension(n) :: x, xdx, dx, dff
            double precision, dimension(n, n) :: J
            double precision :: t_tol
            double precision, optional :: tol

            interface
                function ff(x, n) result (y)
                    implicit none
                    integer :: n
                    double precision :: x(n), y(n)
```

```
            end function
        end interface

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        ok = .TRUE.

        x = x0
        J = B0

        do k=1, t_max_iter
            dx = -MATMUL(inv(J, n, ok), ff(x, n))
            if (.NOT. ok) then
                exit
            end if
            xdx = x + dx
            dff = ff(xdx, n) - ff(x, n)
            if ((norm(dx, n) / norm(xdx, n)) > t_tol) then
                J = J + OUTER_PRODUCT((dff - MATMUL(J, dx)) /
                    DOT_PRODUCT(dx, dx), dx, n)
                x = xdx
            else
                if (VEDGE(x) .OR. (NORM(ff(x, n), n) > t_tol))
                    then
                    ok = .FALSE.
                end if
                return
            end if
        end do
        ok = .FALSE.
        return
    end function
```

Questão 5.: Ajuste de curvas não-lineares

```fortran
        function sys_least_squares(ff, dff, x, y, b0, m, n, ok, tol, max_iter) result (b)
            implicit none
            logical, intent(out) :: ok
            integer :: m, n, k, t_max_iter
            integer, optional :: max_iter
            double precision, dimension(n), intent(in) :: x, y, b0
            double precision, dimension(n) :: b, bdb, db
            double precision :: J(n, n)
            double precision :: t_tol
            double precision, optional :: tol

            interface
                function ff(x, b, m, n) result (z)
                    implicit none
                    integer :: m, n
                    double precision, dimension(n), intent(in) :: x
                    double precision, dimension(m), intent(in) :: b
                    double precision, dimension(n) :: z
                end function
            end interface

            interface
                function dff(x, b, m, n) result (J)
                    implicit none
                    integer :: m, n
                    double precision, dimension(n), intent(in) :: x
                    double precision, dimension(m), intent(in) :: b
                    double precision, dimension(n, m) :: J
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            ok = .TRUE.

            b = b0

            do k=1, t_max_iter
                J = dff(x, b, m, n)
```

```
            db = -MATMUL(inv(MATMUL(TRANSPOSE(J), J), n, ok),
                MATMUL(TRANSPOSE(J), ff(x, b, m, n) - y))
            bdb = b + db

            if (.NOT. ok) then
                exit
            else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
                b = bdb
            else
                if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
                    t_tol)) then
                    ok = .FALSE.
                end if
                return
            end if
        end do
        ok = .FALSE.
        return
    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 * √(g * j))) - 50
: Método da Bissecção :
[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 Bissecção :
 [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) :=
16x^4 + 16y^4 + z^4 = 16
x^2 + y^2 + x^2 = 3
x^3 - 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) :=
c2^2 + 2 c3^2 + 6 c4^2 = 1
8 c3^3 + 6 c3 c2^2 + 36 c3 c2 c4 + 108 c3 c4^4 = θ1
60 * c3^4 + 60 * c3^2 * c2^2 + 576 * c3^2 * c2 * c4 + 2232 * c3^2 * c4^2 + 252 * c4
   ^2 * c2^2 + 1296 * c4^3 c2 + 3348 c4^4 + 24 c2^3 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 $f(x) = b_0 + b_1 x^{b_2}$ 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

```
program main4
    use Func
    use Calc
    use Util
    implicit none

!   Command-line Args
    integer :: argc

!   Random seed definition
    call init_random_seed()

!   Get Command-Line Args
    argc = iargc()

    if (argc == 0) then
        goto 100
    else
        goto 11
    end if

!   ====== Success ===================================
10  call info(':: Sucesso ::')
    goto 1
!   ====== Errors ====================================
11  call error('Este programa não aceita parâmetros.')
    goto 1
!   ====== Finish ====================================
1   stop
!   ==================================================

100 call Q1
    goto 200

200 call Q2
    goto 300

300 call Q3
    goto 400

400 call Q4
    goto 500

500 call Q5
    goto 10
!   ==================================================
    contains
```

```fortran

    subroutine Q1
        implicit none
        logical :: ok
        integer :: k
        double precision :: a, b, x, y, x0
        double precision :: xx(3)

        call blue("1) "//F1_NAME)
        call info(": Método da Bissecção :")
!       == Bounds definition ==
        a = -1000.0D0
        b =  1000.0D0
        call blue("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
!       == Algorithm run ======
        x = bissection(f1, a, b)
        y = f1(x)
!       == Results ============
        call show('x', x)
        call show('y', y)
!       =======================

        call info(": Método de Newton (Zero de função) :")
!       == Bounds definition ==
        ok = .FALSE.
        do while(.NOT. ok)
            x0 = DRAND(0.0D0, b)
!       == Algorithm run ======
            x = newton(f1, df1, x0, ok)
            y = f1(x)
        end do
!       == Results ============
        call show('x0', x0)
        call show('x', x)
        call show('y', y)
!       =======================

        call info(": Método da Secante :")
!       == Bounds definition ==
        ok = .FALSE.
        do while(.NOT. ok)
            x0 = DRAND(0.0D0, b)
!       == Algorithm run ======
            x = secant(f1, x0, ok)
            y = f1(x)
        end do
!       == Results ============
        call show('x0', x0)
        call show('x', x)
        call show('y', y)
!       =======================

        call info("Interpolação Inversa:")
```

```fortran
    !   == Bounds definition ==
        ok = .FALSE.
        do while(.NOT. ok)
            xx = (/ (DRAND(0.0D0, b), k=1,3)/)
    !   == Algorithm run ======
            x = inv_interp(f1, xx, ok)
            y = f1(x)
        end do
    !   == Results ============
        call blue('x1 = '//DSTR(xx(1))//'; x2 = '//DSTR(xx(2))//';
           x3 = '//DSTR(xx(3))//';')
        call show('x', x)
        call show('y', y)
    !   =======================
    end subroutine

    subroutine Q2
        implicit none
        logical :: ok
        integer :: k
        double precision :: a, b, x, y, x0
        double precision :: xx(3)

        call info(ENDL//"2) "//F2_NAME)
        call info(": Método da Bissecção :")
    !   == Bounds definition ==
        a = -1000.0D0
        b =  1000.0D0
        call blue("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
    !   == Algorithm run ======
        x = bissection(f2, a, b)
        y = f2(x)
    !   == Results ============
        call show('x', x)
        call show('y', y)
    !   =======================

        call info(": Método de Newton (Zero de função) :")
    !   == Bounds definition ==
        ok = .FALSE.
        do while(.NOT. ok)
            x0 = DRAND(0.0D0, b)
    !   == Algorithm run ======
            x = newton(f2, df2, x0, ok)
            y = f2(x)
        end do
    !   == Results ============
        call show('x0', x0)
        call show('x', x)
        call show('y', y)
    !   =======================

        call info(": Método da Secante :")
```

```fortran
    !   == Bounds definition ==
        ok = .FALSE.
        do while(.NOT. ok)
            x0 = DRAND(0.0D0, b)
    !   == Algorithm run ======
            x = secant(f2, x0, ok)
            y = f2(x)
        end do
    !   == Results ============
        call show('x0', x0)
        call show('x', x)
        call show('y', y)
    !   =======================

        call info("Interpolação Inversa:")
    !   == Bounds definition ==
    !   allocate(x123(3))
        ok = .FALSE.
        do while(.NOT. ok)
            xx = (/ (DRAND(0.0D0, b), k=1,3)/)
    !   == Algorithm run ======
            x = inv_interp(f2, xx, ok)
            y = f2(x)
        end do
    !   == Results ============
        call blue('x1 = '//DSTR(xx(1))//'; x2 = '//DSTR(xx(2))//'; x3 = '//DSTR(xx(3))//';')
        call show('x', x)
        call show('y', y)
    !   =======================
    end subroutine

    subroutine Q3
        implicit none
        logical :: ok
        integer :: k
        double precision, dimension(F3_N) :: x, y, x0
        double precision, dimension(F3_N, F3_N) :: J0

        call blue(ENDL//'3) '//F3_NAME)
        x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
        call info(": Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Analíticas] :")
    !   == Bounds definition ==
        do k=1, D_MAX_ITER
    !   == Algorithm run ======
            x = sys_newton(f3, df3, x0, F3_N, ok)
            if (.NOT. ok) then
                x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
            else
                exit
            end if
        end do
```

```
        if (.NOT. ok) then
            call error('Este método não convergiu.')
        else
            y = f3(x, F3_N)
        !   == Results ============
            call show_vector('x0', x0, F3_N)
            call show_vector('x', x, F3_N)
            call show_vector('y', y, F3_N)
        !   =======================
        end if

        call info(": Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Numéricas] :")
    !   == Bounds definition ==
        do k=1, D_MAX_ITER
    !   == Algorithm run ======
            x = sys_newton_num(f3, x0, F3_N, ok)
            if (.NOT. ok) then
                x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
            else
                exit
            end if
        end do
        if (.NOT. ok) then
            call error('Este método não convergiu.')
        else
            y = f3(x, F3_N)
        !   == Results ============
            call show_vector('x0', x0, F3_N)
            call show_vector('x', x, F3_N)
            call show_vector('y', y, F3_N)
        !   =======================
        end if

        call info(": Método de Broyden :")
    !   == Bounds definition ==
        J0 = id_matrix(F3_N)
        do k=1, D_MAX_ITER
    !   == Algorithm run ======
            x = sys_broyden(f3, x0, J0, F3_N, ok)
            if (.NOT. ok) then
                x0 = rand_vector(F3_N, 0.0D0, 1.0D0)
            else
                exit
            end if
        end do
        if (.NOT. ok) then
            call error('Este método não convergiu.')
        else
            y = f3(x, F3_N)
        !   == Results ============
            call show_vector('x0', x0, F3_N)
            call show_vector('x', x, F3_N)
```

```fortran
            call show_vector('y', y, F3_N)
        !   =======================
        end if
    end subroutine

    subroutine Q4
        implicit none
        logical :: ok
        integer :: i, k
        double precision, dimension(3) :: x, y, x0
        double precision, dimension(3, 3) :: J0

        call info('4) '//F4_NAME)

        do i=1, F4_N
            F4_T1 = F4_TT1(i)
            F4_T2 = F4_TT2(i)

            call show('θ1', F4_T1)
            call show('θ2', F4_T2)

            call info(": Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Analíticas] :")
            x0 = rand_vector(F4_N, 0.0D0, 1.0D0)
            do k=1, 5 * D_MAX_ITER
        !   == Algorithm run ======
                x = sys_newton(f4, df4, x0, F4_N, ok)
                if (.NOT. ok) then
                    x0 = rand_vector(F4_N, 0.0D0, 1.0D0)
                else
                    exit
                end if
            end do
            if (.NOT. ok) then
                call error('Este método não convergiu.')
            else
                y = f4(x, F4_N)
            !   == Results ============
                call show_vector('x0', x0, F4_N)
                call show_vector('x', x, F4_N)
                call show_vector('y', y, F4_N)
            !   =======================
            end if

            call info(": Método de Newton (Sistemas Não-Lineares) [Derivadas Parciais Numéricas] :")
            do k=1, 5 * D_MAX_ITER
        !   == Algorithm run ======
                x = sys_newton_num(f4, x0, F4_N, ok)
                if (.NOT. ok) then
                    x0 = rand_vector(F4_N, 0.0D0, 1.0D0)
                else
                    exit
```

```
                end if
            end do
            if (.NOT. ok) then
                call error('Este método não convergiu.')
            else
                y = f4(x, F4_N)
            !   == Results ============
                call show_vector('x0', x0, F4_N)
                call show_vector('x', x, F4_N)
                call show_vector('y', y, F4_N)
            !   =======================
            end if

            call info(": Método de Broyden :")
            J0 = id_matrix(F4_N)
            do k=1, 5 * D_MAX_ITER
        !   == Algorithm run ======
                x = sys_broyden(f4, x0, J0, F4_N, ok)
                if (.NOT. ok) then
                    x0 = rand_vector(F4_N, -1.0D0, 1.0D0)
                else
                    exit
                end if
            end do
            if (.NOT. ok) then
                call error('Este método não convergiu.')
            else
                y = f4(x, F4_N)
            !   == Results ============
                call show_vector('x0', x0, F4_N)
                call show_vector('x', x, F4_N)
                call show_vector('y', y, F4_N)
            !   =======================
            end if
        end do
    end subroutine

    subroutine Q5
        implicit none
        logical :: ok
        integer :: k
        double precision, dimension(3) :: x, y, b, b0

        call info("5) "//F5_NAME)

    !   ===============================
        x = (/1.0D0, 2.0D0, 3.0D0/)
        y = (/1.0D0, 2.0D0, 9.0D0/)

        b0 = rand_vector(F5_N, -1.0D0, 1.0D0)

        call info(": Método não-linear de Mínimos Quadrados :")
        do k=1, D_MAX_ITER
```

```fortran
    !    == Algorithm run ======
            b = sys_least_squares(f5, df5, x, y, b0, F5_N, F5_N, ok)
            if (.NOT. ok) then
                b0 = rand_vector(F5_N, 0.8D0, 1.0D0)
            else
                exit
            end if
        end do
        if (.NOT. ok) then
            call error('Este método não convergiu.')
        else
            y = f5(x, b, F5_N, F5_N)
        !    == Results ============
            call show_vector('b0', b0, F5_N)
            call show_vector('b', b, F5_N)
            call show_vector('x', x, F5_N)
            call show_vector('y', y, F5_N)
        !    =======================
        end if

        call info(": Método não-linear de Mínimos Quadrados [Derivadas Numéricas]:")
        do k=1, D_MAX_ITER
    !    == Algorithm run ======
            b = sys_least_squares(f5, df5, x, y, b0, F5_N, F5_N, ok)
            if (.NOT. ok) then
                b0 = rand_vector(F5_N, 0.8D0, 1.0D0)
            else
                exit
            end if
        end do
        if (.NOT. ok) then
            call error('Este método não convergiu.')
        else
            y = f5(x, b, F5_N, F5_N)
        !    == Results ============
            call show_vector('b0', b0, F5_N)
            call show_vector('b', b, F5_N)
            call show_vector('x', x, F5_N)
            call show_vector('y', y, F5_N)
        !    =======================
        end if
    end subroutine
end program main4
```

