Universidade de São Paulo
Instituto de Física de São Carlos

Lista 1

Pedro Calligaris Delbem 5255417

Professor: Attilio Cucchieri

Março de 2025

Sumário

1 Exercício 1

Tarefa: Calcular a área de um círculo. Opções: entrada e saída usando teclado e monitor; entrada e saída usando arquivos; calcular a área em uma subrotina.

Código Escrito:

```fortran
!----------------------------------------------------------------
! File: L1-5255417-ex-1.f90
!
! Description:
!   Computes the area of a circle.
!
! Dependencies:
!   - None
!
! Since:
!   - 03/2025
!
! Authors:
!   - Pedro C. Delbem <pedrodelbem@usp.br>
!----------------------------------------------------------------
program circle_area

    !deactivate implicit typing
    implicit none

    !define variables
    real radius, area

    !request radius value to the user
    write(*,*) 'Insert radius value:'

    !read user input
    read(*,*) radius

    !call calculate_area
    call calculate_area(radius, area)

    !print result
    write(*,*) 'Area of the circle is:', area

contains

    subroutine calculate_area(radius, area)

        !deactivate implicit typing
        implicit none

        !define variables
        real, intent(in) :: radius
        real, intent(out) :: area

        !calculate area (4*atan(1) = pi)
        area = 4*atan(1.)*radius**2

    end subroutine calculate_area
```

```
end program circle_area
```

Resultados:

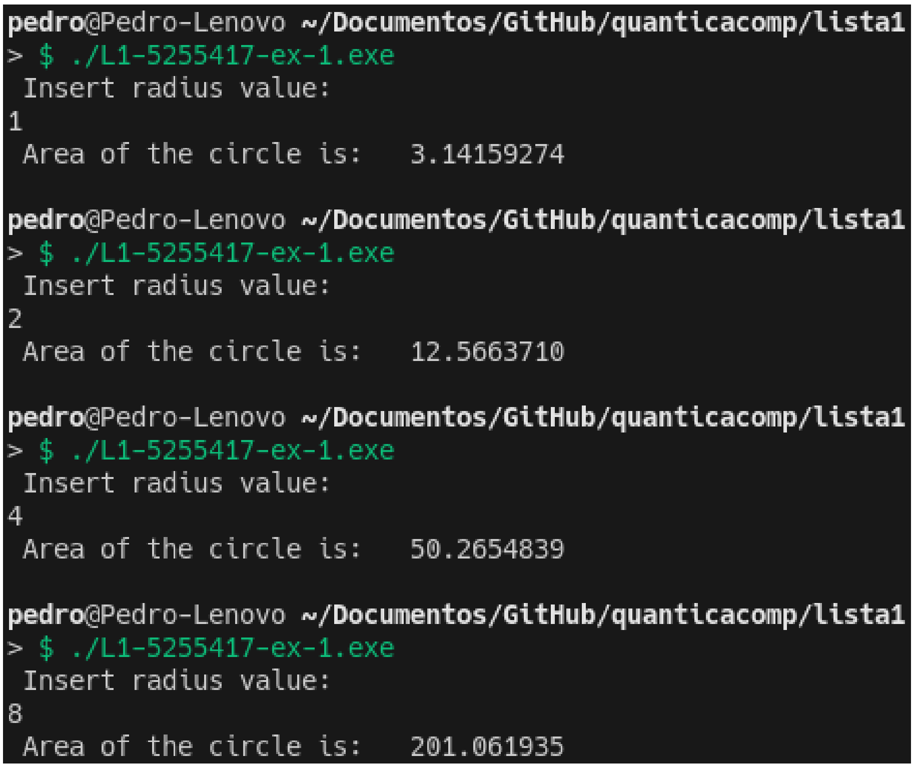


Figura 1: Valores de área para os raio de 1, 2, 4 e 8.

A área para o raio = 1 correponde ao valor de π, de acordo com o esperado. Dobrando o valor do raio espera-se, então, que o valor da área quadruplique e isto foi o que se obteve.

2 Exercício 2

Tarefa: Testar overflow e underflow em precisão simples e dupla.

