

# COC473 - Lista 5

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.: Integração Numérica

A função `num_int(f, a, b, n, kind)` calcula a integral  $\int_a^b f(x)dx$  aproximada por  $n$  pontos. O parâmetro nomeado opcional `kind` permite ao usuário escolher dentre as opções:

`'polynomial'` Integração Polinomial  
`'gauss-legendre'` Quadratura de *Gauss-Legendre*  
`'gauss-hermite'` Quadratura de *Gauss-Hermite*<sup>1</sup>  
`'romberg'` Método de *Romberg*

```
1      function num_int(f, a, b, n, kind) result (s)
2          implicit none
3          integer :: n
4          character (len=*), optional :: kind
5          double precision :: a, b, s
6          interface
7              function f(x) result (y)
8                  double precision :: x, y
9              end function
10         end interface
11
12         if (.NOT. PRESENT(kind)) then
13             kind = "polynomial"
14         end if
15
16         if (kind == "polynomial") then
17             s = polynomial_int(f, a, b, n)
18         else if (kind == "gauss-legendre") then
19             s = gauss_legendre_int(f, a, b, n)
20         else if (kind == "gauss-hermite") then
21             s = gauss_hermite_int(f, a, b, n)
22         else if (kind == "romberg") then
23             s = romberg_int(f, a, b, n)
24         else
25             call error("Unknown integration kind '//kind//'. "//
26                 &
27                 "Available options are: 'polynomial', 'gauss-
28                 legendre', 'gauss-hermite' and 'romberg'." )
29         end if
30     end function
```

---

<sup>1</sup>Demanda condições especiais.

## 1 .: Integração Polinomial

A Integração Polinomial foi implementada através da solução do sistema linear

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ \vdots & \vdots & & \vdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{bmatrix} \begin{bmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_n \end{bmatrix} = \begin{bmatrix} b-a \\ \frac{b^2-a^2}{2} \\ \vdots \\ \frac{b^n-a^n}{n} \end{bmatrix}$$

onde os pesos de integração são dados pelas componentes  $\omega_i$  da solução.

```
1      function polynomial_int(f, a, b, n) result (s)
2          implicit none
3          integer :: n, i
4          double precision :: a, b, s
5          double precision, dimension(n) :: x, y, w
6          double precision, dimension(n, n) :: V
7          interface
8              function f(x) result (y)
9                  double precision :: x, y
10             end function
11         end interface
12
13         x(:) = ((b-a)/(n-1)) * (/ (i, i=0,n-1) /) + a
14         y(:) = (/ ((b**i - a**i)/i, i=1, n) /)
15         V(:, :) = vandermond_matrix(x, n)
16         w(:) = solve(V, y, n)
17         s = 0.0D0
18         do i=1, n
19             s = s + (w(i) * f(x(i)))
20         end do
21         return
22     end function
```

## 2 .: Quadratura de Gauss-Legendre

A quadratura de *Gauss-Legendre* foi calculada previamente para até  $n = 128$  pontos através do sistema de computação algébrica da linguagem *Mathematica*. O Código consta no apêndice.

```
1      function gauss_legendre_int(f, a, b, n) result (s)
2          implicit none
3          integer, intent(in) :: n
4          double precision, intent(in) :: a, b
5          double precision :: s
6          double precision, dimension(n) :: xx, ww
7          integer :: k
8          character(len=*), parameter :: fname =
9              GAUSS_LEGENDRE_QUAD
10         interface
11             function f(x) result (y)
12                 double precision :: x, y
13             end function
14         end interface
```

```

15      call load_quad(xx, ww, n, fname//STR(n)//".txt")
16
17      xx(:) = ((b - a) * xx(:) + (b + a)) / 2
18      s = 0.0D0
19      do k=1, n
20          s = s + (ww(k) * f(xx(k)))
21      end do
22      s = s * ((b - a) / 2)
23      return
24  end function

```

### 3 :: Quadratura de *Gauss-Hermite*

A quadratura de *Gauss-Hermite* foi tabelada da mesma maneira que a anterior. Para utilizar este método, é preciso que a integração ocorra sobre todos os números reais, isto é,  $[a, b] = [-\infty, \infty]$ .

```

1      function gauss_hermite_int(f, a, b, n) result (s)
2          implicit none
3          integer, intent(in) :: n
4          double precision, intent(in) :: a, b
5          double precision :: s
6          double precision, dimension(n) :: xx, ww
7          integer :: k
8          character(len=*), parameter :: fname =
9              GAUSS_HERMITE_QUAD
10         interface
11             function f(x) result (y)
12                 double precision :: x, y
13             end function
14         end interface
15
16         call load_quad(xx, ww, n, fname//STR(n)//".txt")
17
18         if (a /= DNINF .OR. b /= DINF) then
19             call error("O Método de Gauss-Hermite deve ser usado
20                 no intervalo dos reais.")
21             stop
22         end if
23
24         s = 0.0D0
25         do k=1, n
26             s = s + (ww(k) * f(xx(k)))
27         end do
28
29         return
30     end function

```

#### 4 .: Método de *Romberg*

Além dos métodos pedidos, implementei também a integração de *Romberg* para conhecer mais esta técnica.

```
1      integer, intent(in) :: n
2      double precision, intent(in) :: a, b
3      double precision :: s
4      double precision, dimension(n) :: xx, ww
5      integer :: k
6      character(len=*), parameter :: fname =
7          GAUSS_HERMITE_QUAD
8      interface
9          function f(x) result (y)
10             double precision :: x, y
11         end function
12     end interface
13
14     call load_quad(xx, ww, n, fname//STR(n)//".txt")
15
16     if (a /= DNINF .OR. b /= DINF) then
17         call error("O Método de Gauss-Hermite deve ser usado
18             no intervalo dos reais.")
19         stop
20     end if
21
22     s = 0.0D0
23     do k=1, n
24         s = s + (ww(k) * f(xx(k)))
25     end do
26
27     return
28 end function
29
30 recursive function adapt_int(f, a, b, n, tol, kind) result (
31     s)
32     implicit none
33     integer :: n
34     character (len=*), optional :: kind
35     double precision, intent(in) :: a, b
36     double precision :: p, q, e, r, s, t_tol
37     double precision, optional :: tol
38     interface
39         function f(x) result (y)
40             double precision :: x, y
41         end function
42     end interface
43
44     if (.NOT. PRESENT(tol)) then
45         t_tol = D_TOL
46     else
47         t_tol = tol
48     end if
```

```

47         if (n > 1) then
48             p = num_int(f, a, b, n / 2, kind = kind)
49             q = num_int(f, a, b, n, kind = kind)
50             e = DABS(p - q)
51             if (e <= t_tol) then
52                 s = q
53             else
54                 r = (b + a) / 2
55                 s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
56                     + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
57             end if
58             return
59         else
60             s = 0.0D0
61             return
62         end if
63     end function
64
65 function romberg_int(f, a, b, n, tol) result (s)
66     implicit none
67     integer, intent(in) :: n
68     double precision, intent(in) :: a, b
69     double precision, optional :: tol
70     interface
71         function f(x) result (y)
72             double precision :: x, y
73         end function
74     end interface
75     integer :: i, j, k, t_n
76     double precision :: s, dx, t_tol
77     ! Previous row, Current row and Temporary row
78     double precision, dimension(:, :), allocatable :: R
79
80     if (.NOT. PRESENT(tol)) then
81         t_tol = D_TOL
82     else
83         t_tol = tol
84     end if
85
86     t_n = ILOG2(n)
87
88     dx = (b - a)
89
90     allocate(R(t_n + 1, t_n + 1))
91
92     R(1, 1) = (f(a) + f(b)) * dx / 2
93
94     do i = 1, t_n
95         dx = dx / 2
96
97         R(i + 1, 1) = (f(a) + 2 * SUM((/ (f(a + k*dx), k=1,
98             (2**i)-1) /)) + f(b)) * dx / 2;

```

```

98         do j = 1, i
99             k = 4 ** j
100             R(i + 1, j + 1) = (k*R(i + 1, j) - R(i, j)) / (k
- 1)
101         end do
102
103         if (DABS(R(i + 1, i + 1) - R(i, i)) > t_tol) then
104             continue
105         else
106             exit
107         end if
108     end do
109     s = R(i, i)
110
111     deallocate(R)
112 end function

```

## 5 .: Integração Adaptativa

Implementei também a integração adaptativa, que permite calcular integrais dada uma tolerância. Assim, conseguimos resultados mais precisos para funções de comportamento irregular, isto é, aquelas que possuem derivadas com alto valor absoluto no intervalo de integração. Isso é feito subdividindo os trechos do intervalo  $[a, b]$  de maneira que os subintervalos de maior irregularidade sejam analisados por mais pontos de integração. Esse método garante maior resolução sob demanda.

Assim, foi possível obter valores mais precisos para fins de comparação.

```

1      recursive function adapt_int(f, a, b, n, tol, kind) result (
2          s)
3          implicit none
4          integer :: n
5          character (len=*) , optional :: kind
6          double precision, intent(in) :: a, b
7          double precision :: p, q, e, r, s, t_tol
8          double precision, optional :: tol
9          interface
10             function f(x) result (y)
11                 double precision :: x, y
12             end function
13         end interface
14
15         if (.NOT. PRESENT(tol)) then
16             t_tol = D_TOL
17         else
18             t_tol = tol
19         end if
20
21         if (n > 1) then
22             p = num_int(f, a, b, n / 2, kind = kind)
23             q = num_int(f, a, b, n, kind = kind)
24             e = DABS(p - q)
25             if (e <= t_tol) then
26                 s = q

```

```
26         else
27             r = (b + a) / 2
28             s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
                + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
29         end if
30         return
31     else
32         s = 0.0D0
33         return
34     end if
35 end function
```



## Aplicações

**Questão 2.:** Use o programa desenvolvido para obter o resultado numérico das seguintes integrais:

$$I1 = \int_0^1 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$
$$I2 = \int_0^5 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

Usando  $n = 10$  pontos de integração:

```
2)
f(x) = exp(-x^2/2) / sqrt(2 * pi)
[a, b] = [0, 1]
:: Integração Polinomial ::
I1 = ∫f(x) dx ≈ 0.3413447460735613
:: Quadratura de Gauss-Legendre ::
I1 = ∫f(x) dx ≈ 0.34134474606854304
:: Método de Romberg ::
I1 = ∫f(x) dx ≈ 0.34134391691400612
[a, b] = [0, 5]
:: Integração Polinomial ::
I2 = ∫f(x) dx ≈ 0.49957515630078708
:: Quadratura de Gauss-Legendre ::
I2 = ∫f(x) dx ≈ 0.49999971572535451
:: Método de Romberg ::
I2 = ∫f(x) dx ≈ 0.50108201123349327
```

**Questão 3.:** Usando o seu programa e considerando  $S_\sigma(\omega) = \text{RAO}(\omega)^2 S_\eta(\omega)$ , onde

$$\text{RAO}(\omega) = \frac{1}{\sqrt{\left(1 - \left(\frac{\omega}{\omega_n}\right)^2\right)^2 + \left(2\xi\frac{\omega}{\omega_n}\right)^2}}$$

com  $\omega_n = 1.0$  e  $\xi = 0.05$  e  $S_\eta(\omega) = 2.0$ , obtenha  $m_0$  e  $m_2$  dados por:

$$m_0 = \int_0^{10} S_\sigma(\omega) d\omega$$

$$m_2 = \int_0^{10} \omega^2 S_\sigma(\omega) d\omega$$

```
3)
Sσ(ω) = RAO(ω)2 Sη(ω)
      RAO(ω) = 1 / √((1 - (ω/ωn)2)2 + (2ξω/ωn)2)
[a, b] = [0, 10]

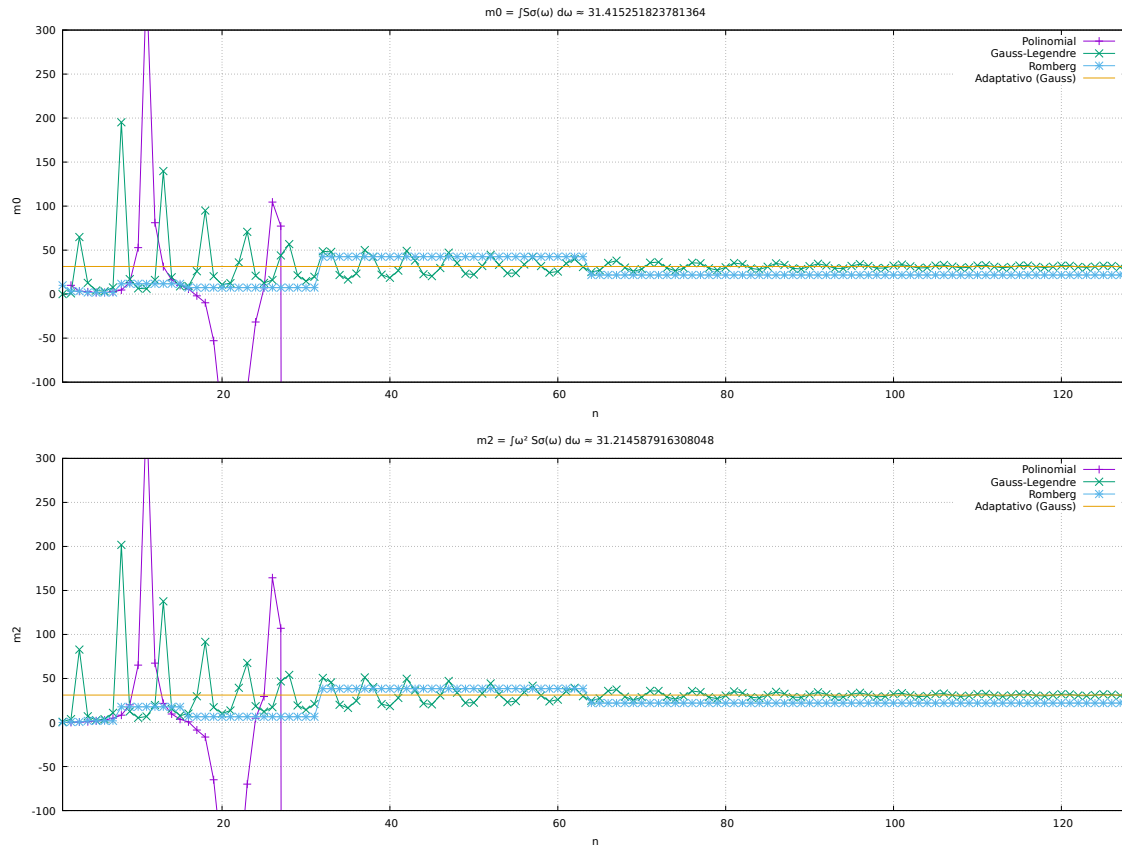
m0 ~ Sη(ω) = 2

:: Valor de referência (Integração Adaptativa) tol = 1E-8 ::
m0 = ∫Sσ(ω) dω ≈ 31.415251823781364
:: Integração Polinomial ::
m0 = ∫Sσ(ω) dω ≈ 52.859249702281744
:: Quadratura de Gauss-Legendre ::
m0 = ∫Sσ(ω) dω ≈ 6.5985596152553274
:: Método de Romberg ::
m0 = ∫Sσ(ω) dω ≈ 11.716280480263372

m2 ~ Sη(ω) = 2

:: Valor de referência (Integração Adaptativa) tol = 1E-8 ::
m2 = ∫Sσ(ω) dω ≈ 31.214587916308048
:: Integração Polinomial ::
m2 = ∫Sσ(ω) dω ≈ 65.209981049518717
:: Quadratura de Gauss-Legendre ::
m2 = ∫Sσ(ω) dω ≈ 5.3569850092067099
:: Método de Romberg ::
m2 = ∫Sσ(ω) dω ≈ 17.819159784396721
```

Com  $n = 10$  pontos de integração, nenhum dos métodos se aproximou de fato do valor de referência. Para um estudo mais aprofundado, elaborei gráficos com os valores de cada método, considerando de 1 até 128 pontos de integração. Vejamos a figura:



O que vemos neste gráfico é, primeiramente, que a Integração Polinomial diverge completamente conforme aumentamos o número de pontos. A Quadratura de *Gauss-Legendre* demonstra comportamento oscilatório ao redor da solução, apresentando valores próximos ao valor de referência com menos de 20 pontos de integração. Este resultando, contudo, ainda não é consistente e pequenas perturbações nas condições poderiam levar a erros catastróficos. Os valores passam a ser confiáveis quando se utiliza mais de 100 pontos de integração.