Código - Definição das Funções

```fortran
!    Func Module

    module Func
        use Util
```

```fortran
        implicit none

!       >> F1 <<
        character (len = *), parameter :: F1_NAME = "f(x) = log(cosh(x * √(g * j))) - 50"
        double precision :: F1_G = 9.80600D0
        double precision :: F1_K = 0.00341D0

!       >> F2 <<
        character (len = *), parameter :: F2_NAME = "f(x) = 4 * cos(x) - exp(2 * x)"

!       >> F3 <<
        character (len = *), parameter :: F3_NAME = "f(x, y, z) :="//ENDL// &
        "16x^4 + 16y^4 + z^4 = 16"//ENDL// &
        "x^2 + y^2 + x^2 = 3"//ENDL// &
        "x^3 - y + z = 1"
        integer :: F3_N = 3

!       >> F4 <<
        character (len = *), parameter :: F4_NAME = "f(c2, c3, c4) :="//ENDL// &
        "c2^2 + 2 c3^2 + 6 c4^2 = 1"//ENDL// &
        "8 c3^3 + 6 c3 c2^2 + 36 c3 c2 c4 + 108 c3 c4^4 = θ1"//ENDL// &
        "60 * c3^4 + 60 * c3^2 * c2^2 + 576 * c3^2 * c2 * c4 + "// &
        "2232 * c3^2 * c4^2 + 252 * c4^2 * c2^2 + "// &
        "1296 * c4^3 c2 + 3348 c4^4 + 24 c2^3 c4 + 3 c2 = θ2"
        double precision :: F4_TT1(3) = (/ 0.0D0, 0.75D0,  0.000D0 /)
        double precision :: F4_TT2(3) = (/ 3.0D0, 6.50D0, 11.667D0 /)
        double precision :: F4_T1 = 0.0D0
        double precision :: F4_T2 = 0.0D0
        integer :: F4_N = 3

!       >> F5 <<
        character (len = *), parameter :: F5_NAME = "f(x) = b1 + b2 x^b3"
        integer :: F5_N = 3

!       >> F6 <<
        character (len = *), parameter :: F6_NAME = "f(x) = exp(-x^2/2) / √(2 π)"

!       >> F7 <<
        character (len = *), parameter :: F7_NAME = "Sσ(ω) = RAO(ω)^2 Sη(ω)"//ENDL//TAB// &
        "RAO(ω) = 1 / √((1 - (ω/ωn)^2)^2 + (2ξω/ωn)^2)"

        character (len = *), parameter :: F7a_NAME = "Sη(ω) = 2"
        character (len = *), parameter :: F7b_NAME = "Sη(ω) = 2"

```

```
!        >> F8 <<
         character (len = *), parameter :: F8_NAME = "Sσ(ω) = RAO(ω)^2 Sη(ω)"//ENDL//TAB// &
         "RAO(ω) = 1 / √((1 - (ω/ωn)^2)^2 + (2ξω/ωn)^2)"

         character (len = *), parameter :: F8a_NAME = "Sη(ω) = ((4 π^3 Hs^2) / (ω^5 Tz^4)) exp(-(16 π^3) / (ω^4 Tz^4))"
         character (len = *), parameter :: F8b_NAME = "Sη(ω) = ((4 π^3 Hs^2) / (ω^5 Tz^4)) exp(-(16 π^3) / (ω^4 Tz^4))"


!        >> F13 <<
         character (len = *), parameter :: F13_NAME = "y'(t) = -2 t y(t)^2"//ENDL//"y(0) = 1"
         double precision :: F13_A = 0.0D0
         double precision :: F13_B = 10.0D0
         double precision :: F13_Y0 = 1.0D0

!        >> F14 <<
         character (len = *), parameter :: F14_NAME = "m y''(t) + c y'(t) + k y(t) = F(t)"//ENDL// &
         "m = 1; c = 0.2; k = 1;"//ENDL// &
         "F(t) = 2 sin(w t) + sin(2 w t) + cos(3 w t)"//ENDL// &
         "w = 0.5;"//ENDL// &
         "y'(0) = 0; y(0) = 0;"
         double precision :: F14_M = 1.0D0
         double precision :: F14_C = 0.2D0
         double precision :: F14_K = 1.0D0
         double precision :: F14_W = 0.5D0
         double precision :: F14_Y0 = 0.0D0
         double precision :: F14_DY0 = 0.0D0
         double precision :: F14_A = 0.0D0
         double precision :: F14_B = 100.0D0

!        >> F15 <<
         character (len = *), parameter :: F15_NAME = "z''(t) = -g -k z'(t) |z'(t)|"//ENDL// &
         "z'(0) = 0; z(0) = 0;"//ENDL// &
         "g = 9.806; k = 1;"
         double precision :: F15_G = 9.80600D0
         double precision :: F15_KD = 1.0D0
         double precision :: F15_BY0 = 100.0D0
         double precision :: F15_Y0 = 0.0D0
         double precision :: F15_DY0 = 0.0D0
         double precision :: F15_A = 0.0D0
         double precision :: F15_B = 20.0D0



         double precision :: t1 = 0.0D0
         double precision :: t2 = 0.0D0

         double precision :: wn = 1.00D0
```

```fortran
        double precision :: xi = 0.05D0
        double precision :: Hs = 3.0D0
        double precision :: Tz = 5.0D0

        character (len = *), parameter :: F9_NAME = "$f(x) = 2 + 2x - x^2 + 3x^3$"

        character (len = *), parameter :: F10_NAME = "$f(x) = 1 / (1 + x^2)$"

        character (len = *), parameter :: F11_NAME = "$f(x) = exp(- x^2/2) / \sqrt(2 \pi)$"
        character (len = *), parameter :: F12_NAME = "$f(x) = x^2 exp(- x^2/2) / \sqrt(2 \pi)$"

!       >> L5-QE <<
        character (len = *), parameter :: FL5_QE1_NAME = '$f(x) = x^3 + exp(-x)$'
        character (len = *), parameter :: DFL5_QE1_NAME = "$f'(x) = 3 x^2 - exp(-x)$"
        character (len = *), parameter :: FL5_QE2_NAME = '$f(x) = \sqrt[3]{x} + log(x)$'
        character (len = *), parameter :: DFL5_QE2_NAME = "$f'(x) = 1 / (3 \sqrt[3]{x^2}) + (1 / x)$"
        character (len = *), parameter :: FL5_QE3_NAME = '$f(x) = 1 - exp(-x^2 / 25)$'
        character (len = *), parameter :: DFL5_QE3_NAME = "$f'(x) = (2 x / 25) exp(-x^2 / 25)$"


    contains

    function FL5_QE1(x) result (y)
        implicit none
        double precision :: x, y
        y = x ** 3 + DEXP(-x)
        return
    end function

    function DFL5_QE1(x) result (y)
        implicit none
        double precision :: x, y
        y = 3 * x ** 2 - DEXP(-x)
        return
    end function

    function FL5_QE2(x) result (y)
        implicit none
        double precision :: x, y
        y = x ** (1.0D0/3.0D0) + DLOG(x)
        return
    end function

