COC473 - Lista 6

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.: Escreva um programa que "integre" uma equação diferencial ordinária de primeira ordem $y'(t) = f(t, y(t))$ onde o usuário pode escolher o método de solução entre as três seguintes possibilidades:

a) *Euler*

b) *Runge-Kutta* de segunda ordem

c) *Runge-Kutta* de quarta ordem

A função `ode_solve(df, y0, t, n, kind)` integra a função `df` sobre o domínio do tempo definido pelo vetor `t[n]`, assumindo que o valor de $y(t)$ correspondente a primeira entrada de `t[n]` é `y0`. O argumento nomeado opcional `kind` define o método a ser utilizado:

'euler' *Euler*

'runge-kutta2' *Runge-Kutta* de segunda ordem

'runge-kutta4' *Runge-Kutta* de quarta ordem

```fortran
!    ======== 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)
```

```fortran
            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))
```

```fortran
77          k2 = df(t(k - 1) + dt, y(k - 1) + k1 * dt)
78          y(k) = y(k - 1) + dt * (k1 + k2) / 2
79       end do
80       return
81    end function
82
83    function runge_kutta4(df, y0, t, n) result (y)
84       implicit none
85       integer :: k, n
86       double precision, intent(in) :: y0
87       double precision :: k1, k2, k3, k4, dt
88       double precision, dimension(n), intent(in) :: t
89       double precision, dimension(n) :: y
90       interface
91          function df(t, y) result (u)
92             implicit none
93             double precision :: t, y, u
94          end function
95       end interface
96
97       y(1) = y0
98       do k=2, n
99          dt = t(k) - t(k - 1)
100          k1 = df(t(k - 1), y(k - 1))
101          k2 = df(t(k - 1) + dt / 2, y(k - 1) + k1 * dt / 2)
102          k3 = df(t(k - 1) + dt / 2, y(k - 1) + k2 * dt / 2)
103          k4 = df(t(k - 1) + dt, y(k - 1) + dt * k3)
104          y(k) = y(k - 1) + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6
105       end do
106       return
107    end function
```
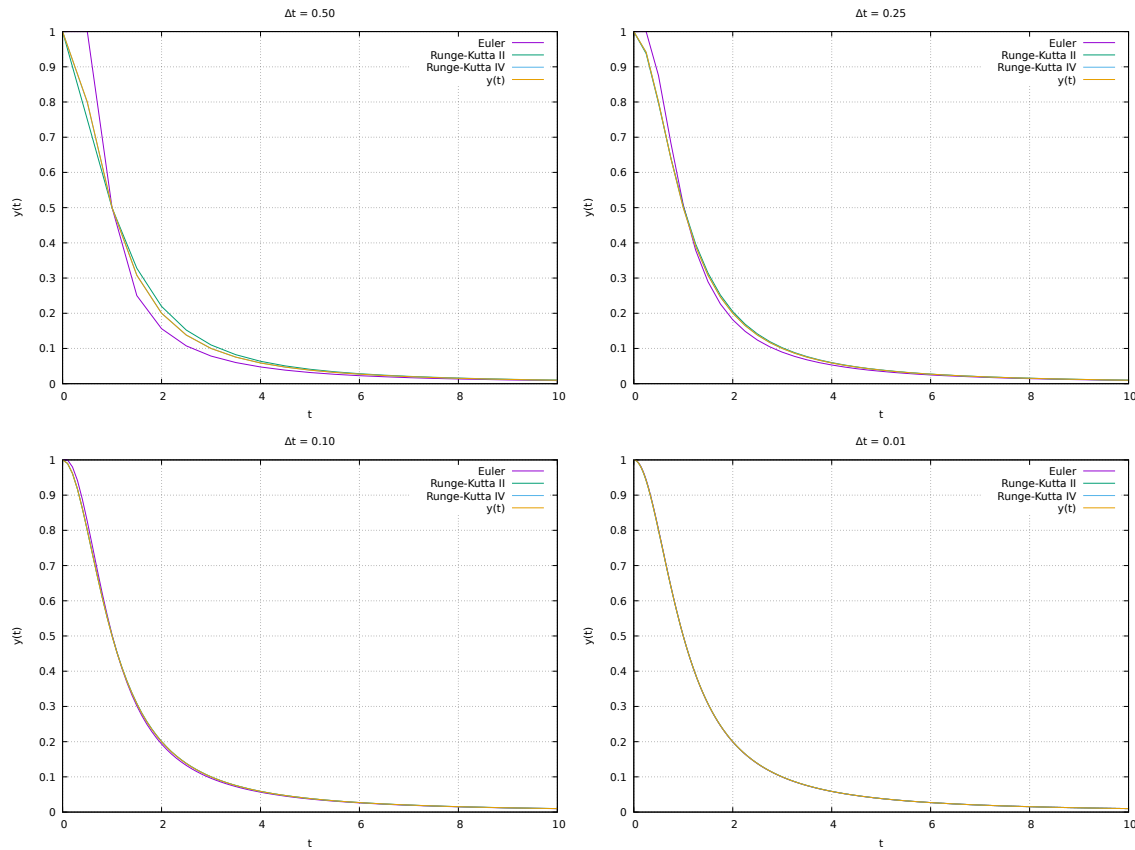
Aplicações

Questão 2.: Utilizando o programa desenvolvido, use os métodos de *Euler*, *Runge-Kutta* de segunda ordem e *Runge-Kutta* de quarta ordem, para $0 \leq t \leq 10$, e resolva a seguinte equação diferencial ordinária:

$$y'(t) = -2ty(t)^2$$
$$y(0) = 1$$

Compare com a solução exata $y(t) = \dfrac{1}{1 + t^2}$

Vemos neste caso que o método de *Runge-Kutta* de quarta ordem converge muito rapidamente para a solução ao longo de toda a curva, sendo difícil distinguir seu gráfico daquele que descreve a solução. O método de segunda ordem é o próximo a garantir sua convergência. Por fim, o método de *Euler* se aproxima da solução para passos ainda menores, uma ordem de grandeza abaixo dos demais.

É interessante notar, neste caso, que apesar de se distanciarem da solução no meio do percurso, as aproximações numéricas atingem valores muito próximos ao fim do processo, mesmo com passos mais generosos.

Programas

Questão 3.: Escreva um programa que "integre" uma equação diferencial ordinária de segunda ordem $y''(t) = f(t, y(t), y'(t))$ onde o usuário pode escolher o método de solução entre as seguintes possibilidades:

a) Aproximação de segunda ordem em Série de *Taylor*

b) *Runge-Kutta-Nystrom*

A função `ode2_solve(d2f, y0, dy0, t, n, kind)` integra a função `d2f` sobre o domínio do tempo definido pelo vetor `t[n]`, assumindo que o valor de $y(t)$ correspondente a primeira entrada de `t[n]` é `y0` e que o valor de $y'(t)$ neste mesmo ponto é `dy0`. O argumento nomeado opcional `kind` define o método a ser utilizado:

> `'taylor'` Aproximação em Série de *Taylor*

`'runge-kutta-nystrom'` Método de *Runge-Kutta-Nystrom*

```fortran
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'.")
```

6

```fortran
                    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
```

```
80            k3 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k2) * dt
                  ) / 2
81            l = (dy + k3) * dt
82            k4 = (d2f(t(k - 1) + dt, y(k - 1) + l, dy + 2* k3) * dt)
                  / 2
83
84            y(k) = y(k - 1) + (dy + (k1 + k2 + k3) / 3) * dt
85            dy = dy + (k1 + 2 * k2 + 2 * k3 + k4) / 3
86        end do
87        return
88    end function
```

Aplicações

Questão 4.: Resolva a seguinte equação diferencial pelos métodos 'Expansão em Série de *Taylor*' e *Runge-Kutta-Nystrom*, $0 \leq t \leq 100$, utilizando o programa desenvolvido na Tarefa 02:
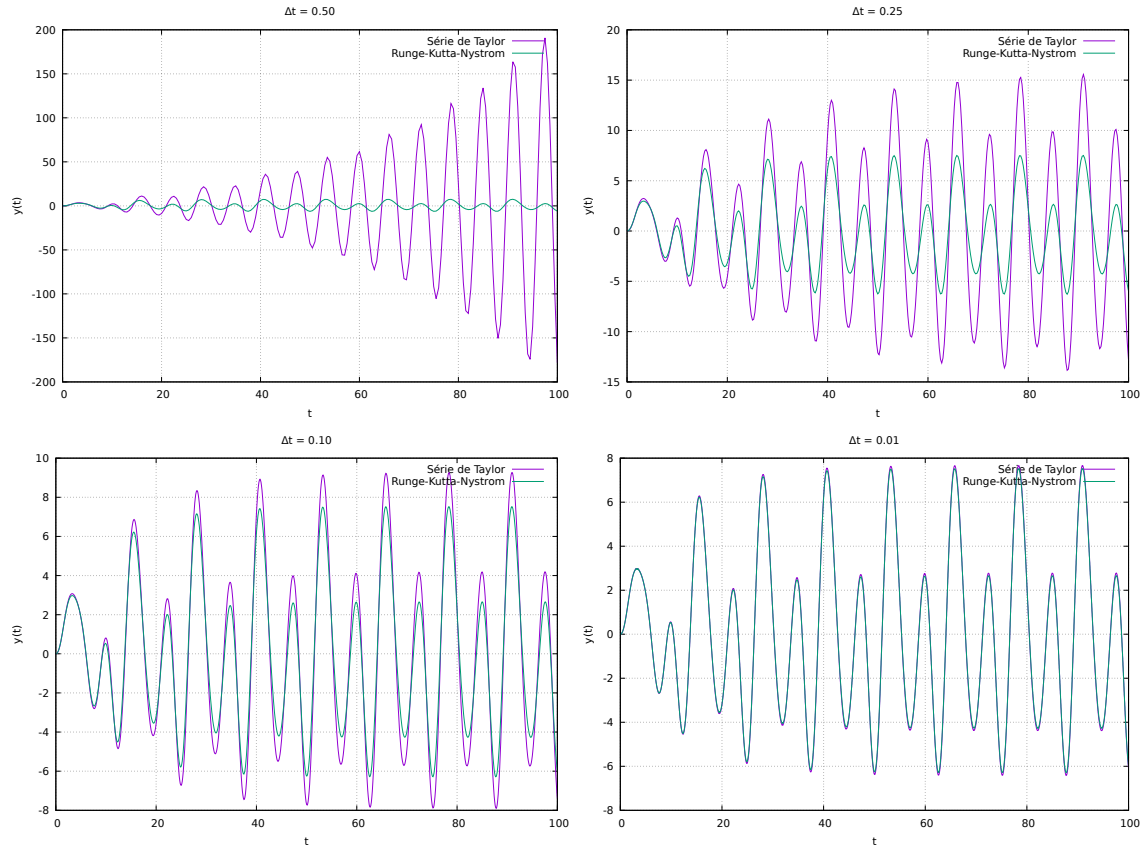
$$my''(t) + cy'(t) + ky(t) = F(t)$$
$$m = 1; c = 0.2; k = 1;$$
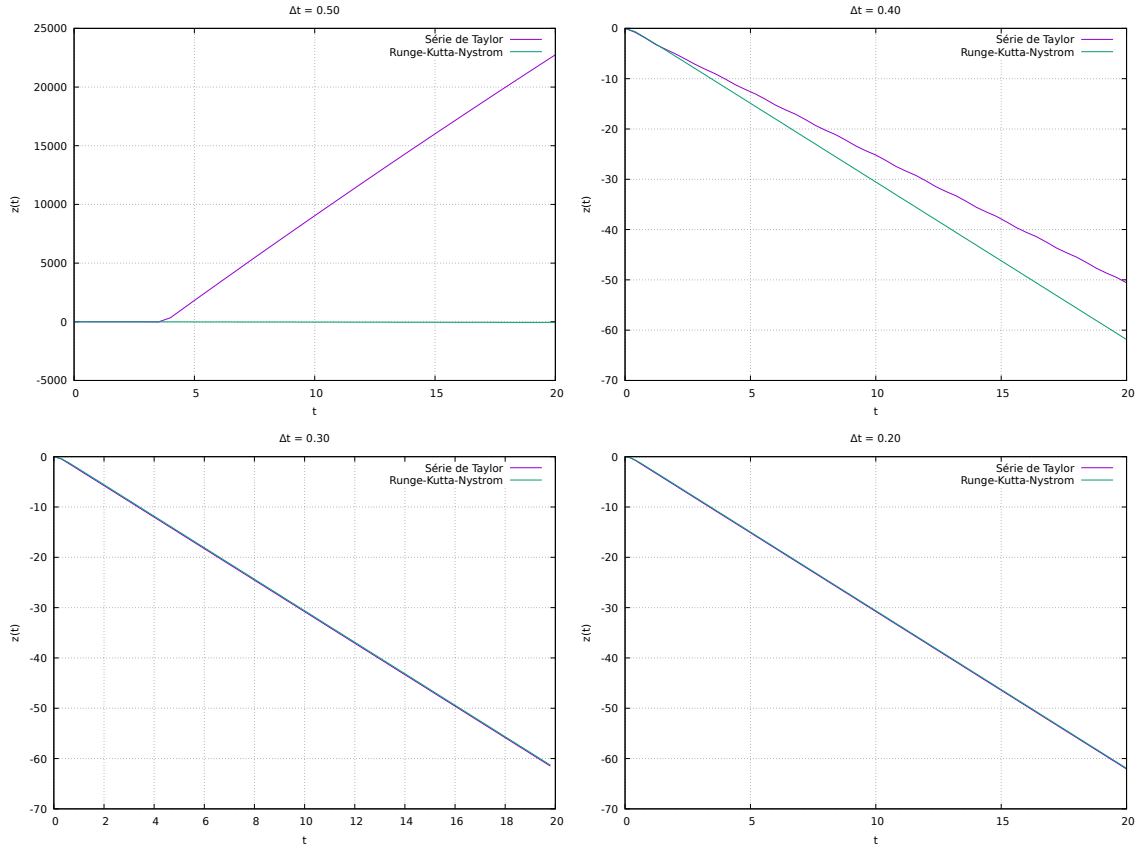$$F(t) = 2\sin(wt) + \sin(2wt) + \cos(3wt)$$
$$w = 0.5;$$
$$y'(0) = y(0) = 0.0$$

