Análise Numérica

Alexandre Silva
João Temudo
Pedro Leite
Pedro Carvalho

March 2021

1 Introdução

Este trabalho foi realizado no âmbito de aprender as bases de aprender as bases de Análise Numérica, particularmente no meio digital. Temos de descobrir um epsilon máquina que iremos usar para o resto dos exercícios. Iremos também tratar de descobrir o valor de uma série com erro menor a um dado valor, comparar o valor que nós obtemos com o valor real da série e tirar conclusões disso.

2 Primeiro Exercício

Epsilon Máquina: a distância entre 1 e o primeiro número maior do que 1 representável no sistema de virgula flutuante utilizado pelo sistema.

No primeiro exercício, utilizamos um método simples para calcular o epsilon máquina do computador que utilizamos. Primeiro, criamos uma variável de valor 0,5, chamada **eps**. De seguida, fomos diminuindo o seu valor pela metade até a diferença entre 1+eps e 1 não ser reconhecida pela máquina. No final do programa, imprimos o dobro (ou seja, o último passo em que reconheceu a diferença entre 1+eps e 1) de eps.

```
eps = 0.5
while 1+eps>1:
    eps = eps/2

print(eps*2)
```

Este código imprimiu resultado 2.220446049250313e-16, ou seja, o nosso epsilon máquina encontra-se na casa decimal de 10 elevado a -16.

3 Segundo Exercício

3.1 Contexto

Para majorar o erro cometido no cálculo da soma de séries, pelo critério de D'Alembert, é necessário encontrar um valor ***L*** *que segue a seguinte fórmula, dada nos powerpoints:*

$$\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = L \tag{1}$$

No caso de o valor de L ser menor que 1, sabemos que a série será convergente e o erro será majorado por:

$$R_n \leq a_{n+1} \frac{1}{1-L} \tag{2}$$

Precisamos então de encontrar um valor de n tal que a fórmula acima seja menor que o erro desejado (por exemplo, 10 elevado a -15)

A fórmula que desejamos cálcular é dada por:

$$S = 2\sum_{k=0}^{\infty} \frac{2^k k!^2}{(2k+1)!} \tag{3}$$

3.2 Alínea A

Para encontrar o nosso L, precisamos primeiro de seguir a fórmula **(1)**. Então, o nosso L será dado por:

$$\frac{a_{n+1}}{a_n} = \frac{\frac{2^{n+2}(n+1)!^2}{(2x+3)!}}{\frac{2^{n+1}n!^2}{(2n+1)!}} = \frac{2^{n+2}(n+1)!^2(2n+1)!}{2^{n+1}n!^2(2n+3)!} = \tag{4}$$

$$= \frac{2(n+1)^2}{(2n+3)(2n+2)} = \frac{n+1}{2n+3} < \frac{n+1}{2n+2} = \frac{1}{2} \tag{5}$$

Daqui podemos concluir que o nosso L é 0,5. Então, para encontrarmos um valor com erro menor que E, temos de encontrar um **n** tal que:

$$a_{n+1} \frac{1}{1-\frac{1}{2}} \leq E \tag{6}$$

Onde **n** será o número de iterações do nosso somatório. Como era necessário fazer um programa que mostrasse o número de iterações de modo ao nosso erro ser menor que um valor dado (10 elevado aos inteiros entre -8 e -15, inclusive), escrevemos o código na página seguinte de modo a imprimir todos esses de uma só vez.

```
import math

def sum_s(error):
    sum=0
    x=0

    r = 2

    while(2r>error):
    #Como concluído previamente o valor de L é 0.5, temos assim
    que(1/1-L)an+1 >= Rn. Sendo assim se 2*an+1 > erro temos de
    retornar a soma de todos os an
        upper_small = 2(int(x)) * math.factorial(int(x))2
        #dividendo da fraçao de an
        lower_small = math.factorial(2int(x) + 1)
        #divisor da fraçao de an
        r = 2(upper/lower)
        #cálculo de an+1
        r_small = 2*(upper_small/lower_small)
        #cálculo de an
        sum = sum + r_small
        #soma de an
        x = x+1

    return('Valor de S com erro < '+ str(error) + ' : ' + str(sum)
    + ', com ' + str(x) + ' termos')

for x in range(8,16):
    print(sum_s(10**-x))
```

Este código deu-nos vários resultados, no entanto, quando chegamos a erros menores que 10^{-13}, 10^{-14} e 10^{-15} os erros não eram expressos no output do nosso programa. O valor dado com erro menor a 10^{-15}, segundo o nosso programa, é **3,1415926535897922**.