Código Escrito:

```
!-------------------------------------------------------------
! File: L1-5255417-ex-2.f90
!
! Description:
!   Computes overflow and underflow
!
! Dependencies:
!   - None
!
! Since:
```

```fortran
!    - 03/2025
!
! Authors:
!    - Pedro C. Delbem <pedrodelbem@usp.br>
!-------------------------------------------------------------
program testing_overflow_and_underflow

    !deactivate implicit typing
    implicit none

    !define variables
    real test_overflow, real_overflow, test_underflow, real_underflow
    real*8 test_overflow8, real_overflow8, test_underflow8, real_underflow8

    !initialize overflow variables
    test_overflow = 1.0
    test_overflow8 = 1.0

    !call overflows computations
    call compute_overflow(test_overflow,real_overflow)
    call compute_overflow8(test_overflow8,real_overflow8)

    !initialize underflow variables
    test_underflow = 1.0
    test_underflow8 = 1.0

    !call underflows computations
    call compute_underflow(test_underflow,real_underflow)
    call compute_underflow8(test_underflow8,real_underflow8)

    !print results
    write(*,*)'Overflow simple real is: ', real_overflow
    write(*,*)'Overflow double real is: ', real_overflow8
    write(*,*)'Underflow simple real is: ', real_underflow
    write(*,*)'Underflow double real is: ', real_underflow8

contains

    !computes single precision overflow
    subroutine compute_overflow(test_overflow,real_overflow)

        !deactivate implicit typing
        implicit none

        !define variables
        real, intent(inout) :: test_overflow
        real, intent(inout) :: real_overflow

        !compute overflow
        do while (test_overflow < 2.0*test_overflow) !test_overflow => 2*test_overflow implies overflow

            !update real_overflow variable which saves the value before exceeding the overflow
            real_overflow = test_overflow

```

```
         !update test_overflow variable to test the next value
         test_overflow = test_overflow*10

     end do

 end subroutine compute_overflow

 !computes double precision overflow
 subroutine compute_overflow8(test_overflow8,real_overflow8)

     !deactivate implicit typing
     implicit none

     !define variables
     real*8, intent(inout) :: test_overflow8
     real*8, intent(inout) :: real_overflow8

     !compute overflow
     do while (test_overflow8 < 2.0*test_overflow8) ! test_overflow8 => 2*test_overflow8 implies overflow

         !update real_overflow8 variable which saves the value before exceeding the overflow
         real_overflow8 = test_overflow8

         !update test_overflow8 variable to test the next value
         test_overflow8 = test_overflow8*10

     end do

 end subroutine compute_overflow8

 !computes single precision underflow
 subroutine compute_underflow(test_underflow,real_underflow)

     !deactivate implicit typing
     implicit none

     !define variables
     real, intent(inout) :: test_underflow
     real, intent(inout) :: real_underflow

     !compute underflow
     do while (test_underflow > 0.5*test_underflow) ! test_underflow > 0.5*test_underflow implies underflow

         !update real_underflow variable which saves the value before exceeding the underflow
         real_underflow = test_underflow

         !update test_underflow variable to test the next value
         test_underflow = test_underflow/10

     end do

 end subroutine compute_underflow

 !computes double precision underflow
```

```
    subroutine compute_underflow8(test_underflow8,real_underflow8)

        !deactivate implicit typing
        implicit none

        !define variables
        real*8, intent(inout) :: test_underflow8
        real*8, intent(inout) :: real_underflow8

        !compute underflow
        do while (test_underflow8 > 0.5*test_underflow) ! test_underflow8 > 0.5*test_underflow8 implies underflow

            !update real_underflow8 variable which saves the value before exceeding the underflow
            real_underflow8 = test_underflow8

            !update test_underflow8 variable to test the next value
            test_underflow8 = test_underflow8/10

        end do

    end subroutine compute_underflow8

end program testing_overflow_and_underflow
```

Resultados:

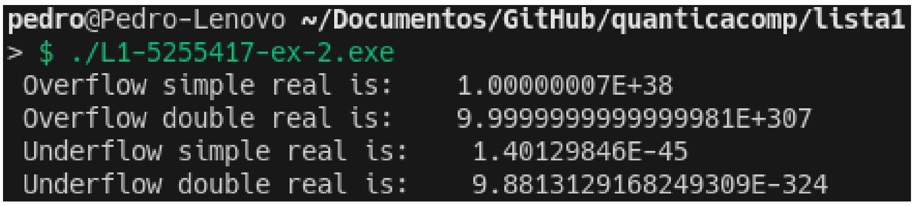


Figura 2: Valores de overflow e underflow para precisão simples e dupla.