Questão 5.: Resolva a equação diferencial de segunda ordem apresentada em sala de aula pelos métodos 'Expansão em Série de *Taylor*' e *Runge-Kutta-Nystrom*, $0 \leq t \leq 20$, utilizando o programa desenvolvido na Tarefa 02:

$$z''(t) = -g - k \cdot z'(t) \cdot |z'(t)|$$
$$k = 1;$$
$$z'(0) = z(0) = 0$$

Como dito em aula, esta equação modela o deslocamento horizontal de um objeto abandonado em repouso sobre a superfície da água. É interessante perceber como a aceleração $z''(t)$ aumenta em módulo no sentido para baixo conforme a gravidade g, enquanto o termo $k \cdot z'(t) \cdot |z'(t)|$ representa o empuxo exercido pelo fluido: Uma vez que a velocidade é negativa (movimenta o objeto para o fundo), o sinal que antecede o produto torna este positivo, a fim de compensar o efeito da gravidade. A constante k se apresenta como o grau de viscosidade do meio.

Questão 6.: Extra: Queda livre na água.

Pensando na interpretação física do problema, achei que seria interessante visualizar uma situação onde o objeto é abandonado de um ponto acima do nível da água. Escolhendo arbitrariamente, supus $z(0) = 100$, enquanto mantive $z'(0) = 0$. Além disso, a equação diferencial ordinária de segunda ordem precisou ser reescrita em partes, para os diferentes meios:

$$z''(t) = \begin{cases} -g & \text{se } z \geq 0 \\ -g - k \cdot z'(t) \cdot |z'(t)| & \text{se } z < 0 \end{cases}$$

Isso, é claro, desprezando a resistência do ar. Os resultados foram os seguintes:
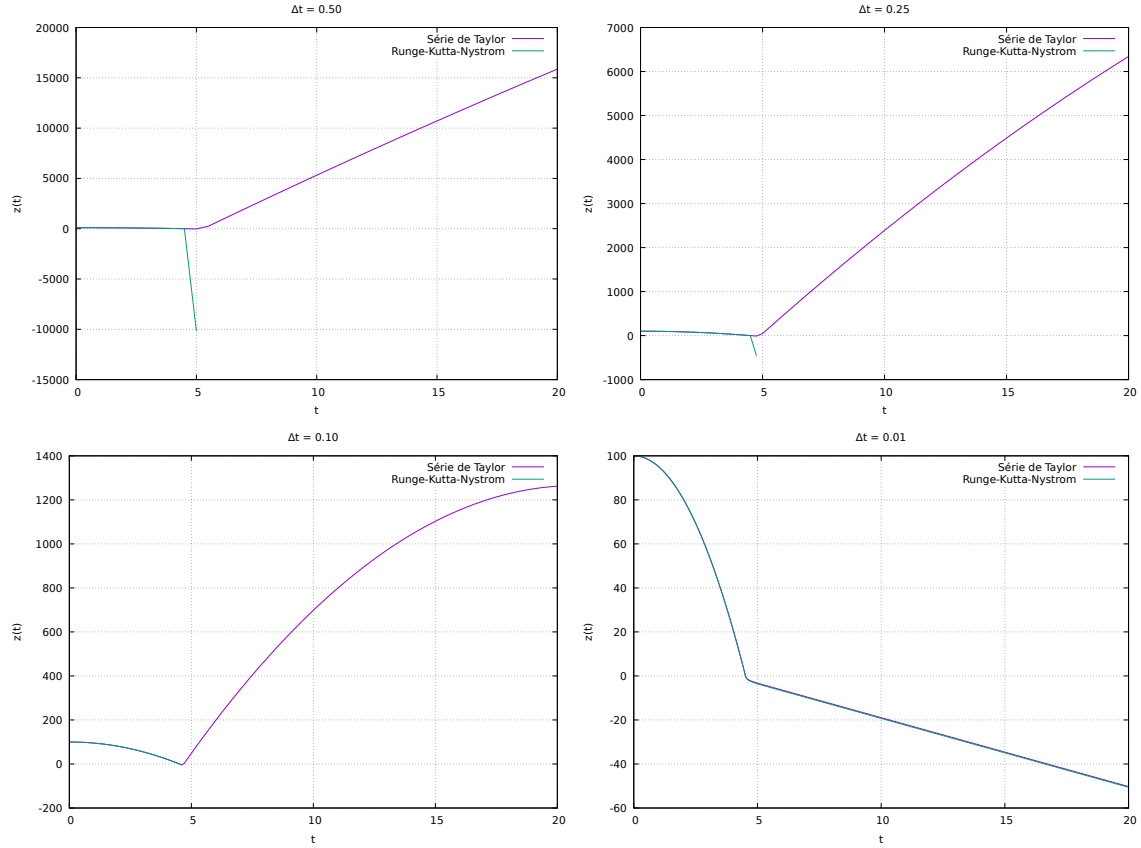
Como esperado, o gráfico da posição do objeto no tempo descreve uma parábola enquanto se encontra no ar e, após atravessar a linha d'água ($z = 0$), se comporta como no caso anterior. O que mais chamou minha atenção foi a divergência e instabilidade numérica para passos da ordem de $\Delta t = 10^{-1}$, faixa na qual os algoritmos se comportaram bem anteriormente.

Achei interessante ver nestes gráficos o "impacto" do objeto sobre a água, causado pela descontinuidade imposta sobre a segunda derivada $z''(t)$.