Um defeito do método utilizado é que apenas verifica com os erros dados pelo exercício, mas isso pode fácilmente ser corrigido com a adição de um sistema I/O básico.

3.3 Alínea B

Para a alínea B, podemos fazer uma simples adição ao código previamente demonstrado à linha *return*, alterando-a para ler assim:

```
    return('Valor de S com erro < '+ str(error) + ' : ' + str(sum)
    + ', com ' + str(x) + ' termos e erro absoluto = ' +
    str(math.pi - sum))
```

Isto irá, então, passar a mostrar a diferença entre a constante π e o nosso valor, dando então o erro exato. Com cada iteração do nosso código, o erro fica cada vez menor e, até pedir um erro menor que 10^{-15}, dá um erro absoluto dentro dos parâmetros dados. Quando chegamos a 10^{-15}, o erro máquina faz com que o nosso python mostre um erro maior do que realmente teriamos com o número de termos, que neste caso é 49. Abaixo se incui uma printscreen com o erro que seria obtido com 49 termos.

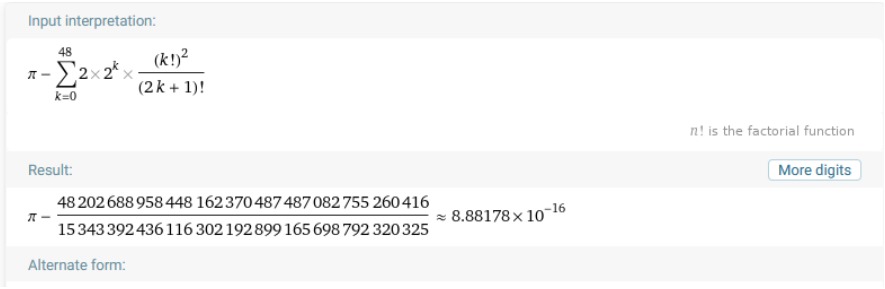

Figure 1: Com 48 termos, o erro é menor que 10^{-15}

4 Terceiro Exercício

Cálculos simples como os pedidos no exercício 3 não requerem muita explicação, visto que são simples substituições no caso do limite e um somatório normal no caso do somatório.

Criamos um programa que contém os métodos para calcular ambos, podendo alterar qual é ao mover o último # entre a última e penúltima linha com código.

```
import math

def an():
    for x in range(1,16):
        r = (1+ 1/(10**x))**(10**x)
        print('Para j=' + str(x) + " o valor aproximado é: " +
        str(r))

def bm():
    sum=0
    for x in range(0,21):
        r = 1/math.factorial(x)
        sum = sum + r
        print('Para m=' + str(x) + " a soma aproximado é: " +
        str(sum))

#an()
bm()
```

4.1 Limite para calcular e

Limite a cálcular:

$$e = \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \tag{7}$$

O código utilizado para cálcular o limite (iniciado por **def an():** e continuando até à linha **def bm():**, não inclusive) faz um ciclo que escreve, sequencialmente, os valores do limite de n→1 até n→15.

Os valores obtidos até n = 12 são os valores esperados e vão ficando mais próximos de e com cada iteração, confirmados com wolframalpha. A partir de n = 13, começa a ocorrer o erro de máquina e vão ficando mais longe de e e aproximando-se de 0. Disto podemos concluir que o computador não consegue processar contas feitas com tantas casas decimais devido a serem guardados em binário. Como a conta involve cálculos decimais com dízimas infinitas em binário (0.1 é uma, por exemplo), o computador arredonda mal, levando a estes erros máquina.

4.2 Somatório para calcular e

Somatório a cálcular:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} \tag{8}$$

O código utilizado para calcular o somatório (iniciado por **def bm():** e continuando até à linha **#an()**, não inclusive) faz o somatório até ao limite determinado, sendo estes limites os números inteiros entre m=1 até m=20.

Os valores obtidos com este cálculo aproximam-se de e com cada iteração, ou seja, o erro máquina ou não ocorre ou é tão pequeno que não afeta o resultado final até m=20, que é o limite até onde testamos.

5 Conclusão

Podemos concluir, tendo em conta o exercício 3, que uma máquina (neste caso uma máquina a utilizar *python*) terá mais facilidade a processar somatórios do que limites, particularmente quando são limites com frações de potências de 10.