Nota-se que os valores obtidos são coerentes com o esperado.

3 Exercício 3

Tarefa: Achar a precisão do computador, i.e., o maior número positivo tal que $1 + \epsilon = 1$, usando precisão simples e dupla.

Código Escrito:

```
!-------------------------------------------------------------
! File: L1-5255417-ex-3.f90
!
! Description:
!   Finds computer precision
!
```

```fortran
! Dependencies:
!    - None
!
! Since:
!    - 03/2025
!
! Authors:
!    - Pedro C. Delbem <pedrodelbem@usp.br>
!-------------------------------------------------------------
program computer_precision

    !deactivate implicit typing
    implicit none

    !define variables
    real*4 real4, sum4, aux4
    real*8 real8, sum8, aux8

    !initialize variables
    real4 = 1.0
    sum4 = 1.0
    aux4 = 1.0
    real8 = 1.0
    sum8 = 1.0
    aux8 = 1.0

    !find machine precision
    do while (sum4+real4 /= sum4) !while sum4+real4 is not equal to sum4

        !define aux4 to save the value before exceeding machine precision
        real4 = aux4

        !divide real4 by 2 and save the value in aux4
        aux4 = real4*0.5

    end do

    !find machine precision
    do while (sum8+real8 /= sum8) !while sum4+real4 is not equal to sum4

        !define aux8 to save the value before exceeding machine precision
        real8 = aux8

        !divide real4 by 2 and save the value in aux8
        aux8 = real8*0.5d0

    end do

    !print results
    write(*,*) "Machine precision for simple precision is:", real4
    write(*,*) "Machine precision for double precision is:", real8

end program computer_precision
```

Resultados:

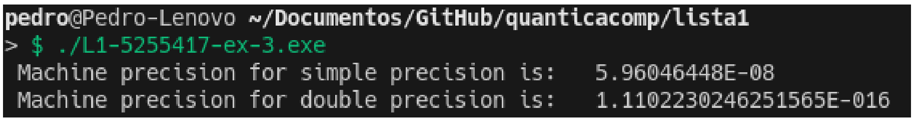


Figura 3: Valores de ϵ para precisão para precisão simples e dupla.

Nota-se que os valores obtidos correspondem aos valores esperados.

4 Exercício 4

Tarefa: Calcular

$$e^{-x} = 1 - x + x^2/2! - x^3/3! + ... \tag{1}$$

para x = 0.1, 1, 10, 100 e 1000 com um erro menor do que 10^{-8}. Problema: quando truncar a série? É preciso calcular o fatorial explicitamente? Comparar o valor obtido usando a série com o resultado exato.

Código Escrito:

```
!-------------------------------------------------------------
! File: L1-5255417-ex-4.f90
!
! Description:
!   Computes taylor series for exponential of -x
!
! Dependencies:
!   - None
!
! Since:
!   - 03/2025
!
! Authors:
!   - Pedro C. Delbem <pedrodelbem@usp.br>
!-------------------------------------------------------------
program exponential_taylor_series

    !deactivate implicit typing
    implicit none

    !define variables
    integer i
    real*16 x(5), e_x, index, old_term, next_term, precision

    !initialize variables
    x = [0.1,1.0,10.0,100.0,1000.0]
    precision = 1_16*10**(-8)
```

```fortran

    do i=1,5

        !initialize variables
        e_x = 1_16
        index = 1_16
        next_term = 1_16
        old_term = 1_16

        !call compute_exponential
        call compute_exponential(x(i), e_x, index, old_term, next_term, precision)

        !print result
        write(*,*) "x = ", x(i)
        write(*,*) "    Computed e^(-", x(i), ") = ", e_x
        write(*,*) "    Real e^(-", x(i), ") = ", exp(-x(i))

    end do

contains

    subroutine compute_exponential(x, e_x, index, old_term, next_term, precision)

        !deactivate implicit typing
        implicit none

        !define variables
        real*16, intent(in) :: x
        real*16, intent(in) :: precision
        real*16, intent(inout) :: index
        real*16, intent(inout) :: next_term
        real*16, intent(inout) :: old_term
        real*16, intent(out) :: e_x

        !compute next term
        old_term = next_term
        next_term = next_term*(-x)/index

        !update e_x
        e_x = e_x + next_term

        !update index and factorial
        index = index + 1_16

        !update e_x while the absolute value of next_term is greater than precision
        do while (abs(next_term-old_term) > abs(precision*e_x))

            !compute next term
            old_term = next_term
            next_term = next_term*(-x)/index

            !update e_x
            e_x = e_x + next_term

            !update index and factorial
```