Appendices

Código - Programa Principal

```fortran
1   program main5
2       use Util
3       use Func
4       use Matrix
5       use Calc
6       use Plotlib
7       implicit none
8
9       double precision :: XMIN, XMAX, YMIN, YMAX
10
11  !   Command-line Args
12      integer :: argc
13
14  !    ENABLE_DEBUG = .TRUE.
15
16  !   Random seed definition
17      call init_random_seed()
18
19  !   Get Command-Line Args
20      argc = iargc()
21
22      if (argc == 0) then
23          goto 100
24      else
25          goto 11
26      end if
27
28  !   ====== Success ================================
29  10   call info(':: Sucesso ::')
30      goto 1
31  !   ====== Errors =================================
32  11   call error('Este programa não aceita parâmetross.')
33      goto 1
34  !   ====== Finish =================================
35  1    stop
36  !   ==============================================
37
38  100  goto 200
39
40  200  call Q2
41      goto 300
42
43  300  call Q3
44      goto 400
45
46  400  call Q4
47      goto 500
```

```fortran
48
49  500   call Q5
50        goto 600
51
52  600   call Q6
53        goto 700
54
55  700   call Q7
56        goto 800
57
58  800   call warn(ENDL//":: Complmento ::"//ENDL)
59        call QE1; call QE2; call QE3;
60        goto 10
61
62  !     ===============================
63
64        contains
65
66        subroutine Q2
67            implicit none
68            integer :: n = 10
69            double precision :: a, b, s
70
71            call info("2)"//ENDL//F6_NAME)
72            a = 0.0D0
73            b = 1.0D0
74            call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
75            call info(":: Integração Polinomial ::")
76            s = num_int(f6, a, b, n, kind="polynomial")
77            call blue("I1 = ∫f(x) dx ≈ "//DSTR(s))
78            call info(":: Quadratura de Gauss-Legendre ::")
79            s = num_int(f6, a, b, n, kind="gauss-legendre")
80            call blue("I1 = ∫f(x) dx ≈ "//DSTR(s))
81            call info(":: Método de Romberg ::")
82            s = num_int(f6, a, b, n, kind="romberg")
83            call blue("I1 = ∫f(x) dx ≈ "//DSTR(s))
84
85            a = 0.0D0
86            b = 5.0D0
87            call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
88            call info(":: Integração Polinomial ::")
89            s = num_int(f6, a, b, n, kind="polynomial")
90            call blue("I2 = ∫f(x) dx ≈ "//DSTR(s))
91            call info(":: Quadratura de Gauss-Legendre ::")
92            s = num_int(f6, a, b, n, kind="gauss-legendre")
93            call blue("I2 = ∫f(x) dx ≈ "//DSTR(s))
94            call info(":: Método de Romberg ::")
95            s = num_int(f6, a, b, n, kind="romberg")
96            call blue("I2 = ∫f(x) dx ≈ "//DSTR(s))
97
98        end subroutine
99
100       subroutine Q3
```

```fortran
            implicit none
            integer :: n
            double precision :: a, b, r
            double precision, dimension(INT_N) :: x
            double precision, dimension(4, INT_N) :: y

            type(StringArray), dimension(:), allocatable :: legend, with

            allocate(legend(4), with(4))

            legend(1)%str = 'Polinomial'
            legend(2)%str = 'Gauss-Legendre'
            legend(3)%str = 'Romberg'
            legend(4)%str = 'Adaptativo (Gauss)'

            with(1)%str = 'linespoints'
            with(2)%str = 'linespoints'
            with(3)%str = 'linespoints'
            with(4)%str = 'lines'

            a = 0.00D0
            b = 10.0D0

            call info(ENDL//"3)"//ENDL//F7_NAME)

            call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")

            INT_N = 128

            XMIN = 1.0D0
            XMAX = INT_N
            YMIN = -100.0D0
            YMAX = 300.0D0

            x = (/ (n, n=1, INT_N) /)

            call begin_plot(fname='L5-Q3')

            call subplots(2, 1)

            call info(ENDL//"m0 ~ "//F7a_NAME//ENDL)

            r = adapt_int(f7a, a, b, INT_N, tol=1.0D-8, kind="gauss-
                legendre")

            call info(":: Valor de referência (Integração Adaptativa)
                tol = 1E-8 ::")
            call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(r))

            do n = 1, INT_N
                y(:, n) = (/ &
                    num_int(f7a, a, b, n, kind="polynomial"), &
!
```

```fortran
152                     num_int(f7a, a, b, n, kind="gauss-legendre"), &
153 !
154                     num_int(f7a, a, b, n, kind="romberg"), &
155 !
156                     r &
157             /)
158         end do
159
160         call info(":: Integração Polinomial ::")
161         call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
162         call info(":: Quadratura de Gauss-Legendre ::")
163         call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
164         call info(":: Método de Romberg ::")
165         call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))
166
167         do n = 1, 4
168             call subplot(1, 1, x, y(n, :), INT_N)
169         end do
170
171         call subplot_config(1, 1, title='m0 = ∫Sσ(ω) dω ≈ '//DSTR(r)
                , xlabel='n', ylabel='m0', grid=.TRUE., &
172             legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=
                    YMIN, ymax=YMAX)
173
174         call info(ENDL//"m2 ~ "//F7b_NAME//ENDL)
175
176         r = adapt_int(f7b, a, b, INT_N, tol=1.0D-8, kind="gauss-
                legendre")
177
178         call info(":: Valor de referência (Integração Adaptativa)
                tol = 1E-8 ::")
179         call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(r))
180
181         do n = 1, INT_N
182             y(:, n) = (/ &
183                 num_int(f7b, a, b, n, kind="polynomial"), &
184 !
185                 num_int(f7b, a, b, n, kind="gauss-legendre"), &
186 !
187                 num_int(f7b, a, b, n, kind="romberg"), &
188 !
189                     r &
190             /)
191         end do
192
193         call info(":: Integração Polinomial ::")
194         call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
195         call info(":: Quadratura de Gauss-Legendre ::")
196         call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
197         call info(":: Método de Romberg ::")
198         call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))
199
200         do n = 1, 4
```

```fortran
201                 call subplot(2, 1, x, y(n, :), INT_N)
202             end do
203
204             call subplot_config(2, 1, title='m2 = ∫ω² Sσ(ω) dω ≈ '//DSTR
                    (r), xlabel='n', ylabel='m2', grid=.TRUE., &
205                 legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=
                        YMIN, ymax=YMAX)
206
207             call render_plot(clean=.TRUE.)
208         end subroutine
209
210         subroutine Q4
211             implicit none
212             integer :: n
213             double precision :: a, b, r
214             double precision, dimension(INT_N) :: x
215             double precision, dimension(4, INT_N) :: y
216
217             type(StringArray), dimension(:), allocatable :: legend, with
218
219             allocate(legend(4), with(4))
220
221             legend(1)%str = 'Polinomial'
222             legend(2)%str = 'Gauss-Legendre'
223             legend(3)%str = 'Romberg'
224             legend(4)%str = 'Adaptativo (Gauss)'
225
226             with(1)%str = 'linespoints'
227             with(2)%str = 'linespoints'
228             with(3)%str = 'linespoints'
229             with(4)%str = 'lines'
230
231             a = 0.00D0
232             b = 10.0D0
233
234             call info(ENDL//"4)"//ENDL//F8_NAME)
235
236             call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
237
238             INT_N = 128
239
240             XMIN = 1.0D0
241             XMAX = INT_N
242             YMIN = -10.0D0
243             YMAX = 100.0D0
244
245             x = (/ (n, n=1, INT_N) /)
246
247             call begin_plot(fname='L5-Q4')
248
249             call subplots(2, 1)
250
251             call info(ENDL//"m0 ~ "//F8a_NAME//ENDL)
```

```fortran
252
253             r = adapt_int(f8a, a, b, INT_N, tol=1.0D-8, kind="gauss-
                    legendre")
254
255             call info(":: Valor de referência (Integração Adaptativa)
                    tol = 1E-8 ::")
256             call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(r))
257
258             do n = 1, INT_N
259                 y(:, n) = (/ &
260                     num_int(f8a, a, b, n, kind="polynomial"), &
261 !
262                     num_int(f8a, a, b, n, kind="gauss-legendre"), &
263 !
264                     num_int(f8a, a, b, n, kind="romberg"), &
265 !
266                     r &
267                 /)
268             end do
269
270             call info(":: Integração Polinomial ::")
271             call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
272             call info(":: Quadratura de Gauss-Legendre ::")
273             call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
274             call info(":: Método de Romberg ::")
275             call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))
276
277             do n = 1, 4
278                 call subplot(1, 1, x, y(n, :), INT_N)
279             end do
280
281             call subplot_config(1, 1, title='m0 = ∫Sσ(ω) dω ≈ '//DSTR(r)
                    , xlabel='n', ylabel='m0', grid=.TRUE., &
282                 legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=
                        YMIN, ymax=YMAX)
283
284             call info(ENDL//"m2 ~ "//F8b_NAME//ENDL)
285
286             r = adapt_int(f8b, a, b, INT_N, tol=1.0D-8, kind="gauss-
                    legendre")
287
288             call info(":: Valor de referência (Integração Adaptativa)
                    tol = 1E-8 ::")
289             call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(r))
290
291             do n = 1, INT_N
292                 y(:, n) = (/ &
293                     num_int(f8b, a, b, n, kind="polynomial"), &
294 !
295                     num_int(f8b, a, b, n, kind="gauss-legendre"), &
296 !
297                     num_int(f8b, a, b, n, kind="romberg"), &
298 !
```

```fortran
299                      r &
300                )
301          end do
302
303          call info(":: Integração Polinomial ::")
304          call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
305          call info(":: Quadratura de Gauss-Legendre ::")
306          call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
307          call info(":: Método de Romberg ::")
308          call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))
309
310          do n = 1, 4
311              call subplot(2, 1, x, y(n, :), INT_N)
312          end do
313
314          call subplot_config(2, 1, title='m2 = ∫ω² Sσ(ω) dω ≈ '//DSTR
                  (r), xlabel='n', ylabel='m2', grid=.TRUE., &
315              legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=
                      YMIN, ymax=YMAX)
316
317          call render_plot(clean=.TRUE.)
318      end subroutine
319
320      subroutine Q5
321          implicit none
322          integer :: n
323          double precision :: a, b, s
324
325          call info(ENDL//"5) "//F9_NAME)
326          a = 0.0D0
327          b = 4.0D0
328          call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
329
330          n = 4
331          call info(":: Integração Polinomial ::")
332          call blue('n = '//STR(n))
333          s = num_int(f9, a, b, n, kind="polynomial")
334          call blue("A = ∫f(x) dx ≈ "//DSTR(s))
335
336          n = 2
337          call info(":: Quadratura de Gauss-Legendre ::")
338          call blue('n = '//STR(n))
339          s = num_int(f9, a, b, n, kind="gauss-legendre")
340          call blue("A = ∫f(x) dx ≈ "//DSTR(s))
341      end subroutine
342
343      subroutine Q6
344          implicit none
345          integer :: n
346          double precision :: a, b, s
347
348          n = 10
349
```

```fortran
350            call blue('n = '//STR(n))
351
352            call info(ENDL//"6) "//F10_NAME)
353            a = 0.0D0
354            b = 3.0D0
355            call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
356            call info(":: Integração Polinomial ::")
357            s = num_int(f10, a, b, n, kind="polynomial")
358            call blue("A = ∫f(x) dx ≈ "//DSTR(s))
359            call info(":: Quadratura de Gauss-Legendre ::")
360            s = num_int(f10, a, b, n, kind="gauss-legendre")
361            call blue("A = ∫f(x) dx ≈ "//DSTR(s))
362            call info(":: Método de Romberg ::")
363            s = num_int(f10, a, b, n, kind="romberg")
364            call blue("A = ∫f(x) dx ≈ "//DSTR(s))
365        end subroutine
366
367        subroutine Q7
368            implicit none
369            integer :: n
370            double precision :: a, b, r, s
371
372            call info(ENDL//"7)")
373
374            n = 10
375
376            call blue('n = '//STR(n))
377
378            call info("A1 ~ "//F11_NAME)
379            a = DNINF
380            b = 1.0D0
381            call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
382            call info(":: Quadratura de Gauss-Hermite e de Gauss-
                  Legendre ::")
383            r = num_int(f11a, a, -a, n, kind="gauss-hermite")
384            s = num_int(f11b, -b, b, n, kind="gauss-legendre")
385            call blue("A1 = ∫f(x) dx ≈ "//DSTR(r+s))
386
387            call info("A2 ~ "//F12_NAME)
388            a = DNINF
389            b = DINF
390            call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
391            call info(":: Quadratura de Gauss-Hermite ::")
392            s = num_int(f12, a, b, n, kind="gauss-hermite")
393            call blue("A2 = ∫f(x) dx ≈ "//DSTR(s))
394        end subroutine
395
396        subroutine QE1
397            implicit none
398            double precision :: x, y, dy
399
400            x = 3.0D0
401
```

```fortran
        call info(ENDL//')')

        call info(FL5_QE1_NAME)
        call info(DFL5_QE1_NAME)

        call info(':: Derivada Analítica ::')
        dy = DFL5_QE1(x)
        call blue("f'("//DSTR(x)//") = "//DSTR(dy))

        call info(":: Diferenças Finitas ::"//ENDL)

        call info(':: Diferença Central (Δx = 1E-2)::')
        y = d(FL5_QE1, x, dx=1.0D-2, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2)::')
        y = d(FL5_QE1, x, dx=1.0D-2, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2)::')
        y = d(FL5_QE1, x, dx=1.0D-2, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(ENDL//":: Extrapolação de Richard ::")

        call info(':: Diferença Central (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE1, x, dx=1.0D-2, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Diferença Central (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE1, x, dx=1.0D-2, p=2.0D0, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE1, x, dx=1.0D-2, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE1, x, dx=1.0D-2, p=2.0D0, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE1, x, dx=1.0D-2, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))
```

```fortran
455        call info(':: Passo atrás (Δx = 1E-2, p = 2)::')
456        y = richard(FL5_QE1, x, dx=1.0D-2, p=2.0D0, kind='backward')
457        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
458        call blue("|δy| = "//DSTR(DABS(y - dy)))
459
460    end subroutine
461
462    subroutine QE2
463        implicit none
464        double precision :: x, y, dy
465
466        x = 2.0D0
467
468        call info(ENDL//'2)')
469
470        call info(FL5_QE2_NAME)
471        call info(DFL5_QE2_NAME)
472
473        call info(':: Derivada Analítica ::')
474        dy = DFL5_QE2(x)
475        call blue("f'("//DSTR(x)//") = "//DSTR(dy))
476
477        call info(":: Diferenças Finitas ::"//ENDL)
478
479        call info(':: Diferença Central (Δx = 1E-2)::')
480        y = d(FL5_QE2, x, dx=1.0D-2, kind='central')
481        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
482        call blue("|δy| = "//DSTR(DABS(y - dy)))
483
484        call info(':: Passo à frente (Δx = 1E-2)::')
485        y = d(FL5_QE2, x, dx=1.0D-2, kind='forward')
486        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
487        call blue("|δy| = "//DSTR(DABS(y - dy)))
488
489        call info(':: Passo atrás (Δx = 1E-2)::')
490        y = d(FL5_QE2, x, dx=1.0D-2, kind='backward')
491        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
492        call blue("|δy| = "//DSTR(DABS(y - dy)))
493
494        call info(ENDL//":: Extrapolação de Richard ::")
495
496        call info(':: Diferença Central (Δx = 1E-2, p = 1)::')
497        y = richard(FL5_QE2, x, dx=1.0D-2, kind='central')
498        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
499        call blue("|δy| = "//DSTR(DABS(y - dy)))
500
501        call info(':: Diferença Central (Δx = 1E-2, p = 2)::')
502        y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='central')
503        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
504        call blue("|δy| = "//DSTR(DABS(y - dy)))
505
506        call info(':: Passo à frente (Δx = 1E-2, p = 1)::')
507        y = richard(FL5_QE2, x, dx=1.0D-2, kind='forward')
```

```fortran
508        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
509        call blue("|δy| = "//DSTR(DABS(y - dy)))
510
511        call info(':: Passo à frente (Δx = 1E-2, p = 2)::')
512        y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='forward')
513        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
514        call blue("|δy| = "//DSTR(DABS(y - dy)))
515
516        call info(':: Passo atrás (Δx = 1E-2, p = 1)::')
517        y = richard(FL5_QE2, x, dx=1.0D-2, kind='backward')
518        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
519        call blue("|δy| = "//DSTR(DABS(y - dy)))
520
521        call info(':: Passo atrás (Δx = 1E-2, p = 2)::')
522        y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='backward')
523        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
524        call blue("|δy| = "//DSTR(DABS(y - dy)))
525    end subroutine
526
527    subroutine QE3
528        implicit none
529        double precision :: x, y, dy
530
531        x = 6.0D0
532
533        call info(ENDL//'3)')
534
535        call info(FL5_QE3_NAME)
536        call info(DFL5_QE3_NAME)
537
538        call info(':: Derivada Analítica ::')
539        dy = DFL5_QE3(x)
540        call blue("f'("//DSTR(x)//") = "//DSTR(dy))
541
542        call info(":: Diferenças Finitas ::"//ENDL)
543
544        call info(':: Diferença Central (Δx = 1E-2)::')
545        y = d(FL5_QE3, x, dx=1.0D-2, kind='central')
546        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
547        call blue("|δy| = "//DSTR(DABS(y - dy)))
548
549        call info(':: Passo à frente (Δx = 1E-2)::')
550        y = d(FL5_QE3, x, dx=1.0D-2, kind='forward')
551        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
552        call blue("|δy| = "//DSTR(DABS(y - dy)))
553
554        call info(':: Passo atrás (Δx = 1E-2)::')
555        y = d(FL5_QE3, x, dx=1.0D-2, kind='backward')
556        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
557        call blue("|δy| = "//DSTR(DABS(y - dy)))
558
559        call info(ENDL//":: Extrapolação de Richard ::")
560
```

```fortran
561              call info(':: Diferença Central (Δx = 1E-2, p = 1)::')
562              y = richard(FL5_QE3, x, dx=1.0D-2, kind='central')
563              call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
564              call blue("|δy| = "//DSTR(DABS(y - dy)))
565
566              call info(':: Diferença Central (Δx = 1E-2, p = 2)::')
567              y = richard(FL5_QE3, x, dx=1.0D-2, p=2.0D0, kind='central')
568              call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
569              call blue("|δy| = "//DSTR(DABS(y - dy)))
570
571              call info(':: Passo à frente (Δx = 1E-2, p = 1)::')
572              y = richard(FL5_QE3, x, dx=1.0D-2, kind='forward')
573              call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
574              call blue("|δy| = "//DSTR(DABS(y - dy)))
575
576              call info(':: Passo à frente (Δx = 1E-2, p = 2)::')
577              y = richard(FL5_QE3, x, dx=1.0D-2, p=2.0D0, kind='forward')
578              call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
579              call blue("|δy| = "//DSTR(DABS(y - dy)))
580
581              call info(':: Passo atrás (Δx = 1E-2, p = 1)::')
582              y = richard(FL5_QE3, x, dx=1.0D-2, kind='backward')
583              call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
584              call blue("|δy| = "//DSTR(DABS(y - dy)))
585
586              call info(':: Passo atrás (Δx = 1E-2, p = 2)::')
587              y = richard(FL5_QE3, x, dx=1.0D-2, p=2.0D0, kind='backward')
588              call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
589              call blue("|δy| = "//DSTR(DABS(y - dy)))
590          end subroutine
591 end program main5
```