```

```fortran
    function DFL5_QE2(x) result (y)
        implicit none
        double precision :: x, y
        y = 1 / (3 * x ** (2.0D0/3.0D0)) + (1 / x)
        return
    end function

    function FL5_QE3(x) result (y)
        implicit none
        double precision :: x, y
        y = 1 - DEXP(-(x ** 2) / 25)
        return
    end function

    function DFL5_QE3(x) result (y)
        implicit none
        double precision :: x, y
        y = (2 * X) / (25 * DEXP((x ** 2) / 25))
        return
    end function


    function f1(x) result (y)
        implicit none
        double precision :: x, y
        y = DLOG(DCOSH(x * DSQRT(F1_G * F1_K))) - 50.0D0
        return
    end function

    function df1(x) result (y)
        implicit none
        double precision :: x, y
        y = (DSINH(x * DSQRT(F1_G * F1_K)) * DSQRT(F1_G * F1_K)) /
           DCOSH(x * DSQRT(F1_G * F1_K))
        return
    end function

    function f2(x) result (y)
        implicit none
        double precision :: x, y
        y = 4 * DCOS(x) - DEXP(2 * x)
        return
    end function

    function df2(x) result (y)
        implicit none
        double precision :: x, y
        y = - 4 * DSIN(x) - 2 * DEXP(2 * x)
        return
    end function

!   ======= R^n -> R^n functions =======
    function f3(x, n) result (y)
```

```fortran
!        R^3 -> R^3 (n == 3)
         implicit none
         integer :: n
         double precision, dimension(n) :: x, y

         y = (/ &
             (16 * x(1) ** 4 + 16 * x(2) ** 4 + x(3) ** 4) - 16.0D0, &
             x(1) ** 2 + x(2) ** 2 +x(3) ** 2 - 3.0D0, &
             x(1) ** 3 - x(2) + x(3) - 1.0D0 &
             /)
         return
     end function

!    ========== Derivative ===========
     function df3(x, n) result (J)
         implicit none
         integer :: n
         double precision, dimension(n) :: x
         double precision, dimension(n, n) :: J

         J(1, :) = (/ 64 * x(1) ** 3, 64 * x(2) ** 3, 4 * x(3) ** 3 /)
         J(2, :) = (/  2 * x(1)      ,  2 * x(2)      , 2 * x(3)      /)
         J(3, :) = (/  3 * x(1) ** 2,          -1.0D0,          1.0D0 /)
         return
     end function

!    ================== Another function =====================
     function f4(x, n) result (y)
         implicit none
         integer :: n
         double precision, dimension(n) :: x, y
         y = (/ &
             x(1)**2+2*x(2)**2+6*x(3)**2, &
             2*x(2)*(3*x(1)**2+4*x(2)**2+18*x(1)*x(3)+54*x(3)**4), &
             3*(x(1)+20*x(1)**2*x(2)**2+20*x(2)**4+8*x(1)*(x(1)**2+24*x(2)**2)*x(3)+&
             12*(7*x(1)**2+62*x(2)**2)*x(3)**2+432*x(1)*x(3)**3+1116*x(3)**4)&
             /) - (/ 1.0D0, F4_T1, F4_T2 /)
         return
     end function

!    ========== Derivatives ===========
     function df4(x, n) result (J)
!        R^3 -> R^3x3 (n == 3)
         implicit none
         integer :: n
         double precision :: x(n), J(n, n)

```

```
        J(1, :) = (/ &
            2*x(1), &
            4*x(2), &
            12*x(3) &
            /)
        J(2, :) = (/ &
            12*x(1)*x(2)+36*x(2)*x(3),                    &
            6*x(1)**2+24*x(2)**2+36*x(1)*x(3)+108*x(3)**4, &
            36*x(1)*x(2)+432*x(2)*x(3)**3                 &
            /)
        J(3, :) = (/ &
            3+120*x(1)*x(2)**2+72*x(1)**2*x(3)+576*x(2)**2*x(3)+504*x(1)*x(3)**2+1296*x(3)**3,           &
            120*x(1)**2*x(2)+240*x(2)**3+1152*x(1)*x(2)*x(3)+4464*x(2)*x(3)**2,                         &
            24*x(1)**3+576*x(1)*x(2)**2+504*x(1)**2*x(3)+4464*x(2)**2*x(3)+3888*x(1)*x(3)**2+13392*x(3)**3 &
            /)
        return
    end function

!   ============ One more function =============
    function f5(x, b, m, n) result (z)
        implicit none
        integer :: m, n
        double precision, dimension(m), intent(in) :: b
        double precision, dimension(n), intent(in) :: x
        double precision, dimension(n) :: z

        z = b(1) + (b(2) * (x ** b(3)))
        return
    end function

!   ========= Derivatives =========
    function df5(x, b, m, n) result (J)
        implicit none
        integer :: m, n
        double precision, dimension(m), intent(in) :: b
        double precision, dimension(n), intent(in) :: x
        double precision, dimension(n, m) :: J
!       m == 3
        J(:, 1) = 1.0D0
        J(:, 2) = x ** b(3)
        J(:, 3) = b(2) * DLOG(x) * (x ** b(3))
        return
    end function

!   ======== Function 6 ============
    function f6(x) result (y)
        implicit none
        double precision :: x, y

        y = DEXP(-(x*x)/2) / DSQRT(2 * PI)
```

```fortran
        return
    end function

!   ======== Functions 7 & 8 ============
    function RAO(w) result (z)
        implicit none
        double precision :: w, z

        z = 1.0 / DSQRT((1.0D0 - (w/wn) ** 2) ** 2 + (2 * xi * (w/wn
           )) ** 2)
    end function

    function Sn1(w) result (z)
        implicit none
        double precision :: w, z
        z = 2.0D0
        return
    end function

    function Sn2(w) result (z)
        implicit none
        double precision :: w, z
        z = (4 * (Hs**2) * (PI**3)) / ( DEXP( (16 * (PI**3))/((Tz*w)
           **4) ) * (Tz**4) * (w**5) )
        return
    end function

    function Ss(w, Sn) result (z)
        implicit none
        double precision :: w, z
        interface
            function Sn(w) result (z)
                implicit none
                double precision :: w, z
            end function
        end interface
        z = (RAO(w) ** 2) * Sn(w)
        return
    end function

    function f7a(w) result (z)
        implicit none
        double precision :: w, z
        z = Ss(w, Sn1)
        return
    end function

    function f7b(w) result (z)
        implicit none
        double precision :: w, z
        z = (w ** 2) * Ss(w, Sn1)
        return
    end function
```

```fortran

    function f8a(w) result (z)
        implicit none
        double precision :: w, z
        z = Ss(w, Sn2)
        return
    end function

    function f8b(w) result (z)
        implicit none
        double precision :: w, z
        z = (w ** 2) * Ss(w, Sn2)
        return
    end function

!   ========== Function 9 ==============
    function f9(x) result (y)
        implicit none
        double precision :: x, y
        y = 2.0D0 + 2.0D0 * x - x ** 2 + 3.0D0 * x ** 3
        return
    end function

!   ========== Function 10 ==============
    function f10(x) result (y)
        implicit none
        double precision :: x, y
        y = 1.0D0 / (1.0D0 + x ** 2)
        return
    end function

!   ========== Function 11 ==============
    function f11a(x) result (y)
        implicit none
        double precision :: x, y
        y = DEXP((x ** 2) / 2) / DSQRT(8 * PI)
        return
    end function

    function f11b(x) result (y)
        implicit none
        double precision :: x, y
        y = DEXP(-(x ** 2) / 2) / DSQRT(8 * PI)
        return
    end function

!   ========== Function 12 ==============
    function f12(x) result (y)
        implicit none
        double precision :: x, y
        y = (x ** 2) * DEXP((x ** 2) / 2) / DSQRT(2 * PI)
        return
    end function
```

```fortran
!    ========== Function 13 ==============
     function df13(t, y) result (u)
         implicit none
         double precision :: t, y, u
         u = - 2 * t * (y ** 2)
         return
     end function

     function f13(t) result (y)
         implicit none
         double precision :: t, y
         y = 1 / (1 + (t**2))
         return
     end function

!    ========== Function 14 ===============
     function F14_F(t) result (y)
         implicit none
         double precision :: t, y
         y = 2 * DSIN(F14_W * t) + DSIN(2 * F14_W * t) + DCOS(3 *
            F14_W * t)
         return
     end function

     function d2f14(t, y, dy) result (u)
         implicit none
         double precision :: t, y, dy, u
         u = (F14_F(t) - F14_K * y - F14_C * dy) / F14_M
         return
     end function

!    ========== Function 15 ===============
     function d2f15(t, y, dy) result (u)
         implicit none
         double precision :: t, y, dy, u
         if (y >= 0) then
             u = - F15_G
         else
             u = - F15_G - F15_KD * dy * DABS(dy)
         end if
         return
     end function

     end module Func
```

