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*[1]
- `'romberg'` Método de *Romberg*

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

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

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

        end function
```

[1] 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 ω_i da solução.

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

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

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.

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

```

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

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

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, $[a, b] = [-\infty, \infty]$.

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

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

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

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

        return
    end function
```

4 .: Método de *Romberg*

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

```
            integer, intent(in) :: n
            double precision, intent(in) :: a, b
            double precision :: s
            double precision, dimension(n) :: xx, ww
            integer :: k
            character(len=*), parameter :: fname = GAUSS_HERMITE_QUAD
            interface
                function f(x) result (y)
                    double precision :: x, y
                end function
            end interface

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

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

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

            return
        end function

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

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

```

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

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

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

            t_n = ILOG2(n)

            dx = (b - a)

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

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

            do i = 1, t_n
                dx = dx / 2

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

```

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

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

            deallocate(R)
        end function
```

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.

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

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

            if (n > 1) then
                p = num_int(f, a, b, n / 2, kind = kind)
                q = num_int(f, a, b, n, kind = kind)
                e = DABS(p - q)
                if (e <= t_tol) then
                    s = q
```

```
            else
                r = (b + a) / 2
                s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
                    + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
            end if
            return
        else
            s = 0.0D0
            return
        end if
    end function
```

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(−x² /2) / √(2 π)
 [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 $S_\sigma(\omega) = \mathrm{RAO}(\omega)^2 S_\eta(\omega)$, onde

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

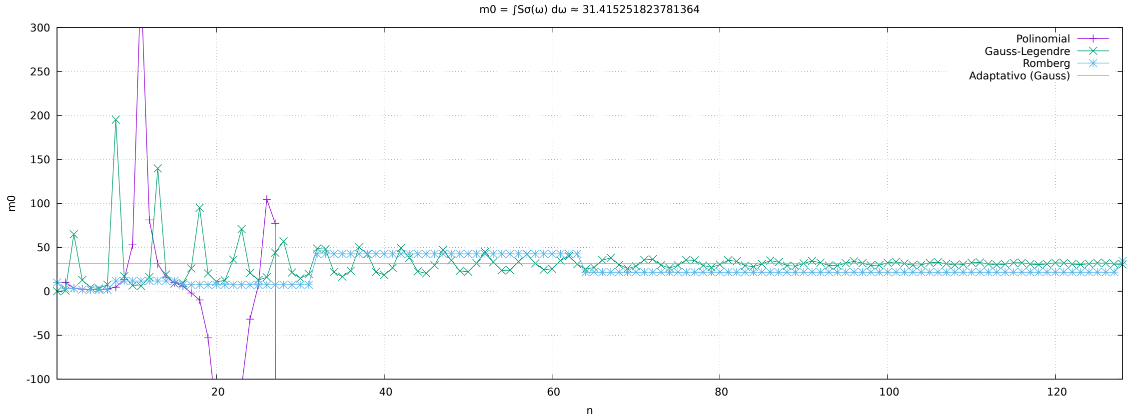
com $\omega_n = 1.0$ e $\xi = 0.05$ e $S_\eta(\omega) = 2.0$, obtenha m_0 e m_2 dados por:

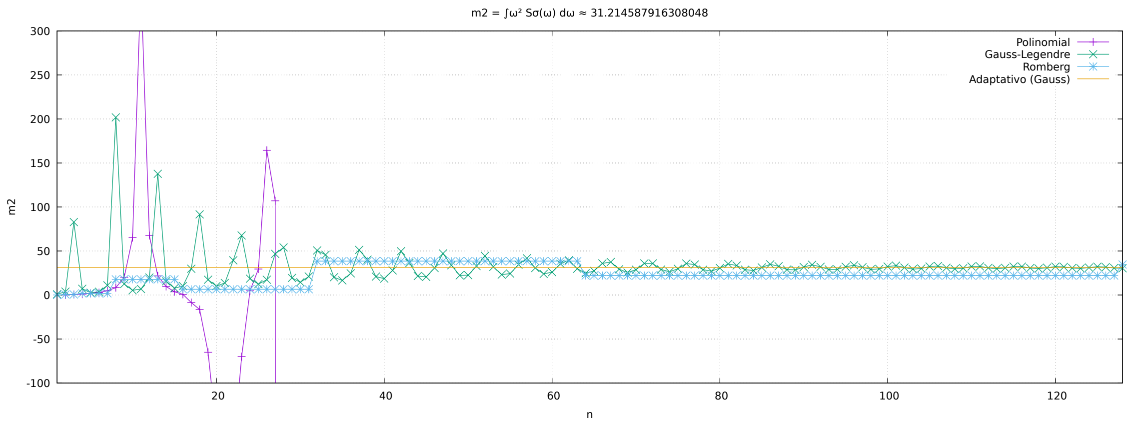
$$m_0 = \int_0^{10} S_\sigma(\omega)\, d\omega$$