Código - Definição das Funções

```fortran
1  !       Func Module
2
3      module Func
4          use Util
5          implicit none
6
7  !          >> F1 <<
8          character (len = *), parameter :: F1_NAME = "f(x) = log(cosh
               (x * √(g * j))) - 50"
9          double precision :: F1_G = 9.80600D0
10         double precision :: F1_K = 0.00341D0
11
12 !          >> F2 <<
13         character (len = *), parameter :: F2_NAME = "f(x) = 4 * cos(
               x) - exp(2 * x)"
14
15 !          >> F3 <<
```

```fortran
16          character (len = *), parameter :: F3_NAME = "f(x, y, z) :="
                //ENDL// &
17          "16x^4 + 16y^4 + z^4 = 16"//ENDL// &
18          "x^2 + y^2 + x^2 = 3"//ENDL// &
19          "x^3 - y + z = 1"
20          integer :: F3_N = 3
21
22 !        >> F4 <<
23          character (len = *), parameter :: F4_NAME = "f(c2, c3, c4)
                :="//ENDL// &
24          "c2^2 + 2 c3^2 + 6 c4^2 = 1"//ENDL// &
25          "8 c3^3 + 6 c3 c2^2 + 36 c3 c2 c4 + 108 c3 c4^4 = θ1"//ENDL// &
26          "60 * c3^4 + 60 * c3^2 * c2^2 + 576 * c3^2 * c2 * c4 + "// &
27          "2232 * c3^2 * c4^2 + 252 * c4^2 * c2^2 + "// &
28          "1296 * c4^3 c2 + 3348 c4^4 + 24 c2^3 c4 + 3 c2 = θ2"
29          double precision :: F4_TT1(3) = (/ 0.0D0, 0.75D0,  0.000D0
                /)
30          double precision :: F4_TT2(3) = (/ 3.0D0, 6.50D0, 11.667D0
                /)
31          double precision :: F4_T1 = 0.0D0
32          double precision :: F4_T2 = 0.0D0
33          integer :: F4_N = 3
34
35 !        >> F5 <<
36          character (len = *), parameter :: F5_NAME = "f(x) = b1 + b2
                x^b3"
37          integer :: F5_N = 3
38
39 !        >> F6 <<
40          character (len = *), parameter :: F6_NAME = "f(x) = exp(-x^2
                /2) / √(2 π)"
41
42 !        >> F7 <<
43          character (len = *), parameter :: F7_NAME = "Sσ(ω) = RAO(ω)^2
                Sη(ω)"//ENDL//TAB// &
44          "RAO(ω) = 1 / √((1 - (ω/ωn)^2)^2 + (2ξω/ωn)^2)"
45
46          character (len = *), parameter :: F7a_NAME = "Sη(ω) = 2"
47          character (len = *), parameter :: F7b_NAME = "Sη(ω) = 2"
48
49 !        >> F8 <<
50          character (len = *), parameter :: F8_NAME = "Sσ(ω) = RAO(ω)^2
                Sη(ω)"//ENDL//TAB// &
51          "RAO(ω) = 1 / √((1 - (ω/ωn)^2)^2 + (2ξω/ωn)^2)"
52
53          character (len = *), parameter :: F8a_NAME = "Sη(ω) = ((4 π^3
                Hs^2) / (ω^5 Tz^4)) exp(-(16 π^3) / (ω^4 Tz^4))"
54          character (len = *), parameter :: F8b_NAME = "Sη(ω) = ((4 π^3
                Hs^2) / (ω^5 Tz^4)) exp(-(16 π^3) / (ω^4 Tz^4))"
55
56
57 !        >> F13 <<
```

```fortran
58          character (len = *), parameter :: F13_NAME = "y'(t) = -2 t y
                (t)^2 "//ENDL//"y(0) = 1 "
59          double precision :: F13_A = 0.0D0
60          double precision :: F13_B = 10.0D0
61          double precision :: F13_Y0 = 1.0D0
62
63 !        >> F14 <<
64          character (len = *), parameter :: F14_NAME = "m y''(t) + c y
                '(t) + k y(t) = F(t)"//ENDL// &
65          "m = 1; c = 0.2; k = 1;"//ENDL// &
66          "F(t) = 2 sin(w t) + sin(2 w t) + cos(3 w t)"//ENDL// &
67          "w = 0.5;"//ENDL// &
68          "y'(0) = 0; y(0) = 0;"
69          double precision :: F14_M = 1.0D0
70          double precision :: F14_C = 0.2D0
71          double precision :: F14_K = 1.0D0
72          double precision :: F14_W = 0.5D0
73          double precision :: F14_Y0 = 0.0D0
74          double precision :: F14_DY0 = 0.0D0
75          double precision :: F14_A = 0.0D0
76          double precision :: F14_B = 100.0D0
77
78 !        >> F15 <<
79          character (len = *), parameter :: F15_NAME = "z''(t) = -g -k
                z'(t) |z'(t)|"//ENDL// &
80          "z'(0) = 0; z(0) = 0;"//ENDL// &
81          "g = 9.806; k = 1;"
82          double precision :: F15_G = 9.80600D0
83          double precision :: F15_KD = 1.0D0
84          double precision :: F15_BY0 = 100.0D0
85          double precision :: F15_Y0 = 0.0D0
86          double precision :: F15_DY0 = 0.0D0
87          double precision :: F15_A = 0.0D0
88          double precision :: F15_B = 20.0D0
89
90
91
92          double precision :: t1 = 0.0D0
93          double precision :: t2 = 0.0D0
94
95          double precision :: wn = 1.00D0
96          double precision :: xi = 0.05D0
97          double precision :: Hs = 3.0D0
98          double precision :: Tz = 5.0D0
99
100         character (len = *), parameter :: F9_NAME = "f(x) = 2 + 2x -
                x^2 + 3x^3 "
101
102         character (len = *), parameter :: F10_NAME = "f(x) = 1 / (1
                + x^2 )"
103
104         character (len = *), parameter :: F11_NAME = "f(x) = exp(- x
                ^2/2) / \sqrt(2 \pi )"
```

```fortran
        character (len = *), parameter :: F12_NAME = "f(x) = x² exp
            (- x²/2) / √(2 π)"

!          >> L5-QE <<
        character (len = *), parameter :: FL5_QE1_NAME = 'f(x) = x³
            + exp(-x)'
        character (len = *), parameter :: DFL5_QE1_NAME = "f'(x) = 3
            x² - exp(-x)"
        character (len = *), parameter :: FL5_QE2_NAME = 'f(x) = ³√x
            + log(x)'
        character (len = *), parameter :: DFL5_QE2_NAME = "f'(x) = 1
            / (3 ³√x²) + (1 / x)"
        character (len = *), parameter :: FL5_QE3_NAME = 'f(x) = 1 -
            exp(-x² / 25)'
        character (len = *), parameter :: DFL5_QE3_NAME = "f'(x) =
            (2 x / 25) exp(-x² / 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

    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
```

```fortran
      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)
!         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
```

```fortran
202          end function
203
204  !       ========== Derivative ===========
205          function df3(x, n) result (J)
206              implicit none
207              integer :: n
208              double precision, dimension(n) :: x
209              double precision, dimension(n, n) :: J
210
211              J(1, :) = (/ 64 * x(1) ** 3, 64 * x(2) ** 3, 4 * x(3) ** 3 &
                     /)
212              J(2, :) = (/  2 * x(1)      ,  2 * x(2)      ,  2 * x(3) &
                     /)
213              J(3, :) = (/  3 * x(1) ** 2,           -1.0D0,          1.0D0 &
                     /)
214              return
215          end function
216
217  !       ================== Another function ====================
218          function f4(x, n) result (y)
219              implicit none
220              integer :: n
221              double precision, dimension(n) :: x, y
222              y = (/ &
223                  x(1)**2+2*x(2)**2+6*x(3)**2, &
224                  2*x(2)*(3*x(1)**2+4*x(2)**2+18*x(1)*x(3)+54*x(3)**4), &
225                  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)+&
226                  12*(7*x(1)**2+62*x(2)**2)*x(3)**2+432*x(1)*x(3)**3+1116* &
                         x(3)**4)&
227                  /) - (/ 1.0D0, F4_T1, F4_T2 /)
228              return
229          end function
230
231  !       ========== Derivatives ===========
232          function df4(x, n) result (J)
233  !           $R^3$ -> $R^3$ x3 (n == 3)
234              implicit none
235              integer :: n
236              double precision :: x(n), J(n, n)
237
238              J(1, :) = (/ &
239                  2*x(1), &
240                  4*x(2), &
241                  12*x(3) &
242                  /)
243              J(2, :) = (/ &
244                  12*x(1)*x(2)+36*x(2)*x(3),                          &
245                  6*x(1)**2+24*x(2)**2+36*x(1)*x(3)+108*x(3)**4, &
246                  36*x(1)*x(2)+432*x(2)*x(3)**3                     &
247                  /)
248              J(3, :) = (/ &
```

```fortran
                        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)
            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
```

```fortran
298
299     function Sn1(w) result (z)
300         implicit none
301         double precision :: w, z
302         z = 2.0D0
303         return
304     end function
305
306     function Sn2(w) result (z)
307         implicit none
308         double precision :: w, z
309         z = (4 * (Hs**2) * (PI**3)) / ( DEXP( (16 * (PI**3))/((Tz*w)
                **4) ) * (Tz**4) * (w**5) )
310         return
311     end function
312
313     function Ss(w, Sn) result (z)
314         implicit none
315         double precision :: w, z
316         interface
317             function Sn(w) result (z)
318                 implicit none
319                 double precision :: w, z
320             end function
321         end interface
322         z = (RAO(w) ** 2) * Sn(w)
323         return
324     end function
325
326     function f7a(w) result (z)
327         implicit none
328         double precision :: w, z
329         z = Ss(w, Sn1)
330         return
331     end function
332
333     function f7b(w) result (z)
334         implicit none
335         double precision :: w, z
336         z = (w ** 2) * Ss(w, Sn1)
337         return
338     end function
339
340     function f8a(w) result (z)
341         implicit none
342         double precision :: w, z
343         z = Ss(w, Sn2)
344         return
345     end function
346
347     function f8b(w) result (z)
348         implicit none
349         double precision :: w, z
```

```fortran
350            z = (w ** 2) * Ss(w, Sn2)
351            return
352        end function
353
354    !    ========== Function 9 ==============
355        function f9(x) result (y)
356            implicit none
357            double precision :: x, y
358            y = 2.0D0 + 2.0D0 * x - x ** 2 + 3.0D0 * x ** 3
359            return
360        end function
361
362    !    ========== Function 10 ==============
363        function f10(x) result (y)
364            implicit none
365            double precision :: x, y
366            y = 1.0D0 / (1.0D0 + x ** 2)
367            return
368        end function
369
370    !    ========== Function 11 ==============
371        function f11a(x) result (y)
372            implicit none
373            double precision :: x, y
374            y = DEXP((x ** 2) / 2) / DSQRT(8 * PI)
375            return
376        end function
377
378        function f11b(x) result (y)
379            implicit none
380            double precision :: x, y
381            y = DEXP(-(x ** 2) / 2) / DSQRT(8 * PI)
382            return
383        end function
384
385    !    ========== Function 12 ==============
386        function f12(x) result (y)
387            implicit none
388            double precision :: x, y
389            y = (x ** 2) * DEXP((x ** 2) / 2) / DSQRT(2 * PI)
390            return
391        end function
392
393    !    ========== Function 13 ==============
394        function df13(t, y) result (u)
395            implicit none
396            double precision :: t, y, u
397            u = - 2 * t * (y ** 2)
398            return
399        end function
400
401        function f13(t) result (y)
402            implicit none
```