Código - Métodos Numéricos

```fortran
!    Calc Module

     module Calc
```

```fortran
        use Util
        use Matrix
        implicit none
        integer :: INT_N = 128
        double precision :: h = 1.0D-5
        !double precision :: D_TOL = 1.0D-5

        character (len=*), parameter :: GAUSS_LEGENDRE_QUAD = "quadratures/gauss-legendre/gauss-legendre"
        character (len=*), parameter :: GAUSS_HERMITE_QUAD = "quadratures/gauss-hermite/gauss-hermite"
    contains
!       ================= Numerical Mathods =================
        function d(f, x, dx, kind) result (y)
            implicit none
            character (len=*), optional :: kind
            double precision, optional :: dx
            character (len=:), allocatable :: t_kind
            double precision :: x, y, t_dx

            interface
                function f(x) result (y)
                    implicit none
                    double precision :: x, y
                end function
            end interface

            if (.NOT. PRESENT(dx)) then
                t_dx = h
            else
                t_dx = dx
            end if

            if (.NOT. PRESENT(kind)) then
                t_kind = "central"
            else
                t_kind = kind
            end if

            if (t_kind == "central") then
                y = (f(x + t_dx) - f(x - t_dx)) / (2 * t_dx)
            else if (t_kind == "forward") then
                y = (f(x + t_dx) - f(x)) / t_dx
            else if (t_kind == "backward") then
                y = (f(x) - f(x - t_dx)) / t_dx
            else
                call error("Unexpected value `"//t_kind//"` for derivative kind."// &
                "Options are: `central`, `forward` and `backward`.")
            end if
            return
        end function

```

```fortran
        function dp(f, x, i, n) result (y)
            implicit none
            integer :: i, n
            double precision :: f
            double precision :: x(n), xh(n)
            double precision :: y

            xh(:) = 0.0D0
            xh(i) = h

            y = (f(x + xh) - f(x - xh)) / (2 * h)
            return
        end function

        function grad(f, x, n) result (y)
            implicit none
            integer :: i, n
            double precision :: f
            double precision :: xh(n), x(n), y(n)

            xh(:) = 0.0D0
            do i=1, n
!               Compute partial derivative with respect to x_i
                xh(i) = h
                y(i) = (f(x + xh) - f(x - xh)) / (2 * h)
                xh(i) = 0.0D0
            end do
            return
        end function

!       ====================================================

        function lagrange(x0, y0, n, x) result (y)
            implicit none
            integer :: n
            double precision :: x0(n), y0(n)
            double precision :: x, y, yi
            integer :: i, j

            y = 0.0D0
            do i = 1, n
                yi = y0(i)
                do j = 1, n
                    if (i /= j) then
                        yi = yi * (x - x0(j)) / (x0(i) - x0(j))
                    end if
                end do
                y = y + yi
            end do

            return
        end function

```

```fortran
    function bissection(f, aa, bb, tol) result (x)
        implicit none
        double precision, intent(in) :: aa, bb
        double precision :: a, b, x, t_tol
        double precision, optional :: tol

        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        if (bb < aa) then
            a = bb
            b = aa
        else
            a = aa
            b = bb
        end if

        do while (DABS(a - b) > t_tol)
            x = (a + b) / 2
            if (f(a) > f(b)) then
                if (f(x) > 0) then
                    a = x
                else
                    b = x
                end if
            else
                if (f(x) < 0) then
                    a = x
                else
                    b = x
                end if
            end if
        end do
        x = (a + b) / 2
        return
    end function

    function newton(f, df, x0, ok, tol, max_iter) result (x)
        implicit none
        integer :: k, t_max_iter
        integer, optional :: max_iter
        double precision, intent(in) :: x0
        double precision :: x, xk, t_tol
        double precision, optional :: tol
```

```
            logical, intent(out) :: ok

            interface
                function f(x) result (y)
                    double precision :: x, y
                end function
            end interface

            interface
                function df(x) result (y)
                    double precision :: x, y
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            ok = .TRUE.
            xk = x0
            do k = 1, t_max_iter
                x = xk - f(xk) / df(xk)
                if (DABS(x - xk) > t_tol) then
                    xk = x
                else
                    if (ISNAN(x) .OR. x == DINF .OR. x == DNINF) then
                        ok = .FALSE.
                    end if
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

        function secant(f, x0, ok, tol, max_iter) result (x)
            implicit none
            integer :: k, t_max_iter
            integer, optional :: max_iter
            double precision :: xk(3), yk(2)
            double precision, intent(in) :: x0
            double precision :: x, t_tol
            double precision, optional :: tol
            logical, intent(out) :: ok
```

```
            interface
                function f(x) result (y)
                    implicit none
                    double precision :: x, y
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            ok = .TRUE.

            xk(1) = x0
            xk(2) = x0 + h
            yk(1) = f(xk(1))
            do k = 1, t_max_iter
                yk(2) = f(xk(2))
                xk(3) = xk(2) - (yk(2) * (xk(2) - xk(1))) / (yk(2) -
                    yk(1))
                if (DABS(xk(3) - xk(2)) > t_tol) then
                    xk(1:2) = xk(2:3)
                    yk(1) = yk(2)
                else
                    x = xk(3)
                    if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
                        then
                        ok = .FALSE.
                    end if
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

        function inv_interp(f, x00, ok, tol, max_iter) result (x)
            implicit none
            logical, intent(out) :: ok
            integer :: i, j(1), k, t_max_iter
            integer, optional :: max_iter
            double precision :: x, xk, t_tol
            double precision, optional :: tol
            double precision, intent(in) :: x00(3)
            double precision :: x0(3), y0(3)
```

```fortran

            interface
                function f(x) result (y)
                    double precision :: x, y
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            x0(:) = x00(:)
            xk = 1.0D+308

            ok = .TRUE.

            do k = 1, t_max_iter
                call cross_sort(x0, y0, 3)

!               Cálculo de y
                do i = 1, 3
                    y0(i) = f(x0(i))
                end do

                x = lagrange(y0, x0, 3, 0.0D0)

                if (DABS(x - xk) > t_tol) then
                    j(:) = MAXLOC(DABS(y0))
                    i = j(1)
                    x0(i) = x
                    y0(i) = f(x)
                    xk = x
                else
                    if (ISNAN(x) .OR. x == DINF .OR. x == DNINF) then
                        ok = .FALSE.
                    end if
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

```

```
    function sys_newton(ff, dff, x0, n, ok, tol, max_iter) result (x)
        implicit none
        logical, intent(out) :: ok
        integer :: n, k, t_max_iter
        integer, optional :: max_iter
        double precision, dimension(n), intent(in) :: x0
        double precision, dimension(n) :: x, xdx, dx
        double precision, dimension(n, n) :: J
        double precision :: t_tol
        double precision, optional :: tol

        interface
            function ff(x, n) result (y)
                implicit none
                integer :: n
                double precision :: x(n), y(n)
            end function
        end interface

        interface
            function dff(x, n) result (J)
                implicit none
                integer :: n
                double precision :: x(n), J(n, n)
            end function
        end interface

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        ok = .TRUE.

        x = x0

        do k=1, t_max_iter
            J = dff(x, n)
            dx = -MATMUL(inv(J, n, ok), ff(x, n))
            xdx = x + dx

            if (.NOT. ok) then
                exit
            else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
                x = xdx
```

```fortran
            else
                if (VEDGE(x)) then
                    ok = .FALSE.
                end if
                return
            end if
        end do
        ok = .FALSE.
        return
    end function

    function sys_newton_num(ff, x0, n, ok, tol, max_iter) result (x)
!       Same as previous function, with numerical partial derivatives
        implicit none
        logical, intent(out) :: ok
        integer :: n, i, k, t_max_iter
        integer, optional :: max_iter
        double precision, dimension(n), intent(in) :: x0
        double precision, dimension(n):: x, xdx, xh, dx
        double precision, dimension(n, n) :: J
        double precision :: t_tol
        double precision, optional :: tol

        interface
            function ff(x, n) result (y)
                implicit none
                integer :: n
                double precision :: x(n), y(n)
            end function
        end interface

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        ok = .TRUE.

        x = x0
        xh = 0.0D0

        do k=1, t_max_iter
!           Compute Jacobian Matrix
            do i=1, n
```

```
!                    Partial derivative with respect do the i-th
    coordinates
                     xh(i) = h
                     J(:, i) = (ff(x + xh, n) - ff(x - xh, n)) / (2 *
                         h)
                     xh(i) = 0.0D0
                 end do

                 dx = -MATMUL(inv(J, n, ok), ff(x, n))
                 xdx = x + dx

                 if (.NOT. ok) then
                     exit
                 else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
                     x = xdx
                 else
                     if (VEDGE(x)) then
                         ok = .FALSE.
                     end if
                     return
                 end if
             end do
             ok = .FALSE.
             return
         end function

         function sys_broyden(ff, x0, B0, n, ok, tol, max_iter)
            result (x)
             implicit none
             logical, intent(out) :: ok
             integer :: n, k, t_max_iter
             integer, optional :: max_iter
             double precision, dimension(n), intent(in) :: x0
             double precision, dimension(n, n), intent(in) :: B0
             double precision, dimension(n) :: x, xdx, dx, dff
             double precision, dimension(n, n) :: J
             double precision :: t_tol
             double precision, optional :: tol

             interface
                 function ff(x, n) result (y)
                     implicit none
                     integer :: n
                     double precision :: x(n), y(n)
                 end function
             end interface

             if (.NOT. PRESENT(max_iter)) then
                 t_max_iter = D_MAX_ITER
             else
                 t_max_iter = max_iter
             end if