O método de *Romberg* possui uma certa sutileza. Iniciar o algoritmo com  $n$  pontos de entrada faz com que este avalie a função em  $2^n$  pontos. Por isso, para comparação com os algoritmos, são utilizados  $\log_2(n)$  pontos como entrada. Isso faz com que se utilize  $n$  pontos no total somente quando  $n$  é potência de 2. Apesar disso, vemos que o método apresenta comportamento similar ao da quadratura, acompanhando a "amplitude" da oscilação.

**Questão 4.:** Repita o exercício anterior considerando

$$S_{\eta}(\omega) = \frac{4\pi^3 H s^2}{\omega^5 T z^4} \exp\left(-\frac{16\pi^3}{\omega^4 T z^4}\right) \text{ com } H s = 3.0 \text{ e } T z = 5.0$$

4)

$$S\sigma(\omega) = \text{RAO}(\omega)^2 S_{\eta}(\omega)$$

$$\text{RAO}(\omega) = 1 / \sqrt{(1 - (\omega/\omega_n)^2)^2 + (2\xi\omega/\omega_n)^2}$$

$$[a, b] = [0, 10]$$

$$m0 \sim S_{\eta}(\omega) = ((4 \pi^3 H s^2) / (\omega^5 T z^4)) \exp(-(16 \pi^3) / (\omega^4 T z^4))$$

```

:: Valor de referência (Integração Adaptativa) tol = 1E-8 ::
m0 = ∫Sσ(ω) dω ≈ 11.227882149614736
:: Integração Polinomial ::
m0 = ∫Sσ(ω) dω ≈ ?
:: Quadratura de Gauss–Legendre ::
m0 = ∫Sσ(ω) dω ≈ 0.75329021329714352
:: Método de Romberg ::
m0 = ∫Sσ(ω) dω ≈ ?

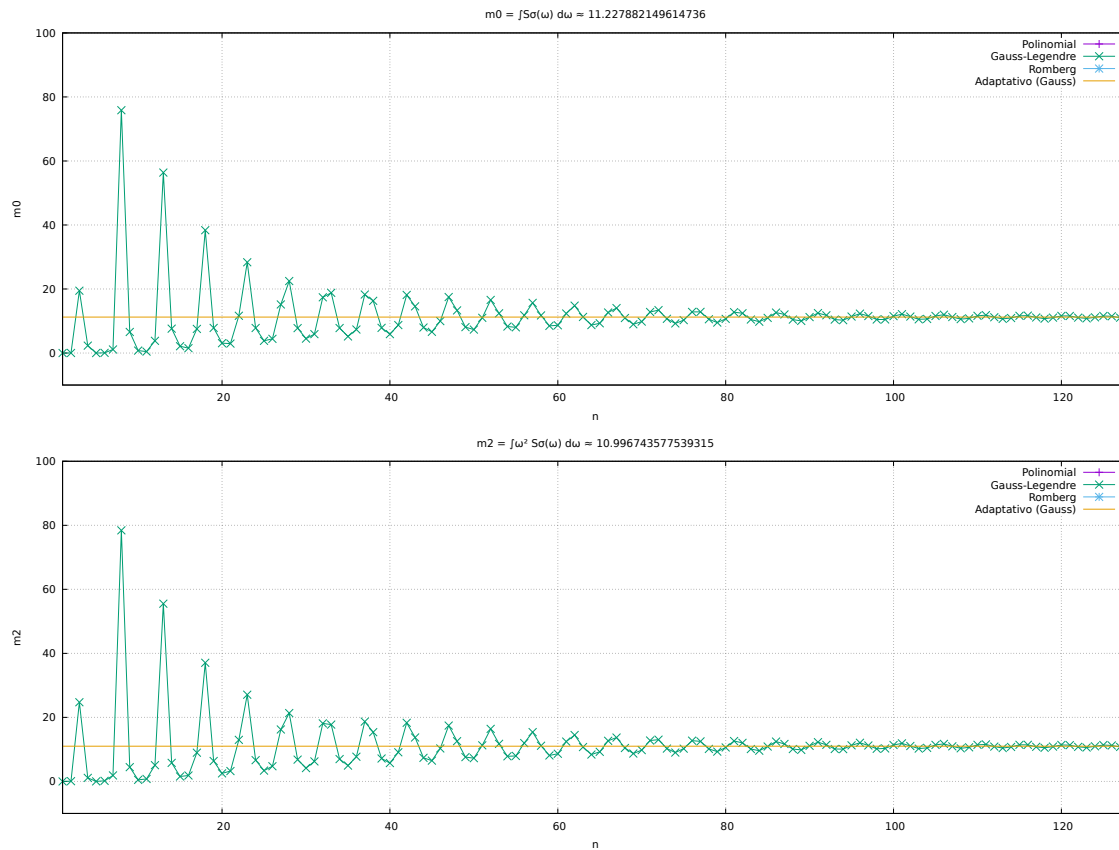
```

$$m2 \sim S\eta(\omega) = ((4 \pi^3 Hs^2) / (\omega^5 Tz^4)) \exp(-(16 \pi^3) / (\omega^4 Tz^4))$$

```

:: Valor de referência (Integração Adaptativa) tol = 1E-8 ::
m2 = ∫Sσ(ω) dω ≈ 10.996743577539315
:: Integração Polinomial ::
m2 = ∫Sσ(ω) dω ≈ ?
:: Quadratura de Gauss–Legendre ::
m2 = ∫Sσ(ω) dω ≈ 0.48426836843985055
:: Método de Romberg ::
m2 = ∫Sσ(ω) dω ≈ ?

```



A Quadratura de *Gauss-Legendre* se mostrou oscilatória ao redor da solução como na questão anterior. Os métodos de Integração Polinomial e de *Romberg*, no entanto, divergiram e rapidamente apresentaram valores inválidos (**NaN**) em seu resultado. O método de *Romberg* é construído sobre a regra do trapézio e, portanto, apesar de ser um método adaptativo, está sujeito as mesmas vulnerabilidades.

**Questão 5.:** Com o programa desenvolvido, use o número mínimo de pontos de integração para integrar exatamente a integral abaixo pelos métodos da Integração Polinomial e da Quadratura de *Gauss*.

$$f(x) = 2 + 2x - x^2 + 3x^3$$

$$A = \int_0^4 f(x) \, dx$$

Como o polinômio  $f(x)$  tem grau 3, precisamos de  $n = 4$  pontos para uma Integração Polinomial e  $n = 2$  pontos para obter o valor exato pela Quadratura de *Gauss*.

```
5) f(x) = 2 + 2x - x^2 + 3x^3
[a, b] = [0, 4]
:: Integração Polinomial ::
n = 4
A = ∫f(x) dx ≈ 194.66666666666669
:: Quadratura de Gauss-Legendre ::
n = 2
A = ∫f(x) dx ≈ 194.66666666666669
```

De fato, analiticamente temos

$$A = \int_0^4 2 + 2x - x^2 + 3x^3 \, dx = \left[ 2x + x^2 - \frac{x^3}{3} + \frac{3x^4}{4} \right]_0^4 = \frac{584}{3} = 194.\bar{6}$$

**Questão 6.:** Use os valores da regra do Ponto médio e do Trapézio para estimar um valor mais aproximado para a integral abaixo. Obtenha também, a partir destes dois valores, qual seria o valor da integral caso tivesse sido usada a Regra de *Simpson*. Resolva numericamente esta integral com o programa desenvolvido e compare os valores obtidos.

1 .: Regra do Ponto médio

$$\begin{aligned} A_M &\approx (b-a) \cdot f\left(\frac{b+a}{2}\right) = 3 \cdot f\left(\frac{3}{2}\right) \\ &= 3 \cdot \frac{1}{1+\frac{9}{4}} \\ &= \frac{12}{13} \approx 0.923 \end{aligned}$$

2 .: Regra do Trapézio

$$\begin{aligned} A_T &\approx (b-a) \cdot \frac{f(a)+f(b)}{2} = 3 \cdot \frac{f(0)+f(3)}{2} \\ &= 3 \cdot \frac{\frac{1}{1+0} + \frac{1}{1+9}}{2} \\ &= \frac{3}{2} \cdot \frac{11}{10} = \frac{33}{20} = 1.65 \end{aligned}$$

3 .: Regra de *Simpson*

$$A_S \approx \frac{2}{3} \cdot A_M + \frac{1}{3} \cdot A_T = \frac{2}{3} \cdot \frac{12}{13} + \frac{1}{3} \cdot \frac{33}{20} = \frac{303}{260} \approx 1.16538$$

Calculando numericamente com  $n = 10$  pontos de integração:

```
6) f(x) = 1 / (1 + x^2)
[a, b] = [0, 3]
:: Integração Polinomial ::
A = ∫ f(x) dx ≈ 1.2494163058828742
:: Quadratura de Gauss-Legendre ::
A = ∫ f(x) dx ≈ 1.2490458082502331
:: Método de Romberg ::
A = ∫ f(x) dx ≈ 1.2499809332223779
```

O erro relativo é de aproximadamente  $\frac{|1.249 - 1.165|}{|1.249|} \approx 6.71\%$

**Questão 7.:** A quadratura de *Gauss* conforme apresentada em aula é usada para integrais com limites de integração conhecidos e é também chamada de Quadratura de *Gauss-Legendre*. Para integrais com um ou ambos limites de integração envolvendo  $-\infty$  ou  $\infty$  usa-se a quadratura *Gauss-Hermite*. Pesquise sobre esta técnica e desenvolva uma rotina (similar ao Exercício 1) para resolver as seguintes integrais:

$$A_1 = \int_{-\infty}^1 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

$$A_2 = \int_{-\infty}^{+\infty} \frac{x^2}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

A quadratura de *Gauss-Hermite* se aplica a integrais na forma

$$\int_{-\infty}^{+\infty} K(x)f(x) dx$$

onde dizemos que  $K(x) = e^{-x^2}$  é o núcleo da integral. Mesmo que a função que desejamos integrar não esteja sendo multiplicada por este termo, podemos sempre utilizar a propriedade fundamental da exponenciação  $e^a \cdot e^b = e^{a+b}$  para separar a função deste núcleo. Dizemos que  $1 = e^0 = e^{x^2-x^2} = e^{x^2} \cdot e^{-x^2}$  e assim construímos a função  $\tilde{f}(x) = f(x) \cdot e^{x^2}$ . Com isso concluímos que

$$\int_{-\infty}^{+\infty} f(x) dx = \int_{-\infty}^{+\infty} K(x)\tilde{f}(x) dx$$

**1 .:**  $A_1$

No cálculo de  $A_1$  vamos separar a integral em duas partes, fatorando as constantes:

$$A_1 = \int_{-\infty}^1 \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx$$

$$= \frac{1}{\sqrt{2\pi}} \left[ \int_{-\infty}^0 \exp\left(-\frac{x^2}{2}\right) dx + \int_0^1 \exp\left(-\frac{x^2}{2}\right) dx \right]$$

Em seguida, multiplicamos o primeiro termo por  $e^{x^2} \cdot e^{-x^2} = 1$ , o que não altera o valor da integral:

$$A_1 = \frac{1}{\sqrt{2\pi}} \left[ \int_{-\infty}^0 e^{-x^2} \cdot e^{x^2} \cdot \exp\left(-\frac{x^2}{2}\right) dx + \int_0^1 \exp\left(-\frac{x^2}{2}\right) dx \right]$$

$$= \frac{1}{\sqrt{2\pi}} \left[ \int_{-\infty}^0 e^{-x^2} \cdot \exp\left(\frac{x^2}{2}\right) dx + \int_0^1 \exp\left(-\frac{x^2}{2}\right) dx \right]$$

Por fim, usamos o fato de que ambas as integrais atuam sobre funções pares para usar a seguinte relação:

$$f(x) = f(-x) \quad \forall x \implies \int_{-L}^0 f(x) dx = \frac{1}{2} \int_{-L}^{+L} f(x) dx \quad \forall L \geq 0$$

Portanto,

$$A_1 = \frac{1}{\sqrt{8\pi}} \left[ \int_{-\infty}^{+\infty} e^{-x^2} \cdot \exp\left(\frac{x^2}{2}\right) dx + \int_{-1}^1 \exp\left(-\frac{x^2}{2}\right) dx \right]$$

Agora estamos prontos para calcular a integral do primeiro termo pela quadratura de *Gauss-Hermite* com  $f(x) = \exp(x^2/2)$  assim como a integral do segundo termo pela quadratura de *Gauss-Legendre* com  $f(x) = \exp(-x^2/2)$ . No fim, dividimos o resultado por  $\sqrt{8\pi}$ .