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

Código - Métodos Numéricos

```fortran
1  !   Calc Module
2
3      module Calc
4          use Util
5          use Matrix
6          implicit none
7          integer :: INT_N = 128
8          double precision :: h = 1.0D-5
9          !double precision :: D_TOL = 1.0D-5
10
11          character (len=*), parameter :: GAUSS_LEGENDRE_QUAD = "
                  quadratures/gauss-legendre/gauss-legendre"
12          character (len=*), parameter :: GAUSS_HERMITE_QUAD = "
                  quadratures/gauss-hermite/gauss-hermite"
```

```fortran
13          contains
14      !            ================= Numerical Mathods =================
15          function d(f, x, dx, kind) result (y)
16              implicit none
17              character (len=*), optional :: kind
18              double precision, optional :: dx
19              character (len=:), allocatable :: t_kind
20              double precision :: x, y, t_dx
21
22              interface
23                  function f(x) result (y)
24                      implicit none
25                      double precision :: x, y
26                  end function
27              end interface
28
29              if (.NOT. PRESENT(dx)) then
30                  t_dx = h
31              else
32                  t_dx = dx
33              end if
34
35              if (.NOT. PRESENT(kind)) then
36                  t_kind = "central"
37              else
38                  t_kind = kind
39              end if
40
41              if (t_kind == "central") then
42                  y = (f(x + t_dx) - f(x - t_dx)) / (2 * t_dx)
43              else if (t_kind == "forward") then
44                  y = (f(x + t_dx) - f(x)) / t_dx
45              else if (t_kind == "backward") then
46                  y = (f(x) - f(x - t_dx)) / t_dx
47              else
48                  call error("Unexpected value '"//t_kind//"' for
                      derivative kind."// &
49                  "Options are: 'central', 'forward' and 'backward'.")
50              end if
51              return
52          end function
53
54          function dp(f, x, i, n) result (y)
55              implicit none
56              integer :: i, n
57              double precision :: f
58              double precision :: x(n), xh(n)
59              double precision :: y
60
61              xh(:) = 0.0D0
62              xh(i) = h
63
64              y = (f(x + xh) - f(x - xh)) / (2 * h)
```

```fortran
                    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

            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
```

34

```fortran
118
119             if (.NOT. PRESENT(tol)) then
120                 t_tol = D_TOL
121             else
122                 t_tol = tol
123             end if
124
125             if (bb < aa) then
126                 a = bb
127                 b = aa
128             else
129                 a = aa
130                 b = bb
131             end if
132
133             do while (DABS(a - b) > t_tol)
134                 x = (a + b) / 2
135                 if (f(a) > f(b)) then
136                     if (f(x) > 0) then
137                         a = x
138                     else
139                         b = x
140                     end if
141                 else
142                     if (f(x) < 0) then
143                         a = x
144                     else
145                         b = x
146                     end if
147                 end if
148             end do
149             x = (a + b) / 2
150             return
151         end function
152
153         function newton(f, df, x0, ok, tol, max_iter) result (x)
154             implicit none
155             integer :: k, t_max_iter
156             integer, optional :: max_iter
157             double precision, intent(in) :: x0
158             double precision :: x, xk, t_tol
159             double precision, optional :: tol
160             logical, intent(out) :: ok
161
162             interface
163                 function f(x) result (y)
164                     double precision :: x, y
165                 end function
166             end interface
167
168             interface
169                 function df(x) result (y)
170                     double precision :: x, y
```

```fortran
                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
```

```fortran
            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)

            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
```

```fortran
274                end if
275
276                if (.NOT. PRESENT(tol)) then
277                    t_tol = D_TOL
278                else
279                    t_tol = tol
280                end if
281
282            x0(:) = x00(:)
283            xk = 1.0D+308
284
285            ok = .TRUE.
286
287            do k = 1, t_max_iter
288                call cross_sort(x0, y0, 3)
289
290 !              Cálculo de y
291                do i = 1, 3
292                    y0(i) = f(x0(i))
293                end do
294
295                x = lagrange(y0, x0, 3, 0.0D0)
296
297                if (DABS(x - xk) > t_tol) then
298                    j(:) = MAXLOC(DABS(y0))
299                    i = j(1)
300                    x0(i) = x
301                    y0(i) = f(x)
302                    xk = x
303                else
304                    if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
                             then
305                        ok = .FALSE.
306                    end if
307                    return
308                end if
309            end do
310            ok = .FALSE.
311            return
312        end function
313
314        function sys_newton(ff, dff, x0, n, ok, tol, max_iter)
                 result (x)
315            implicit none
316            logical, intent(out) :: ok
317            integer :: n, k, t_max_iter
318            integer, optional :: max_iter
319            double precision, dimension(n), intent(in) :: x0
320            double precision, dimension(n) :: x, xdx, dx
321            double precision, dimension(n, n) :: J
322            double precision :: t_tol
323            double precision, optional :: tol
324
```

```fortran
            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
```

```fortran
        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
```

```fortran
426                if (.NOT. ok) then
427                    exit
428                else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
429                    x = xdx
430                else
431                    if (VEDGE(x)) then
432                        ok = .FALSE.
433                    end if
434                    return
435                end if
436            end do
437            ok = .FALSE.
438            return
439        end function
440
441        function sys_broyden(ff, x0, B0, n, ok, tol, max_iter)
                 result (x)
442            implicit none
443            logical, intent(out) :: ok
444            integer :: n, k, t_max_iter
445            integer, optional :: max_iter
446            double precision, dimension(n), intent(in) :: x0
447            double precision, dimension(n, n), intent(in) :: B0
448            double precision, dimension(n) :: x, xdx, dx, dff
449            double precision, dimension(n, n) :: J
450            double precision :: t_tol
451            double precision, optional :: tol
452
453            interface
454                function ff(x, n) result (y)
455                    implicit none
456                    integer :: n
457                    double precision :: x(n), y(n)
458                end function
459            end interface
460
461            if (.NOT. PRESENT(max_iter)) then
462                t_max_iter = D_MAX_ITER
463            else
464                t_max_iter = max_iter
465            end if
466
467            if (.NOT. PRESENT(tol)) then
468                t_tol = D_TOL
469            else
470                t_tol = tol
471            end if
472
473            ok = .TRUE.
474
475            x = x0
476            J = B0
477
```

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

```fortran
            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)
!           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
```

```fortran
            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

                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
```

```fortran
626                  end do
627                  ok = .FALSE.
628                  return
629              end function
630
631 !         ============ Numerical Integration ========
632          subroutine load_quad(x, w, k, fname)
633 !             Load Quadrature
634              implicit none
635              integer :: k, m, n
636              character (len=*) :: fname
637              double precision, dimension(k) :: x, w
638              double precision, dimension(:, :), allocatable :: xw
639              call read_matrix(fname, xw, m, n)
640              if (n /= 2 .OR. m /= k) then
641                  call error("Invalid Matrix dimensions.")
642                  stop "ERROR"
643              end if
644              x(:) = xw(:, 1)
645              w(:) = xw(:, 2)
646              deallocate(xw)
647          end subroutine
648
649          function num_int(f, a, b, n, kind) result (s)
650              implicit none
651              integer :: n
652              character (len=*), optional :: kind
653              double precision :: a, b, s
654              interface
655                  function f(x) result (y)
656                      double precision :: x, y
657                  end function
658              end interface
659
660              if (.NOT. PRESENT(kind)) then
661                  kind = "polynomial"
662              end if
663
664              if (kind == "polynomial") then
665                  s = polynomial_int(f, a, b, n)
666              else if (kind == "gauss-legendre") then
667                  s = gauss_legendre_int(f, a, b, n)
668              else if (kind == "gauss-hermite") then
669                  s = gauss_hermite_int(f, a, b, n)
670              else if (kind == "romberg") then
671                  s = romberg_int(f, a, b, n)
672              else
673                  call error("Unknown integration kind '"//kind//"."// &
                         &
674                  "Available options are: 'polynomial', 'gauss-&
                      legendre', 'gauss-hermite' and 'romberg'.")
675              end if
676
```

```fortran
677              end function
678
679          function polynomial_int(f, a, b, n) result (s)
680              implicit none
681              integer :: n, i
682              double precision :: a, b, s
683              double precision, dimension(n) :: x, y, w
684              double precision, dimension(n, n) :: V
685              interface
686                  function f(x) result (y)
687                      double precision :: x, y
688                  end function
689              end interface
690
691              x(:) = ((b-a)/(n-1)) * (/ (i, i=0,n-1) /) + a
692              y(:) = (/ ((b**i - a**i)/i, i=1, n) /)
693              V(:, :) = vandermond_matrix(x, n)
694              w(:) = solve(V, y, n)
695              s = 0.0D0
696              do i=1, n
697                  s = s + (w(i) * f(x(i)))
698              end do
699              return
700          end function
701
702          function gauss_legendre_int(f, a, b, n) result (s)
703              implicit none
704              integer, intent(in) :: n
705              double precision, intent(in) :: a, b
706              double precision :: s
707              double precision, dimension(n) :: xx, ww
708              integer :: k
709              character(len=*), parameter :: fname =
                      GAUSS_LEGENDRE_QUAD
710              interface
711                  function f(x) result (y)
712                      double precision :: x, y
713                  end function
714              end interface
715
716              call load_quad(xx, ww, n, fname//STR(n)//".txt")
717
718              xx(:) = ((b - a) * xx(:) + (b + a)) / 2
719              s = 0.0D0
720              do k=1, n
721                  s = s + (ww(k) * f(xx(k)))
722              end do
723              s = s * ((b - a) / 2)
724              return
725          end function
726
727          function gauss_hermite_int(f, a, b, n) result (s)
728              implicit none
```

```fortran
            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

            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)
```

```fortran
779                    if (e <= t_tol) then
780                        s = q
781                    else
782                        r = (b + a) / 2
783                        s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
                                + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
784                    end if
785                    return
786                else
787                    s = 0.0D0
788                    return
789                end if
790            end function
791
792            function romberg_int(f, a, b, n, tol) result (s)
793                implicit none
794                integer, intent(in) :: n
795                double precision, intent(in) :: a, b
796                double precision, optional :: tol
797                interface
798                    function f(x) result (y)
799                        double precision :: x, y
800                    end function
801                end interface
802                integer :: i, j, k, t_n
803                double precision :: s, dx, t_tol
804 !              Previous row, Current row and Temporary row
805                double precision, dimension(:, :), allocatable :: R
806
807                if (.NOT. PRESENT(tol)) then
808                    t_tol = D_TOL
809                else
810                    t_tol = tol
811                end if
812
813                t_n = ILOG2(n)
814
815                dx = (b - a)
816
817                allocate(R(t_n + 1, t_n + 1))
818
819                R(1, 1) = (f(a) + f(b)) * dx / 2
820
821                do i = 1, t_n
822                    dx = dx / 2
823
824                    R(i + 1, 1) = (f(a) + 2 * SUM((/ (f(a + k*dx), k=1,
                        (2**i)-1) /)) + f(b)) * dx / 2;
825
826                    do j = 1, i
827                        k = 4 ** j
828                        R(i + 1, j + 1) = (k*R(i + 1, j) - R(i, j)) / (k
                                - 1)
```