```

```fortran
        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        ok = .TRUE.

        x = x0
        J = B0

        do k=1, t_max_iter
            dx = -MATMUL(inv(J, n, ok), ff(x, n))
            if (.NOT. ok) then
                exit
            end if
            xdx = x + dx
            dff = ff(xdx, n) - ff(x, n)
            if ((norm(dx, n) / norm(xdx, n)) > t_tol) then
                J = J + OUTER_PRODUCT((dff - MATMUL(J, dx)) /
                    DOT_PRODUCT(dx, dx), dx, n)
                x = xdx
            else
                if (VEDGE(x) .OR. (NORM(ff(x, n), n) > t_tol))
                    then
                     ok = .FALSE.
                end if
                return
            end if
        end do
        ok = .FALSE.
        return
    end function

    function sys_least_squares(ff, dff, x, y, b0, m, n, ok, tol,
        max_iter) result (b)
        implicit none
        logical, intent(out) :: ok
        integer :: m, n, k, t_max_iter
        integer, optional :: max_iter
        double precision, dimension(n), intent(in) :: x, y, b0
        double precision, dimension(n) :: b, bdb, db
        double precision :: J(n, n)
        double precision :: t_tol
        double precision, optional :: tol

        interface
            function ff(x, b, m, n) result (z)
                implicit none
                integer :: m, n
                double precision, dimension(n), intent(in) :: x
                double precision, dimension(m), intent(in) :: b
                double precision, dimension(n) :: z
```

```
                end function
            end interface

            interface
                function dff(x, b, m, n) result (J)
                    implicit none
                    integer :: m, n
                    double precision, dimension(n), intent(in) :: x
                    double precision, dimension(m), intent(in) :: b
                    double precision, dimension(n, m) :: J
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            ok = .TRUE.

            b = b0

            do k=1, t_max_iter
                J = dff(x, b, m, n)
                db = -MATMUL(inv(MATMUL(TRANSPOSE(J), J), n, ok), MATMUL(TRANSPOSE(J), ff(x, b, m, n) - y))
                bdb = b + db

                if (.NOT. ok) then
                    exit
                else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
                    b = bdb
                else
                    if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) > t_tol)) then
                        ok = .FALSE.
                    end if
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

        function sys_least_squares_num(ff, x, y, b0, m, n, ok, tol, max_iter) result (b)
```

```fortran
!            Same as previous function, with numerical partial derivatives
            implicit none
            integer :: m, n, i, k, t_max_iter
            integer, optional :: max_iter
            double precision, dimension(n), intent(in) :: x, y, b0
            double precision, dimension(n) :: b, bdb, db, bh
            double precision :: J(n, n)
            double precision :: t_tol
            double precision, optional :: tol

            logical, intent(out) :: ok
            interface
                function ff(x, b, m, n) result (z)
                    implicit none
                    integer :: m, n
                    double precision, dimension(n), intent(in) :: x
                    double precision, dimension(m), intent(in) :: b
                    double precision, dimension(n) :: z
                end function
            end interface

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            ok = .TRUE.

            bh = 0.0D0
            b = b0

            do k=1, t_max_iter
!                Compute Jacobian Matrix
                do i=1, m
!                    Partial derivative with respect do the i-th coordinates
                    bh(i) = h
                    J(:, i) = (ff(x, b + bh, m, n) - ff(x, b - bh, m, n)) / (2 * h)
                    bh(i) = 0.0D0
                end do
                db = -MATMUL(inv(MATMUL(TRANSPOSE(J), J), n, ok), MATMUL(TRANSPOSE(J), ff(x, b, m, n) - y))
                bdb = b + db

```

```fortran
            if (.NOT. ok) then
                exit
            else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
                b = bdb
            else
                if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
                    t_tol)) then
                    ok = .FALSE.
                end if
                return
            end if
        end do
        ok = .FALSE.
        return
    end function

!   ============ Numerical Integration ========
    subroutine load_quad(x, w, k, fname)
!       Load Quadrature
        implicit none
        integer :: k, m, n
        character (len=*) :: fname
        double precision, dimension(k) :: x, w
        double precision, dimension(:, :), allocatable :: xw
        call read_matrix(fname, xw, m, n)
        if (n /= 2 .OR. m /= k) then
            call error("Invalid Matrix dimensions.")
            stop "ERROR"
        end if
        x(:) = xw(:, 1)
        w(:) = xw(:, 2)
        deallocate(xw)
    end subroutine

    function num_int(f, a, b, n, kind) result (s)
        implicit none
        integer :: n
        character (len=*), optional :: kind
        double precision :: a, b, s
        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface

        if (.NOT. PRESENT(kind)) then
            kind = "polynomial"
        end if

        if (kind == "polynomial") then
            s = polynomial_int(f, a, b, n)
        else if (kind == "gauss-legendre") then
            s = gauss_legendre_int(f, a, b, n)
```

```fortran
        else if (kind == "gauss-hermite") then
            s = gauss_hermite_int(f, a, b, n)
        else if (kind == "romberg") then
            s = romberg_int(f, a, b, n)
        else
            call error("Unknown integration kind `"//kind//"."// &
            "Available options are: `polynomial`, `gauss-legendre`, `gauss-hermite` and `romberg`.")
        end if

    end function

    function polynomial_int(f, a, b, n) result (s)
        implicit none
        integer :: n, i
        double precision :: a, b, s
        double precision, dimension(n) :: x, y, w
        double precision, dimension(n, n) :: V
        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface

        x(:) = ((b-a)/(n-1)) * (/ (i, i=0,n-1) /) + a
        y(:) = (/ ((b**i - a**i)/i, i=1, n) /)
        V(:, :) = vandermond_matrix(x, n)
        w(:) = solve(V, y, n)
        s = 0.0D0
        do i=1, n
            s = s + (w(i) * f(x(i)))
        end do
        return
    end function

    function gauss_legendre_int(f, a, b, n) result (s)
        implicit none
        integer, intent(in) :: n
        double precision, intent(in) :: a, b
        double precision :: s
        double precision, dimension(n) :: xx, ww
        integer :: k
        character(len=*), parameter :: fname = GAUSS_LEGENDRE_QUAD
        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface

        call load_quad(xx, ww, n, fname//STR(n)//".txt")