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

```
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.

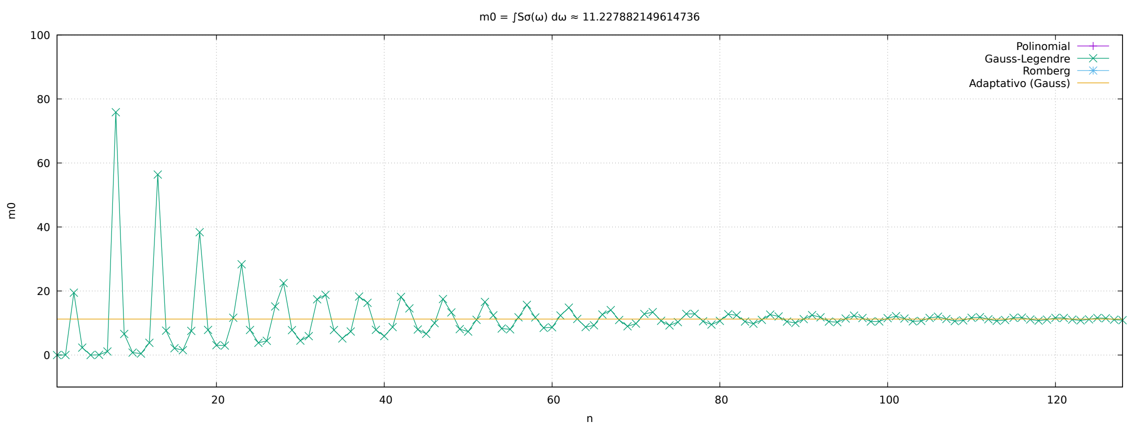
O método de *Romberg* possui uma certa sutileza. Iniciar o algoritmo com n pontos de entrada faz com que este avalie a função em 2^n pontos. Por isso, para comparação com os algoritmos, são utilizados $\log_2(n)$ pontos como entrada. Isso faz com que se utilize n pontos no total somente quando n é potência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

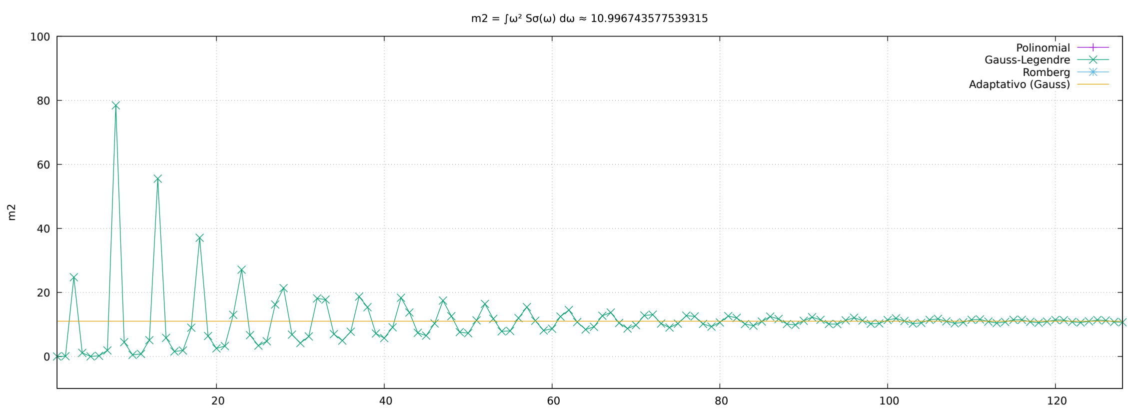
Questão 4.: Repita o exercício anterior considerando

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

```
4)
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) = ((4 π³ Hs²) / (ω⁵ Tz⁴)) exp(−(16 π³) / (ω⁴ Tz⁴))
```

```
 :: 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 ~ Sη(ω) = ((4 π³ Hs²) / (ω⁵ Tz⁴)) exp(−(16 π³) / (ω⁴ Tz⁴))

 :: 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 − x² + 3x³
 [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

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

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

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

2 .: Regra do Trapézio

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

3 .: Regra de *Simpson*

$$A_{\mathbf{S}} \approx \frac{2}{3}\cdot A_{\mathbf{M}} + \frac{1}{3}\cdot A_{\mathbf{T}} = \frac{2}{3}\cdot\frac{12}{13} + \frac{1}{3}\cdot\frac{33}{20} = \frac{303}{260} \approx 1.16538$$

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

```
6) f(x) = 1 / (1 + x²)
 [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 $\dfrac{|1.249 - 1.165|}{|1.249|} \approx 6.71\%$

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 ∞ 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:

$$\begin{aligned} A_1 &= \int_{-\infty}^{1} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)\, dx \\ &= \frac{1}{\sqrt{2\pi}} \left[\int_{-\infty}^{0} \exp\left(-\frac{x^2}{2}\right)\, dx + \int_{0}^{1} \exp\left(-\frac{x^2}{2}\right)\, dx\right] \end{aligned}$$

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

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

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

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

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 $f(x) = \exp\left(x^2/2\right)$ assim como a integral do segundo termo pela quadratura de *Gauss-Legendre* com $f(x) = \exp\left(-x^2/2\right)$. 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 $f(x) = \dfrac{x^2}{\sqrt{2\pi}} \cdot \exp\left(\dfrac{x^2}{2}\right)$ segundo a quadratura de *Gauss-Hermite*.

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

```
7)
 n = 10
 A1 ~ f(x) = exp(- x²/2) / √(2 π)
 [a, b] = [-∞, 1]
 :: Quadratura de Gauss-Hermite e de Gauss-Legendre ::
 A1 = ∫f(x) dx ≈ 0.84133856560070919
 A2 ~ f(x) = x² exp(- x²/2) / √(2 π)
 [a, b] = [-∞, ∞]
 :: 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

```
        function d(f, x, dx, kind) result (y)
            implicit none
            character (len=*), optional :: kind
            double precision, optional :: dx
            character (len=:), allocatable :: t_kind
            double precision :: x, y, t_dx

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

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

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

            if (t_kind == "central") then
                y = (f(x + t_dx) - f(x - t_dx)) / (2 * t_dx)
            else if (t_kind == "forward") then
                y = (f(x + t_dx) - f(x)) / t_dx
            else if (t_kind == "backward") then
                y = (f(x) - f(x - t_dx)) / t_dx
            else
                call error("Unexpected value `"//t_kind//" for
                    derivative kind."// &
```

```fortran
                "Options are: 'central', 'forward' and 'backward'.")
        end if
        return
    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.

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

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

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

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

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

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

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

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 (Δx = 1E−2)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 5.1389023598691352E−006
 :: Passo à frente (Δx = 1E−2)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 1.5948580328932760E−003
 :: Passo atrás (Δx = 1E−2)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 1.6051358376130143E−003

:: Extrapolação de Richard ::
 :: Diferença Central (Δx = 1E−2, p = 1)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 2.5694791288000118E−006
 :: Diferença Central (Δx = 1E−2, p = 2)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 1.8632539955376615E−011
 :: Passo à frente (Δx = 1E−2, p = 1)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 2.5553296916225321E−006
 :: Passo à frente (Δx = 1E−2, p = 2)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 5.3332289742547001E−004
 :: Passo atrás (Δx = 1E−2, p = 1)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 2.5836285659774916E−006
 :: Passo atrás (Δx = 1E−2, p = 2)::
 f'(2) ≈ 0.70998684164914549
 |δy| = 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 (Δx = 1E−2)::
 f'(6) ≈ 0.11372532416741847
 |δy| = 1.8196394321878806E−008
 :: Passo à frente (Δx = 1E−2)::
 f'(6) ≈ 0.11372532416741847
 |δy| = 1.7818749274416124E−004
 :: Passo atrás (Δx = 1E−2)::
 f'(6) ≈ 0.11372532416741847
 |δy| = 1.7815109995551748E−004

:: Extrapolação de Richard ::
 :: Diferença Central (Δx = 1E−2, p = 1)::
 f'(6) ≈ 0.11372532416741847
 |δy| = 9.0983055117677125E−009
 :: Diferença Central (Δx = 1E−2, p = 2)::
 f'(6) ≈ 0.11372532416741847
 |δy| = 7.2233885539674247E−014
```

```
:: Passo à frente (Δx = 1E-2, p = 1)::
f'(6) ≈ 0.11372532416741847
|δy| = 8.8146768356667238E-009
:: Passo à frente (Δx = 1E-2, p = 2)::
f'(6) ≈ 0.11372532416741847
|δy| = 5.9389954463501260E-005
:: Passo atrás (Δx = 1E-2, p = 1)::
f'(6) ≈ 0.11372532416741847
|δy| = 9.3819341878687013E-009
:: Passo atrás (Δx = 1E-2, p = 2)::
f'(6) ≈ 0.11372532416741847
|δy| = 5.9389954607969031E-005
```