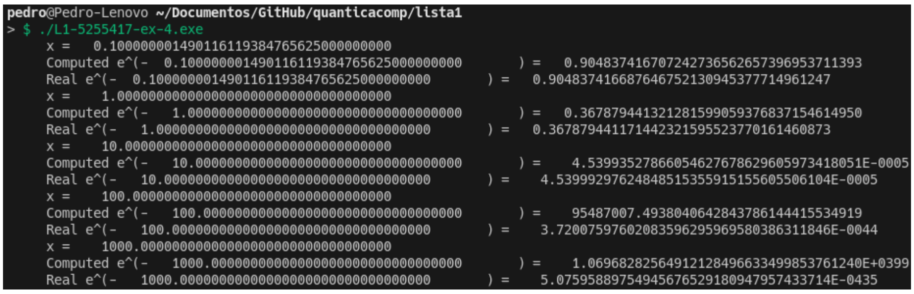
```fortran
            index = index + 1_16

        end do

    end subroutine compute_exponential

end program exponential_taylor_series
```

Resultados:



```
pedro@Pedro-Lenovo ~/Documentos/GitHub/quanticacomp/lista1
> $ ./L1-5255417-ex-4.exe
 x =   0.100000001490116119384765625000000000
 Computed e^(-  0.100000001490116119384765625000000000        ) =   0.904837416707242736562657396953711393
 Real e^(-  0.100000001490116119384765625000000000        ) =   0.904837416687646752130945377714961247
 x =    1.00000000000000000000000000000000000
 Computed e^(-   1.00000000000000000000000000000000000        ) =   0.367879441321281599059376837154614950
 Real e^(-   1.00000000000000000000000000000000000        ) =   0.367879441171442321595523770161460873
 x =    10.0000000000000000000000000000000000
 Computed e^(-   10.0000000000000000000000000000000000        ) =    4.53993527866054627678629605973418051E-0005
 Real e^(-   10.0000000000000000000000000000000000        ) =    4.53999297624848515355915155605506104E-0005
 x =    100.000000000000000000000000000000000
 Computed e^(-   100.000000000000000000000000000000000        ) =    95487007.4938040642843786144415534919
 Real e^(-   100.000000000000000000000000000000000        ) =    3.72007597602083596295969580386311846E-0044
 x =    1000.00000000000000000000000000000000
 Computed e^(-   1000.00000000000000000000000000000000        ) =    1.06968282564912128496633499853761240E+0399
 Real e^(-   1000.00000000000000000000000000000000        ) =    5.07595889754945676529180947957433714E-0435
```

Figura 4: Valores de e^{-x} para x = 0.1, 1, 10, 100 e 1000.

Descrição: O código faz um loop onde, em cada iteração, calcula o valor da série para um valor de x diferente. Na subrotina "compute-exponencial" calcula-se a série definindo o próximo termo como a multiplicação do termo anterior por -x/n, pois - deste modo - não se faz necessário calcular o fatorial explicitamente. Ademais, interrompe-se a soma quando o termo atual for menor que 10^{-8} garantindo a precisão desejada, uma vez que cada termo da série é menor - em módulo - que o anterior.

Por fim, percebe-se que o resultado eperado foi obtido para todos os casos testados - com exeção dos casos onde o resultado é menor que a precisão.

5 Exercício 5

Tarefa: Considerar a somatória

$$\Sigma(N) = \sum_{n=1}^{2N} (-1)^n \frac{n}{n+1} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)} \quad (2)$$

e calcular $\Sigma(N)$ para N = 1, 2, ... , 10^6 usando as três fórmulas acima. Comparar os resultados usando precis ão simples.