## 2 ∴ $A_2$

A integral  $A_2$ , por sua vez, se encontra mais próxima da forma que se espera para aplicar a quadratura de *Gauss-Hermite*. Multiplicando a integral pelo produto de exponenciais como fizemos anteriormente obtemos:

$$\begin{aligned} A_2 &= \int_{-\infty}^{+\infty} e^{-x^2} \cdot \frac{x^2}{\sqrt{2\pi}} \cdot e^{x^2} \cdot \exp\left(-\frac{x^2}{2}\right) dx \\ &= \int_{-\infty}^{+\infty} e^{-x^2} \cdot \frac{x^2}{\sqrt{2\pi}} \cdot \exp\left(\frac{x^2}{2}\right) dx \end{aligned}$$

Portanto, basta integrar  $f(x) = \frac{x^2}{\sqrt{2\pi}} \cdot \exp\left(\frac{x^2}{2}\right)$  segundo a quadratura de *Gauss-Hermite*.

Com  $n = 10$  pontos de integração obtive os seguintes resultados:

```

7)
n = 10
A1 ~ f(x) = exp(- x^2/2) / sqrt(2 pi)
[a, b] = [-inf, 1]
:: Quadratura de Gauss-Hermite e de Gauss-Legendre ::
A1 = ∫ f(x) dx ≈ 0.84133856560070919
A2 ~ f(x) = x^2 exp(- x^2/2) / sqrt(2 pi)
[a, b] = [-inf, inf]
:: Quadratura de Gauss-Hermite ::
A2 = ∫ f(x) dx ≈ 0.99966839273752539

```



## Complemento - Derivadas Numéricas

**Questão 1.:** Escreva um programa que permita o cálculo numérico da derivada de uma função num ponto  $x$  pelas regras de diferenças finitas:

- a) Diferença central
- b) Passo à frente
- c) Passo atrás

```
1      function d(f, x, dx, kind) result (y)
2          implicit none
3          character (len=*), optional :: kind
4          double precision, optional :: dx
5          character (len=:), allocatable :: t_kind
6          double precision :: x, y, t_dx
7
8          interface
9              function f(x) result (y)
10                  implicit none
11                  double precision :: x, y
12              end function
13          end interface
14
15          if (.NOT. PRESENT(dx)) then
16              t_dx = h
17          else
18              t_dx = dx
19          end if
20
21          if (.NOT. PRESENT(kind)) then
22              t_kind = "central"
23          else
24              t_kind = kind
25          end if
26
27          if (t_kind == "central") then
28              y = (f(x + t_dx) - f(x - t_dx)) / (2 * t_dx)
29          else if (t_kind == "forward") then
30              y = (f(x + t_dx) - f(x)) / t_dx
31          else if (t_kind == "backward") then
32              y = (f(x) - f(x - t_dx)) / t_dx
33          else
34              call error("Unexpected value '//t_kind//' for
35                          derivative kind."// &
```

```

35         "Options are: 'central', 'forward' and 'backward'.")
36     end if
37     return
38 end function

```

**Questão 2.:** Automatize no programa anterior o procedimento de extrapolação de Richard ( $p = 1$  ou  $p = 2$ , a ser escolhido pelo usuário) para melhorar a estimativa da derivada de uma função  $f(x)$  num ponto  $x$  qualquer.

```

1  function richard(f, x, p, q, dx, kind) result (y)
2  !      Richard Extrapolation
3      implicit none
4      double precision, optional :: dx, p, q
5      character(len=*), optional :: kind
6      double precision :: x, y, t_p, t_q, t_dx, dx1, dx2, d1,
7          d2
8      interface
9          function f(x) result (y)
10             implicit none
11             double precision :: x, y
12         end function
13     end interface
14     if (.NOT. PRESENT(dx)) then
15         t_dx = h
16     else
17         t_dx = dx
18     end if
19
20     if (.NOT. PRESENT(p)) then
21         t_p = 1.0D0
22     else
23         t_p = p
24     end if
25
26     if (.NOT. PRESENT(q)) then
27         t_q = 2.0D0
28     else
29         t_q = q
30     end if
31
32     dx1 = t_dx
33     d1 = d(f, x, dx1, kind = kind)
34     dx2 = dx1 / t_q
35     d2 = d(f, x, dx2, kind = kind)
36
37     y = d1 + (d1 - d2) / ((t_q ** (-t_p)) - 1.0D0)
38     return
39 end function

```

**Questão 3.:** Utilizando os programas desenvolvidos nas Tarefas 1 e 2, calcule as derivadas das seguintes funções nos pontos indicados e compare com os valores analíticos.

1.  $f(x) = x^3 + e^{-x}$ ;  $x = 3$ ;
2.  $f(x) = x^{1/3} + \log(x)$ ;  $x = 2$ ;
3.  $f(x) = 1 - \exp(-x^2/25)$ ;  $x = 6$ ;

Nos resultados vemos os valores aproximados por cada modalidade de derivada, seguidos pelo erro  $|\delta y|$ , calculado em relação ao valor da derivada analítica.

```

1)
f(x) = x3 + exp(-x)
f'(x) = 3 x2 - exp(-x)
:: Derivada Analítica ::
f'(3) = 26.950212931632137
:: Diferenças Finitas ::

:: Diferença Central (Δx = 1E-2)::
f'(3) ≈ 26.950212931632137
|δy| = 9.9170210795307412E-005
:: Passo à frente (Δx = 1E-2)::
f'(3) ≈ 26.950212931632137
|δy| = 9.0348107627221452E-002
:: Passo atrás (Δx = 1E-2)::
f'(3) ≈ 26.950212931632137
|δy| = 9.0149767205630837E-002

:: Extrapolação de Richard ::
:: Diferença Central (Δx = 1E-2, p = 1)::
f'(3) ≈ 26.950212931632137
|δy| = 4.9585105180938172E-005
:: Diferença Central (Δx = 1E-2, p = 2)::
f'(3) ≈ 26.950212931632137
|δy| = 1.4566126083082054E-013
:: Passo à frente (Δx = 1E-2, p = 1)::
f'(3) ≈ 26.950212931632137
|δy| = 4.9586659315536963E-005
:: Passo à frente (Δx = 1E-2, p = 2)::
f'(3) ≈ 26.950212931632137
|δy| = 3.0082978102864644E-002
:: Passo atrás (Δx = 1E-2, p = 1)::
f'(3) ≈ 26.950212931632137
|δy| = 4.9583551046339380E-005
:: Passo atrás (Δx = 1E-2, p = 2)::
f'(3) ≈ 26.950212931632137
|δy| = 3.0082978102573321E-002

2)
f(x) = 3√x + log(x)
f'(x) = 1 / (3 3√x2) + (1 / x)

```

```

:: Derivada Analítica ::
f'(2) = 0.70998684164914549
:: Diferenças Finitas ::

:: Diferença Central ( $\Delta x = 1E-2$ )::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 5.1389023598691352E-006$ 
:: Passo à frente ( $\Delta x = 1E-2$ )::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 1.5948580328932760E-003$ 
:: Passo atrás ( $\Delta x = 1E-2$ )::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 1.6051358376130143E-003$ 

:: Extrapolação de Richard ::
:: Diferença Central ( $\Delta x = 1E-2$ , p = 1)::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 2.5694791288000118E-006$ 
:: Diferença Central ( $\Delta x = 1E-2$ , p = 2)::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 1.8632539955376615E-011$ 
:: Passo à frente ( $\Delta x = 1E-2$ , p = 1)::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 2.5553296916225321E-006$ 
:: Passo à frente ( $\Delta x = 1E-2$ , p = 2)::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 5.3332289742547001E-004$ 
:: Passo atrás ( $\Delta x = 1E-2$ , p = 1)::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 2.5836285659774916E-006$ 
:: Passo atrás ( $\Delta x = 1E-2$ , p = 2)::
f'(2)  $\approx$  0.70998684164914549
 $|\delta y| = 5.3332286016039010E-004$ 

3)
f(x) = 1 - exp(-x2 / 25)
f'(x) = (2 x / 25) exp(-x2 / 25)
:: Derivada Analítica ::
f'(6) = 0.11372532416741847
:: Diferenças Finitas ::

:: Diferença Central ( $\Delta x = 1E-2$ )::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 1.8196394321878806E-008$ 
:: Passo à frente ( $\Delta x = 1E-2$ )::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 1.7818749274416124E-004$ 
:: Passo atrás ( $\Delta x = 1E-2$ )::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 1.7815109995551748E-004$ 

:: Extrapolação de Richard ::
:: Diferença Central ( $\Delta x = 1E-2$ , p = 1)::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 9.0983055117677125E-009$ 
:: Diferença Central ( $\Delta x = 1E-2$ , p = 2)::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 7.2233885539674247E-014$ 

```

```

:: Passo à frente ( $\Delta x = 1E-2$ ,  $p = 1$ ) ::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 8.8146768356667238E-009$ 
:: Passo à frente ( $\Delta x = 1E-2$ ,  $p = 2$ ) ::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 5.9389954463501260E-005$ 
:: Passo atrás ( $\Delta x = 1E-2$ ,  $p = 1$ ) ::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 9.3819341878687013E-009$ 
:: Passo atrás ( $\Delta x = 1E-2$ ,  $p = 2$ ) ::
f'(6)  $\approx$  0.11372532416741847
 $|\delta y| = 5.9389954607969031E-005$ 

```

# Appendices

## Código - Programa Principal

```
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
```

```

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// ":: Complemento ::"//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 =  $\int f(x) dx \approx$  "//DSTR(s))
78     call info(":: Quadratura de Gauss-Legendre ::")
79     s = num_int(f6, a, b, n, kind="gauss-legendre")
80     call blue("I1 =  $\int f(x) dx \approx$  "//DSTR(s))
81     call info(":: Método de Romberg ::")
82     s = num_int(f6, a, b, n, kind="romberg")
83     call blue("I1 =  $\int f(x) dx \approx$  "//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 =  $\int f(x) dx \approx$  "//DSTR(s))
91     call info(":: Quadratura de Gauss-Legendre ::")
92     s = num_int(f6, a, b, n, kind="gauss-legendre")
93     call blue("I2 =  $\int f(x) dx \approx$  "//DSTR(s))
94     call info(":: Método de Romberg ::")
95     s = num_int(f6, a, b, n, kind="romberg")
96     call blue("I2 =  $\int f(x) dx \approx$  "//DSTR(s))
97
98 end subroutine
99
100 subroutine Q3

```

```

101     implicit none
102     integer :: n
103     double precision :: a, b, r
104     double precision, dimension(INT_N) :: x
105     double precision, dimension(4, INT_N) :: y
106
107     type(StringArray), dimension(:), allocatable :: legend, with
108
109     allocate(legend(4), with(4))
110
111     legend(1)%str = 'Polinomial'
112     legend(2)%str = 'Gauss-Legendre'
113     legend(3)%str = 'Romberg'
114     legend(4)%str = 'Adaptativo (Gauss)'
115
116     with(1)%str = 'linespoints'
117     with(2)%str = 'linespoints'
118     with(3)%str = 'linespoints'
119     with(4)%str = 'lines'
120
121     a = 0.00D0
122     b = 10.00D0
123
124     call info(ENDL// "3" //ENDL//F7_NAME)
125
126     call info("[a, b] = [ " //DSTR(a) // ", " //DSTR(b) // "]" )
127
128     INT_N = 128
129
130     XMIN = 1.0D0
131     XMAX = INT_N
132     YMIN = -100.0D0
133     YMAX = 300.0D0
134
135     x = (/ (n, n=1, INT_N) /)
136
137     call begin_plot(fname='L5-Q3')
138
139     call subplots(2, 1)
140
141     call info(ENDL// "m0 ~ " //F7a_NAME//ENDL)
142
143     r = adapt_int(f7a, a, b, INT_N, tol=1.0D-8, kind="gauss-
        legendre")
144
145     call info(":: Valor de referência (Integração Adaptativa)
        tol = 1E-8 ::")
146     call blue("m0 =  $\int S\sigma(\omega) d\omega \approx$  " //DSTR(r))
147
148     do n = 1, INT_N
149         y(:, n) = (/ &
150             num_int(f7a, a, b, n, kind="polynomial"), &
151

```



```

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 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(1, 10)))
162 call info(":: Quadratura de Gauss-Legendre ::")
163 call blue("m0 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(2, 10)))
164 call info(":: Método de Romberg ::")
165 call blue("m0 =  $\int S\sigma(\omega) d\omega \approx$  "//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 =  $\int S\sigma(\omega) d\omega \approx$  '//DSTR(r)
172     , xlabel='n', ylabel='m0', grid=.TRUE., &
173     legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=
174     YMIN, ymax=YMAX)
175
176 call info(ENDL//"m2 ~ "//F7b_NAME//ENDL)
177
178 r = adapt_int(f7b, a, b, INT_N, tol=1.0D-8, kind="gauss-
179     legendre")
180
181 call info(":: Valor de referência (Integração Adaptativa)
182     tol = 1E-8 ::")
183 call blue("m2 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(r))
184
185 do n = 1, INT_N
186     y(:, n) = (/ &
187         num_int(f7b, a, b, n, kind="polynomial"), &
188         !
189         num_int(f7b, a, b, n, kind="gauss-legendre"), &
190         !
191         num_int(f7b, a, b, n, kind="romberg"), &
192         !
193         r &
194     /)
195 end do
196
197 call info(":: Integração Polinomial ::")
198 call blue("m2 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(1, 10)))
199 call info(":: Quadratura de Gauss-Legendre ::")
200 call blue("m2 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(2, 10)))
201 call info(":: Método de Romberg ::")
202 call blue("m2 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(3, 10)))
203
204 do n = 1, 4

```

```

201      call subplot(2, 1, x, y(n, :), INT_N)
202  end do
203
204      call subplot_config(2, 1, title='m2 =  $\int \omega^2 S\sigma(\omega) d\omega \approx$  '//DSTR
205          (r), xlabel='n', ylabel='m2', grid=.TRUE., &
206          legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=
207          YMIN, ymax=YMAX)
208
209      call render_plot(clean=.TRUE.)
210  end subroutine
211
212  subroutine Q4
213      implicit none
214      integer :: n
215      double precision :: a, b, r
216      double precision, dimension(INT_N) :: x
217      double precision, dimension(4, INT_N) :: y
218
219      type(StringArray), dimension(:), allocatable :: legend, with
220
221      allocate(legend(4), with(4))
222
223      legend(1)%str = 'Polinomial'
224      legend(2)%str = 'Gauss-Legendre'
225      legend(3)%str = 'Romberg'
226      legend(4)%str = 'Adaptativo (Gauss)'
227
228      with(1)%str = 'linespoints'
229      with(2)%str = 'linespoints'
230      with(3)%str = 'linespoints'
231      with(4)%str = 'lines'
232
233      a = 0.00D0
234      b = 10.00D0
235
236      call info(ENDL// "4" //ENDL//F8_NAME)
237
238      call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
239
240      INT_N = 128
241
242      XMIN = 1.0D0
243      XMAX = INT_N
244      YMIN = -10.0D0
245      YMAX = 100.0D0
246
247      x = (/ (n, n=1, INT_N) /)
248
249      call begin_plot(fname='L5-Q4')
250
251      call subplots(2, 1)
252
253      call info(ENDL// "m0 ~ " //F8a_NAME//ENDL)

```

```

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 =  $\int S\sigma(\omega) d\omega \approx$  "//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 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(1, 10)))
272     call info(":: Quadratura de Gauss-Legendre ::")
273     call blue("m0 =  $\int S\sigma(\omega) d\omega \approx$  "//DSTR(y(2, 10)))
274     call info(":: Método de Romberg ::")
275     call blue("m0 =  $\int S\sigma(\omega) d\omega \approx$  "//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 =  $\int S\sigma(\omega) d\omega \approx$  '//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 =  $\int S\sigma(\omega) d\omega \approx$  "//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             !