Appendices

Código - Programa Principal

```
program main5
    use Util
    use Func
    use Matrix
    use Calc
    use Plotlib
    implicit none

    double precision :: XMIN, XMAX, YMIN, YMAX

!   Command-line Args
    integer :: argc

!    ENABLE_DEBUG = .TRUE.

!   Random seed definition
    call init_random_seed()

!   Get Command-Line Args
    argc = iargc()

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

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

100 goto 200

200 call Q2
    goto 300

300 call Q3
    goto 400

400 call Q4
    goto 500
```

```

500 call Q5
    goto 600

600 call Q6
    goto 700

700 call Q7
    goto 800

800 call warn(ENDL//":: Complmento ::"//ENDL)
    call QE1; call QE2; call QE3;
    goto 10

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

    contains

    subroutine Q2
        implicit none
        integer :: n = 10
        double precision :: a, b, s

        call info("2)"//ENDL//F6_NAME)
        a = 0.0D0
        b = 1.0D0
        call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
        call info(":: Integração Polinomial ::")
        s = num_int(f6, a, b, n, kind="polynomial")
        call blue("I1 = ∫f(x) dx ≈ "//DSTR(s))
        call info(":: Quadratura de Gauss-Legendre ::")
        s = num_int(f6, a, b, n, kind="gauss-legendre")
        call blue("I1 = ∫f(x) dx ≈ "//DSTR(s))
        call info(":: Método de Romberg ::")
        s = num_int(f6, a, b, n, kind="romberg")
        call blue("I1 = ∫f(x) dx ≈ "//DSTR(s))

        a = 0.0D0
        b = 5.0D0
        call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
        call info(":: Integração Polinomial ::")
        s = num_int(f6, a, b, n, kind="polynomial")
        call blue("I2 = ∫f(x) dx ≈ "//DSTR(s))
        call info(":: Quadratura de Gauss-Legendre ::")
        s = num_int(f6, a, b, n, kind="gauss-legendre")
        call blue("I2 = ∫f(x) dx ≈ "//DSTR(s))
        call info(":: Método de Romberg ::")
        s = num_int(f6, a, b, n, kind="romberg")
        call blue("I2 = ∫f(x) dx ≈ "//DSTR(s))

    end subroutine

    subroutine Q3
```

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

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

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

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

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

        a = 0.00D0
        b = 10.0D0

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

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

        INT_N = 128

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

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

        call begin_plot(fname='L5-Q3')

        call subplots(2, 1)