```

```fortran
        xx(:) = ((b - a) * xx(:) + (b + a)) / 2
        s = 0.0D0
        do k=1, n
            s = s + (ww(k) * f(xx(k)))
        end do
        s = s * ((b - a) / 2)
        return
    end function

    function gauss_hermite_int(f, a, b, n) result (s)
        implicit none
        integer, intent(in) :: n
        double precision, intent(in) :: a, b
        double precision :: s
        double precision, dimension(n) :: xx, ww
        integer :: k
        character(len=*), parameter :: fname = GAUSS_HERMITE_QUAD
        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface

        call load_quad(xx, ww, n, fname//STR(n)//".txt")

        if (a /= DNINF .OR. b /= DINF) then
            call error("O Método de Gauss-Hermite deve ser usado no intervalo dos reais.")
            stop
        end if

        s = 0.0D0
        do k=1, n
            s = s + (ww(k) * f(xx(k)))
        end do

        return
    end function

    recursive function adapt_int(f, a, b, n, tol, kind) result (s)
        implicit none
        integer :: n
        character (len=*), optional :: kind
        double precision, intent(in) :: a, b
        double precision :: p, q, e, r, s, t_tol
        double precision, optional :: tol
        interface
            function f(x) result (y)
                double precision :: x, y
            end function
        end interface
```

```fortran

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            if (n > 1) then
                p = num_int(f, a, b, n / 2, kind = kind)
                q = num_int(f, a, b, n, kind = kind)
                e = DABS(p - q)
                if (e <= t_tol) then
                    s = q
                else
                    r = (b + a) / 2
                    s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
                        + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
                end if
                return
            else
                s = 0.0D0
                return
            end if
        end function

        function romberg_int(f, a, b, n, tol) result (s)
            implicit none
            integer, intent(in) :: n
            double precision, intent(in) :: a, b
            double precision, optional :: tol
            interface
                function f(x) result (y)
                    double precision :: x, y
                end function
            end interface
            integer :: i, j, k, t_n
            double precision :: s, dx, t_tol
!           Previous row, Current row and Temporary row
            double precision, dimension(:, :), allocatable :: R

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            t_n = ILOG2(n)

            dx = (b - a)

            allocate(R(t_n + 1, t_n + 1))

            R(1, 1) = (f(a) + f(b)) * dx / 2
```

```fortran

            do i = 1, t_n
                dx = dx / 2

                R(i + 1, 1) = (f(a) + 2 * SUM((/ (f(a + k*dx), k=1, (2**i)-1) /)) + f(b)) * dx / 2;

                do j = 1, i
                    k = 4 ** j
                    R(i + 1, j + 1) = (k*R(i + 1, j) - R(i, j)) / (k - 1)
                end do

                if (DABS(R(i + 1, i + 1) - R(i, i)) > t_tol) then
                    continue
                else
                    exit
                end if
            end do
            s = R(i, i)

            deallocate(R)
        end function

        function richard(f, x, p, q, dx, kind) result (y)
!           Richard Extrapolation
            implicit none
            double precision, optional :: dx, p, q
            character(len=*), optional :: kind
            double precision :: x, y, t_p, t_q, t_dx, dx1, dx2, d1, d2
            interface
                function f(x) result (y)
                    implicit none
                    double precision :: x, y
                end function
            end interface

            if (.NOT. PRESENT(dx)) then
                t_dx = h
            else
                t_dx = dx
            end if

            if (.NOT. PRESENT(p)) then
                t_p = 1.0D0
            else
                t_p = p
            end if

            if (.NOT. PRESENT(q)) then
                t_q = 2.0D0
            else
```

```
                t_q = q
            end if

            dx1 = t_dx
            d1 = d(f, x, dx1, kind = kind)
            dx2 = dx1 / t_q
            d2 = d(f, x, dx2, kind = kind)

            y = d1 + (d1 - d2) / ((t_q ** (-t_p)) - 1.0D0)
            return
        end function

!    ======== Ordinary Differential Equations ==========
    function ode_solve(df, y0, t, n, kind) result (y)
        implicit none
        integer :: n
        double precision, intent(in) :: y0
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        character(len=*), optional :: kind
        character(len=:), allocatable :: t_kind
        interface
            function df(t, y) result (u)
                implicit none
                double precision :: t, y, u
            end function
        end interface

        if (.NOT. PRESENT(kind)) then
            t_kind = 'euler'
        else
            t_kind = kind
        end if

        if (t_kind == 'euler') then
            y = euler(df, y0, t, n)
        else if (t_kind == 'runge-kutta2') then
            y = runge_kutta2(df, y0, t, n)
        else if (t_kind == 'runge-kutta4') then
            y = runge_kutta4(df, y0, t, n)
        else
            call error("As opções são: `euler`, `runge-kutta2` e `
               runge-kutta4`.")
            stop
        end if
        return
    end function


    function euler(df, y0, t, n) result (y)
        implicit none
        integer :: k, n
        double precision, intent(in) :: y0
```

```
        double precision :: dt
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        interface
            function df(t, y) result (u)
                implicit none
                double precision :: t, y, u
            end function
        end interface

        y(1) = y0
        do k=2, n
            dt = t(k) - t(k - 1)
            y(k) = y(k - 1) + df(t(k - 1), y(k - 1)) * dt
        end do
        return
    end function

    function runge_kutta2(df, y0, t, n) result (y)
        implicit none
        integer :: k, n
        double precision, intent(in) :: y0
        double precision :: k1, k2, dt
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        interface
            function df(t, y) result (u)
                implicit none
                double precision :: t, y, u
            end function
        end interface

        y(1) = y0
        do k=2, n
            dt = t(k) - t(k - 1)
            k1 = df(t(k - 1), y(k - 1))
            k2 = df(t(k - 1) + dt, y(k - 1) + k1 * dt)
            y(k) = y(k - 1) + dt * (k1 + k2) / 2
        end do
        return
    end function

    function runge_kutta4(df, y0, t, n) result (y)
        implicit none
        integer :: k, n
        double precision, intent(in) :: y0
        double precision :: k1, k2, k3, k4, dt
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        interface
            function df(t, y) result (u)
                implicit none
                double precision :: t, y, u
```

```
            end function
        end interface

        y(1) = y0
        do k=2, n
            dt = t(k) - t(k - 1)
            k1 = df(t(k - 1), y(k - 1))
            k2 = df(t(k - 1) + dt / 2, y(k - 1) + k1 * dt / 2)
            k3 = df(t(k - 1) + dt / 2, y(k - 1) + k2 * dt / 2)
            k4 = df(t(k - 1) + dt, y(k - 1) + dt * k3)
            y(k) = y(k - 1) + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6
        end do
        return
    end function

    function ode2_solve(d2f, y0, dy0, t, n, kind) result (y)
        implicit none
        integer :: n
        double precision, intent(in) :: y0, dy0
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        character(len=*), optional :: kind
        character(len=:), allocatable :: t_kind
        interface
            function d2f(t, y, dy) result (u)
                implicit none
                double precision :: t, y, dy, u
            end function
        end interface

        if (.NOT. PRESENT(kind)) then
            t_kind = 'taylor'
        else
            t_kind = kind
        end if

        if (t_kind == 'taylor') then
            y = taylor(d2f, y0, dy0, t, n)
        else if (t_kind == 'runge-kutta-nystrom') then
            y = runge_kutta_nystrom(d2f, y0, dy0, t, n)
        else
            call error("As opções são: ‘taylor‘, ‘runge-kutta-nystrom‘.")
            stop
        end if
        return
    end function

    function taylor(d2f, y0, dy0, t, n) result (y)
        implicit none
        integer :: k, n
        double precision, intent(in) :: y0, dy0
        double precision :: dt, dy, d2y
```

```
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        interface
            function d2f(t, y, dy) result (d2y)
                implicit none
                double precision :: t, y, dy, d2y
            end function
        end interface
!       Solution
        y(1) = y0
!       1st derivative
        dy = dy0
        do k=2, n
            dt = t(k) - t(k - 1)
            d2y = d2f(t(k - 1), y(k - 1), dy)
            y(k) = y(k - 1) + (dy * dt) + (d2y * dt ** 2) / 2
            dy = dy + d2y * dt
        end do
        return
    end function

    function runge_kutta_nystrom(d2f, y0, dy0, t, n) result (y)
        implicit none
        integer :: k, n
        double precision, intent(in) :: y0, dy0
        double precision :: k1, k2, k3, k4, dt, dy, l, q
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        interface
            function d2f(t, y, dy) result (u)
                implicit none
                double precision :: t, y, dy, u
            end function
        end interface

        y(1) = y0
        dy = dy0
        do k=2, n
            dt = t(k) - t(k - 1)
            k1 = (d2f(t(k - 1), y(k - 1), dy) * dt) / 2
            q = ((dy + k1 / 2) * dt) / 2
            k2 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k1) * dt
                ) / 2
            k3 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k2) * dt
                ) / 2
            l = (dy + k3) * dt
            k4 = (d2f(t(k - 1) + dt, y(k - 1) + l, dy + 2* k3) * dt)
                 / 2

            y(k) = y(k - 1) + (dy + (k1 + k2 + k3) / 3) * dt
            dy = dy + (k1 + 2 * k2 + 2 * k3 + k4) / 3
        end do
        return
```

```
    end function
    end module Calc
```

Código - Métodos com Matrizes

```
!    Matrix Module

    module Matrix
        use Util
        implicit none
        integer :: D_MAX_ITER = 1000
        double precision :: D_TOL = 1.0D-5
    contains
        subroutine ill_cond()
!           Prompts the user with an ill-conditioning warning.
            implicit none
            call error('Matriz mal-condicionada: este método não irá convergir.')
        end subroutine

        subroutine show_matrix(var, A, m, n)
            implicit none
            integer :: m, n
            character(len=*) :: var
            double precision, dimension(m, n), intent(in) :: A
            write (*, *) ''//achar(27)//'[36m'//var//' = '
            call print_matrix(A, m, n)
            write (*, *) ''//achar(27)//'[0m'
        end subroutine

        subroutine print_matrix(A, m, n)
            implicit none
            integer :: m, n
            double precision :: A(m, n)
            integer :: i, j
20          format(' |', F32.12, ' ')
21          format(F30.12, '|')
22          format(F30.12, ' ')
            do i = 1, m
                do j = 1, n
                    if (j == 1) then
                        write(*, 20, advance='no') A(i, j)
                    elseif (j == n) then
                        write(*, 21, advance='yes') A(i, j)
                    else
                        write(*, 22, advance='no') A(i, j)
                    end if
                end do
            end do
        end subroutine

```

```fortran
        subroutine read_matrix(fname, A, m, n)
            implicit none
            character(len=*) :: fname
            integer :: m, n
            double precision, dimension(:, :), allocatable :: A
            integer :: i
            open(unit=33, file=fname, status='old', action='read')
            read(33, *) m
            read(33, *) n
            allocate(A(m, n))
            do i = 1, m
                read(33,*) A(i,:)
            end do
            close(33)
        end subroutine

        subroutine print_vector(x, n)
            implicit none
            integer :: n
            double precision :: x(n)
            integer :: i
30          format(' |', F30.12, '|')
            do i = 1, n
                write(*, 30) x(i)
            end do
        end subroutine

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

            open(unit=33, file=fname, status='old', action='read')
            read(33, *) n
            allocate(b(n))
            read(33, *) b(:)
            close(33)
        end subroutine

        subroutine show_vector(var, x, n)
            implicit none
            integer :: n
            character(len=*) :: var
            double precision :: x(n)
            write (*, *) ''//achar(27)//'[36m'//var//' = '
            call print_vector(x, n)
            write (*, *) ''//achar(27)//'[0m'
        end subroutine


!       =========== Matrix Methods ============