Código Escrito:

```fortran
!---------------------------------------------------------------
! File: L1-5255417-ex-5.f90
```

```
!
! Description:
!   Computes sums of series
!
! Dependencies:
!   - None
!
! Since:
!   - 03/2025
!
! Authors:
!   - Pedro C. Delbem <pedrodelbem@usp.br>
!------------------------------------------------------------
program exponential_taylor_series

    !deactivate implicit typing
    implicit none

    !define variables
    integer N(14), i
    real sum1, sum2, sum3

    !initialize variables
    N = [1,2,3,4,5,6,7,9,10**1,10**2,10**3,10**4,10**5,10**6]

    do i=1,14

        !initialize variables
        sum1 = 0.0
        sum2 = 0.0
        sum3 = 0.0

        !compute series
        call compute_series(sum1, sum2, sum3, N(i))

        !print result
        write(*,*) N(i), sum1, sum2, sum3

    end do

contains

    subroutine compute_series(sum1, sum2, sum3, N)

        !deactivate implicit typing
        implicit none

        !define variables
        integer i
        integer, intent(in) :: N
        real x, sum2a, sum2b
        real, intent(inout) :: sum1
        real, intent(inout) :: sum2
        real, intent(inout) :: sum3

        !initialize partial sums
        sum2a = 0.0
        sum2b = 0.0
```

```fortran

        !compute sum1
        do i=1,N

            !update x
            x = i

            !compute sums
            sum1 = sum1 + (-1)**(x)*x/(x+1)
            sum2a = sum2a + (2.0*x-1)/(2.0*x)
            sum2b = sum2b + (2.0*x)/(2.0*x+1)
            sum3 = sum3 + 1/(2.0*x*(2.0*x+1))

        end do!

        !update sum1
        do i=N,2*N

            !update x
            x = i

            !compute sums
            sum1 = sum1 + (-1)**(x)*x/(x+1)

        end do

        !update sum2
        sum2 = -sum2a + sum2b

    end subroutine compute_series

end program exponential_taylor_series
```

Resultados:

```
pedro@Pedro-Lenovo ~/Documentos/GitHub/quanticacomp/lista1
> $ ./L1-5255417-ex-5.exe
           1 -0.333333313      0.166666687      0.166666672
           2  0.883333385      0.216666698      0.216666669
           3 -0.509523690      0.240476370      0.240476191
           4   1.05436516      0.254365206      0.254365087
           5 -0.569877207      0.263456345      0.263455987
           6   1.12700903      0.269866943      0.269866258
           7 -0.600371778      0.274629116      0.274628162
           9 -0.618771255      0.281229973      0.281228602
          10   1.19270074      0.283611298      0.283609569
         100   1.29447055      0.304382324      0.304371417
        1000   1.30560207      0.306640625      0.306603163
       10000   1.30670857      0.306640625      0.306800693
      100000   1.30684614      0.304687500      0.306800693
     1000000   1.30686092      0.312500000      0.306800693
```

Figura 5: Valores de $\Sigma(N)$ para N $= 1, 2, \ldots, 10^6$ usando precisão simples.

Nota-se que a primeira série demora é mais instável do que as demais. Além disso, a terceira série se mostra mais estável que a segunda - sendo então a melhor versão.

6 Exercício 6

Tarefa: Estudar numericamente o erro da aproximação

$$e^{-x} \approx \sum_{n=0}^{N} \frac{(-x)^n}{n!} \tag{3}$$

em função de N, para diferentes valores de x. Sugestão: faça um gráfico do erro em função de N. O que acontece quando e^{-x} é calculado usando a série $e^x = \sum_{n=0}^{N} \frac{x^n}{n!}$ e, depois, calculando $1/e^x$?

Código Escrito:

```fortran
!------------------------------------------------------------
! File: L1-5255417-ex-6.f90
!
! Description:
!   Computes taylor series for exponential of -x with function of N
!
! Dependencies:
!   - None
!
! Since:
!   - 03/2025
!
! Authors:
!   - Pedro C. Delbem <pedrodelbem@usp.br>
!------------------------------------------------------------
program exponential_taylor_series

    !deactivate implicit typing
    implicit none

    !define variables
    integer N(14), i, j
    real exponential_of_minus_x, exponential_of_x, x(5)

    !initialize variables
    N = [1,2,3,4,5,6,7,9,10**1,10**2,10**3,10**4,10**5,10**6]
    x = [0.1,1.0,10.0,100.0,1000.0]

    open(unit=1, file='exponential_taylor_series.txt', status='replace')

    do i=1,5

        !initialize variables
        exponential_of_minus_x = 0.0
        exponential_of_x = 0.0
```

```fortran

        write(1,*) 'x:', x(i)

        do j=1,14
            !compute series
            call compute_series(exponential_of_minus_x, exponential_of_x, N(j), x(i))

            !compute exponential of -x
            exponential_of_x = 1/exponential_of_x

            !print result
            write(1,*) 'N:', N(j), exponential_of_minus_x, exponential_of_x, exp(-x(i))

        end do

    end do

    close(1)

contains

    subroutine compute_series(exponential_of_minus_x, exponential_of_x, N, x)

        !deactivate implicit typing
        implicit none

        !define variables
        integer i
        integer, intent(in) :: N
        real j, next_term_minus_x, next_term_x
        real, intent(in) :: x
        real, intent(inout) :: exponential_of_minus_x
        real, intent(inout) :: exponential_of_x

        !initialize next terms
        next_term_minus_x = 1.0
        next_term_x = 1.0

        !compute sum1
        do i=1,N

            !update j
            j = i

            !compute next terms
            next_term_minus_x = next_term_minus_x*(-x)/j
            next_term_x = next_term_x*x/j

            !compute sums
            exponential_of_minus_x = exponential_of_minus_x + next_term_minus_x
            exponential_of_x = exponential_of_x + next_term_x

        end do