```

```

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

```

```

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 =  $\int f(x) dx \approx$  "//DSTR(s))
359     call info(":: Quadratura de Gauss-Legendre ::")
360     s = num_int(f10, a, b, n, kind="gauss-legendre")
361     call blue("A =  $\int f(x) dx \approx$  "//DSTR(s))
362     call info(":: Método de Romberg ::")
363     s = num_int(f10, a, b, n, kind="romberg")
364     call blue("A =  $\int f(x) dx \approx$  "//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-
383             Legendre ::")
384     r = num_int(f11a, a, -a, n, kind="gauss-hermite")
385     s = num_int(f11b, -b, b, n, kind="gauss-legendre")
386     call blue("A1 =  $\int f(x) dx \approx$  "//DSTR(r+s))
387
388     call info("A2 ~ "//F12_NAME)
389     a = DNINF
390     b = DINF
391     call info("[a, b] = [("//DSTR(a)//", "//DSTR(b)//"]")
392     call info(":: Quadratura de Gauss-Hermite ::")
393     s = num_int(f12, a, b, n, kind="gauss-hermite")
394     call blue("A2 =  $\int f(x) dx \approx$  "//DSTR(s))
395 end subroutine
396
397 subroutine QE1
398     implicit none
399     double precision :: x, y, dy
400
401     x = 3.0D0

```

```

402     call info(ENDL// '1')
403
404     call info(FL5_QE1_NAME)
405     call info(DFL5_QE1_NAME)
406
407     call info(':: Derivada Analítica ::')
408     dy = DFL5_QE1(x)
409     call blue("f'("//DSTR(x)//") = "//DSTR(dy))
410
411     call info(":: Diferenças Finitas ::"//ENDL)
412
413     call info(':: Diferença Central ( $\Delta x = 1E-2$ )::')
414     y = d(FL5_QE1, x, dx=1.0D-2, kind='central')
415     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
416     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
417
418     call info(':: Passo à frente ( $\Delta x = 1E-2$ )::')
419     y = d(FL5_QE1, x, dx=1.0D-2, kind='forward')
420     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
421     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
422
423     call info(':: Passo atrás ( $\Delta x = 1E-2$ )::')
424     y = d(FL5_QE1, x, dx=1.0D-2, kind='backward')
425     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
426     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
427
428     call info(ENDL//":: Extrapolação de Richard ::")
429
430     call info(':: Diferença Central ( $\Delta x = 1E-2$ ,  $p = 1$ )::')
431     y = richard(FL5_QE1, x, dx=1.0D-2, kind='central')
432     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
433     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
434
435     call info(':: Diferença Central ( $\Delta x = 1E-2$ ,  $p = 2$ )::')
436     y = richard(FL5_QE1, x, dx=1.0D-2, p=2.0D0, kind='central')
437     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
438     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
439
440     call info(':: Passo à frente ( $\Delta x = 1E-2$ ,  $p = 1$ )::')
441     y = richard(FL5_QE1, x, dx=1.0D-2, kind='forward')
442     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
443     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
444
445     call info(':: Passo à frente ( $\Delta x = 1E-2$ ,  $p = 2$ )::')
446     y = richard(FL5_QE1, x, dx=1.0D-2, p=2.0D0, kind='forward')
447     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
448     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
449
450     call info(':: Passo atrás ( $\Delta x = 1E-2$ ,  $p = 1$ )::')
451     y = richard(FL5_QE1, x, dx=1.0D-2, kind='backward')
452     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
453     call blue("| $\delta y$ | = "//DSTR(DABS(y - dy)))
454

```

```

455     call info(':: Passo atrás ( $\Delta x = 1E-2$ ,  $p = 2$ )::')
456     y = richard(FL5_QE1, x, dx=1.0D-2, p=2.0D0, kind='backward')
457     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
458     call blue("/ $\delta y$ | = "//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 ( $\Delta x = 1E-2$ )::')
480     y = d(FL5_QE2, x, dx=1.0D-2, kind='central')
481     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
482     call blue("/ $\delta y$ | = "//DSTR(DABS(y - dy)))
483
484     call info(':: Passo à frente ( $\Delta x = 1E-2$ )::')
485     y = d(FL5_QE2, x, dx=1.0D-2, kind='forward')
486     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
487     call blue("/ $\delta y$ | = "//DSTR(DABS(y - dy)))
488
489     call info(':: Passo atrás ( $\Delta x = 1E-2$ )::')
490     y = d(FL5_QE2, x, dx=1.0D-2, kind='backward')
491     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
492     call blue("/ $\delta y$ | = "//DSTR(DABS(y - dy)))
493
494     call info(ENDL// ":: Extrapolação de Richard ::")
495
496     call info(':: Diferença Central ( $\Delta x = 1E-2$ ,  $p = 1$ )::')
497     y = richard(FL5_QE2, x, dx=1.0D-2, kind='central')
498     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
499     call blue("/ $\delta y$ | = "//DSTR(DABS(y - dy)))
500
501     call info(':: Diferença Central ( $\Delta x = 1E-2$ ,  $p = 2$ )::')
502     y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='central')
503     call blue("f'("//DSTR(x)//")  $\approx$  "//DSTR(dy))
504     call blue("/ $\delta y$ | = "//DSTR(DABS(y - dy)))
505
506     call info(':: Passo à frente ( $\Delta x = 1E-2$ ,  $p = 1$ )::')
507     y = richard(FL5_QE2, x, dx=1.0D-2, kind='forward')

```

```

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

```



```

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

```

## Código - Definição das Funções

```

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

```

```

16      character (len = *), parameter :: F3_NAME = "f(x, y, z) :="
17          //ENDL// &
18      "16x4 + 16y4 + z4 = 16"//ENDL// &
19      "x2 + y2 + x2 = 3"//ENDL// &
20      "x3 - y + z = 1"
21      integer :: F3_N = 3
22  !
23      >> F4 <<
24      character (len = *), parameter :: F4_NAME = "f(c2, c3, c4)
25          :="//ENDL// &
26      "c22 + 2 c32 + 6 c42 = 1"//ENDL// &
27      "8 c33 + 6 c3 c22 + 36 c3 c2 c4 + 108 c3 c44 =  $\theta_1$ "//ENDL// &
28      "60 * c34 + 60 * c32 * c22 + 576 * c32 * c2 * c4 + //" &
29      "2232 * c32 * c42 + 252 * c42 * c22 + //" &
30      "1296 * c43 c2 + 3348 c44 + 24 c23 c4 + 3 c2 =  $\theta_2$ "
31      double precision :: F4_TT1(3) = (/ 0.0D0, 0.75D0, 0.000D0
32          /)
33      double precision :: F4_TT2(3) = (/ 3.0D0, 6.50D0, 11.667D0
34          /)
35      double precision :: F4_T1 = 0.0D0
36      double precision :: F4_T2 = 0.0D0
37      integer :: F4_N = 3
38  !
39      >> F5 <<
40      character (len = *), parameter :: F5_NAME = "f(x) = b1 + b2
41          x~b3"
42      integer :: F5_N = 3
43  !
44      >> F6 <<
45      character (len = *), parameter :: F6_NAME = "f(x) = exp(-x2
46          /2) /  $\sqrt{(2 \pi)}$ "
47  !
48      >> F7 <<
49      character (len = *), parameter :: F7_NAME = "S $\sigma(\omega)$  = RAO( $\omega$ )2
50          S $\eta(\omega)$ "//ENDL//TAB// &
51      "RAO( $\omega$ ) = 1 /  $\sqrt{((1 - (\omega/\omega_n)^2)^2 + (2\xi\omega/\omega_n)^2)}$ "
52
53      character (len = *), parameter :: F7a_NAME = "S $\eta(\omega)$  = 2"
54      character (len = *), parameter :: F7b_NAME = "S $\eta(\omega)$  = 2"
55  !
56      >> F8 <<
57      character (len = *), parameter :: F8_NAME = "S $\sigma(\omega)$  = RAO( $\omega$ )2
58          S $\eta(\omega)$ "//ENDL//TAB// &
59      "RAO( $\omega$ ) = 1 /  $\sqrt{((1 - (\omega/\omega_n)^2)^2 + (2\xi\omega/\omega_n)^2)}$ "
60
61      character (len = *), parameter :: F8a_NAME = "S $\eta(\omega)$  = ((4  $\pi^3$ 
62          Hs2) / ( $\omega^5$  Tz4)) exp(-(16  $\pi^3$ ) / ( $\omega^4$  Tz4))"
63      character (len = *), parameter :: F8b_NAME = "S $\eta(\omega)$  = ((4  $\pi^3$ 
64          Hs2) / ( $\omega^5$  Tz4)) exp(-(16  $\pi^3$ ) / ( $\omega^4$  Tz4))"
65
66      >> F13 <<

```

```

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 -
      x2 + 3x3"
101
102     character (len = *), parameter :: F10_NAME = "f(x) = 1 / (1
      + x2)"
103
104     character (len = *), parameter :: F11_NAME = "f(x) = exp(- x
      2/2) / sqrt(2 pi)"

```

```

105         character (len = *), parameter :: F12_NAME = "f(x) = x2 exp  

           (- x2/2) / √(2 π)"
106
107 !      >> L5-QE <<
108         character (len = *), parameter :: FL5_QE1_NAME = 'f(x) = x3  

           + exp(-x)'
109         character (len = *), parameter :: DFL5_QE1_NAME = "f'(x) = 3  

           x2 - exp(-x)"
110         character (len = *), parameter :: FL5_QE2_NAME = 'f(x) = 3√x  

           + log(x)'
111         character (len = *), parameter :: DFL5_QE2_NAME = "f'(x) = 1  

           / (3 3√x2) + (1 / x)"
112         character (len = *), parameter :: FL5_QE3_NAME = 'f(x) = 1 -  

           exp(-x2 / 25)'
113         character (len = *), parameter :: DFL5_QE3_NAME = "f'(x) =  

           (2 x / 25) exp(-x2 / 25)"
114
115
116 contains
117
118 function FL5_QE1(x) result (y)
119     implicit none
120     double precision :: x, y
121     y = x ** 3 + DEXP(-x)
122     return
123 end function
124
125 function DFL5_QE1(x) result (y)
126     implicit none
127     double precision :: x, y
128     y = 3 * x ** 2 - DEXP(-x)
129     return
130 end function
131
132 function FL5_QE2(x) result (y)
133     implicit none
134     double precision :: x, y
135     y = x ** (1.0D0/3.0D0) + DLOG(x)
136     return
137 end function
138
139 function DFL5_QE2(x) result (y)
140     implicit none
141     double precision :: x, y
142     y = 1 / (3 * x ** (2.0D0/3.0D0)) + (1 / x)
143     return
144 end function
145
146 function FL5_QE3(x) result (y)
147     implicit none
148     double precision :: x, y
149     y = 1 - DEXP(-(x ** 2) / 25)
150     return

```

```

151     end function
152
153     function DFL5_QE3(x) result (y)
154         implicit none
155         double precision :: x, y
156         y = (2 * X) / (25 * DEXP((x ** 2) / 25))
157         return
158     end function
159
160
161     function f1(x) result (y)
162         implicit none
163         double precision :: x, y
164         y = DLOG(DCOSH(x * DSQRT(F1_G * F1_K))) - 50.0D0
165         return
166     end function
167
168     function df1(x) result (y)
169         implicit none
170         double precision :: x, y
171         y = (DSINH(x * DSQRT(F1_G * F1_K)) * DSQRT(F1_G * F1_K)) /
172             DCOSH(x * DSQRT(F1_G * F1_K))
173         return
174     end function
175
176     function f2(x) result (y)
177         implicit none
178         double precision :: x, y
179         y = 4 * DCOS(x) - DEXP(2 * x)
180         return
181     end function
182
183     function df2(x) result (y)
184         implicit none
185         double precision :: x, y
186         y = - 4 * DSIN(x) - 2 * DEXP(2 * x)
187         return
188     end function
189
190     ! ===== R^n -> R^n functions =====
191     function f3(x, n) result (y)
192         ! R^3 -> R^3 (n == 3)
193         implicit none
194         integer :: n
195         double precision, dimension(n) :: x, y
196
197         y = (/ &
198             (16 * x(1) ** 4 + 16 * x(2) ** 4 + x(3) ** 4) - 16.0D0,
199             &
200             x(1) ** 2 + x(2) ** 2 + x(3) ** 2 - 3.0D0, &
201             x(1) ** 3 - x(2) + x(3) - 1.0D0 &
202             /)
203         return

```

```

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                /)
213     J(2, :) = (/ 2 * x(1)      , 2 * x(2)      , 2 * x(3)
214                /)
215     J(3, :) = (/ 3 * x(1) ** 2,      -1.0D0,      1.0D0
216                /)
217     return
218 end function
219
220 ! ===== Another function =====
221 function f4(x, n) result (y)
222     implicit none
223     integer :: n
224     double precision, dimension(n) :: x, y
225     y = (/ &
226          x(1)**2+2*x(2)**2+6*x(3)**2, &
227          2*x(2)*(3*x(1)**2+4*x(2)**2+18*x(1)*x(3)+54*x(3)**4), &
228          3*(x(1)+20*x(1)**2*x(2)**2+20*x(2)**4+8*x(1)*(x(1)
229              **2+24*x(2)**2)*x(3))+&
230          12*(7*x(1)**2+62*x(2)**2)*x(3)**2+432*x(1)*x(3)**3+1116*
231              x(3)**4)&
232          /) - (/ 1.0D0, F4_T1, F4_T2 /)
233     return
234 end function
235
236 ! ===== Derivatives =====
237 function df4(x, n) result (J)
238 !  $R^3 \rightarrow R^3 \otimes^3 (n == 3)$ 
239     implicit none
240     integer :: n
241     double precision :: x(n), J(n, n)
242
243     J(1, :) = (/ &
244                2*x(1), &
245                4*x(2), &
246                12*x(3) &
247                /)
248     J(2, :) = (/ &
249                12*x(1)*x(2)+36*x(2)*x(3), &
250                6*x(1)**2+24*x(2)**2+36*x(1)*x(3)+108*x(3)**4, &
251                36*x(1)*x(2)+432*x(2)*x(3)**3 &
252                /)
253     J(3, :) = (/ &