48

```fortran
                    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
```

```fortran
! ======== 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
```

50

```fortran
        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
```

```fortran
986          end do
987          return
988      end function
989
990      function ode2_solve(d2f, y0, dy0, t, n, kind) result (y)
991          implicit none
992          integer :: n
993          double precision, intent(in) :: y0, dy0
994          double precision, dimension(n), intent(in) :: t
995          double precision, dimension(n) :: y
996          character(len=*), optional :: kind
997          character(len=:), allocatable :: t_kind
998          interface
999              function d2f(t, y, dy) result (u)
1000                  implicit none
1001                  double precision :: t, y, dy, u
1002              end function
1003          end interface
1004
1005          if (.NOT. PRESENT(kind)) then
1006              t_kind = 'taylor'
1007          else
1008              t_kind = kind
1009          end if
1010
1011          if (t_kind == 'taylor') then
1012              y = taylor(d2f, y0, dy0, t, n)
1013          else if (t_kind == 'runge-kutta-nystrom') then
1014              y = runge_kutta_nystrom(d2f, y0, dy0, t, n)
1015          else
1016              call error("As opções são: 'taylor', 'runge-kutta-
                      nystrom'.")
1017              stop
1018          end if
1019          return
1020      end function
1021
1022      function taylor(d2f, y0, dy0, t, n) result (y)
1023          implicit none
1024          integer :: k, n
1025          double precision, intent(in) :: y0, dy0
1026          double precision :: dt, dy, d2y
1027          double precision, dimension(n), intent(in) :: t
1028          double precision, dimension(n) :: y
1029          interface
1030              function d2f(t, y, dy) result (d2y)
1031                  implicit none
1032                  double precision :: t, y, dy, d2y
1033              end function
1034          end interface
1035  !       Solution
1036          y(1) = y0
1037  !       1st derivative
```

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

Código - Métodos com Matrizes

```fortran
1  !     Matrix Module
2
3      module Matrix
4          use Util
```

```fortran
        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

    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
```

```fortran
57              read(33,*) A(i,:)
58          end do
59          close(33)
60      end subroutine
61
62      subroutine print_vector(x, n)
63          implicit none
64          integer :: n
65          double precision :: x(n)
66          integer :: i
67  30      format(' |', F30.12, '|')
68          do i = 1, n
69              write(*, 30) x(i)
70          end do
71      end subroutine
72
73      subroutine read_vector(fname, b, n)
74          implicit none
75          character(len=*) :: fname
76          integer :: n
77          double precision, allocatable :: b(:)
78
79          open(unit=33, file=fname, status='old', action='read')
80          read(33, *) n
81          allocate(b(n))
82          read(33, *) b(:)
83          close(33)
84      end subroutine
85
86      subroutine show_vector(var, x, n)
87          implicit none
88          integer :: n
89          character(len=*) :: var
90          double precision :: x(n)
91          write (*, *) ''//achar(27)//'[36m'//var//' = '
92          call print_vector(x, n)
93          write (*, *) ''//achar(27)//'[0m'
94      end subroutine
95
96
97  !       =========== Matrix Methods ============
98
99      function clip(x, n, a, b) result (y)
100         integer, intent(in) :: n
101         integer :: k
102         double precision, intent(in) :: a, b
103         double precision, dimension(n), intent(in) :: x
104         double precision, dimension(n) :: y
105
106         do k=1, n
107             if ((a <= x(k)) .AND. (x(k) <= b)) then
108                 y(k) = x(k)
109             else
```

```fortran
110                        y(k) = DNAN
111                    end if
112                end do
113                return
114            end function
115
116            function rand_vector(n, a, b) result (r)
117                implicit none
118                integer :: n, i
119                double precision, dimension(n) :: r
120                double precision, optional :: a, b
121                double precision :: t_a, t_b
122
123                if (.NOT. PRESENT(a)) then
124                    t_a = -1.0D0
125                else
126                    t_a = a
127                end if
128
129                if (.NOT. PRESENT(b)) then
130                    t_b = 1.0D0
131                else
132                    t_b = b
133                end if
134
135                do i = 1, n
136                    r(i) = DRAND(t_a, t_b)
137                end do
138                return
139            end function
140
141            function rand_matrix(m, n, a, b) result (R)
142                implicit none
143                integer :: m, n, i
144                double precision, dimension(m, n) :: R
145                double precision, optional :: a, b
146
147                do i = 1, m
148                    R(i, :) = rand_vector(n, a=a, b=b)
149                end do
150                return
151            end function
152
153            function id_matrix(n) result (A)
154                implicit none
155                integer :: n
156                double precision :: A(n, n)
157                integer :: j
158                A(:, :) = 0.0D0
159                do j = 1, n
160                    A(j, j) = 1.0D0
161                end do
162                return
```

```fortran
            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
            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
```

```fortran
        end function

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

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

            logical :: ok

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

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

            double precision :: A(n, n)

            integer :: i, j
            logical :: ok

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

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

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

            temp(:) = A(i, :)
            A(i, :) = A(j, :)
```

```fortran
267                     A(j, :) = temp(:)
268             end subroutine
269
270             function outer_product(x, y, n) result (A)
271                 implicit none
272                 integer :: n
273                 double precision, dimension(n), intent(in) :: x, y
274                 double precision, dimension(n, n) :: A
275                 integer :: i, j
276                 do i=1,n
277                     do j=1,n
278                         A(i, j) = x(i) * y(j)
279                     end do
280                 end do
281                 return
282             end function
283
284 !           ================= Matrix Method ====================
285             function inv(A, n, ok) result (Ainv)
286                 integer :: n
287                 double precision :: A(n, n), Ainv(n, n)
288                 double precision :: work(n)
289                 integer :: ipiv(n)    ! pivot indices
290                 integer :: info
291
292                 logical :: ok
293
294                 ! External procedures defined in LAPACK
295                 external DGETRF
296                 external DGETRI
297
298                 ! Store A in Ainv to prevent it from being overwritten
299                 !    by LAPACK
299                 Ainv(:, :) = A(:, :)
300
301                 ! DGETRF computes an LU factorization of a general M-by-
                  !    N matrix A
302                 ! using partial pivoting with row interchanges.
303                 call DGETRF(n, n, Ainv, n, ipiv, info)
304
305                 if (info /= 0) then
306                     ok = .FALSE.
307                     return
308                 end if
309
310                 ! DGETRI computes the inverse of a matrix using the LU
                  !    factorization
311                 ! computed by DGETRF.
312                 call DGETRI(n, Ainv, n, ipiv, work, n, info)
313
314                 if (info /= 0) then
315                     ok = .FALSE.
316                     return
```

```fortran
                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
```

```fortran
            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
                    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)
```

```fortran
                            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

                    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
```

```fortran
476          double precision :: A(n, n)
477          logical :: ok
478          ok = symmetrical(A, n) .AND. positive_definite(A, n)
479          return
480      end function
481
482      function PLU_cond(A, n) result (ok)
483          implicit none
484          integer :: n
485          double precision A(n, n)
486          integer :: i, j
487          double precision :: s
488          logical :: ok
489          do j = 1, n
490              s = 0.0D0
491              do i = 1, j
492                  if (A(i, j) > s) then
493                      s = A(i, j)
494                  end if
495              end do
496          end do
497          ok = (s < 0.01D0)
498          return
499      end function
500
501      function LU_cond(A, n) result (ok)
502          implicit none
503          integer :: n
504          double precision A(n, n)
505          logical :: ok
506          ok = positive_definite(A, n)
507          return
508      end function
509 !          _       _____  _____ _____           __
510 !         / |     |_    _|/ ____|__     __|/\      /_ |
511 !        / /        | | | (___      | |  /  \       | |
512 !       / /         | |  \___ \     | | / /\ \      | |
513 !      / |____  _/  |_  ____) |    | |/ ____ \     | |
514 !     /_____|_____|_____/    |_/_/    \_\   |_|
515 !      ==========================================
516
517 !         ======= Matrix Factorization Methods ========
518      function PLU_decomp(A, P, L, U, n) result (ok)
519          implicit none
520          integer :: n
521          double precision :: A(n,n), P(n,n), L(n,n), U(n,n)
522          logical :: ok
523 !          Permutation Matrix
524          P = pivot_matrix(A, n)
525 !          Decomposition over Row-Swapped Matrix
526          ok = LU_decomp(matmul(P, A), L, U, n)
527          return
528      end function
```