```

```
    end subroutine compute_series

end program exponential_taylor_series
```

Resultados:

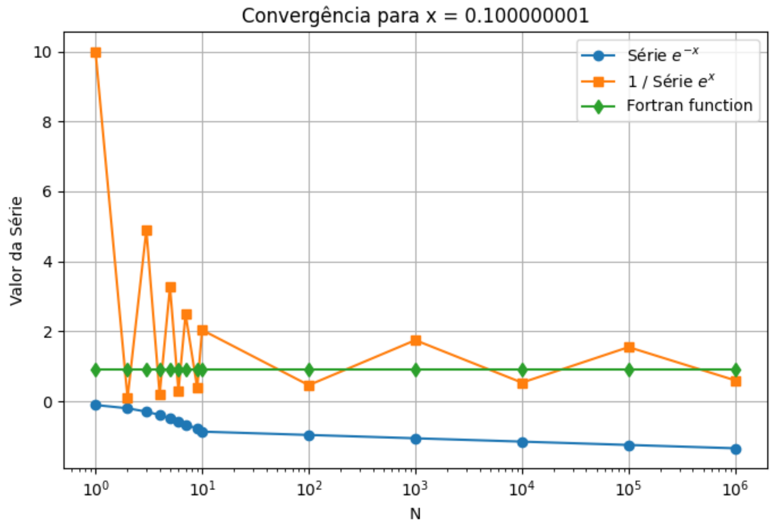


Figura 6: Gráfico do erro em função de N para x = 0.100000001.

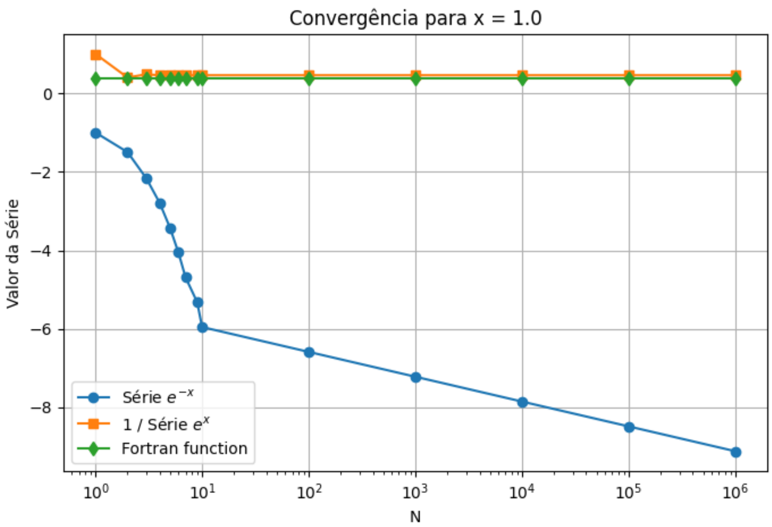


Figura 7: Gráfico do erro em função de N para x = 1.