```

```

249      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,      &
250      120*x(1)**2*x(2)+240*x(2)**3+1152*x(1)*x(2)*x(3)+4464*x
      (2)*x(3)**2,      &
251      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 &
252      /)
253      return
254  end function
255
256  ! ===== One more function =====
257  function f5(x, b, m, n) result (z)
258      implicit none
259      integer :: m, n
260      double precision, dimension(m), intent(in) :: b
261      double precision, dimension(n), intent(in) :: x
262      double precision, dimension(n) :: z
263
264      z = b(1) + (b(2) * (x ** b(3)))
265      return
266  end function
267
268  ! ===== Derivatives =====
269  function df5(x, b, m, n) result (J)
270      implicit none
271      integer :: m, n
272      double precision, dimension(m), intent(in) :: b
273      double precision, dimension(n), intent(in) :: x
274      double precision, dimension(n, m) :: J
275      ! m == 3
276      J(:, 1) = 1.0D0
277      J(:, 2) = x ** b(3)
278      J(:, 3) = b(2) * DLOG(x) * (x ** b(3))
279      return
280  end function
281
282  ! ===== Function 6 =====
283  function f6(x) result (y)
284      implicit none
285      double precision :: x, y
286
287      y = DEXP(-(x*x)/2) / DSQRT(2 * PI)
288      return
289  end function
290
291  ! ===== Functions 7 & 8 =====
292  function RAO(w) result (z)
293      implicit none
294      double precision :: w, z
295
296      z = 1.0 / DSQRT((1.0D0 - (w/wn) ** 2) ** 2 + (2 * xi * (w/wn
      ) ** 2)
297  end function

```

```

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)
310             **4) ) * (Tz**4) * (w**5) )
311         return
312     end function
313
314     function Ss(w, Sn) result (z)
315         implicit none
316         double precision :: w, z
317         interface
318             function Sn(w) result (z)
319                 implicit none
320                 double precision :: w, z
321             end function
322         end interface
323         z = (RA0(w) ** 2) * Sn(w)
324         return
325     end function
326
327     function f7a(w) result (z)
328         implicit none
329         double precision :: w, z
330         z = Ss(w, Sn1)
331         return
332     end function
333
334     function f7b(w) result (z)
335         implicit none
336         double precision :: w, z
337         z = (w ** 2) * Ss(w, Sn1)
338         return
339     end function
340
341     function f8a(w) result (z)
342         implicit none
343         double precision :: w, z
344         z = Ss(w, Sn2)
345         return
346     end function
347
348     function f8b(w) result (z)
349         implicit none
350         double precision :: w, z

```



```

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

```

```

403     double precision :: t, y
404     y = 1 / (1 + (t**2))
405     return
406 end function
407
408 ! ===== Function 14 =====
409 function F14_F(t) result (y)
410     implicit none
411     double precision :: t, y
412     y = 2 * DSIN(F14_W * t) + DSIN(2 * F14_W * t) + DCOS(3 *
         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

```

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"

```

```

13 contains
14 ! ===== Numerical Methods =====
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
49             derivative kind."// &
50             "Options are: 'central', 'forward' and 'backward'.")
51     end if
52     return
53 end function
54
55 function dp(f, x, i, n) result (y)
56     implicit none
57     integer :: i, n
58     double precision :: f
59     double precision :: x(n), xh(n)
60     double precision :: y
61
62     xh(:) = 0.0D0
63     xh(i) = h
64
65     y = (f(x + xh) - f(x - xh)) / (2 * h)

```

```

65         return
66     end function
67
68     function grad(f, x, n) result (y)
69         implicit none
70         integer :: i, n
71         double precision :: f
72         double precision :: xh(n), x(n), y(n)
73
74         xh(:) = 0.0D0
75         do i=1, n
76             !      Compute partial derivative with respect to x_i
77             xh(i) = h
78             y(i) = (f(x + xh) - f(x - xh)) / (2 * h)
79             xh(i) = 0.0D0
80         end do
81         return
82     end function
83
84     !      =====
85
86     function lagrange(x0, y0, n, x) result (y)
87         implicit none
88         integer :: n
89         double precision :: x0(n), y0(n)
90         double precision :: x, y, yi
91         integer :: i, j
92
93         y = 0.0D0
94         do i = 1, n
95             yi = y0(i)
96             do j = 1, n
97                 if (i /= j) then
98                     yi = yi * (x - x0(j)) / (x0(i) - x0(j))
99                 end if
100             end do
101             y = y + yi
102         end do
103
104         return
105     end function
106
107     function bisection(f, aa, bb, tol) result (x)
108         implicit none
109         double precision, intent(in) :: aa, bb
110         double precision :: a, b, x, t_tol
111         double precision, optional :: tol
112
113         interface
114             function f(x) result (y)
115                 double precision :: x, y
116             end function
117         end interface

```

```

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

```

```

171         end function
172     end interface
173
174     if (.NOT. PRESENT(max_iter)) then
175         t_max_iter = D_MAX_ITER
176     else
177         t_max_iter = max_iter
178     end if
179
180     if (.NOT. PRESENT(tol)) then
181         t_tol = D_TOL
182     else
183         t_tol = tol
184     end if
185
186     ok = .TRUE.
187     xk = x0
188     do k = 1, t_max_iter
189         x = xk - f(xk) / df(xk)
190         if (DABS(x - xk) > t_tol) then
191             xk = x
192         else
193             if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
194                 then
195                     ok = .FALSE.
196                 end if
197             return
198         end if
199     end do
200     ok = .FALSE.
201     return
202 end function
203
204 function secant(f, x0, ok, tol, max_iter) result (x)
205     implicit none
206     integer :: k, t_max_iter
207     integer, optional :: max_iter
208     double precision :: xk(3), yk(2)
209     double precision, intent(in) :: x0
210     double precision :: x, t_tol
211     double precision, optional :: tol
212     logical, intent(out) :: ok
213     interface
214         function f(x) result (y)
215             implicit none
216             double precision :: x, y
217         end function
218     end interface
219
220     if (.NOT. PRESENT(max_iter)) then
221         t_max_iter = D_MAX_ITER
222     else
223         t_max_iter = max_iter

```

```

223     end if
224
225     if (.NOT. PRESENT(tol)) then
226         t_tol = D_TOL
227     else
228         t_tol = tol
229     end if
230
231     ok = .TRUE.
232
233     xk(1) = x0
234     xk(2) = x0 + h
235     yk(1) = f(xk(1))
236     do k = 1, t_max_iter
237         yk(2) = f(xk(2))
238         xk(3) = xk(2) - (yk(2) * (xk(2) - xk(1))) / (yk(2) -
239             yk(1))
240         if (DABS(xk(3) - xk(2)) > t_tol) then
241             xk(1:2) = xk(2:3)
242             yk(1) = yk(2)
243         else
244             x = xk(3)
245             if (ISNAN(x) .OR. x == DINF .OR. x == DNINF)
246                 then
247                 ok = .FALSE.
248             end if
249             return
250         end if
251     end do
252     ok = .FALSE.
253     return
254 end function
255
256 function inv_interp(f, x00, ok, tol, max_iter) result (x)
257     implicit none
258     logical, intent(out) :: ok
259     integer :: i, j(1), k, t_max_iter
260     integer, optional :: max_iter
261     double precision :: x, xk, t_tol
262     double precision, optional :: tol
263     double precision, intent(in) :: x00(3)
264     double precision :: x0(3), y0(3)
265
266     interface
267         function f(x) result (y)
268             double precision :: x, y
269         end function
270     end interface
271
272     if (.NOT. PRESENT(max_iter)) then
273         t_max_iter = D_MAX_ITER
274     else
275         t_max_iter = max_iter
276     end if

```

```

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

```



```

325     interface
326         function ff(x, n) result (y)
327             implicit none
328             integer :: n
329             double precision :: x(n), y(n)
330         end function
331     end interface
332
333     interface
334         function dff(x, n) result (J)
335             implicit none
336             integer :: n
337             double precision :: x(n), J(n, n)
338         end function
339     end interface
340
341     if (.NOT. PRESENT(max_iter)) then
342         t_max_iter = D_MAX_ITER
343     else
344         t_max_iter = max_iter
345     end if
346
347     if (.NOT. PRESENT(tol)) then
348         t_tol = D_TOL
349     else
350         t_tol = tol
351     end if
352
353     ok = .TRUE.
354
355     x = x0
356
357     do k=1, t_max_iter
358         J = dff(x, n)
359         dx = -MATMUL(inv(J, n, ok), ff(x, n))
360         xdx = x + dx
361
362         if (.NOT. ok) then
363             exit
364         else if ((NORM(dx, n) / NORM(xdx, n)) > t_tol) then
365             x = xdx
366         else
367             if (VEDGE(x)) then
368                 ok = .FALSE.
369             end if
370             return
371         end if
372     end do
373     ok = .FALSE.
374     return
375 end function
376

```

```

377         function sys_newton_num(ff, x0, n, ok, tol, max_iter) result
378             (x)
379             ! Same as previous function, with numerical partial
380             derivatives
381             implicit none
382             logical, intent(out) :: ok
383             integer :: n, i, k, t_max_iter
384             integer, optional :: max_iter
385             double precision, dimension(n), intent(in) :: x0
386             double precision, dimension(n):: x, xdx, xh, dx
387             double precision, dimension(n, n) :: J
388             double precision :: t_tol
389             double precision, optional :: tol
390
391             interface
392                 function ff(x, n) result (y)
393                     implicit none
394                     integer :: n
395                     double precision :: x(n), y(n)
396                 end function
397             end interface
398
399             if (.NOT. PRESENT(max_iter)) then
400                 t_max_iter = D_MAX_ITER
401             else
402                 t_max_iter = max_iter
403             end if
404
405             if (.NOT. PRESENT(tol)) then
406                 t_tol = D_TOL
407             else
408                 t_tol = tol
409             end if
410
411             ok = .TRUE.
412
413             x = x0
414             xh = 0.0D0
415
416             do k=1, t_max_iter
417                 ! Compute Jacobian Matrix
418                 do i=1, n
419                     ! Partial derivative with respect to the i-th
420                     coordinates
421                     xh(i) = h
422                     J(:, i) = (ff(x + xh, n) - ff(x - xh, n)) / (2 *
423                             h)
424                     xh(i) = 0.0D0
425                 end do
426
427                 dx = -MATMUL(inv(J, n, ok), ff(x, n))
428                 xdx = x + dx

```

```

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)
442     result (x)
443     implicit none
444     logical, intent(out) :: ok
445     integer :: n, k, t_max_iter
446     integer, optional :: max_iter
447     double precision, dimension(n), intent(in) :: x0
448     double precision, dimension(n, n), intent(in) :: B0
449     double precision, dimension(n) :: x, xdx, dx, dff
450     double precision, dimension(n, n) :: J
451     double precision :: t_tol
452     double precision, optional :: tol
453
454     interface
455         function ff(x, n) result (y)
456             implicit none
457             integer :: n
458             double precision :: x(n), y(n)
459         end function
460     end interface
461
462     if (.NOT. PRESENT(max_iter)) then
463         t_max_iter = D_MAX_ITER
464     else
465         t_max_iter = max_iter
466     end if
467
468     if (.NOT. PRESENT(tol)) then
469         t_tol = D_TOL
470     else
471         t_tol = tol
472     end if
473
474     ok = .TRUE.
475
476     x = x0
477     J = B0

```

```

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

```

```

528         end interface
529
530         if (.NOT. PRESENT(max_iter)) then
531             t_max_iter = D_MAX_ITER
532         else
533             t_max_iter = max_iter
534         end if
535
536         if (.NOT. PRESENT(tol)) then
537             t_tol = D_TOL
538         else
539             t_tol = tol
540         end if
541
542         ok = .TRUE.
543
544         b = b0
545
546         do k=1, t_max_iter
547             J = dff(x, b, m, n)
548             db = -MATMUL(inv(MATMUL(TRANPOSE(J), J), n, ok),
549                          MATMUL(TRANPOSE(J), ff(x, b, m, n) - y))
550             bdb = b + db
551
552             if (.NOT. ok) then
553                 exit
554             else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
555                 b = bdb
556             else
557                 if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
558                                t_tol)) then
559                     ok = .FALSE.
560                 end if
561                 return
562             end if
563         end do
564         ok = .FALSE.
565         return
566     end function
567
568     function sys_least_squares_num(ff, x, y, b0, m, n, ok, tol,
569                                   max_iter) result (b)
570         ! Same as previous function, with numerical partial
571         derivatives
572         implicit none
573         integer :: m, n, i, k, t_max_iter
574         integer, optional :: max_iter
575         double precision, dimension(n), intent(in) :: x, y, b0
576         double precision, dimension(n) :: b, bdb, db, bh
577         double precision :: J(n, n)
578         double precision :: t_tol
579         double precision, optional :: tol

```

```

577     logical, intent(out) :: ok
578     interface
579         function ff(x, b, m, n) result (z)
580             implicit none
581             integer :: m, n
582             double precision, dimension(n), intent(in) :: x
583             double precision, dimension(m), intent(in) :: b
584             double precision, dimension(n) :: z
585         end function
586     end interface
587
588     if (.NOT. PRESENT(max_iter)) then
589         t_max_iter = D_MAX_ITER
590     else
591         t_max_iter = max_iter
592     end if
593
594     if (.NOT. PRESENT(tol)) then
595         t_tol = D_TOL
596     else
597         t_tol = tol
598     end if
599
600     ok = .TRUE.
601
602     bh = 0.0D0
603     b = b0
604
605     do k=1, t_max_iter
606         ! Compute Jacobian Matrix
607         do i=1, m
608             ! Partial derivative with respect do the i-th
        coordinates
609             bh(i) = h
610             J(:, i) = (ff(x, b + bh, m, n) - ff(x, b - bh, m
611                 , n)) / (2 * h)
612             bh(i) = 0.0D0
613         end do
614         db = -MATMUL(inv(MATMUL(TRANPOSE(J), J), n, ok),
615             MATMUL(TRANPOSE(J), ff(x, b, m, n) - y))
616         bdb = b + db
617
618         if (.NOT. ok) then
619             exit
620         else if ((NORM(db, m) / NORM(bdb, m)) > t_tol) then
621             b = bdb
622         else
623             if (VEDGE(b) .OR. (NORM(ff(x, b, m, n) - y, n) >
624                 t_tol)) then
625                 ok = .FALSE.
626             end if
627             return
628         end if
629     end do
630 end if

```