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

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

              ok = .TRUE.
              return

          end function

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

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

              logical :: ok

              integer :: i, j

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

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

```fortran
581                         L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)
                            )) / L(i, i)
582                     end do
583                 end do
584
585             ok = .TRUE.
586             return
587         end function
588
589         function Jacobi_cond(A, n) result (ok)
590             implicit none
591
592             integer :: n
593
594             double precision :: A(n, n)
595
596             logical :: ok
597
598             if (.NOT. spectral_radius(A, n) < 1.0D0) then
599                 ok = .FALSE.
600                 call ill_cond()
601                 return
602             else
603                 ok = .TRUE.
604                 return
605             end if
606         end function
607
608         function Jacobi(A, x, b, e, n, tol, max_iter) result (ok)
609             implicit none
610
611             logical :: ok
612
613             integer :: n, i, k, t_max_iter
614             integer, optional :: max_iter
615
616             double precision :: A(n, n)
617             double precision :: b(n), x(n), x0(n)
618             double precision :: e, t_tol
619             double precision, optional :: tol
620
621             if (.NOT. PRESENT(tol)) then
622                 t_tol = D_TOL
623             else
624                 t_tol = tol
625             end if
626
627             if (.NOT. PRESENT(max_iter)) then
628                 t_max_iter = D_MAX_ITER
629             else
630                 t_max_iter = max_iter
631             end if
632
```

```fortran
            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
```

```fortran
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
        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
```

```fortran
736            double precision :: L(n, n), U(n, n)
737            double precision :: b(n), x(n), y(n)
738            integer :: i
739 !          Ly = b (Forward Substitution)
740            do i = 1, n
741                y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i
                       )
742            end do
743 !          Ux = y (Backsubstitution)
744            do i = n, 1, -1
745                x(i) = (y(i) - SUM(U(i,i+1:n) * x(i+1:n))) / U(i, i)
746            end do
747        end subroutine
748
749        function LU_solve(A, x, y, b, n) result (ok)
750            implicit none
751
752            integer :: n
753
754            double precision :: A(n, n), L(n, n), U(n, n)
755            double precision :: b(n), x(n), y(n)
756
757            logical :: ok
758
759            ok = LU_decomp(A, L, U, n)
760
761            if (.NOT. ok) then
762                return
763            end if
764
765            call LU_backsub(L, U, x, y, b, n)
766
767            return
768        end function
769
770        function PLU_solve(A, x, y, b, n) result (ok)
771            implicit none
772
773            integer :: n
774
775            double precision :: A(n, n), P(n,n), L(n, n), U(n, n)
776            double precision :: b(n), x(n), y(n)
777
778            logical :: ok
779
780            ok = PLU_decomp(A, P, L, U, n)
781
782            if (.NOT. ok) then
783                return
784            end if
785
786            call LU_backsub(L, U, x, y, matmul(P, b), n)
787
```

```fortran
            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
```

```fortran
!        ============ 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
```

```fortran
841                 if (.NOT. PRESENT(max_iter)) then
842                     t_max_iter = D_MAX_ITER
843                 else
844                     t_max_iter = max_iter
845                 end if
846
847 !           Begin with random normal vector and set 1st component to
        zero
848             x(:) = rand_vector(n)
849             x(1) = 1.0D0
850
851 !           Initialize Eigenvalues
852             l = 0.0D0
853
854 !           Checks if error tolerance was reached
855             do k=1, t_max_iter
856                 ll = l
857
858                 x(:) = matmul(A, x)
859
860 !               Retrieve Eigenvalue
861                 l = x(1)
862
863 !               Retrieve Eigenvector
864                 x(:) = x(:) / l
865
866                 if (dabs((l - ll) / l) < t_tol) then
867                     ok = .TRUE.
868                     return
869                 end if
870             end do
871             ok = .FALSE.
872             return
873         end function
874
875         function Jacobi_eigen(A, n, L, X, tol, max_iter) result (ok)
876             implicit none
877             logical :: ok
878             integer :: n, i, j, k, u, v, t_max_iter
879             integer, optional :: max_iter
880             double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
881             double precision :: y, z, t_tol
882             double precision, optional :: tol
883
884             if (.NOT. PRESENT(tol)) then
885                 t_tol = D_TOL
886             else
887                 t_tol = tol
888             end if
889
890             if (.NOT. PRESENT(max_iter)) then
891                 t_max_iter = D_MAX_ITER
892             else
```

```fortran
893                           t_max_iter = max_iter
894                   end if
895
896               X(:, :) = id_matrix(n)
897               L(:, :) = A(:, :)
898
899               do k=1, t_max_iter
900                   z = 0.0D0
901                   do i = 1, n
902                       do j = 1, i - 1
903                           y = DABS(L(i, j))
904
905 !                               Found new maximum absolute value
906                           if (y > z) then
907                                   u = i
908                                   v = j
909                                   z = y
910                           end if
911                       end do
912                   end do
913
914                   if (z >= t_tol) then
915                       P(:, :) = given_matrix(L, n, u, v)
916                       L(:, :) = matmul(matmul(transpose(P), L), P)
917                       X(:, :) = matmul(X, P)
918                   else
919                       ok = .TRUE.
920                       return
921                   end if
922               end do
923               ok = .FALSE.
924               return
925           end function
926
927 !           _        _____  _____  _____        _____
928 !          | |     |_   _|/ ____|/ __  \     |  __ \
929 !          | |       | | | (___      | |  / \        ) /
930 !          | |       | |  \___ \     | | / /\ \       /_   \
931 !          | |____  _| |_ ____) |    | |/ ____ \    ___) |
932 !          |_____|_____|_____/     |_/_/     \_\ /_____/
933 !           ========================================
934
935           function least_squares(x, y, s, n) result (ok)
936               implicit none
937               integer :: n
938
939               logical :: ok
940
941               double precision :: A(2,2), b(2), s(2), r(2), x(n), y(n)
942
943               A(1, 1) = n
944               A(1, 2) = SUM(x)
945               A(2, 1) = SUM(x)
```

```fortran
946                  A(2, 2) = dot_product(x, x)
947
948              b(1) = SUM(y)
949              b(2) = dot_product(x, y)
950
951              ok = Cholesky_solve(A, s, r, b, n)
952              return
953          end function
954
955 !          ========== Extra Stuff ========
956
957          function Gauss_solve(A0, x, b0, n) result (ok)
958              implicit none
959              integer n
960              double precision, dimension(n, n), intent(in) :: A0
961              double precision, dimension(n, n) :: A
962              double precision, dimension(n), intent(in) :: b0
963              double precision, dimension(n) :: b, x, s
964              double precision :: c, pivot, store
965              integer i, j, k, l
966
967              logical :: ok
968
969              ok = .TRUE.
970
971              A(:, :) = A0(:, :)
972              b(:) = b0(:)
973
974              do k=1, n-1
975                  do i=k,n
976                      s(i) = 0.0
977                      do j=k,n
978                          s(i) = MAX(s(i), DABS(A(i,j)))
979                      end do
980                  end do
981
982                  pivot = DABS(A(k,k) / s(k))
983                  l = k
984                  do j=k+1,n
985                      if(DABS(A(j,k) / s(j)) > pivot) then
986                          pivot = DABS(A(j,k) / s(j))
987                          l = j
988                      end if
989                  end do
990
991                  if(pivot == 0.0) then
992                      ok = .FALSE.
993                      return
994                  end if
995
996                  if (l /= k) then
997                      do j=k,n
998                          store = A(k,j)
```

```fortran
                                  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
              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)
```

```fortran
1052            else if (t_kind == "cholesky") then
1053                ok = Cholesky_solve(A, x, y, b, n)
1054            else if (t_kind == "gauss") then
1055                ok = Gauss_solve(A, x, b, n)
1056            else
1057                ok = .FALSE.
1058            end if
1059
1060            call debug(":: Solved via '"//t_kind//"' ::")
1061
1062            if (.NOT. ok) then
1063                call error("Failed to solve system Ax = b.")
1064            end if
1065
1066            return
1067        end function
1068
1069    end module Matrix
```

Código - Biblioteca Auxiliar

```fortran
1   !    Util Module
2       module Util
3           implicit none
4           character, parameter :: ENDL = ACHAR(10)
5           character, parameter :: TAB = ACHAR(9)
6
7           double precision :: DINF, DNINF, DNAN
8           DATA DINF/x'7ff0000000000000'/, DNINF/x'fff0000000000000'/,
                DNAN/x'7ff8000000000000'/
9
10          double precision :: PI = 4.0D0 * DATAN(1.0D0)
11
12          logical :: DEBUG_MODE = .FALSE.
13          logical :: QUIET_MODE = .FALSE.
14
15          type StringArray
16              character (:), allocatable :: str
17          end type StringArray
18      contains
19
20      function EDGE(x) result (y)
21          double precision, intent(in) :: x
22          logical :: y
23
24          y = ISNAN(x) .OR. (x == DINF) .OR. (x == DNINF)
25          return
26      end function
27
28      function VEDGE(x) result (y)
29          double precision, dimension(:), intent(in) :: x
```

```fortran
      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
       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(:)
```

```fortran
 83              call RANDOM_SEED(SIZE=n)
 84              allocate(seed(n))
 85              call SYSTEM_CLOCK(COUNT=clock)
 86              seed = clock + 37 * (/ (i - 1, i = 1, n) /)
 87              call RANDOM_SEED(PUT=seed)
 88              deallocate(seed)
 89          end subroutine
 90
 91          function DRAND(a, b) result (y)
 92              implicit none
 93              double precision :: a, b, x, y
 94              ! x in [0, 1)
 95              call RANDOM_NUMBER(x)
 96              y = (x * (b - a)) + a
 97              return
 98          end function
 99
100  !          ===== I/O Metods =====
101          function STR(k) result (t)
102  !          "Convert an integer to string."
103              integer, intent(in) :: k
104              character(len=128) :: s
105              character(len=:), allocatable :: t
106              write(s, *) k
107              t = TRIM(ADJUSTL(s))
108              return
109              return
110          end function
111
112          function DSTR(x) result (q)
113              integer :: j, k
114              double precision, intent(in) :: x
115              character(len=64) :: s
116              character(len=:), allocatable :: p, q
117
118              if (ISNAN(x)) then
119                  q = '?'
120                  return
121              else if (x == DINF) then
122                  q = '∞'
123                  return
124              else if (x == DNINF) then
125                  q = '-∞'
126                  return
127              end if
128
129              write(s, *) x
130              p = TRIM(ADJUSTL(s))
131              do j=LEN(p), 1, -1
132                  if (p(j:j) == '0') then
133                      continue
134                  else if (p(j:j) == '.') then
135                      k = j - 1
```

76

```fortran
136                         exit
137                     else
138                         k = j
139                         exit
140                     end if
141                 end do
142             q = p(:k)
143                 return
144         end function
145
146         subroutine display(text, ansi_code)
147             implicit none
148             character(len=*) :: text
149             character(len=*), optional :: ansi_code
150             if (QUIET_MODE) then
151                 return
152             else
153                 if (PRESENT(ansi_code)) then
154                     write (*, *) ''//achar(27)//'['//ansi_code//'m'
                            //text//''//achar(27)//'[0m'
155                 else
156                     write (*, *) text
157                 end if
158             end if
159         end subroutine
160
161         subroutine error(text)
162 !           Red Text
163             implicit none
164             character(len=*) :: text
165             call display(text, '31')
166         end subroutine
167
168         subroutine warn(text)
169 !           Yellow Text
170             implicit none
171             character(len=*) :: text
172             call display(text, '93')
173         end subroutine
174
175         subroutine debug(text)
176 !           Yellow Text
177             implicit none
178             character(len=*) :: text
179             if (DEBUG_MODE) then
180                 call display('[DEBUG] '//text, '93')
181             end if
182         end subroutine
183
184         subroutine info(text)
185 !           Green Text
186             implicit none
187             character(len=*) :: text
```

```fortran
188            call display(text, '32')
189        end subroutine
190
191        subroutine blue(text)
192 !          Blue Text
193            implicit none
194            character(len=*) :: text
195            call display(text, '36')
196        end subroutine
197
198        subroutine show(var, val)
199 !          Violet Text
200            implicit none
201            character(len=*) :: var
202            double precision :: val
203            write (*, *) ''//achar(27)//'[36m'//var//' = '//DSTR(val
                 )//''//achar(27)//'[0m'
204        end subroutine
205
206        recursive subroutine cross_quick_sort(x, y, u, v, n)
207            integer :: n, i, j, u, v
208            double precision :: p, aux, auy
209            double precision :: x(n), y(n)
210
211            i = u
212            j = v
213
214            p = x((u + v) / 2)
215
216            do while (i <= j)
217                do while (x(i) < p)
218                    i = i + 1
219                end do
220                do while(x(j) > p)
221                    j = j - 1
222                end do
223                if (i <= j) then
224                    aux = x(i)
225                    auy = y(i)
226                    x(i) = x(j)
227                    y(i) = y(j)
228                    x(j) = aux
229                    y(j) = auy
230                    i = i + 1
231                    j = j - 1
232                end if
233            end do
234
235            if (u < j) then
236                call cross_quick_sort(x, y, u, j, n)
237            end if
238            if (i < v) then
239                call cross_quick_sort(x, y, i, v, n)
```

```fortran
240              end if
241              return
242          end subroutine
243
244          subroutine cross_sort(x, y, n)
245              implicit none
246              integer :: n
247              double precision :: x(n), y(n)
248
249              call cross_quick_sort(x, y, 1, n, n)
250          end subroutine
251
252          subroutine linspace(a, b, dt, n, t)
253              implicit none
254              integer :: k, n
255              double precision, intent(in) :: a, b, dt
256              double precision, dimension(:), allocatable :: t
257              n = 1 + FLOOR((b - a) / dt)
258              allocate(t(n))
259              t(:) = dt * (/ (k, k=0, n-1) /)
260          end subroutine
261      end module Util
```