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

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

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

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

```fortran
                num_int(f7a, a, b, n, kind="gauss-legendre"), &
!
                num_int(f7a, a, b, n, kind="romberg"), &
!
                r &
            /)
        end do

        call info(":: Integração Polinomial ::")
        call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
        call info(":: Quadratura de Gauss-Legendre ::")
        call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
        call info(":: Método de Romberg ::")
        call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))

        do n = 1, 4
            call subplot(1, 1, x, y(n, :), INT_N)
        end do

        call subplot_config(1, 1, title='m0 = ∫Sσ(ω) dω ≈ '//DSTR(r), xlabel='n', ylabel='m0', grid=.TRUE., &
            legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=YMIN, ymax=YMAX)

        call info(ENDL//"m2 ~ "//F7b_NAME//ENDL)

        r = adapt_int(f7b, a, b, INT_N, tol=1.0D-8, kind="gauss-legendre")

        call info(":: Valor de referência (Integração Adaptativa) tol = 1E-8 ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(r))

        do n = 1, INT_N
            y(:, n) = (/ &
                num_int(f7b, a, b, n, kind="polynomial"), &
!
                num_int(f7b, a, b, n, kind="gauss-legendre"), &
!
                num_int(f7b, a, b, n, kind="romberg"), &
!
                r &
            /)
        end do

        call info(":: Integração Polinomial ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
        call info(":: Quadratura de Gauss-Legendre ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
        call info(":: Método de Romberg ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))

        do n = 1, 4
```

```fortran
            call subplot(2, 1, x, y(n, :), INT_N)
        end do

        call subplot_config(2, 1, title='m2 = ∫ω^2 Sσ(ω) dω ≈ '//DSTR(r), xlabel='n', ylabel='m2', grid=.TRUE., &
            legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=YMIN, ymax=YMAX)

        call render_plot(clean=.TRUE.)
    end subroutine

    subroutine Q4
        implicit none
        integer :: n
        double precision :: a, b, r
        double precision, dimension(INT_N) :: x
        double precision, dimension(4, INT_N) :: y

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

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

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

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

        a = 0.00D0
        b = 10.0D0

        call info(ENDL//"4)"//ENDL//F8_NAME)

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

        INT_N = 128

        XMIN = 1.0D0
        XMAX = INT_N
        YMIN = -10.0D0
        YMAX = 100.0D0

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

        call begin_plot(fname='L5-Q4')

        call subplots(2, 1)

        call info(ENDL//"m0 ~ "//F8a_NAME//ENDL)
```

```fortran

        r = adapt_int(f8a, a, b, INT_N, tol=1.0D-8, kind="gauss-legendre")

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

        do n = 1, INT_N
            y(:, n) = (/ &
                num_int(f8a, a, b, n, kind="polynomial"), &
!
                num_int(f8a, a, b, n, kind="gauss-legendre"), &
!
                num_int(f8a, a, b, n, kind="romberg"), &
!
                r &
            /)
        end do

        call info(":: Integração Polinomial ::")
        call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
        call info(":: Quadratura de Gauss-Legendre ::")
        call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
        call info(":: Método de Romberg ::")
        call blue("m0 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))

        do n = 1, 4
            call subplot(1, 1, x, y(n, :), INT_N)
        end do

        call subplot_config(1, 1, title='m0 = ∫Sσ(ω) dω ≈ '//DSTR(r), xlabel='n', ylabel='m0', grid=.TRUE., &
            legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=YMIN, ymax=YMAX)

        call info(ENDL//"m2 ~ "//F8b_NAME//ENDL)

        r = adapt_int(f8b, a, b, INT_N, tol=1.0D-8, kind="gauss-legendre")

        call info(":: Valor de referência (Integração Adaptativa) tol = 1E-8 ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(r))

        do n = 1, INT_N
            y(:, n) = (/ &
                num_int(f8b, a, b, n, kind="polynomial"), &
!
                num_int(f8b, a, b, n, kind="gauss-legendre"), &
!
                num_int(f8b, a, b, n, kind="romberg"), &
!
```

```fortran
                    r &
                /)
        end do

        call info(":: Integração Polinomial ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(1, 10)))
        call info(":: Quadratura de Gauss-Legendre ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(2, 10)))
        call info(":: Método de Romberg ::")
        call blue("m2 = ∫Sσ(ω) dω ≈ "//DSTR(y(3, 10)))

        do n = 1, 4
            call subplot(2, 1, x, y(n, :), INT_N)
        end do

        call subplot_config(2, 1, title='m2 = ∫ω² Sσ(ω) dω ≈ '//DSTR(r), xlabel='n', ylabel='m2', grid=.TRUE., &
            legend=legend, with=with, xmin=XMIN, xmax=XMAX, ymin=YMIN, ymax=YMAX)

        call render_plot(clean=.TRUE.)
    end subroutine

    subroutine Q5
        implicit none
        integer :: n
        double precision :: a, b, s

        call info(ENDL//"5) "//F9_NAME)
        a = 0.0D0
        b = 4.0D0
        call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")

        n = 4
        call info(":: Integração Polinomial ::")
        call blue('n = '//STR(n))
        s = num_int(f9, a, b, n, kind="polynomial")
        call blue("A = ∫f(x) dx ≈ "//DSTR(s))

        n = 2
        call info(":: Quadratura de Gauss-Legendre ::")
        call blue('n = '//STR(n))
        s = num_int(f9, a, b, n, kind="gauss-legendre")
        call blue("A = ∫f(x) dx ≈ "//DSTR(s))
    end subroutine

    subroutine Q6
        implicit none
        integer :: n
        double precision :: a, b, s

        n = 10

```

```fortran
        call blue('n = '//STR(n))

        call info(ENDL//"6) "//F10_NAME)
        a = 0.0D0
        b = 3.0D0
        call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
        call info(":: Integração Polinomial ::")
        s = num_int(f10, a, b, n, kind="polynomial")
        call blue("A = ∫f(x) dx ≈ "//DSTR(s))
        call info(":: Quadratura de Gauss-Legendre ::")
        s = num_int(f10, a, b, n, kind="gauss-legendre")
        call blue("A = ∫f(x) dx ≈ "//DSTR(s))
        call info(":: Método de Romberg ::")
        s = num_int(f10, a, b, n, kind="romberg")
        call blue("A = ∫f(x) dx ≈ "//DSTR(s))
    end subroutine

    subroutine Q7
        implicit none
        integer :: n
        double precision :: a, b, r, s

        call info(ENDL//"7)")

        n = 10

        call blue('n = '//STR(n))

        call info("A1 ~ "//F11_NAME)
        a = DNINF
        b = 1.0D0
        call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
        call info(":: Quadratura de Gauss-Hermite e de Gauss-
            Legendre ::")
        r = num_int(f11a, a, -a, n, kind="gauss-hermite")
        s = num_int(f11b, -b, b, n, kind="gauss-legendre")
        call blue("A1 = ∫f(x) dx ≈ "//DSTR(r+s))

        call info("A2 ~ "//F12_NAME)
        a = DNINF
        b = DINF
        call info("[a, b] = ["//DSTR(a)//", "//DSTR(b)//"]")
        call info(":: Quadratura de Gauss-Hermite ::")
        s = num_int(f12, a, b, n, kind="gauss-hermite")
        call blue("A2 = ∫f(x) dx ≈ "//DSTR(s))
    end subroutine

    subroutine QE1
        implicit none
        double precision :: x, y, dy

        x = 3.0D0

```

```
        call info(ENDL//'1)')

        call info(FL5_QE1_NAME)
        call info(DFL5_QE1_NAME)

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

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

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

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

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

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

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

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

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

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

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

```

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

    end subroutine

    subroutine QE2
        implicit none
        double precision :: x, y, dy

        x = 2.0D0

        call info(ENDL//'2)')

        call info(FL5_QE2_NAME)
        call info(DFL5_QE2_NAME)

        call info(':: Derivada Analítica ::')
        dy = DFL5_QE2(x)
        call blue("f'("//DSTR(x)//") = "//DSTR(dy))

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

        call info(':: Diferença Central (Δx = 1E-2)::')
        y = d(FL5_QE2, x, dx=1.0D-2, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2)::')
        y = d(FL5_QE2, x, dx=1.0D-2, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2)::')
        y = d(FL5_QE2, x, dx=1.0D-2, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

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

        call info(':: Diferença Central (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE2, x, dx=1.0D-2, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Diferença Central (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE2, x, dx=1.0D-2, kind='forward')
```

```
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE2, x, dx=1.0D-2, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE2, x, dx=1.0D-2, p=2.0D0, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))
    end subroutine

    subroutine QE3
        implicit none
        double precision :: x, y, dy

        x = 6.0D0

        call info(ENDL//'3)')

        call info(FL5_QE3_NAME)
        call info(DFL5_QE3_NAME)

        call info(':: Derivada Analítica ::')
        dy = DFL5_QE3(x)
        call blue("f'("//DSTR(x)//") = "//DSTR(dy))