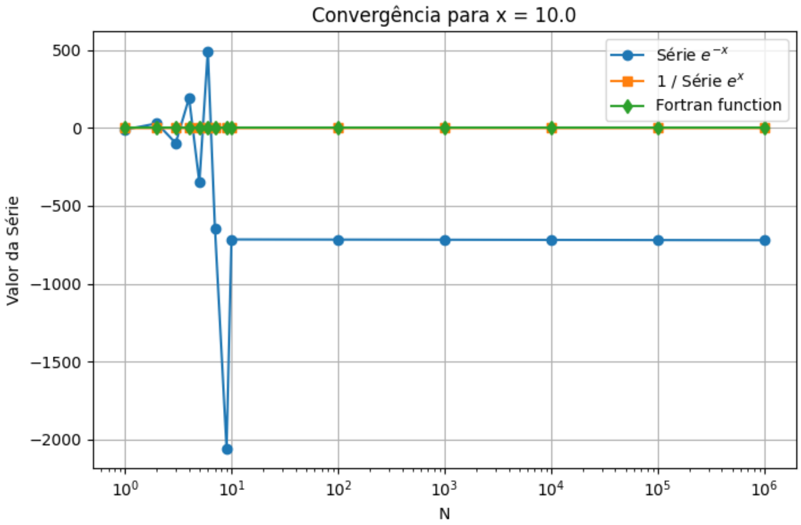


Figura 8: Gráfico do erro em função de N para x = 10.

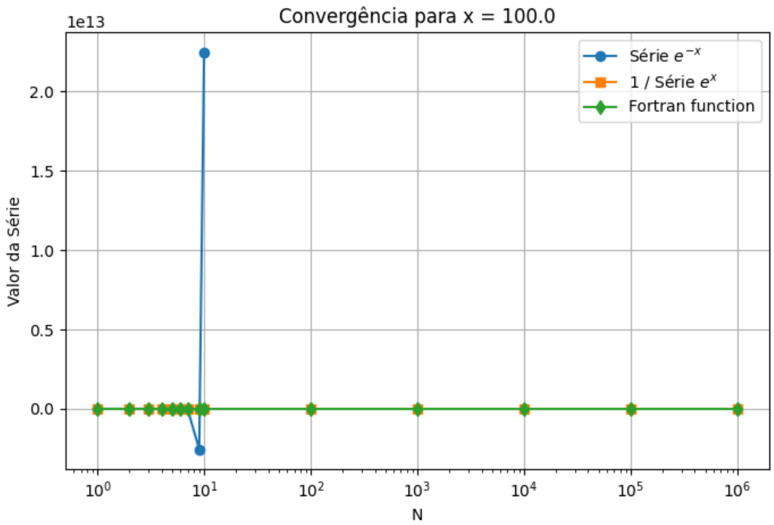


Figura 9: Gráfico do erro em função de N para x = 100.

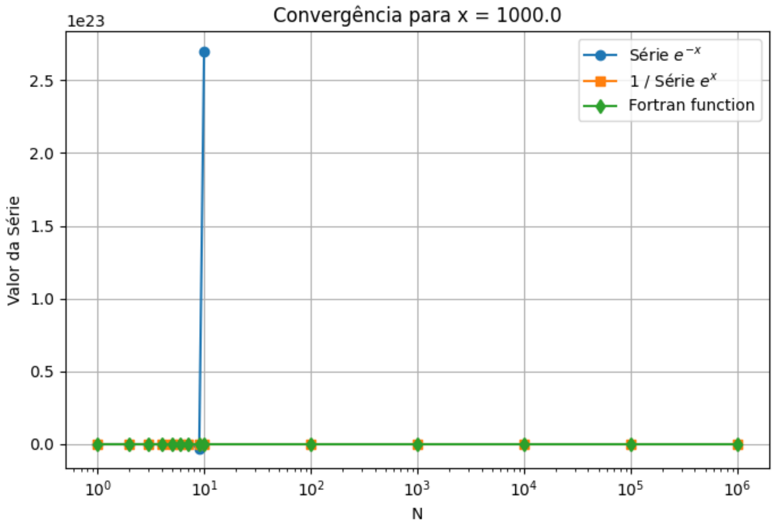


Figura 10: Gráfico do erro em função de N para x = 1000.

Percebe-se - claramente - que calcular a série de e^x e fazer $1/e^x$ o resultado é mais preciso do que calcular a série, diretamente, de e^{-x}. Além disso, é notório que ao calcular a série de e^{-x} diretamente para valores de N muito grande o resultado apresenta uma divergência muito grande do resultado real o que se deve ao fato de que os termos somados se tornam muito pequenos ao ponto de serem menores que o Machine Precision.