```

```fortran
        function clip(x, n, a, b) result (y)
            integer, intent(in) :: n
            integer :: k
            double precision, intent(in) :: a, b
            double precision, dimension(n), intent(in) :: x
            double precision, dimension(n) :: y

            do k=1, n
                if ((a <= x(k)) .AND. (x(k) <= b)) then
                    y(k) = x(k)
                else
                    y(k) = DNAN
                end if
            end do
            return
        end function

        function rand_vector(n, a, b) result (r)
            implicit none
            integer :: n, i
            double precision, dimension(n) :: r
            double precision, optional :: a, b
            double precision :: t_a, t_b

            if (.NOT. PRESENT(a)) then
                t_a = -1.0D0
            else
                t_a = a
            end if

            if (.NOT. PRESENT(b)) then
                t_b = 1.0D0
            else
                t_b = b
            end if

            do i = 1, n
                r(i) = DRAND(t_a, t_b)
            end do
            return
        end function

        function rand_matrix(m, n, a, b) result (R)
            implicit none
            integer :: m, n, i
            double precision, dimension(m, n) :: R
            double precision, optional :: a, b

            do i = 1, m
                R(i, :) = rand_vector(n, a=a, b=b)
            end do
            return
        end function
```

```fortran
        function id_matrix(n) result (A)
            implicit none
            integer :: n
            double precision :: A(n, n)
            integer :: j
            A(:, :) = 0.0D0
            do j = 1, n
                A(j, j) = 1.0D0
            end do
            return
        end function

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

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

            G(:, :) = id_matrix(n)

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

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

            return
        end function

        function vandermond_matrix(x, n) result (V)
            implicit none
            integer :: n, i
            double precision, dimension(n), intent(in) :: x
            double precision, dimension(n, n) :: V
            V(1, :) = 1.0D0
            do i=2, n
                V(i, :) = V(i-1, :) * x(:)
            end do
            return
        end function

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

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

            logical :: ok
```

```fortran
        integer :: i

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

    recursive function positive_definite(A, n) result (ok)
!       Checks wether a matrix is positive definite
!       according to Sylvester's criterion.
        implicit none

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

        logical :: ok

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

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

        double precision :: A(n, n)

        integer :: i, j
        logical :: ok

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

```fortran

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

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

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

        function outer_product(x, y, n) result (A)
            implicit none
            integer :: n
            double precision, dimension(n), intent(in) :: x, y
            double precision, dimension(n, n) :: A
            integer :: i, j
            do i=1,n
                do j=1,n
                    A(i, j) = x(i) * y(j)
                end do
            end do
            return
        end function

!       ================= Matrix Method ====================
        function inv(A, n, ok) result (Ainv)
            integer :: n
            double precision :: A(n, n), Ainv(n, n)
            double precision :: work(n)
            integer :: ipiv(n)   ! pivot indices
            integer :: info

            logical :: ok

            ! External procedures defined in LAPACK
            external DGETRF
            external DGETRI

            ! Store A in Ainv to prevent it from being overwritten by LAPACK
            Ainv(:, :) = A(:, :)

            ! DGETRF computes an LU factorization of a general M-by-N matrix A
            ! using partial pivoting with row interchanges.
            call DGETRF(n, n, Ainv, n, ipiv, info)

            if (info /= 0) then
                ok = .FALSE.
```

```fortran
                return
            end if

            ! DGETRI computes the inverse of a matrix using the LU factorization
            ! computed by DGETRF.
            call DGETRI(n, Ainv, n, ipiv, work, n, info)

            if (info /= 0) then
                ok = .FALSE.
                return
            end if

            return
        end function

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

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

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

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

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

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

            double precision :: P(n, n)

            integer :: j, k

            P = id_matrix(n)

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

```
            return
        end function

        function vector_norm(x, n) result (s)
            implicit none
            integer :: n
            double precision :: x(n)
            double precision :: s
            s = sqrt(dot_product(x, x))
            return
        end function

        function NORM(x, n) result (s)
            implicit none
            integer :: n
            double precision :: x(n)
            double precision :: s
            s = SQRT(DOT_PRODUCT(x, x))
            return
        end function

        function matrix_norm(A, n) result (s)
!           Frobenius norm
            implicit none
            integer :: n
            double precision :: A(n, n)
            double precision :: s

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

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

            integer :: n
            double precision :: A(n, n), x(n)
            double precision :: r, l
            logical :: ok
            ok = power_method(A, n, x, l)
            r = DABS(l)
            return
        end function

        recursive function det(A, n) result (d)
            implicit none
            integer :: n
            double precision, dimension(n, n) :: A
            double precision, dimension(n-1, n-1) :: X
            integer :: i
            double precision :: d, s

            if (n == 1) then
```

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

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

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

                d = 0.0D0

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

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

                return
            end function

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

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

                integer :: i

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

```

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

!       === Matrix Factorization Conditions ===
        function Cholesky_cond(A, n) result (ok)
            implicit none
            integer :: n
            double precision :: A(n, n)
            logical :: ok
            ok = symmetrical(A, n) .AND. positive_definite(A, n)
            return
        end function

        function PLU_cond(A, n) result (ok)
            implicit none
            integer :: n
            double precision A(n, n)
            integer :: i, j
            double precision :: s
            logical :: ok
            do j = 1, n
                s = 0.0D0
                do i = 1, j
                    if (A(i, j) > s) then
                        s = A(i, j)
                    end if
                end do
            end do
            ok = (s < 0.01D0)
            return
        end function

        function LU_cond(A, n) result (ok)
            implicit none
            integer :: n
            double precision A(n, n)
            logical :: ok
            ok = positive_definite(A, n)
            return
        end function
!        _      _____  _____ _______           __
!       | |    |_   _|/ ____|__   __|/\       /_ |
!       | |      | | | (___    | |  /  \       | |
!       | |      | |  \___ \   | | / /\ \      | |
!       | |____ _| |_ ____) |  | |/ ____ \     | |
!       |______|_____|_____/   |_/_/    \_\    |_|
!       ==========================================

!       ======= Matrix Factorization Methods ========
```

```fortran
        function PLU_decomp(A, P, L, U, n) result (ok)
            implicit none
            integer :: n
            double precision :: A(n,n), P(n,n), L(n,n), U(n,n)
            logical :: ok
!           Permutation Matrix
            P = pivot_matrix(A, n)
!           Decomposition over Row-Swapped Matrix
            ok = LU_decomp(matmul(P, A), L, U, n)
            return
        end function

        function LU_decomp(A, L, U, n) result (ok)
            implicit none
            integer :: n
            double precision :: A(n, n), L(n, n), U(n,n), M(n, n)
            logical :: ok
            integer :: i, j, k
!           Results Matrix
            M(:, :) = A(:, :)
            if (.NOT. LU_cond(A, n)) then
                call ill_cond()
                ok = .FALSE.
                return
            end if
            do k = 1, n-1
                do i = k+1, n
                    M(i, k) = M(i, k) / M(k, k)
                end do
                do j = k+1, n
                    do i = k+1, n
                        M(i, j) = M(i, j) - M(i, k) * M(k, j)
                    end do
                end do
            end do

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

            ok = .TRUE.
            return

        end function

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

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

            logical :: ok

            integer :: i, j
```

```fortran

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

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

            ok = .TRUE.
            return
        end function

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

            integer :: n

            double precision :: A(n, n)

            logical :: ok

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

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

            logical :: ok

            integer :: n, i, k, t_max_iter
            integer, optional :: max_iter

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

            if (.NOT. PRESENT(tol)) then
```

```
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        x0 = rand_vector(n)

        ok = Jacobi_cond(A, n)

        if (.NOT. ok) then
            return
        end if

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

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

        integer :: n

        double precision :: A(n, n)

        logical :: ok

        integer :: i

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

```

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

    function Gauss_Seidel(A, x, b, e, n, tol, max_iter) result (ok)
        implicit none
        logical :: ok
        integer :: n, i, j, k, t_max_iter
        integer, optional :: max_iter
        double precision :: A(n, n)
        double precision :: b(n), x(n)
        double precision :: e, s, t_tol
        double precision, optional :: tol

        if (.NOT. PRESENT(tol)) then
            t_tol = D_TOL
        else
            t_tol = tol
        end if

        if (.NOT. PRESENT(max_iter)) then
            t_max_iter = D_MAX_ITER
        else
            t_max_iter = max_iter
        end if

        ok = Gauss_Seidel_cond(A, n)

        if (.NOT. ok) then
            return
        end if

        do k = 1, t_max_iter
            do i = 1, n
                s = 0.0D0
                do j = 1, n
                    if (i /= j) then
                        s = s + A(i, j) * x(j)
                    end if
                end do
                x(i) = (b(i) - s) / A(i, i)
            end do
            e = vector_norm(matmul(A, x) - b, n)
            if (e < t_tol) then
                return
```

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

!       Decomposição LU e afins
        subroutine LU_backsub(L, U, x, y, b, n)
            implicit none
            integer :: n
            double precision :: L(n, n), U(n, n)
            double precision :: b(n), x(n), y(n)
            integer :: i
!           Ly = b (Forward Substitution)
            do i = 1, n
                y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i
                    )
            end do
!           Ux = y (Backsubstitution)
            do i = n, 1, -1
                x(i) = (y(i) - SUM(U(i,i+1:n) * x(i+1:n))) / U(i, i)
            end do
        end subroutine

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

            integer :: n

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

            logical :: ok

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

            if (.NOT. ok) then
                return
            end if

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

            return
        end function

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

            integer :: n

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

```fortran

            logical :: ok

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

            if (.NOT. ok) then
                return
            end if

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

            x(:) = matmul(P, x)

            return
        end function

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

            integer :: n

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

            logical :: ok

            ok = Cholesky_decomp(A, L, n)

            if (.NOT. ok) then
                return
            end if

            U = transpose(L)

