Análise de Erros em Séries

FCUP
Análise Numérica (M2018) 2018/2019
Trabalho de Grupo 1
Ângelo Gomes – 201703990 – MIERSI
Eduardo Morgado – 201706894 – MIERSI
Simão Cardoso – 201604595 – MIERSI
Sónia Rocha – 201704679 – MIERSI

1. Considerações Iniciais:

Na realização deste trabalho, utilizamos como linguagem de implementação Python, como tal, tendo em conta que, pontos de virgula flutuante em Python são sempre de precisão dupla, a evolução dos dados não é tão notória. Uma das soluções para este problema seria, através de bibliotecas como *numpy*, forçar o programa a trabalhar apenas em precisão simples, no entanto, todos os cálculos seriam realizados em precisão dupla e só depois convertidos/arredondados para precisão simples, o que acabaria por transmitir erros de arredondamento para os resultados. Sendo assim, durante todo este trabalho, iremos sempre ter em atenção esse fator, fator esse que influencia no cálculo do *epsilon* máquina.

Uma nota importante para utilização dos próximos métodos em Python é a necessidade de configurar a divisão inteira como sendo uma divisão de pontos flutuantes, caso contrário, operações como 1/4=0 e não 0.25, para isso é necessário importar um parâmetro de uma biblioteca: `from __future__ import division`

Todos estes programas podem ser encontrados no Github no repositório da equipa, https://github.com/thejoblessducks/Trabalho1AN

2. Cálculo do *Epsilon* Máquina:

Antes de qualquer cálculo, é necessário perceber que o *epsilon* máquina é o menor número que, quando somado a 1, produza um resultado diferente de 1, ou seja, é o primeiro valor que, quando somado a 1, não leve a arredondamento, representando a exatidão relativa de um computador. Este valor surge provém da precisão finita de pontos flutuantes, uma vez que, em qualquer computador existem um número limitado de *bits* para representar um número, normalmente 32 *bits* ou 64 *bits*.

Para este trabalho, procuramos diferentes formas de calcular o *epsilon* (eps), as próximas implementações foram as mais simples de realizar:

- `eps= numpy.finfo(float).eps`
- `eps=numpy.spacing(1.0)`
- `eps=2**(-23)`

Tanto a primeira como a segunda implementação, utilizam funções pré-definidas pela biblioteca *numpy*, a primeira produz um cálculo em precisão dupla

($\mathbf{2.220446049250313x10^{-16}}$), enquanto que a segunda produz um cálculo em precisão simples pela distância de 1 até ao primeiro número diferente de 1 ($\mathbf{1.1920928955078125x10^{-7}}$). A terceira implementação, provem da limitação de representação para pontos de vírgula flutuante. Em qualquer computador com implementação em precisão simples, dispomos apenas de 23 *bits* para a mantissa de um qualquer número, como tal, o menor valor possível a representar é $\mathbf{2^{-23}}$. Para o resto deste trabalho e, tendo em conta, restrições de linguagem, iremos considerar como valor de *epsilon* $2.220446049250313x10^{-16}$.

3. Resolução de Série de Termos Positivos (Exercício 2) $S = \frac{9}{2\sqrt{3}} \sum_{k=0}^{\infty} \frac{k!^2}{(2k+1)!}$

Qualquer série pode ser decomposta em $S = S_n + R_n$ onde $S_n = \sum_{k=0}^{n} a_k$ *e* $R_n = \sum_{k=n+1}^{\infty} a_k$, pelo que, $|S - S_n| = |R_n|$, logo, aproximar a série será encontrar um valor n tal que, $|R_n| < \varepsilon$, $\varepsilon = 10^{-8}, \dots, 10^{-15}$, uma vez que, a série tratada é uma série de termos positivos, podemos aplicar o critério de D'Alembert, o que permite concluir que $\frac{a_{n+1}}{1-L} < \varepsilon$, onde $\lim_{n\to\infty} \frac{a_{n+1}}{a_n}$=L.