Código - Biblioteca de Plotagem

```fortran
1  module PlotLib
2      use Util
3      implicit none
4      character(len=*), parameter :: DEFAULT_FNAME = 'plotfile'
5      character(len=*), parameter :: DEFAULT_SIZE_W = '12in'
6      character(len=*), parameter :: DEFAULT_SIZE_H = '9in'
7      character(len=*), parameter :: PLOT_ENDL = ',\'//ENDL
8
9      logical :: g_INPLOT = .FALSE.
10      logical :: g_INMULTIPLOT = .FALSE.
11
12      character(len=:), allocatable :: g_FNAME, g_OUTP_FNAME,
           g_PLOT_FNAME, g_SIZE_W, g_SIZE_H
13
14      integer :: g_M, g_N
15
16      type SPLOT
17          integer :: i, j
18          integer :: n = 0
19          logical :: grid = .FALSE.
20          logical :: done = .FALSE.
21
22          character(len=:), allocatable :: title, xlabel, ylabel
23          type(StringArray), dimension(:), allocatable :: legend
24          type(StringArray), dimension(:), allocatable :: with
25
```

```fortran
26   !        Plot boundaries
27         logical :: l_xmin = .FALSE., l_xmax = .FALSE.
28         logical :: l_ymin = .FALSE., l_ymax = .FALSE.
29         double precision :: xmin, xmax, ymin, ymax
30     end type
31
32     type(SPLOT), dimension(:, :), allocatable :: g_SUBPLOTS
33
34     contains
35
36     function REMOVE_TEMP_FILES(fname) result (cmd)
37         implicit none
38         character(len=*) :: fname
39         character(len=:), allocatable :: plt
40         character(len=:), allocatable :: dat
41         character(len=:), allocatable :: cmd
42
43         plt = 'plot/'//fname//'*.plt'
44         dat = 'plot/'//fname//'*.dat'
45
46         cmd = 'rm '//plt//' '//dat
47         return
48     end function
49
50     function PLOT_FNAME(fname) result (path)
51         implicit none
52         character(len=*) :: fname
53         character(len=:), allocatable :: path
54         path = 'plot/'//fname//'.plt'
55         return
56     end function
57
58     function DATA_FNAME(fname, i, j, n) result (path)
59         implicit none
60         integer, optional, intent(in) :: i, j, n
61         character(len=*) :: fname
62         character(len=:), allocatable :: path, t_i, t_j, t_n
63
64         if (.NOT. PRESENT(i)) then
65             t_i = '1'
66         else
67             t_i = STR(i)
68         end if
69
70         if (.NOT. PRESENT(j)) then
71             t_j = '1'
72         else
73             t_j = STR(j)
74         end if
75
76         if (.NOT. PRESENT(n)) then
77             t_n = '1'
78         else
```

```fortran
79                    t_n = STR(n)
80              end if
81
82          path = 'plot/'//fname//'_'//t_i//'_'//t_j//'_'//t_n//'.dat'
83          return
84      end function
85
86      function OUTP_FNAME(fname) result (path)
87          implicit none
88          character(len=*) :: fname
89          character(len=:), allocatable :: path
90          path = 'plot/'//fname//'.pdf'
91          return
92      end function
93
94      function GNU_PLOT_CMD(fname) result (cmd)
95          implicit none
96          character(len=*) :: fname
97          character(len=:), allocatable :: cmd
98          cmd = 'gnuplot -p '//PLOT_FNAME(fname)
99      end function
100
101     subroutine subplot_config(i, j, title, xlabel, ylabel, xmin,
            xmax, ymin, ymax, legend, with, grid)
102         integer, intent(in) :: i, j
103         integer :: k, n
104
105         logical, optional :: grid
106
107         character(len=*), optional :: title, ylabel, xlabel
108
109         double precision, optional :: xmin, xmax, ymin, ymax
110
111         type(StringArray), dimension(:), optional :: legend, with
112
113         if (g_SUBPLOTS(i, j)%done) then
114             call error("Duplicate configuration of subplot ("//STR(i
                )//", "//STR(j)//")")
115             stop "ERROR"
116         end if
117
118         n = g_SUBPLOTS(i, j)%n
119
120         allocate(g_SUBPLOTS(i, j)%legend(n))
121         allocate(g_SUBPLOTS(i, j)%with(n))
122
123         if (PRESENT(xmin)) then
124             g_SUBPLOTS(i, j)%xmin = xmin
125             g_SUBPLOTS(i, j)%l_xmin = .TRUE.
126         end if
127
128         if (PRESENT(xmax)) then
129             g_SUBPLOTS(i, j)%xmax = xmax
```

```
130              g_SUBPLOTS(i, j)%l_xmax = .TRUE.
131          end if
132
133          if (PRESENT(ymin)) then
134              g_SUBPLOTS(i, j)%ymin = ymin
135              g_SUBPLOTS(i, j)%l_ymin = .TRUE.
136          end if
137
138          if (PRESENT(ymax)) then
139              g_SUBPLOTS(i, j)%ymax = ymax
140              g_SUBPLOTS(i, j)%l_ymax = .TRUE.
141          end if
142
143          if (.NOT. PRESENT(legend)) then
144              do k=1,n
145                  g_SUBPLOTS(i, j)%legend(k)%str = 't '//quote(STR(i))
146              end do
147          else
148              do k=1,n
149                  g_SUBPLOTS(i, j)%legend(k)%str = 't '//quote(legend(
                         k)%str)
150              end do
151          end if
152
153          if (.NOT. PRESENT(with)) then
154              do k=1,n
155                  g_SUBPLOTS(i, j)%with(k)%str = 'w lines'
156              end do
157          else
158              do k=1,n
159                  g_SUBPLOTS(i, j)%with(k)%str = 'w '//with(k)%str
160              end do
161          end if
162
163          if (.NOT. PRESENT(grid)) then
164              g_SUBPLOTS(i, j)%grid = .TRUE.
165          else
166              g_SUBPLOTS(i, j)%grid = grid
167          end if
168
169          if (.NOT. PRESENT(title)) then
170              g_SUBPLOTS(i, j)%title = ''
171          else
172              g_SUBPLOTS(i, j)%title = title
173          end if
174
175          if (.NOT. PRESENT(xlabel)) then
176              g_SUBPLOTS(i, j)%xlabel = 'x'
177          else
178              g_SUBPLOTS(i, j)%xlabel = xlabel
179          end if
180
181          if (.NOT. PRESENT(ylabel)) then
```

```fortran
182                    g_SUBPLOTS(i, j)%ylabel = 'y'
183                else
184                    g_SUBPLOTS(i, j)%ylabel = ylabel
185                end if
186
187                g_SUBPLOTS(i, j)%done = .TRUE.
188
189        end subroutine
190
191        subroutine subplot(i, j, x, y, n)
192            integer :: file, k
193            integer, intent(in) :: i, j, n
194            double precision, dimension(n), intent(in) :: x
195            double precision, dimension(n), intent(in) :: y
196
197            character(len=:), allocatable :: s_data_fname
198
199            if (g_SUBPLOTS(i, j)%done) then
200                call error("Plot over finished subplot ("//STR(i)//", "
                        //STR(j)//")")
201                stop "ERROR"
202            else
203                g_SUBPLOTS(i, j)%n = g_SUBPLOTS(i, j)%n + 1
204            end if
205
206            s_data_fname = DATA_FNAME(g_FNAME, i, j, g_SUBPLOTS(i, j)%n)
207
208 !         =================== Touch Plot File
        ================================
209            open(newunit=file, file=s_data_fname, status="replace",
                action="write")
210                write(file, *) "# file: "//s_data_fname
211            close(file)
212 !
        =========================================================================

213
214 !         =================== Write to Plot File
        ================================
215 10       format(F16.8, ' ')
216 11       format(F16.8, ' ')
217            open(newunit=file, file=s_data_fname, status="old", position
                ="append", action="write")
218            write(file, *)
219            do k=1, n
220                write(file, 10, advance='no') x(k)
221                write(file, 11, advance='yes') y(k)
222            end do
223            close(file)
224 !
        =========================================================================

225        end subroutine
```

```fortran
!     ======= Pipeline ============
      subroutine begin_plot(fname, size_w, size_h)
          integer :: file
          character(len=*), optional :: fname, size_w, size_h

          if (.NOT. PRESENT(size_w)) then
              g_SIZE_W = DEFAULT_SIZE_W
          else
              g_SIZE_W = size_w
          end if

          if (.NOT. PRESENT(size_h)) then
              g_SIZE_H = DEFAULT_SIZE_H
          else
              g_SIZE_H = size_h
          end if

          if (.NOT. PRESENT(fname)) then
              g_FNAME = DEFAULT_FNAME
          else
              g_FNAME = fname
          end if

          g_PLOT_FNAME = PLOT_FNAME(g_FNAME)
          g_OUTP_FNAME = OUTP_FNAME(g_FNAME)

          open(newunit=file, file=g_PLOT_FNAME, status="new", action="&
              write")
          write(file, *) 'set terminal pdf size '//g_SIZE_W//', '//&
              g_SIZE_H//';'
          write(file, *) 'set output '//quote(g_OUTP_FNAME)//';'
          close(file)

          g_INPLOT = .TRUE.

      end subroutine

      subroutine subplots(m, n)
          integer, optional, intent(in) :: m, n
          integer :: t_m, t_n

          if ((.NOT. PRESENT(m)) .OR. (m <= 0)) then
              t_m = 1
          else
              t_m = m
          end if

          if ((.NOT. PRESENT(n)) .OR. (n <= 0)) then
              t_n = 1
          else
              t_n = n
          end if
```

```fortran
            if (.NOT. g_INPLOT) then
                call begin_plot()
            end if

!           ===== Allocate Variables =====
            allocate(g_SUBPLOTS(t_m, t_n))
            g_M = t_m
            g_N = t_n
            g_INMULTIPLOT = .TRUE.
!           ==============================
      end subroutine

      subroutine render_plot(clean)
          integer :: file, i, j, k, m, n

          logical, optional :: clean
          logical :: t_clean

          if (.NOT. PRESENT(clean)) then
              t_clean = .FALSE.
          else
              t_clean = clean
          end if

!         === Check Plot =========
          if (.NOT. g_INPLOT) then
              call error("No active plot to render.")
              stop "ERROR"
          end if
!         =======================

          m = g_M
          n = g_N

!         ==================== Write to Plot File
      ====================================================
          open(newunit=file, file=g_PLOT_FNAME, status="old", position
              ="append", action="write")
          write(file, *) 'set origin 0,0;'

          if (g_INMULTIPLOT) then
              write(file, *) 'set multiplot layout '//STR(m)//','//STR
                  (n)//' rowsfirst;'
          end if

10        format(A, ' ')
!         =========== Plot data =============
          do i= 1, m
              do j = 1, n
                  g_SUBPLOTS(i, j) = g_SUBPLOTS(i, j)
```

```fortran
326                    write(file, *) 'set title '//quote(g_SUBPLOTS(i, j)%&
                           title)//';'
327                    write(file, *) 'set xlabel '//quote(g_SUBPLOTS(i, j)&
                           %xlabel)//';'
328                    write(file, *) 'set ylabel '//quote(g_SUBPLOTS(i, j)&
                           %ylabel)//';'
329
330    !              ============= Set XRANGE
          =====================================
331                    if (g_SUBPLOTS(i, j)%l_xmin .AND. g_SUBPLOTS(i, j)%&
                           l_xmax) then
332                        write(file, *) 'set xrange ['//DSTR(g_SUBPLOTS(i&
                               , j)%xmin)//':'//DSTR(g_SUBPLOTS(i, j)%xmax)&
                               //'];'
333                    else if (g_SUBPLOTS(i, j)%l_xmin) then
334                        write(file, *) 'set xrange ['//DSTR(g_SUBPLOTS(i&
                               , j)%xmin)//':*];'
335                    else if (g_SUBPLOTS(i, j)%l_xmax) then
336                        write(file, *) 'set xrange [*:'//DSTR(g_SUBPLOTS&
                               (i, j)%xmax)//'];'
337                    else
338                        write(file, *) 'set xrange [*:*];'
339                    end if
340    !
          ================================================================
341
342    !              ============= Set YRANGE
          =====================================
343                    if (g_SUBPLOTS(i, j)%l_ymin .AND. g_SUBPLOTS(i, j)%&
                           l_ymax) then
344                        write(file, *) 'set yrange ['//DSTR(g_SUBPLOTS(i&
                               , j)%ymin)//':'//DSTR(g_SUBPLOTS(i, j)%ymax)&
                               //'];'
345                    else if (g_SUBPLOTS(i, j)%l_ymin) then
346                        write(file, *) 'set yrange ['//DSTR(g_SUBPLOTS(i&
                               , j)%ymin)//':*];'
347                    else if (g_SUBPLOTS(i, j)%l_ymax) then
348                        write(file, *) 'set yrange [*:'//DSTR(g_SUBPLOTS&
                               (i, j)%ymax)//'];'
349                    else
350                        write(file, *) 'set yrange [*:*];'
351                    end if
352    !
          ================================================================
353
354                    if (g_SUBPLOTS(i, j)%grid) then
355                        write(file, *) 'set grid;'
356                    else
357                        write(file, *) 'unset grid;'
358                    end if
359
360                    write(file, 10, advance='no') 'plot'
361
```

```fortran
362                 do k = 1, g_SUBPLOTS(i, j)%n
363                     write(file, 10, advance='no') quote(DATA_FNAME(
                            g_FNAME, i, j, k))
364                     write(file, 10, advance='no') 'u 1:2'
365                     write(file, 10, advance='no') g_SUBPLOTS(i, j)%
                            legend(k)%str
366                     write(file, 10, advance='no') g_SUBPLOTS(i, j)%
                            with(k)%str
367
368                     if (k == (g_SUBPLOTS(i, j)%n)) then
369                         write(file, *) ';'
370                     else
371                         write(file, *) ',\'
372                     end if
373                 end do
374             end do
375         end do
376 !       ==================================
377
378 !       == Finish Multiplot ===
379         if (g_INMULTIPLOT) then
380             write(file, *) 'unset multiplot'
381             g_INMULTIPLOT = .FALSE.
382         end if
383 !       =====================
384         close(file)
385 !
    ===============================================================================

386
387 !       ===== Call GNUPLOT and remove temporary files ===========
388         call EXECUTE_COMMAND_LINE(GNU_PLOT_CMD(g_FNAME))
389
390         if (t_clean) then
391             call EXECUTE_COMMAND_LINE(REMOVE_TEMP_FILES(g_FNAME))
392         end if
393 !       ===========================================================
394
395 !       ========= Free Variables =====================
396         deallocate(g_FNAME, g_OUTP_FNAME, g_PLOT_FNAME)
397         deallocate(g_SUBPLOTS)
398         g_INPLOT = .FALSE.
399 !       ==============================================
400     end subroutine
401 end module PlotLib
```