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

        call info(':: Diferença Central (Δx = 1E-2)::')
        y = d(FL5_QE3, x, dx=1.0D-2, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2)::')
        y = d(FL5_QE3, x, dx=1.0D-2, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2)::')
        y = d(FL5_QE3, x, dx=1.0D-2, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

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

```

```fortran
        call info(':: Diferença Central (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE3, x, dx=1.0D-2, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Diferença Central (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE3, x, dx=1.0D-2, p=2.0D0, kind='central')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE3, x, dx=1.0D-2, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo à frente (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE3, x, dx=1.0D-2, p=2.0D0, kind='forward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2, p = 1)::')
        y = richard(FL5_QE3, x, dx=1.0D-2, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))

        call info(':: Passo atrás (Δx = 1E-2, p = 2)::')
        y = richard(FL5_QE3, x, dx=1.0D-2, p=2.0D0, kind='backward')
        call blue("f'("//DSTR(x)//") ≈ "//DSTR(dy))
        call blue("|δy| = "//DSTR(DABS(y - dy)))
    end subroutine
end program main5
```

Código - Definição das Funções

```fortran
!   Func Module

    module Func
        use Util
        implicit none

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

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

!       >> F3 <<
```

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

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

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

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

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

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

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

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


!       >> F13 <<
```

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

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

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



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

        double precision :: wn = 1.00D0
        double precision :: xi = 0.05D0
        double precision :: Hs = 3.0D0
        double precision :: Tz = 5.0D0

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

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

        character (len = *), parameter :: F11_NAME = "f(x) = exp(- x^2/2) / √(2 π)"
```

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

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


    contains

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

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

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

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

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

```
    end function

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


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

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

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

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

!   ======= R^n -> R^n functions =======
    function f3(x, n) result (y)
!       R^3 -> R^3 (n == 3)
        implicit none
        integer :: n
        double precision, dimension(n) :: x, y

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

```
    end function

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

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

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

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

        J(1, :) = (/ &
            2*x(1), &
            4*x(2), &
            12*x(3) &
            /)
        J(2, :) = (/ &
            12*x(1)*x(2)+36*x(2)*x(3),                          &
            6*x(1)**2+24*x(2)**2+36*x(1)*x(3)+108*x(3)**4,  &
            36*x(1)*x(2)+432*x(2)*x(3)**3                       &
            /)
        J(3, :) = (/ &
```

```fortran
            3+120*x(1)*x(2)**2+72*x(1)**2*x(3)+576*x(2)**2*x(3)+504*x(1)*x(3)**2+1296*x(3)**3,           &
            120*x(1)**2*x(2)+240*x(2)**3+1152*x(1)*x(2)*x(3)+4464*x(2)*x(3)**2,           &
            24*x(1)**3+576*x(1)*x(2)**2+504*x(1)**2*x(3)+4464*x(2)**2*x(3)+3888*x(1)*x(3)**2+13392*x(3)**3 &
            /)
        return
    end function

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

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

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

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

        y = DEXP(-(x*x)/2) / DSQRT(2 * PI)
        return
    end function

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

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

```fortran

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

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

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

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

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

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

    function f8b(w) result (z)
        implicit none
        double precision :: w, z
```

```fortran
            z = (w ** 2) * Ss(w, Sn2)
            return
        end function

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

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

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

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

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

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

        function f13(t) result (y)
            implicit none
```

```fortran
        double precision :: t, y
        y = 1 / (1 + (t**2))
        return
    end function

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

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

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

    end module Func
```

Código - Métodos Numéricos

```fortran
!   Calc Module

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

        character (len=*), parameter :: GAUSS_LEGENDRE_QUAD = "
           quadratures/gauss-legendre/gauss-legendre"
        character (len=*), parameter :: GAUSS_HERMITE_QUAD = "
           quadratures/gauss-hermite/gauss-hermite"
```

```fortran
    contains
!        ================= Numerical Mathods =================
        function d(f, x, dx, kind) result (y)
            implicit none
            character (len=*), optional :: kind
            double precision, optional :: dx
            character (len=:), allocatable :: t_kind
            double precision :: x, y, t_dx

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

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

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

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

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

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

            y = (f(x + xh) - f(x - xh)) / (2 * h)
```

```
            return
        end function

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

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

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

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

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

            return
        end function

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

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

```fortran

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

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

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

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

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

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

```
            end function
        end interface

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

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

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

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

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

```fortran
            end if

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

            ok = .TRUE.

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

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

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

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

```
        end if

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

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

        ok = .TRUE.

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

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

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

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

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

```

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

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

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

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

            ok = .TRUE.

            x = x0

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

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

```

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

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

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

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

            ok = .TRUE.

            x = x0
            xh = 0.0D0

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

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

```

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

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

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

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

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

            ok = .TRUE.

            x = x0
            J = B0

```

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

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

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

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

```
            end interface

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

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

            ok = .TRUE.

            b = b0

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

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

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

```

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

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

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

        ok = .TRUE.

        bh = 0.0D0
        b = b0

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

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

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

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

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

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

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

```

```
    end function

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

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

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

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

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

    function gauss_hermite_int(f, a, b, n) result (s)
        implicit none
```

```
            integer, intent(in) :: n
            double precision, intent(in) :: a, b
            double precision :: s
            double precision, dimension(n) :: xx, ww
            integer :: k
            character(len=*), parameter :: fname =
               GAUSS_HERMITE_QUAD
            interface
                function f(x) result (y)
                    double precision :: x, y
                end function
            end interface

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

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

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

            return
        end function

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

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

            if (n > 1) then
                p = num_int(f, a, b, n / 2, kind = kind)
                q = num_int(f, a, b, n, kind = kind)
                e = DABS(p - q)
```

```fortran
                if (e <= t_tol) then
                    s = q
                else
                    r = (b + a) / 2
                    s = adapt_int(f, a, r, n, tol=t_tol, kind=kind)
                        + adapt_int(f, r, b, n, tol=t_tol, kind=kind)
                end if
                return
            else
                s = 0.0D0
                return
            end if
        end function

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

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

            t_n = ILOG2(n)

            dx = (b - a)

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

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

            do i = 1, t_n
                dx = dx / 2

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

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

```fortran
                end do

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

            deallocate(R)
        end function

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

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

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

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

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

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