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

            return
        end function

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

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

```
            double precision :: A(n, n)
            double precision :: x(n)
            double precision :: l, ll, t_tol
            double precision, optional :: tol

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

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

!           Initialize Eigenvalues
            l = 0.0D0

!           Checks if error tolerance was reached
            do k=1, t_max_iter
                ll = l

                x(:) = matmul(A, x)

!               Retrieve Eigenvalue
                l = x(1)

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

                if (dabs((l - ll) / l) < t_tol) then
                    ok = .TRUE.
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

        function Jacobi_eigen(A, n, L, X, tol, max_iter) result (ok)
            implicit none
            logical :: ok
            integer :: n, i, j, k, u, v, t_max_iter
            integer, optional :: max_iter
            double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
            double precision :: y, z, t_tol
```

```
            double precision, optional :: tol

            if (.NOT. PRESENT(tol)) then
                t_tol = D_TOL
            else
                t_tol = tol
            end if

            if (.NOT. PRESENT(max_iter)) then
                t_max_iter = D_MAX_ITER
            else
                t_max_iter = max_iter
            end if

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

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

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

                if (z >= t_tol) then
                    P(:, :) = given_matrix(L, n, u, v)
                    L(:, :) = matmul(matmul(transpose(P), L), P)
                    X(:, :) = matmul(X, P)
                else
                    ok = .TRUE.
                    return
                end if
            end do
            ok = .FALSE.
            return
        end function

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

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

            logical :: ok

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

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

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

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

!       ========== Extra Stuff ========

        function Gauss_solve(A0, x, b0, n) result (ok)
            implicit none
            integer n
            double precision, dimension(n, n), intent(in) :: A0
            double precision, dimension(n, n) :: A
            double precision, dimension(n), intent(in) :: b0
            double precision, dimension(n) :: b, x, s
            double precision :: c, pivot, store
            integer i, j, k, l

            logical :: ok

            ok = .TRUE.

            A(:, :) = A0(:, :)
            b(:) = b0(:)

            do k=1, n-1
                do i=k,n
                    s(i) = 0.0
                    do j=k,n
                        s(i) = MAX(s(i), DABS(A(i,j)))
                    end do
                end do

                pivot = DABS(A(k,k) / s(k))
                l = k
                do j=k+1,n
                    if(DABS(A(j,k) / s(j)) > pivot) then
                        pivot = DABS(A(j,k) / s(j))
                        l = j
```

```
                    end if
                end do

                if(pivot == 0.0) then
                    ok = .FALSE.
                    return
                end if

                if (l /= k) then
                    do j=k,n
                        store = A(k,j)
                        A(k,j) = A(l,j)
                        A(l,j) = store
                    end do
                    store = b(k)
                    b(k) = b(l)
                    b(l) = store
                end if

                do i=k+1,n
                    c = A(i,k) / A(k,k)
                    A(i,k) = 0.0D0
                    b(i) = b(i)- c*b(k)
                    do j=k+1,n
                        A(i,j) = A(i,j) - c * A(k,j)
                    end do
                end do
            end do

            x(n) = b(n) / A(n,n)
            do i=n-1,1,-1
                c = 0.0D0
                do j=i+1,n
                    c = c + A(i,j) * x(j)
                end do
                x(i) = (b(i)- c) / A(i,i)
            end do

            return
        end function

        function solve(A, b, n, kind) result (x)
            implicit none
            integer :: n
            double precision, dimension(n), intent(in) :: b
            double precision, dimension(n) :: x, y
            double precision, dimension(n, n), intent(in) :: A
            character(len=*), optional :: kind
            character(len=:), allocatable :: t_kind

            logical :: ok = .TRUE.

            if (.NOT. PRESENT(kind)) then
```

```fortran
                call debug("Indeed, not present.")
                t_kind = "gauss"
            else
                t_kind = kind
            end if

            call debug("Now it is: "//t_kind)
            if (t_kind == "LU") then
                ok = LU_solve(A, x, y, b, n)
            else if (t_kind == "PLU") then
                ok = PLU_solve(A, x, y, b, n)
            else if (t_kind =="cholesky") then
                ok = Cholesky_solve(A, x, y, b, n)
            else if (t_kind =="gauss") then
                ok = Gauss_solve(A, x, b, n)
            else
                ok = .FALSE.
            end if

            call debug(":: Solved via `"//t_kind//"` ::")

            if (.NOT. ok) then
                call error("Failed to solve system Ax = b.")
            end if

            return
        end function

    end module Matrix
```

Código - Biblioteca Auxiliar

```fortran
!   Util Module
    module Util
        implicit none
        character, parameter :: ENDL = ACHAR(10)
        character, parameter :: TAB = ACHAR(9)

        double precision :: DINF, DNINF, DNAN
        DATA DINF/x'7ff0000000000000'/, DNINF/x'fff0000000000000'/,
           DNAN/x'7ff8000000000000'/

        double precision :: PI = 4.0D0 * DATAN(1.0D0)

        logical :: DEBUG_MODE = .FALSE.
        logical :: QUIET_MODE = .FALSE.

        type StringArray
            character (:), allocatable :: str
        end type StringArray
    contains
```

```fortran

    function EDGE(x) result (y)
        double precision, intent(in) :: x
        logical :: y

        y = ISNAN(x) .OR. (x == DINF) .OR. (x == DNINF)
        return
    end function

    function VEDGE(x) result (y)
        double precision, dimension(:), intent(in) :: x
        logical :: y

        y = ANY(ISNAN(x)) .OR. ANY(x == DINF) .OR. ANY(x == DNINF)
        return
    end function

    function MEDGE(x) result (y)
        double precision, dimension(:, :), intent(in) :: x
        logical :: y

        y = ANY(ISNAN(x)) .OR. ANY(x == DINF) .OR. ANY(x == DNINF)
        return
    end function

        function quote(s, q) result (r)
            character(len=*), intent(in) :: s
            character(len=*), optional, intent(in) :: q
            character(len=:), allocatable :: t_q
            character(len=:), allocatable :: r

            if (.NOT. PRESENT(q)) then
                t_q = '"'
            else
                t_q = q
            end if

            r = t_q//s//t_q
        end function

        function DLOG2(x) result (y)
            implicit none
            double precision, intent(in) :: x
            double precision :: y

            y = DLOG(x) / DLOG(2.0D0)
            return
        end function

        function ILOG2(n) result (k)
            integer, intent(in) :: n
            integer :: k
            double precision :: x
```

```fortran
        x = n
        x = DLOG2(x)
        k = FLOOR(x)
        return
    end function

!   ==== Random seed Initialization ====
    subroutine init_random_seed()
        integer :: i, n, clock
        integer, allocatable :: seed(:)

        call RANDOM_SEED(SIZE=n)
        allocate(seed(n))
        call SYSTEM_CLOCK(COUNT=clock)
        seed = clock + 37 * (/ (i - 1, i = 1, n) /)
        call RANDOM_SEED(PUT=seed)
        deallocate(seed)
    end subroutine

    function DRAND(a, b) result (y)
        implicit none
        double precision :: a, b, x, y
        ! x in [0, 1)
        call RANDOM_NUMBER(x)
        y = (x * (b - a)) + a
        return
    end function

!   ===== I/O Metods =====
    function STR(k) result (t)
!   "Convert an integer to string."
        integer, intent(in) :: k
        character(len=128) :: s
        character(len=:), allocatable :: t
        write(s, *) k
        t = TRIM(ADJUSTL(s))
        return
        return
    end function

    function DSTR(x) result (q)
        integer :: j, k
        double precision, intent(in) :: x
        character(len=64) :: s
        character(len=:), allocatable :: p, q

        if (ISNAN(x)) then
            q = '?'
            return
        else if (x == DINF) then
            q = '∞'
            return
        else if (x == DNINF) then
```

```
            q = '-∞'
            return
        end if

        write(s, *) x
        p = TRIM(ADJUSTL(s))
        do j=LEN(p), 1, -1
            if (p(j:j) == '0') then
                continue
            else if (p(j:j) == '.') then
                k = j - 1
                exit
            else
                k = j
                exit
            end if
        end do
        q = p(:k)
        return
    end function

    subroutine display(text, ansi_code)
        implicit none
        character(len=*) :: text
        character(len=*), optional :: ansi_code
        if (QUIET_MODE) then
            return
        else
            if (PRESENT(ansi_code)) then
                write (*, *) ''//achar(27)//'['//ansi_code//'m'//text//''//achar(27)//'[0m'
            else
                write (*, *) text
            end if
        end if
    end subroutine

    subroutine error(text)
!       Red Text
        implicit none
        character(len=*) :: text
        call display(text, '31')
    end subroutine

    subroutine warn(text)
!       Yellow Text
        implicit none
        character(len=*) :: text
        call display(text, '93')
    end subroutine

    subroutine debug(text)
!       Yellow Text
```

```
        implicit none
        character(len=*) :: text
        if (DEBUG_MODE) then
            call display('[DEBUG] '//text, '93')
        end if
    end subroutine

    subroutine info(text)
!       Green Text
        implicit none
        character(len=*) :: text
        call display(text, '32')
    end subroutine

    subroutine blue(text)
!       Blue Text
        implicit none
        character(len=*) :: text
        call display(text, '36')
    end subroutine

    subroutine show(var, val)
!       Violet Text
        implicit none
        character(len=*) :: var
        double precision :: val
        write (*, *) ''//achar(27)//'[36m'//var//' = '//DSTR(val)//''//achar(27)//'[0m'
    end subroutine

    recursive subroutine cross_quick_sort(x, y, u, v, n)
        integer :: n, i, j, u, v
        double precision :: p, aux, auy
        double precision :: x(n), y(n)

        i = u
        j = v

        p = x((u + v) / 2)

        do while (i <= j)
            do while (x(i) < p)
                i = i + 1
            end do
            do while(x(j) > p)
                j = j - 1
            end do
            if (i <= j) then
                aux = x(i)
                auy = y(i)
                x(i) = x(j)
                y(i) = y(j)
                x(j) = aux
```

```fortran
                    y(j) = auy
                    i = i + 1
                    j = j - 1
                end if
            end do

            if (u < j) then
                call cross_quick_sort(x, y, u, j, n)
            end if
            if (i < v) then
                call cross_quick_sort(x, y, i, v, n)
            end if
            return
        end subroutine

        subroutine cross_sort(x, y, n)
            implicit none
            integer :: n
            double precision :: x(n), y(n)

            call cross_quick_sort(x, y, 1, n, n)
        end subroutine

        subroutine linspace(a, b, dt, n, t)
            implicit none
            integer :: k, n
            double precision, intent(in) :: a, b, dt
            double precision, dimension(:), allocatable :: t
            n = 1 + FLOOR((b - a) / dt)
            allocate(t(n))
            t(:) = dt * (/ (k, k=0, n-1) /)
        end subroutine
    end module Util
```