```

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                 &
675                 "Available options are: 'polynomial', 'gauss-legendre', 'gauss-hermite' and 'romberg'.")
676         end if
677     end function
678
679

```

```

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 =
710             GAUSS_LEGENDRE_QUAD
711         interface
712             function f(x) result (y)
713                 double precision :: x, y
714             end function
715         end interface
716
717         call load_quad(xx, ww, n, fname//STR(n)//".txt")
718
719         xx(:) = ((b - a) * xx(:) + (b + a)) / 2
720         s = 0.0D0
721         do k=1, n
722             s = s + (ww(k) * f(xx(k)))
723         end do
724         s = s * ((b - a) / 2)
725         return
726     end function
727
728     function gauss_hermite_int(f, a, b, n) result (s)
729         implicit none

```



```

729 integer, intent(in) :: n
730 double precision, intent(in) :: a, b
731 double precision :: s
732 double precision, dimension(n) :: xx, ww
733 integer :: k
734 character(len=*), parameter :: fname =
      GAUSS_HERMITE_QUAD
735 interface
736     function f(x) result (y)
737         double precision :: x, y
738     end function
739 end interface
740
741 call load_quad(xx, ww, n, fname//STR(n)//".txt")
742
743 if (a /= DNINF .OR. b /= DINF) then
744     call error("O Método de Gauss-Hermite deve ser usado
745               no intervalo dos reais.")
746     stop
747 end if
748
749 s = 0.0D0
750 do k=1, n
751     s = s + (ww(k) * f(xx(k)))
752 end do
753
754 return
755 end function
756
757 recursive function adapt_int(f, a, b, n, tol, kind) result (
758 s)
759     implicit none
760     integer :: n
761     character (len=*), optional :: kind
762     double precision, intent(in) :: a, b
763     double precision :: p, q, e, r, s, t_tol
764     double precision, optional :: tol
765     interface
766         function f(x) result (y)
767             double precision :: x, y
768         end function
769     end interface
770
771 if (.NOT. PRESENT(tol)) then
772     t_tol = D_TOL
773 else
774     t_tol = tol
775 end if
776
777 if (n > 1) then
778     p = num_int(f, a, b, n / 2, kind = kind)
779     q = num_int(f, a, b, n, kind = kind)
780     e = DABS(p - q)

```

```

779         if (e <= t_tol) then
780             s = q
781         else
782             r = (b + a) / 2
783             s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
              + 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)

```

```

829         end do
830
831         if (DABS(R(i + 1, i + 1) - R(i, i)) > t_tol) then
832             continue
833         else
834             exit
835         end if
836     end do
837     s = R(i, i)
838
839     deallocate(R)
840 end function
841
842 function richard(f, x, p, q, dx, kind) result (y)
843 !      Richard Extrapolation
844     implicit none
845     double precision, optional :: dx, p, q
846     character(len=*), optional :: kind
847     double precision :: x, y, t_p, t_q, t_dx, dx1, dx2, d1,
848         d2
849     interface
850         function f(x) result (y)
851             implicit none
852             double precision :: x, y
853         end function
854     end interface
855
856     if (.NOT. PRESENT(dx)) then
857         t_dx = h
858     else
859         t_dx = dx
860     end if
861
862     if (.NOT. PRESENT(p)) then
863         t_p = 1.0D0
864     else
865         t_p = p
866     end if
867
868     if (.NOT. PRESENT(q)) then
869         t_q = 2.0D0
870     else
871         t_q = q
872     end if
873
874     dx1 = t_dx
875     d1 = d(f, x, dx1, kind = kind)
876     dx2 = dx1 / t_q
877     d2 = d(f, x, dx2, kind = kind)
878
879     y = d1 + (d1 - d2) / ((t_q ** (-t_p)) - 1.0D0)
880     return
881 end function

```

```

881
882 ! ===== Ordinary Differential Equations =====
883 function ode_solve(df, y0, t, n, kind) result (y)
884     implicit none
885     integer :: n
886     double precision, intent(in) :: y0
887     double precision, dimension(n), intent(in) :: t
888     double precision, dimension(n) :: y
889     character(len=*), optional :: kind
890     character(len=:), allocatable :: t_kind
891     interface
892         function df(t, y) result (u)
893             implicit none
894             double precision :: t, y, u
895         end function
896     end interface
897
898     if (.NOT. PRESENT(kind)) then
899         t_kind = 'euler'
900     else
901         t_kind = kind
902     end if
903
904     if (t_kind == 'euler') then
905         y = euler(df, y0, t, n)
906     else if (t_kind == 'runge-kutta2') then
907         y = runge_kutta2(df, y0, t, n)
908     else if (t_kind == 'runge-kutta4') then
909         y = runge_kutta4(df, y0, t, n)
910     else
911         call error("As opções são: 'euler', 'runge-kutta2' e '
912             runge-kutta4'".)
913         stop
914     end if
915     return
916 end function
917
918 function euler(df, y0, t, n) result (y)
919     implicit none
920     integer :: k, n
921     double precision, intent(in) :: y0
922     double precision :: dt
923     double precision, dimension(n), intent(in) :: t
924     double precision, dimension(n) :: y
925     interface
926         function df(t, y) result (u)
927             implicit none
928             double precision :: t, y, u
929         end function
930     end interface
931
932     y(1) = y0

```

```

933         do k=2, n
934             dt = t(k) - t(k - 1)
935             y(k) = y(k - 1) + df(t(k - 1), y(k - 1)) * dt
936         end do
937         return
938     end function
939
940     function runge_kutta2(df, y0, t, n) result (y)
941         implicit none
942         integer :: k, n
943         double precision, intent(in) :: y0
944         double precision :: k1, k2, dt
945         double precision, dimension(n), intent(in) :: t
946         double precision, dimension(n) :: y
947         interface
948             function df(t, y) result (u)
949                 implicit none
950                 double precision :: t, y, u
951             end function
952         end interface
953
954         y(1) = y0
955         do k=2, n
956             dt = t(k) - t(k - 1)
957             k1 = df(t(k - 1), y(k - 1))
958             k2 = df(t(k - 1) + dt, y(k - 1) + k1 * dt)
959             y(k) = y(k - 1) + dt * (k1 + k2) / 2
960         end do
961         return
962     end function
963
964     function runge_kutta4(df, y0, t, n) result (y)
965         implicit none
966         integer :: k, n
967         double precision, intent(in) :: y0
968         double precision :: k1, k2, k3, k4, dt
969         double precision, dimension(n), intent(in) :: t
970         double precision, dimension(n) :: y
971         interface
972             function df(t, y) result (u)
973                 implicit none
974                 double precision :: t, y, u
975             end function
976         end interface
977
978         y(1) = y0
979         do k=2, n
980             dt = t(k) - t(k - 1)
981             k1 = df(t(k - 1), y(k - 1))
982             k2 = df(t(k - 1) + dt / 2, y(k - 1) + k1 * dt / 2)
983             k3 = df(t(k - 1) + dt / 2, y(k - 1) + k2 * dt / 2)
984             k4 = df(t(k - 1) + dt, y(k - 1) + dt * k3)
985             y(k) = y(k - 1) + dt * (k1 + 2 * k2 + 2 * k3 + k4) / 6

```

```

986         end do
987         return
988     end function
989
990     function ode2_solve(d2f, y0, dy0, t, n, kind) result (y)
991         implicit none
992         integer :: n
993         double precision, intent(in) :: y0, dy0
994         double precision, dimension(n), intent(in) :: t
995         double precision, dimension(n) :: y
996         character(len=*), optional :: kind
997         character(len=:), allocatable :: t_kind
998         interface
999             function d2f(t, y, dy) result (u)
1000                 implicit none
1001                 double precision :: t, y, dy, u
1002             end function
1003         end interface
1004
1005         if (.NOT. PRESENT(kind)) then
1006             t_kind = 'taylor'
1007         else
1008             t_kind = kind
1009         end if
1010
1011         if (t_kind == 'taylor') then
1012             y = taylor(d2f, y0, dy0, t, n)
1013         else if (t_kind == 'runge-kutta-nystrom') then
1014             y = runge_kutta_nystrom(d2f, y0, dy0, t, n)
1015         else
1016             call error("As opções são: 'taylor', 'runge-kutta-
1017                          nystrom'.")
1018             stop
1019         end if
1020         return
1021     end function
1022
1023     function taylor(d2f, y0, dy0, t, n) result (y)
1024         implicit none
1025         integer :: k, n
1026         double precision, intent(in) :: y0, dy0
1027         double precision :: dt, dy, d2y
1028         double precision, dimension(n), intent(in) :: t
1029         double precision, dimension(n) :: y
1030         interface
1031             function d2f(t, y, dy) result (d2y)
1032                 implicit none
1033                 double precision :: t, y, dy, d2y
1034             end function
1035         end interface
1036         ! Solution
1037         y(1) = y0
1038         ! 1st derivative

```

```

1038     dy = dy0
1039     do k=2, n
1040         dt = t(k) - t(k - 1)
1041         d2y = d2f(t(k - 1), y(k - 1), dy)
1042         y(k) = y(k - 1) + (dy * dt) + (d2y * dt ** 2) / 2
1043         dy = dy + d2y * dt
1044     end do
1045     return
1046 end function

1047
1048 function runge_kutta_nystrom(d2f, y0, dy0, t, n) result (y)
1049     implicit none
1050     integer :: k, n
1051     double precision, intent(in) :: y0, dy0
1052     double precision :: k1, k2, k3, k4, dt, dy, l, q
1053     double precision, dimension(n), intent(in) :: t
1054     double precision, dimension(n) :: y
1055     interface
1056         function d2f(t, y, dy) result (u)
1057             implicit none
1058             double precision :: t, y, dy, u
1059         end function
1060     end interface

1061
1062     y(1) = y0
1063     dy = dy0
1064     do k=2, n
1065         dt = t(k) - t(k - 1)
1066         k1 = (d2f(t(k - 1), y(k - 1), dy) * dt) / 2
1067         q = ((dy + k1 / 2) * dt) / 2
1068         k2 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k1) * dt
1069             ) / 2
1070         k3 = (d2f(t(k - 1) + dt / 2, y(k - 1) + q, dy + k2) * dt
1071             ) / 2
1072         l = (dy + k3) * dt
1073         k4 = (d2f(t(k - 1) + dt, y(k - 1) + l, dy + 2 * k3) * dt)
1074             / 2
1075
1076         y(k) = y(k - 1) + (dy + (k1 + k2 + k3) / 3) * dt
1077         dy = dy + (k1 + 2 * k2 + 2 * k3 + k4) / 3
1078     end do
1079     return
1080 end function

1081 end module Calc

```

## Código - Métodos com Matrizes

```

1  !   Matrix Module
2
3  module Matrix
4      use Util

```

```

5      implicit none
6      integer :: D_MAX_ITER = 1000
7      double precision :: D_TOL = 1.0D-5
8  contains
9      subroutine ill_cond()
10     !      Prompts the user with an ill-conditioning warning.
11     implicit none
12     call error('Matriz mal-condicionada: este método não irá
13             convergir.')
14 end subroutine
15
16 subroutine show_matrix(var, A, m, n)
17     implicit none
18     integer :: m, n
19     character(len=*) :: var
20     double precision, dimension(m, n), intent(in) :: A
21     write (*, *) '//achar(27)//'[36m'//var//' = '
22     call print_matrix(A, m, n)
23     write (*, *) '//achar(27)//'[0m'
24 end subroutine
25
26 subroutine print_matrix(A, m, n)
27     implicit none
28     integer :: m, n
29     double precision :: A(m, n)
30     integer :: i, j
31     format(' /', F32.12, ' ')
32     format(F30.12, '/')
33     format(F30.12, ' ')
34     do i = 1, m
35         do j = 1, n
36             if (j == 1) then
37                 write(*, 20, advance='no') A(i, j)
38             elseif (j == n) then
39                 write(*, 21, advance='yes') A(i, j)
40             else
41                 write(*, 22, advance='no') A(i, j)
42             end if
43         end do
44     end do
45 end subroutine
46
47 subroutine read_matrix(fname, A, m, n)
48     implicit none
49     character(len=*) :: fname
50     integer :: m, n
51     double precision, dimension(:, :), allocatable :: A
52     integer :: i
53     open(unit=33, file=fname, status='old', action='read')
54     read(33, *) m
55     read(33, *) n
56     allocate(A(m, n))
57     do i = 1, m

```



```

57         read(33,*) A(i,:)
58     end do
59     close(33)
60 end subroutine
61
62 subroutine print_vector(x, n)
63     implicit none
64     integer :: n
65     double precision :: x(n)
66     integer :: i
67 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

```

```

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

```

```

163     end function
164
165     function given_matrix(A, n, i, j) result (G)
166         implicit none
167
168         integer :: n, i, j
169         double precision :: A(n, n), G(n, n)
170         double precision :: t, c, s
171
172         G(:, :) = id_matrix(n)
173
174         t = 0.5D0 * DATAN2(2.0D0 * A(i, j), A(i, i) - A(j, j))
175         s = DSIN(t)
176         c = DCOS(t)
177
178         G(i, i) = c
179         G(j, j) = c
180         G(i, j) = -s
181         G(j, i) = s
182
183         return
184     end function
185
186     function vandermond_matrix(x, n) result (V)
187         implicit none
188         integer :: n, i
189         double precision, dimension(n), intent(in) :: x
190         double precision, dimension(n, n) :: V
191         V(1, :) = 1.0D0
192         do i=2, n
193             V(i, :) = V(i-1, :) * x(i)
194         end do
195         return
196     end function
197
198     function diagonally_dominant(A, n) result (ok)
199         implicit none
200
201         integer :: n
202         double precision :: A(n, n)
203
204         logical :: ok
205         integer :: i
206
207         do i = 1, n
208             if (DABS(A(i, i)) < SUM(DABS(A(i, :i-1))) + SUM(DABS
209                 (A(i, i+1:)))) then
210                 ok = .FALSE.
211                 return
212             end if
213         end do
214         ok = .TRUE.
215         return

```