```fortran

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

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

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


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

        y(1) = y0
```

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

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

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

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

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

```
        end do
        return
    end function

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

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

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

    function taylor(d2f, y0, dy0, t, n) result (y)
        implicit none
        integer :: k, n
        double precision, intent(in) :: y0, dy0
        double precision :: dt, dy, d2y
        double precision, dimension(n), intent(in) :: t
        double precision, dimension(n) :: y
        interface
            function d2f(t, y, dy) result (d2y)
                implicit none
                double precision :: t, y, dy, d2y
            end function
        end interface
!       Solution
        y(1) = y0
!       1st derivative
```

```
        dy = dy0
        do k=2, n
            dt = t(k) - t(k - 1)
            d2y = d2f(t(k - 1), y(k - 1), dy)
            y(k) = y(k - 1) + (dy * dt) + (d2y * dt ** 2) / 2
            dy = dy + d2y * dt
        end do
        return
    end function

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

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

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

Código - Métodos com Matrizes

```
!   Matrix Module

    module Matrix
        use Util
```

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

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

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

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

```fortran
                read(33,*) A(i,:)
            end do
            close(33)
        end subroutine

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

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

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

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


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

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

            do k=1, n
                if ((a <= x(k)) .AND. (x(k) <= b)) then
                    y(k) = x(k)
                else
```

```fortran
                    y(k) = DNAN
                end if
            end do
            return
        end function

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

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

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

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

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

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

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

```
        end function

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

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

            G(:, :) = id_matrix(n)

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

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

            return
        end function

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

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

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

            logical :: ok
            integer :: i

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

```fortran
        end function

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

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

            logical :: ok

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

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

            double precision :: A(n, n)

            integer :: i, j
            logical :: ok

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

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

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

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

```fortran
            A(j, :) = temp(:)
        end subroutine

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

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

            logical :: ok

            ! External procedures defined in LAPACK
            external DGETRF
            external DGETRI

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

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

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

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

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

```fortran
        end if

        return
    end function

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

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

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

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

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

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

        double precision :: P(n, n)

        integer :: j, k

        P = id_matrix(n)

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

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

```fortran

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

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

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

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

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

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

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

```
                    X(:, i:    ) = A(2:, i+1:    )
                    d = s * det(X, n-1) * A(1, i) + d
                    s = -s
                end do
            end if
            return
        end function

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

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

            d = 0.0D0

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

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

            return
        end function

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

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

            integer :: i

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

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

!       === Matrix Factorization Conditions ===
        function Cholesky_cond(A, n) result (ok)
            implicit none
            integer :: n
```

```fortran
            double precision :: A(n, n)
            logical :: ok
            ok = symmetrical(A, n) .AND. positive_definite(A, n)
            return
        end function

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

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

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

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

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

             ok = .TRUE.
             return

         end function

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

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

             logical :: ok

             integer :: i, j

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

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

```fortran
                L(j, i) = (A(i, j) - sum(L(i, :i-1) * L(j, :i-1))) / L(i, i)
            end do
        end do

        ok = .TRUE.
        return
    end function

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

        integer :: n

        double precision :: A(n, n)

        logical :: ok

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

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

        logical :: ok

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

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

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

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

```

```fortran
        x0 = rand_vector(n)

        ok = Jacobi_cond(A, n)

        if (.NOT. ok) then
            return
        end if

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

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

        integer :: n

        double precision :: A(n, n)

        logical :: ok

        integer :: i

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

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

```

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

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

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

            ok = Gauss_Seidel_cond(A, n)

            if (.NOT. ok) then
                return
            end if

            do k = 1, t_max_iter
                do i = 1, n
                    s = 0.0D0
                    do j = 1, n
                        if (i /= j) then
                            s = s + A(i, j) * x(j)
                        end if
                    end do
                    x(i) = (b(i) - s) / A(i, i)
                end do
                e = vector_norm(matmul(A, x) - b, n)
                if (e < t_tol) then
                    return
                end if
            end do
            call error('Erro: Esse método não convergiu.')
            ok = .FALSE.
            return
        end function

!       Decomposição LU e afins
        subroutine LU_backsub(L, U, x, y, b, n)
            implicit none
            integer :: n
```

```fortran
            double precision :: L(n, n), U(n, n)
            double precision :: b(n), x(n), y(n)
            integer :: i
!           Ly = b (Forward Substitution)
            do i = 1, n
                y(i) = (b(i) - SUM(L(i, 1:i-1) * y(1:i-1))) / L(i, i
                    )
            end do
!           Ux = y (Backsubstitution)
            do i = n, 1, -1
                x(i) = (y(i) - SUM(U(i,i+1:n) * x(i+1:n))) / U(i, i)
            end do
        end subroutine

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

            integer :: n

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

            logical :: ok

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

            if (.NOT. ok) then
                return
            end if

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

            return
        end function

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

            integer :: n

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

            logical :: ok

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

            if (.NOT. ok) then
                return
            end if

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

```

```fortran
            x(:) = matmul(P, x)

            return
        end function

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

            integer :: n

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

            logical :: ok

            ok = Cholesky_decomp(A, L, n)

            if (.NOT. ok) then
                return
            end if

            U = transpose(L)

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

            return
        end function

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

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

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

```

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

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

!           Initialize Eigenvalues
            l = 0.0D0

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

                x(:) = matmul(A, x)

!               Retrieve Eigenvalue
                l = x(1)

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

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

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

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

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

```
                t_max_iter = max_iter
            end if

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

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

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

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

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

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

            logical :: ok

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

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

```
            A(2, 2) = dot_product(x, x)

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

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

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

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

            logical :: ok

            ok = .TRUE.