Para resolver este problema, primeiro é necessário verificar a convergência e determinar L:

$$\lim_{n\to\infty} \frac{a_{n+1}}{a_n} = \lim_{n\to\infty} \frac{\frac{9}{2\sqrt{3}} \frac{(n+1)!^2}{2(n+1)+1)!}}{\frac{9}{2\sqrt{3}} \frac{n!^2}{2n+1)!}} = \lim_{n\to\infty} \frac{(2n+1)!\cdot(n+1)!^2}{(2n+3)!\cdot n!^2} = \lim_{n\to\infty} \frac{\mathbf{n^2+2n+1}}{\mathbf{4n^2+10n+6}} = {}^1/_4$$

L=1/4<1 pelo que a série converge, logo, basta encontrar n tal que, $\frac{a_{n+1}}{1-1/4} < \varepsilon$, é também importante referir que, $\frac{\boldsymbol{a_{n+1}}}{\boldsymbol{a_n}} = \frac{\mathbf{n^2+2n+1}}{\mathbf{4n^2+10n+6}}$ uma vez que, este resultado irá facilitar o cálculo dos termos da série.

A *Figura 1* representa o pseudocódigo a implementar, e a *Figura 2* a sua respetiva implemenntação em Python (em inglês).

```
AproximacaoSerie:

    a->1;  n->0;  soma->a
    a->próximo(a)
    Para cada erro fazer
        Enquanto |a/(1-L)|>erro fazer
            n->n+1
            soma->soma+a
            a=próximo(a)
        escreve(n)
        escreve(soma)
```

Figura 1-Pseudocódigo

A *Figura 3* apresenta a tabela com o valores dos n's e das somas para os erros.

```
#ratioSeries2
def ratioSeries2(n):
    return ((n**2)+2*n+1)/((4*(n**2))+10*n+6)
#E2
def Ex2():#Applying the D'Alembert method
    L=1/4 #hence,|Rn|<=a(n+1)/1-L<eps
    bf=9/(2*mt.sqrt(3)) #value before sum
    a=1
    n=0
    sum=a
    a=ratioSeries2(n)*a
    for i in range(8,16):
        e = error(i) #returns 10^-i-- error = lambda x : 1.0*(10**-x)
        print "Erro "+str(e)+":"
        while (abs(a/(1-L))>e):
            n+=1
            sum+=a
            a=ratioSeries2(n)*a
        print "      |-n: "+str(n-1)
        print "      |-Sn: " +str(dm.Decimal(bf*sum))
    return
```

Figura 2- Implementação em Python 3.7

ε	n	S_n
10^{-8}	11	**3.1415926**41
10^{-9}	13	**3.1415926**529
10^{-10}	14	**3.14159265**342
10^{-11}	16	**3.14159265**3579
10^{-12}	17	**3.14159265358**73
10^{-13}	19	**3.14159265358**964
10^{-14}	21	**3.14159265358978**5
10^{-15}	22	**3.14159265358979**18

Figura 3- Resultados Exercício 2

Por análise dos resultados, podemos concluir que os valores aproximam o $\pi = 3.141592653589793$, para cada valor da tabela, a negrito estão os valores corretos do resultado, a vermelho os valores por propagação do erro, seguido pelo erro de aproximação excluído do erro permitido. Para cada erro, 10^{-m}, o resultado está representado com m-1 casas decimais corretas. Assumindo agora, por observação dos dados, que a série aproxima π podemos calcular o erro relativo desta série, a *Figura 4* apresenta esse erro, podemos então concluir que a série aproxima com exatidão o valor de π a partir do elemento n=16.

ε	$\left\lvert\frac{\Delta S_n}{S}\right\rvert$
10^{-8}	0.999999996
10^{-9}	0.9999999995
10^{-10}	0.9999999999
10^{-11}	1
10^{-12}	
10^{-13}	
10^{-14}	
10^{-15}	

Figura 4-Erro relativo de série

4. Resolução de Série Alternada (Exercício 3) $S = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$

Para séries alternadas, aproximar uma série é encontrar n tal que, $|R_n| < |a_{n+1}| < \varepsilon$ isto se a série for convergente. Uma série alternada $\sum a_n$ converge se $\sum |a_n|$ convergir, isto é, se $\forall k,\ |a_k| \geq |a_{n+1}|$ e se $\lim_{n\to\infty} a_n = 0$, tanto a primeira condição como a segunda são trivialmente verdadeiras , uma vez que, $2k+1 \to +\infty$.