```

215     end function
216
217     recursive function positive_definite(A, n) result (ok)
218     ! Checks wether a matrix is positive definite
219     ! according to Sylvester's criterion.
220         implicit none
221
222         integer :: n
223         double precision A(n, n)
224
225         logical :: ok
226
227         if (n == 1) then
228             ok = (A(1, 1) > 0)
229             return
230         else
231             ok = positive_definite(A(:n-1, :n-1), n-1) .AND. (
232                 det(A, n) > 0)
233             return
234         end if
235     end function
236
237     ! function symmetrical(A, n) result (ok)
238     ! Check if the Matrix is symmetrical
239     integer :: n
240
241     double precision :: A(n, n)
242
243     integer :: i, j
244     logical :: ok
245
246     do i = 1, n
247         do j = 1, i-1
248             if (A(i, j) /= A(j, i)) then
249                 ok = .FALSE.
250                 return
251             end if
252         end do
253     end do
254     ok = .TRUE.
255     return
256 end function
257
258 subroutine swap_rows(A, i, j, n)
259     implicit none
260
261     integer :: n
262     integer :: i, j
263     double precision A(n, n)
264     double precision temp(n)
265
266     temp(:) = A(i, :)
267     A(i, :) = A(j, :)

```

```

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

```

```

317         end if
318
319         return
320     end function
321
322     function row_max(A, j, n) result(k)
323         implicit none
324
325         integer :: n
326         double precision A(n, n)
327
328         integer :: i, j, k
329         double precision :: s
330
331         s = 0.0D0
332         do i = j, n
333             if (A(i, j) > s) then
334                 s = A(i, j)
335                 k = i
336             end if
337         end do
338         return
339     end function
340
341     function pivot_matrix(A, n) result (P)
342         implicit none
343
344         integer :: n
345         double precision :: A(n, n)
346
347         double precision :: P(n, n)
348
349         integer :: j, k
350
351         P = id_matrix(n)
352
353         do j = 1, n
354             k = row_max(A, j, n)
355             if (j /= k) then
356                 call swap_rows(P, j, k, n)
357             end if
358         end do
359         return
360     end function
361
362     function vector_norm(x, n) result (s)
363         implicit none
364         integer :: n
365         double precision :: x(n)
366         double precision :: s
367         s = sqrt(dot_product(x, x))
368         return
369     end function

```

```

370
371 function NORM(x, n) result (s)
372     implicit none
373     integer :: n
374     double precision :: x(n)
375     double precision :: s
376     s = SQRT(DOT_PRODUCT(x, x))
377     return
378 end function
379
380 function matrix_norm(A, n) result (s)
381 !     Frobenius norm
382     implicit none
383     integer :: n
384     double precision :: A(n, n)
385     double precision :: s
386
387     s = DSQRT(SUM(A * A))
388     return
389 end function
390
391 function spectral_radius(A, n) result (r)
392     implicit none
393
394     integer :: n
395     double precision :: A(n, n), x(n)
396     double precision :: r, l
397     logical :: ok
398     ok = power_method(A, n, x, 1)
399     r = DABS(l)
400     return
401 end function
402
403 recursive function det(A, n) result (d)
404     implicit none
405     integer :: n
406     double precision, dimension(n, n) :: A
407     double precision, dimension(n-1, n-1) :: X
408     integer :: i
409     double precision :: d, s
410
411     if (n == 1) then
412         d = A(1, 1)
413         return
414     elseif (n == 2) then
415         d = A(1, 1) * A(2, 2) - A(1, 2) * A(2, 1)
416         return
417     else
418         d = 0.0D0
419         s = 1.0D0
420         do i = 1, n
421 !             Compute submatrix X
422             X(:, :i-1) = A(2:, :i-1)

```

```

423         X(:, i: ) = A(2:, i+1: )
424         d = s * det(X, n-1) * A(1, i) + d
425         s = -s
426     end do
427 end if
428 return
429 end function
430
431 function LU_det(A, n) result (d)
432     implicit none
433
434     integer :: n
435     integer :: i
436     double precision :: A(n, n), L(n, n), U(n, n)
437     double precision :: d
438
439     d = 0.0D0
440
441     if (.NOT. LU_decomp(A, L, U, n)) then
442         call ill_cond()
443         return
444     end if
445
446     do i = 1, n
447         d = d * L(i, i) * U(i, i)
448     end do
449
450     return
451 end function
452
453 subroutine LU_matrix(A, L, U, n)
454 !     Splits Matrix in Lower and Upper-Triangular
455     implicit none
456
457     integer :: n
458     double precision :: A(n, n), L(n, n), U(n, n)
459
460     integer :: i
461
462     L(:, :) = 0.0D0
463     U(:, :) = 0.0D0
464
465     do i = 1, n
466         L(i, i) = 1.0D0
467         L(i, :i-1) = A(i, :i-1)
468         U(i, i: ) = A(i, i: )
469     end do
470 end subroutine
471
472 !     === Matrix Factorization Conditions ===
473 function Cholesky_cond(A, n) result (ok)
474     implicit none
475     integer :: n

```



```

476      double precision :: A(n, n)
477      logical :: ok
478      ok = symmetrical(A, n) .AND. positive_definite(A, n)
479      return
480  end function

481
482  function PLU_cond(A, n) result (ok)
483      implicit none
484      integer :: n
485      double precision A(n, n)
486      integer :: i, j
487      double precision :: s
488      logical :: ok
489      do j = 1, n
490          s = 0.0D0
491          do i = 1, j
492              if (A(i, j) > s) then
493                  s = A(i, j)
494              end if
495          end do
496      end do
497      ok = (s < 0.01D0)
498      return
499  end function

500
501  function LU_cond(A, n) result (ok)
502      implicit none
503      integer :: n
504      double precision A(n, n)
505      logical :: ok
506      ok = positive_definite(A, n)
507      return
508  end function

509  !
510  !      -      - - - -      - - - -      - - - -      - -
511  !      / /      / -      - / /      - -      - - / \      / - /
512  !      / /      / / / ( _ _      / / / \      / /
513  !      / /      / / \ _ _ \      / / / / \ \      / /
514  !      / / _ _ _ _      - / / _ _ _ _ ) /      / / / _ _ _      \      / /
515  !      / _ _ _ _ / _ _ _ _ / _ _ _ _ /      / _ / _ /      \ \      / _ /
516  !      =====
517  !
518  !      ===== Matrix Factorization Methods =====
519  function PLU_decomp(A, P, L, U, n) result (ok)
520      implicit none
521      integer :: n
522      double precision :: A(n,n), P(n,n), L(n,n), U(n,n)
523      logical :: ok
524      !      Permutation Matrix
525      P = pivot_matrix(A, n)
526      !      Decomposition over Row-Swapped Matrix
527      ok = LU_decomp(matmul(P, A), L, U, n)
528      return
529  end function

```

```

529
530 function LU_decomp(A, L, U, n) result (ok)
531     implicit none
532     integer :: n
533     double precision :: A(n, n), L(n, n), U(n,n), M(n, n)
534     logical :: ok
535     integer :: i, j, k
536 ! Results Matrix
537 M(:, :) = A(:, :)
538 if (.NOT. LU_cond(A, n)) then
539     call ill_cond()
540     ok = .FALSE.
541     return
542 end if
543 do k = 1, n-1
544     do i = k+1, n
545         M(i, k) = M(i, k) / M(k, k)
546     end do
547     do j = k+1, n
548         do i = k+1, n
549             M(i, j) = M(i, j) - M(i, k) * M(k, j)
550         end do
551     end do
552 end do
553
554 ! Splits M into L & U
555 call LU_matrix(M, L, U, n)
556
557 ok = .TRUE.
558 return
559
560 end function
561
562 function Cholesky_decomp(A, L, n) result (ok)
563     implicit none
564
565     integer :: n
566     double precision :: A(n, n), L(n, n)
567
568     logical :: ok
569
570     integer :: i, j
571
572     if (.NOT. Cholesky_cond(A, n)) then
573         call ill_cond()
574         ok = .FALSE.
575         return
576     end if
577
578     do i = 1, n
579         L(i, i) = sqrt(A(i, i) - sum(L(i, :i-1) * L(i, :i-1)
580             ))
581         do j = 1 + 1, n

```

```

581         L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1)
582             )) / L(i, i)
583     end do
584 end do
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

```

```

633     x0 = rand_vector(n)
634
635     ok = Jacobi_cond(A, n)
636
637     if (.NOT. ok) then
638         return
639     end if
640
641     do k = 1, t_max_iter
642         do i = 1, n
643             x(i) = (b(i) - dot_product(A(i, :), x0)) / A(i,
644                                     i)
645         end do
646         x0(:) = x(:)
647         e = vector_norm(matmul(A, x) - b, n)
648         if (e < t_tol) then
649             return
650         end if
651     end do
652     call error('Erro: Esse método não convergiu.')
653     ok = .FALSE.
654     return
655 end function
656
657 function Gauss_Seidel_cond(A, n) result (ok)
658     implicit none
659
660     integer :: n
661
662     double precision :: A(n, n)
663
664     logical :: ok
665
666     integer :: i
667
668     do i = 1, n
669         if (A(i, i) == 0.0D0) then
670             ok = .FALSE.
671             call ill_cond()
672             return
673         end if
674     end do
675
676     if (symmetrical(A, n) .AND. positive_definite(A, n))
677         then
678         ok = .TRUE.
679         return
680     else
681         call warn('Aviso: Esse método pode não convergir.')
682         return
683     end if
684 end function

```

```

684 function Gauss_Seidel(A, x, b, e, n, tol, max_iter) result (
      ok)
685     implicit none
686     logical :: ok
687     integer :: n, i, j, k, t_max_iter
688     integer, optional :: max_iter
689     double precision :: A(n, n)
690     double precision :: b(n), x(n)
691     double precision :: e, s, t_tol
692     double precision, optional :: tol
693
694     if (.NOT. PRESENT(tol)) then
695         t_tol = D_TOL
696     else
697         t_tol = tol
698     end if
699
700     if (.NOT. PRESENT(max_iter)) then
701         t_max_iter = D_MAX_ITER
702     else
703         t_max_iter = max_iter
704     end if
705
706     ok = Gauss_Seidel_cond(A, n)
707
708     if (.NOT. ok) then
709         return
710     end if
711
712     do k = 1, t_max_iter
713         do i = 1, n
714             s = 0.0D0
715             do j = 1, n
716                 if (i /= j) then
717                     s = s + A(i, j) * x(j)
718                 end if
719             end do
720             x(i) = (b(i) - s) / A(i, i)
721         end do
722         e = vector_norm(matmul(A, x) - b, n)
723         if (e < t_tol) then
724             return
725         end if
726     end do
727     call error('Erro: Esse método não convergiu.')
728     ok = .FALSE.
729     return
730 end function
731
732 ! Decomposição LU e afins
733 subroutine LU_backsub(L, U, x, y, b, n)
734     implicit none
735     integer :: n

```

```

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

```

```

788         x(:) = matmul(P, x)
789
790     return
791 end function
792
793 function Cholesky_solve(A, x, y, b, n) result (ok)
794     implicit none
795
796     integer :: n
797
798     double precision :: A(n, n), L(n, n), U(n, n)
799     double precision :: b(n), x(n), y(n)
800
801     logical :: ok
802
803     ok = Cholesky_decomp(A, L, n)
804
805     if (.NOT. ok) then
806         return
807     end if
808
809     U = transpose(L)
810
811     call LU_backsub(L, U, x, y, b, n)
812
813     return
814 end function
815
816 !
817 !      -      -----      -----      -----      ---
818 !      / /      /_      _/ /      _      _      / \      /_      \
819 !      / /      / / / ( _      / / / \      \      ) /
820 !      / /      / / \ _      \      / / / \      \      / /
821 !      / /_      _/ /_      _      ) /      / / _      \      / /_
822 !      /_      _/ _      _/ _      _      /_      _      \_      \ /_      _/
823 !      =====
824 !
825 ===== Power Method =====
826 function power_method(A, n, x, l, tol, max_iter) result (ok)
827     implicit none
828     logical :: ok
829     integer :: n, k, t_max_iter
830     integer, optional :: max_iter
831     double precision :: A(n, n)
832     double precision :: x(n)
833     double precision :: l, ll, t_tol
834     double precision, optional :: tol
835
836     if (.NOT. PRESENT(tol)) then
837         t_tol = D_TOL
838     else
839         t_tol = tol
840     end if

```

```

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             l1 = 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 - l1) / l) < t_tol) then
867                 ok = .TRUE.
868                 return
869             end if
870         end do
871         ok = .FALSE.
872         return
873     end function
874
875 function Jacobi_eigen(A, n, L, X, tol, max_iter) result (ok)
876     implicit none
877     logical :: ok
878     integer :: n, i, j, k, u, v, t_max_iter
879     integer, optional :: max_iter
880     double precision :: A(n, n), L(n, n), X(n, n), P(n, n)
881     double precision :: y, z, t_tol
882     double precision, optional :: tol
883
884     if (.NOT. PRESENT(tol)) then
885         t_tol = D_TOL
886     else
887         t_tol = tol
888     end if
889
890     if (.NOT. PRESENT(max_iter)) then
891         t_max_iter = D_MAX_ITER
892     else

```



```

893         t_max_iter = max_iter
894     end if
895
896     X(:, :) = id_matrix(n)
897     L(:, :) = A(:, :)
898
899     do k=1, t_max_iter
900         z = 0.0D0
901         do i = 1, n
902             do j = 1, i - 1
903                 y = DABS(L(i, j))
904
905             !
906             Found new maximum absolute value
907             if (y > z) then
908                 u = i
909                 v = j
910                 z = y
911             end if
912         end do
913     end do
914
915     if (z >= t_tol) then
916         P(:, :) = given_matrix(L, n, u, v)
917         L(:, :) = matmul(matmul(transpose(P), L), P)
918         X(:, :) = matmul(X, P)
919     else
920         ok = .TRUE.
921         return
922     end if
923 end do
924 ok = .FALSE.
925 return
926 end function
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)

```

```

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

```

```

999             A(k,j) = A(l,j)
1000             A(l,j) = store
1001         end do
1002         store = b(k)
1003         b(k) = b(l)
1004         b(l) = store
1005     end if
1006
1007     do i=k+1,n
1008         c = A(i,k) / A(k,k)
1009         A(i,k) = 0.0D0
1010         b(i) = b(i) - c*b(k)
1011         do j=k+1,n
1012             A(i,j) = A(i,j) - c * A(k,j)
1013         end do
1014     end do
1015 end do
1016
1017 x(n) = b(n) / A(n,n)
1018 do i=n-1,1,-1
1019     c = 0.0D0
1020     do j=i+1,n
1021         c = c + A(i,j) * x(j)
1022     end do
1023     x(i) = (b(i) - c) / A(i,i)
1024 end do
1025
1026 return
1027 end function
1028
1029 function solve(A, b, n, kind) result (x)
1030     implicit none
1031     integer :: n
1032     double precision, dimension(n), intent(in) :: b
1033     double precision, dimension(n) :: x, y
1034     double precision, dimension(n, n), intent(in) :: A
1035     character(len=*), optional :: kind
1036     character(len=:), allocatable :: t_kind
1037
1038     logical :: ok = .TRUE.
1039
1040     if (.NOT. PRESENT(kind)) then
1041         call debug("Indeed, not present.")
1042         t_kind = "gauss"
1043     else
1044         t_kind = kind
1045     end if
1046
1047     call debug("Now it is: "//t_kind)
1048     if (t_kind == "LU") then
1049         ok = LU_solve(A, x, y, b, n)
1050     else if (t_kind == "PLU") then
1051         ok = PLU_solve(A, x, y, b, n)

```