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

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

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

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

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

```
                        A(k,j) = A(l,j)
                        A(l,j) = store
                    end do
                    store = b(k)
                    b(k) = b(l)
                    b(l) = store
                end if

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

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

            return
        end function

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

            logical :: ok = .TRUE.

            if (.NOT. PRESENT(kind)) then
                call debug("Indeed, not present.")
                t_kind = "gauss"
            else
                t_kind = kind
            end if

            call debug("Now it is: "//t_kind)
            if (t_kind == "LU") then
                ok = LU_solve(A, x, y, b, n)
            else if (t_kind == "PLU") then
                ok = PLU_solve(A, x, y, b, n)
```

```
            else if (t_kind =="cholesky") then
                ok = Cholesky_solve(A, x, y, b, n)
            else if (t_kind =="gauss") then
                ok = Gauss_solve(A, x, b, n)
            else
                ok = .FALSE.
            end if

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

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

            return
        end function

    end module Matrix
```

Código - Biblioteca Auxiliar

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

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

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

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

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

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

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

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

```
        logical :: y

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

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

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

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

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

            r = t_q//s//t_q
        end function

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

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

        function ILOG2(n) result (k)
            integer, intent(in) :: n
            integer :: k
            double precision :: x
            x = n
            x = DLOG2(x)
            k = FLOOR(x)
            return
        end function

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

```

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

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

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

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

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

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

```
                    exit
                else
                    k = j
                    exit
                end if
            end do
            q = p(:k)
            return
        end function

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

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

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

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

        subroutine info(text)
!           Green Text
            implicit none
            character(len=*) :: text
```

```
            call display(text, '32')
        end subroutine

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

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

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

            i = u
            j = v

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

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

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

```
            end if
            return
        end subroutine

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

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

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

Código - Biblioteca de Plotagem

```
module PlotLib
    use Util
    implicit none
    character(len=*), parameter :: DEFAULT_FNAME = 'plotfile'
    character(len=*), parameter :: DEFAULT_SIZE_W = '12in'
    character(len=*), parameter :: DEFAULT_SIZE_H = '9in'
    character(len=*), parameter :: PLOT_ENDL = ',\'//ENDL

    logical :: g_INPLOT = .FALSE.
    logical :: g_INMULTIPLOT = .FALSE.

    character(len=:), allocatable :: g_FNAME, g_OUTP_FNAME,
       g_PLOT_FNAME, g_SIZE_W, g_SIZE_H

    integer :: g_M, g_N

    type SPLOT
        integer :: i, j
        integer :: n = 0
        logical :: grid = .FALSE.
        logical :: done = .FALSE.

        character(len=:), allocatable :: title, xlabel, ylabel
        type(StringArray), dimension(:), allocatable :: legend
        type(StringArray), dimension(:), allocatable :: with

```

```
!        Plot boundaries
         logical :: l_xmin = .FALSE., l_xmax = .FALSE.
         logical :: l_ymin = .FALSE., l_ymax = .FALSE.
         double precision :: xmin, xmax, ymin, ymax
     end type

     type(SPLOT), dimension(:, :), allocatable :: g_SUBPLOTS

     contains

     function REMOVE_TEMP_FILES(fname) result (cmd)
         implicit none
         character(len=*) :: fname
         character(len=:), allocatable :: plt
         character(len=:), allocatable :: dat
         character(len=:), allocatable :: cmd

         plt = 'plot/'//fname//'*.plt'
         dat = 'plot/'//fname//'*.dat'

         cmd = 'rm '//plt//' '//dat
         return
     end function

     function PLOT_FNAME(fname) result (path)
         implicit none
         character(len=*) :: fname
         character(len=:), allocatable :: path
         path = 'plot/'//fname//'.plt'
         return
     end function

     function DATA_FNAME(fname, i, j, n) result (path)
         implicit none
         integer, optional, intent(in) :: i, j, n
         character(len=*) :: fname
         character(len=:), allocatable :: path, t_i, t_j, t_n

         if (.NOT. PRESENT(i)) then
             t_i = '1'
         else
             t_i = STR(i)
         end if

         if (.NOT. PRESENT(j)) then
             t_j = '1'
         else
             t_j = STR(j)
         end if

         if (.NOT. PRESENT(n)) then
             t_n = '1'
         else
```

```fortran
            t_n = STR(n)
        end if

        path = 'plot/'//fname//'_'//t_i//'_'//t_j//'_'//t_n//'.dat'
        return
    end function

    function OUTP_FNAME(fname) result (path)
        implicit none
        character(len=*) :: fname
        character(len=:), allocatable :: path
        path = 'plot/'//fname//'.pdf'
        return
    end function

    function GNU_PLOT_CMD(fname) result (cmd)
        implicit none
        character(len=*) :: fname
        character(len=:), allocatable :: cmd
        cmd = 'gnuplot -p '//PLOT_FNAME(fname)
    end function

    subroutine subplot_config(i, j, title, xlabel, ylabel, xmin, xmax, ymin, ymax, legend, with, grid)
        integer, intent(in) :: i, j
        integer :: k, n

        logical, optional :: grid

        character(len=*), optional :: title, ylabel, xlabel

        double precision, optional :: xmin, xmax, ymin, ymax

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

        if (g_SUBPLOTS(i, j)%done) then
            call error("Duplicate configuration of subplot ("//STR(i)//", "//STR(j)//")")
            stop "ERROR"
        end if

        n = g_SUBPLOTS(i, j)%n

        allocate(g_SUBPLOTS(i, j)%legend(n))
        allocate(g_SUBPLOTS(i, j)%with(n))

        if (PRESENT(xmin)) then
            g_SUBPLOTS(i, j)%xmin = xmin
            g_SUBPLOTS(i, j)%l_xmin = .TRUE.
        end if

        if (PRESENT(xmax)) then
            g_SUBPLOTS(i, j)%xmax = xmax
```

```fortran
            g_SUBPLOTS(i, j)%l_xmax = .TRUE.
        end if

        if (PRESENT(ymin)) then
            g_SUBPLOTS(i, j)%ymin = ymin
            g_SUBPLOTS(i, j)%l_ymin = .TRUE.
        end if

        if (PRESENT(ymax)) then
            g_SUBPLOTS(i, j)%ymax = ymax
            g_SUBPLOTS(i, j)%l_ymax = .TRUE.
        end if

        if (.NOT. PRESENT(legend)) then
            do k=1,n
                g_SUBPLOTS(i, j)%legend(k)%str = 't '//quote(STR(i))
            end do
        else
            do k=1,n
                g_SUBPLOTS(i, j)%legend(k)%str = 't '//quote(legend(k)%str)
            end do
        end if

        if (.NOT. PRESENT(with)) then
            do k=1,n
                g_SUBPLOTS(i, j)%with(k)%str = 'w lines'
            end do
        else
            do k=1,n
                g_SUBPLOTS(i, j)%with(k)%str = 'w '//with(k)%str
            end do
        end if

        if (.NOT. PRESENT(grid)) then
            g_SUBPLOTS(i, j)%grid = .TRUE.
        else
            g_SUBPLOTS(i, j)%grid = grid
        end if

        if (.NOT. PRESENT(title)) then
            g_SUBPLOTS(i, j)%title = ''
        else
            g_SUBPLOTS(i, j)%title = title
        end if

        if (.NOT. PRESENT(xlabel)) then
            g_SUBPLOTS(i, j)%xlabel = 'x'
        else
            g_SUBPLOTS(i, j)%xlabel = xlabel
        end if

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