```

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

```

## Código - Biblioteca Auxiliar

```

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

```

```

30      logical :: y
31
32      y = ANY(ISNAN(x)) .OR. ANY(x == DINF) .OR. ANY(x == DNINF)
33      return
34  end function
35
36  function MEDGE(x) result (y)
37      double precision, dimension(:, :), intent(in) :: x
38      logical :: y
39
40      y = ANY(ISNAN(x)) .OR. ANY(x == DINF) .OR. ANY(x == DNINF)
41      return
42  end function
43
44  function quote(s, q) result (r)
45      character(len=*), intent(in) :: s
46      character(len=*), optional, intent(in) :: q
47      character(len=:), allocatable :: t_q
48      character(len=:), allocatable :: r
49
50      if (.NOT. PRESENT(q)) then
51          t_q = ' '
52      else
53          t_q = q
54      end if
55
56      r = t_q//s//t_q
57  end function
58
59  function DLOG2(x) result (y)
60      implicit none
61      double precision, intent(in) :: x
62      double precision :: y
63
64      y = DLOG(x) / DLOG(2.0D0)
65      return
66  end function
67
68  function ILOG2(n) result (k)
69      integer, intent(in) :: n
70      integer :: k
71      double precision :: x
72      x = n
73      x = DLOG2(x)
74      k = FLOOR(x)
75      return
76  end function
77
78  ! ===== Random seed Initialization =====
79  subroutine init_random_seed()
80      integer :: i, n, clock
81      integer, allocatable :: seed(:)
82

```

```

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

```

```

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

```

```

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

```



```

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

```

## Código - Biblioteca de Plotagem

```

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 = ',\n'//ENDL
8
9      logical :: g_INPLOT = .FALSE.
10     logical :: g_INMULTIPLY = .FALSE.
11
12     character(len=:), allocatable :: g_FNAME, g_OUTP_FNAME,
13         g_PLOT_FNAME, g_SIZE_W, g_SIZE_H
14
15     integer :: g_M, g_N
16
17     type SPLOT
18         integer :: i, j
19         integer :: n = 0
20         logical :: grid = .FALSE.
21         logical :: done = .FALSE.
22
23         character(len=:), allocatable :: title, xlabel, ylabel
24         type(StringArray), dimension(:), allocatable :: legend
25         type(StringArray), dimension(:), allocatable :: with

```

```

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

```

```

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,
102     xmax, ymin, ymax, legend, with, grid)
103     integer, intent(in) :: i, j
104     integer :: k, n
105
106     logical, optional :: grid
107
108     character(len=*), optional :: title, ylabel, xlabel
109
110     double precision, optional :: xmin, xmax, ymin, ymax
111
112     type(StringArray), dimension(:), optional :: legend, with
113
114     if (g_SUBPLOTS(i, j)%done) then
115         call error("Duplicate configuration of subplot ("//STR(i
116             )//", "//STR(j)//")")
117         stop "ERROR"
118     end if
119
120     n = g_SUBPLOTS(i, j)%n
121
122     allocate(g_SUBPLOTS(i, j)%legend(n))
123     allocate(g_SUBPLOTS(i, j)%with(n))
124
125     if (PRESENT(xmin)) then
126         g_SUBPLOTS(i, j)%xmin = xmin
127         g_SUBPLOTS(i, j)%l_xmin = .TRUE.
128     end if
129
130     if (PRESENT(xmax)) then
131         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(
150                 k)%str)
151         end do
152     end if
153
154     if (.NOT. PRESENT(with)) then
155         do k=1,n
156             g_SUBPLOTS(i, j)%with(k)%str = 'w lines'
157         end do
158     else
159         do k=1,n
160             g_SUBPLOTS(i, j)%with(k)%str = 'w '//with(k)%str
161         end do
162     end if
163
164     if (.NOT. PRESENT(grid)) then
165         g_SUBPLOTS(i, j)%grid = .TRUE.
166     else
167         g_SUBPLOTS(i, j)%grid = grid
168     end if
169
170     if (.NOT. PRESENT(title)) then
171         g_SUBPLOTS(i, j)%title = ''
172     else
173         g_SUBPLOTS(i, j)%title = title
174     end if
175
176     if (.NOT. PRESENT(xlabel)) then
177         g_SUBPLOTS(i, j)%xlabel = 'x'
178     else
179         g_SUBPLOTS(i, j)%xlabel = xlabel
180     end if
181
182     if (.NOT. PRESENT(ylabel)) then

```

```

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)//", "
201                        //STR(j)//")")
202             stop "ERROR"
203         else
204             g_SUBPLOTS(i, j)%n = g_SUBPLOTS(i, j)%n + 1
205         end if
206
207         s_data_fname = DATA_FNAME(g_FNAME, i, j, g_SUBPLOTS(i, j)%n)
208         ! ===== Touch Plot File
209         =====
210         open(newunit=file, file=s_data_fname, status="replace",
211              action="write")
212         write(file, *) "# file: "//s_data_fname
213         close(file)
214         ! =====
215
216         ! ===== Write to Plot File
217         =====
218         10 format(F16.8, ' ')
219         11 format(F16.8, ' ')
220         open(newunit=file, file=s_data_fname, status="old", position
221              ="append", action="write")
222         write(file, *)
223         do k=1, n
224             write(file, 10, advance='no') x(k)
225             write(file, 11, advance='yes') y(k)
226         end do
227         close(file)
228         ! =====
229
230     end subroutine

```

```

226
227 ! ===== Pipeline =====
228 subroutine begin_plot(fname, size_w, size_h)
229     integer :: file
230     character(len=*), optional :: fname, size_w, size_h
231
232     if (.NOT. PRESENT(size_w)) then
233         g_SIZE_W = DEFAULT_SIZE_W
234     else
235         g_SIZE_W = size_w
236     end if
237
238     if (.NOT. PRESENT(size_h)) then
239         g_SIZE_H = DEFAULT_SIZE_H
240     else
241         g_SIZE_H = size_h
242     end if
243
244     if (.NOT. PRESENT(fname)) then
245         g_FNAME = DEFAULT_FNAME
246     else
247         g_FNAME = fname
248     end if
249
250     g_PLOT_FNAME = PLOT_FNAME(g_FNAME)
251     g_OUTP_FNAME = OUTP_FNAME(g_FNAME)
252
253     open(newunit=file, file=g_PLOT_FNAME, status="new", action="
254         write")
255     write(file, *) 'set terminal pdf size '//g_SIZE_W//', '//
256         g_SIZE_H//';'
257     write(file, *) 'set output '//quote(g_OUTP_FNAME)//';'
258     close(file)
259
260     g_INPLOT = .TRUE.
261
262 end subroutine
263
264 subroutine subplots(m, n)
265     integer, optional, intent(in) :: m, n
266     integer :: t_m, t_n
267
268     if ((.NOT. PRESENT(m)) .OR. (m <= 0)) then
269         t_m = 1
270     else
271         t_m = m
272     end if
273
274     if ((.NOT. PRESENT(n)) .OR. (n <= 0)) then
275         t_n = 1
276     else
277         t_n = n
278     end if

```

```

277
278     if (.NOT. g_INPLOT) then
279         call begin_plot()
280     end if
281
282     !      ===== Allocate Variables =====
283     allocate(g_SUBPLOTS(t_m, t_n))
284     g_M = t_m
285     g_N = t_n
286     g_INMULTILOT = .TRUE.
287     !      =====
288     end subroutine
289
290     subroutine render_plot(clean)
291         integer :: file, i, j, k, m, n
292
293         logical, optional :: clean
294         logical :: t_clean
295
296         if (.NOT. PRESENT(clean)) then
297             t_clean = .FALSE.
298         else
299             t_clean = clean
300         end if
301
302         !      === Check Plot =====
303         if (.NOT. g_INPLOT) then
304             call error("No active plot to render.")
305             stop "ERROR"
306         end if
307         !      =====
308
309         m = g_M
310         n = g_N
311
312         !      ===== Write to Plot File =====
313         open(newunit=file, file=g_PLOT_FNAME, status="old", position
314             ="append", action="write")
315         write(file, *) 'set origin 0,0;'
316
317         if (g_INMULTILOT) then
318             write(file, *) 'set multiplot layout '//STR(m)//', '//STR
319                 (n)//' rowsfirst;'
320         end if
321
322         format(A, ' ')
323         !      ===== Plot data =====
324         do i= 1, m
325             do j = 1, n
326                 g_SUBPLOTS(i, j) = g_SUBPLOTS(i, j)
327             end do
328         end do

```

```

326         write(file, *) 'set title '//quote(g_SUBPLOTS(i, j)%
327             title)//';'
328         write(file, *) 'set xlabel '//quote(g_SUBPLOTS(i, j)
329             %xlabel)//';'
330         write(file, *) 'set ylabel '//quote(g_SUBPLOTS(i, j)
331             %ylabel)//';'
332
333         !
334         ===== Set X RANGE
335         =====
336         if (g_SUBPLOTS(i, j)%l_xmin .AND. g_SUBPLOTS(i, j)%
337             l_xmax) then
338             write(file, *) 'set xrange ['//DSTR(g_SUBPLOTS(i
339                 , j)%xmin)//': '//DSTR(g_SUBPLOTS(i, j)%xmax)
340                 //']';'
341         else if (g_SUBPLOTS(i, j)%l_xmin) then
342             write(file, *) 'set xrange ['//DSTR(g_SUBPLOTS(i
343                 , j)%xmin)//':*];'
344         else if (g_SUBPLOTS(i, j)%l_xmax) then
345             write(file, *) 'set xrange [*://DSTR(g_SUBPLOTS
346                 (i, j)%xmax)//']';'
347         else
348             write(file, *) 'set xrange [*:*];'
349         end if
350
351         !
352         =====
353
354         !
355         ===== Set Y RANGE
356         =====
357         if (g_SUBPLOTS(i, j)%l_ymin .AND. g_SUBPLOTS(i, j)%
358             l_ymax) then
359             write(file, *) 'set yrange ['//DSTR(g_SUBPLOTS(i
360                 , j)%ymin)//': '//DSTR(g_SUBPLOTS(i, j)%ymax)
361                 //']';'
362         else if (g_SUBPLOTS(i, j)%l_ymin) then
363             write(file, *) 'set yrange ['//DSTR(g_SUBPLOTS(i
364                 , j)%ymin)//':*];'
365         else if (g_SUBPLOTS(i, j)%l_ymax) then
366             write(file, *) 'set yrange [*://DSTR(g_SUBPLOTS
367                 (i, j)%ymax)//']';'
368         else
369             write(file, *) 'set yrange [*:*];'
370         end if
371
372         !
373         =====
374
375         if (g_SUBPLOTS(i, j)%grid) then
376             write(file, *) 'set grid;'
377         else
378             write(file, *) 'unset grid;'
379         end if
380
381         write(file, 10, advance='no') 'plot'
382
383

```



```

362         do k = 1, g_SUBPLOTS(i, j)%n
363             write(file, 10, advance='no') quote(DATA_FNAME(
364                 g_FNAME, i, j, k))
365             write(file, 10, advance='no') 'u 1:2'
366             write(file, 10, advance='no') g_SUBPLOTS(i, j)%
367                 legend(k)%str
368             write(file, 10, advance='no') g_SUBPLOTS(i, j)%
369                 with(k)%str
370
371             if (k == (g_SUBPLOTS(i, j)%n)) then
372                 write(file, *) ';'
373             else
374                 write(file, *) ',\'
375             end if
376         end do
377     end do
378
379     == Finish Multiplot ==
380     if (g_INMULTILOT) then
381         write(file, *) 'unset multiplot'
382         g_INMULTILOT = .FALSE.
383     end if
384
385     close(file)
386
387     =====
388
389     == Call GNUPLOT and remove temporary files ==
390     call EXECUTE_COMMAND_LINE(GNU_PLOT_CMD(g_FNAME))
391
392     if (t_clean) then
393         call EXECUTE_COMMAND_LINE(REMOVE_TEMP_FILES(g_FNAME))
394     end if
395
396     =====
397
398     == Free Variables ==
399     deallocate(g_FNAME, g_OUTP_FNAME, g_PLOT_FNAME)
400     deallocate(g_SUBPLOTS)
401     g_INPLOT = .FALSE.
402
403     =====
404
405     end subroutine
406 end module PlotLib

```

## Código - Quadraturas pelo *Mathematica*

```

(* Set directory to current one *)
SetDirectory[NotebookDirectory[] <> "/gauss-legendre"]

symboliclegendre[n_, x_] := Solve[LegendreP[n, x] == 0];
legendreprime[n_, a_] := D[LegendreP[n, x], x] /. x -> a;
weights[n_, x_] := 2 / ((1 - x^2) legendreprime[n, x]^2);

(*how many terms should be generated*)
m = 128;

(*what numerical precision is desired?*)
precision = 32;

For[n = 1, n ≤ m, n++,
  nlist := symboliclegendre[n, x];
  xnlist = x /. nlist;
  slist := symboliclegendre[n, x];
  xslist = x /. slist;
  file = OpenWrite["gauss-legendre" <> ToString[n] <> ".txt"];
  Write[file, n];
  Write[file, 2];
  For[k = 1, k ≤ n, k++,
    xs = ToString[ToString[#, FortranForm] & /@ N[xnlist[[k]], precision]] ×
    ws =
    ToString[ToString[#, FortranForm] & /@ N[weights[n, xslist[[k]]], precision]] ×
    WriteString[file, xs, " ", ws, "\n"];
  ] ×
Close[file];
]

```

```

In[6]:= (* Set directory to current one *)
SetDirectory[NotebookDirectory[] <> "/gauss-hermite"]
W[n_, x_] := (2^(n - 1) * (n!) * Sqrt[ $\pi$ ]) / (n HermiteH[n - 1, x])^2;
(*how many terms should be generated*)
m = 128;

(*what numerical precision is desired?*)
precision = 32;

For[n = 1, n ≤ m, n++,
  X = x /. Solve[HermiteH[n, x] == 0];
  file = OpenWrite["gauss-hermite" <> ToString[n] <> ".txt"];
  Write[file, n];
  Write[file, 2];
  For[k = 1, k ≤ n, k++,
    WriteString[file,
      FortranForm@N[X[[k]], precision],
      " ",
      FortranForm@N[W[n, X[[k]]], precision],
      "\n"
    ];
  ] ×
Close[file];
]

```