```fortran
                g_SUBPLOTS(i, j)%ylabel = 'y'
            else
                g_SUBPLOTS(i, j)%ylabel = ylabel
            end if

            g_SUBPLOTS(i, j)%done = .TRUE.

        end subroutine

        subroutine subplot(i, j, x, y, n)
            integer :: file, k
            integer, intent(in) :: i, j, n
            double precision, dimension(n), intent(in) :: x
            double precision, dimension(n), intent(in) :: y

            character(len=:), allocatable :: s_data_fname

            if (g_SUBPLOTS(i, j)%done) then
                call error("Plot over finished subplot ("//STR(i)//", "//STR(j)//")")
                stop "ERROR"
            else
                g_SUBPLOTS(i, j)%n = g_SUBPLOTS(i, j)%n + 1
            end if

            s_data_fname = DATA_FNAME(g_FNAME, i, j, g_SUBPLOTS(i, j)%n)

!           =================== Touch Plot File ===================================
            open(newunit=file, file=s_data_fname, status="replace", action="write")
                write(file, *) "# file: "//s_data_fname
            close(file)
!           ========================================================================

!           ==================== Write to Plot File ==============================================
10          format(F16.8, ' ')
11          format(F16.8, ' ')
            open(newunit=file, file=s_data_fname, status="old", position="append", action="write")
            write(file, *)
            do k=1, n
                write(file, 10, advance='no') x(k)
                write(file, 11, advance='yes') y(k)
            end do
            close(file)
!           ========================================================================================

        end subroutine
```

```fortran

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

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

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

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

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

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

        g_INPLOT = .TRUE.

    end subroutine

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

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

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

```fortran

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

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

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

        logical, optional :: clean
        logical :: t_clean

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

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

        m = g_M
        n = g_N

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

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

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

```

```fortran
                write(file, *) 'set title '//quote(g_SUBPLOTS(i, j)%title)//';'
                write(file, *) 'set xlabel '//quote(g_SUBPLOTS(i, j)%xlabel)//';'
                write(file, *) 'set ylabel '//quote(g_SUBPLOTS(i, j)%ylabel)//';'

!               ============= Set XRANGE =======================================
                if (g_SUBPLOTS(i, j)%l_xmin .AND. g_SUBPLOTS(i, j)%l_xmax) then
                    write(file, *) 'set xrange ['//DSTR(g_SUBPLOTS(i, j)%xmin)//':'//DSTR(g_SUBPLOTS(i, j)%xmax)//'];'
                else if (g_SUBPLOTS(i, j)%l_xmin) then
                    write(file, *) 'set xrange ['//DSTR(g_SUBPLOTS(i, j)%xmin)//':*];'
                else if (g_SUBPLOTS(i, j)%l_xmax) then
                    write(file, *) 'set xrange [*:'//DSTR(g_SUBPLOTS(i, j)%xmax)//'];'
                else
                    write(file, *) 'set xrange [*:*];'
                end if
!               ================================================================

!               ============= Set YRANGE =======================================
                if (g_SUBPLOTS(i, j)%l_ymin .AND. g_SUBPLOTS(i, j)%l_ymax) then
                    write(file, *) 'set yrange ['//DSTR(g_SUBPLOTS(i, j)%ymin)//':'//DSTR(g_SUBPLOTS(i, j)%ymax)//'];'
                else if (g_SUBPLOTS(i, j)%l_ymin) then
                    write(file, *) 'set yrange ['//DSTR(g_SUBPLOTS(i, j)%ymin)//':*];'
                else if (g_SUBPLOTS(i, j)%l_ymax) then
                    write(file, *) 'set yrange [*:'//DSTR(g_SUBPLOTS(i, j)%ymax)//'];'
                else
                    write(file, *) 'set yrange [*:*];'
                end if
!               ================================================================

                if (g_SUBPLOTS(i, j)%grid) then
                    write(file, *) 'set grid;'
                else
                    write(file, *) 'unset grid;'
                end if

                write(file, 10, advance='no') 'plot'

```

```fortran
                do k = 1, g_SUBPLOTS(i, j)%n
                    write(file, 10, advance='no') quote(DATA_FNAME(g_FNAME, i, j, k))
                    write(file, 10, advance='no') 'u 1:2'
                    write(file, 10, advance='no') g_SUBPLOTS(i, j)%legend(k)%str
                    write(file, 10, advance='no') g_SUBPLOTS(i, j)%with(k)%str

                    if (k == (g_SUBPLOTS(i, j)%n)) then
                        write(file, *) ';'
                    else
                        write(file, *) ',\'
                    end if
                end do
            end do
        end do
!       ===================================

!       == Finish Multiplot ===
        if (g_INMULTIPLOT) then
            write(file, *) 'unset multiplot'
            g_INMULTIPLOT = .FALSE.
        end if
!       =======================
        close(file)
!   ===============================================================================

!       ===== Call GNUPLOT and remove temporary files ===========
        call EXECUTE_COMMAND_LINE(GNU_PLOT_CMD(g_FNAME))

        if (t_clean) then
            call EXECUTE_COMMAND_LINE(REMOVE_TEMP_FILES(g_FNAME))
        end if
!       =========================================================

!       ========= Free Variables ======================
        deallocate(g_FNAME, g_OUTP_FNAME, g_PLOT_FNAME)
        deallocate(g_SUBPLOTS)
        g_INPLOT = .FALSE.
!       ===============================================
    end subroutine
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[ ]:= (* Set directory to current one *)
SetDirectory[NotebookDirectory[] <> "/gauss-hermite"]
W[n_, x_] := (2^(n-1) * (n!) * Sqrt[π]) / (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];
  ]
```