A *Figura 5*, *6* e *7* representam o pseudocódigo a implementar, a sua implementação em Python e a sua tabela de resultados respetivamente.

```
AproximacaoSerie:

        a->1;   n->0;   soma->a
        a->próximo(a)
        Para cada erro fazer
                Enquanto |a|>erro fazer
                        n->n+1
                        soma->soma+a
                        a=próximo(a)
                escreve(n)
                escreve(soma)
```

Figura 5-Pseudocódigo Alternada

```
#Ex3
def Ex3():
    #|Rn|<|an+1|<e
    a=1
    n=0
    sum=a
    a=((-1)**(n+1))/(2*(n+1)+1)
    for i in range(8,16):
        e = error(i)
        print "Erro "+str(e)+":"
        while (abs(a)>e):
            n+=1
            sum+=a
            a=((-1)**(n+1))/(2*(n+1)+1)
        print "     |-n: "+str(n)
        print "     |-Sn: "+str(dm.Decimal(4*sum))
    return
```

Figura 6-Implementação em Python 3.7

Algo que notamos ao implementar este programa, é o facto da série convergir muito lentamente, o que leva a um cálculo muito lento dos resultados. Uma das razões poderá ser o facto da implementação não ser a mais eficiente, ou o facto dos termos que estamos a somar, serem de tal forma pequenos que, são menores que o *epsilon* máquina o que não conduz a um aumento "real" da soma. Como tal, a *Figura 7* apenas apresenta alguns resultados.

ε	n	S_n
10^{-8}	49999999	**3.1415926**33
10^{-9}	499999999	**3.1415926551**6
10^{-10}	Não conseguimos obter resultados por excesso de tempo	

Figura 7-Tabela de Resultados

Por uma primeira observação dos dados, verificamos que esta implementação converge muito lentamente, quando comparada à anterior, o que conduz a um elevado valor de n, para um valor aproximado menos correto. Quando comparado à tabela do exercício 2, verificamos que a casa de propagação do erro (vermelho) no 2, encontra-se muito mais próxima da sua versão correta para o próximo erro, em comparação com esta implementação. Logo, esta série não é uma boa série para aproximar a S, quando comparada à série anterior.

5. Aplicação para valores Exatos (Exercício 4)

Para esta secção, já sabemos *a priori* o valor exato da soma S=π, ou seja, a condição de dos ciclos "Enquanto" para cada série será $|\pi - S_n| > \varepsilon$.

5.1. Implementação de Exercício 2 para Valor de Série Conhecido

A *Figura 8* apresenta o código para o problema e a *Figura 9* a tabela dos resultados. Para este problema, consideramos $\pi = 3.141592653589793$.

```
#ratioSeries2
def ratioSeries2(n):
    return ((n**2)+2*n+1)/((4*(n**2))+10*n+6)

#Ex2 in Ex4
def Ex2n4():
    exact=mt.pi
    bf=9/(2*mt.sqrt(3))
    a=1
    sum=a
    Sn=bf*sum
    n=0
    for i in range(8,16):
        e = error(i) #returns 10^-i-- error = lambda x : 1.0*(10**-x)
        print "Erro "+str(e)+":"
        while (abs(exact-Sn)>e):
            a=ratioSeries2(n)*a
            n+=1
            sum+=a
            Sn=bf*sum
        print "     |-n: "+str(n)
        print "     |-Sn: "+str(dm.Decimal(bf*sum))
        print "     |-|S-Sn|: "+str(dm.Decimal(abs(exact-Sn)))
    return
```

Figura 8-Implementação em Python 3.7

ε	n	S_n	$\lvert\pi - S_n\rvert$	$\left\lvert\frac{\Delta S_n}{S}\right\rvert$
10^{-8}	13	**3.14159265**06	**2.9465234**69x10^{-9}	0.9999999989
10^{-9}	14	**3.1415926528**8	**7.133125201**6x10^{-10}	0.9999999997
10^{-10}	16	**3.14159265354**8	**4.203970505**05x10^{-11}	1
10^{-11}	18	**3.1415926535873**	**2.49267273488**8x10^{-12}	
10^{-12}	19	**3.14159265358919**	**6.0795812828474** x10^{-13}	
10^{-13}	21	**3.141592653589757**	**3.59712259978551** x10^{-14}	
10^{-14}	22	**3.1415926535897851**	**7.993605777301127** x10^{-15}	
10^{-15}	24	**3.14159265358979356**	**4.4408920985006262** x10^{-16}	

Figura 9-Tabela de Resultados

Por observação dos dados, esta nova implementação, necessita de mais passos para chegar a uma aproximação inferior ao erro estipulado, no entanto, é mais fidedigna (apresenta mais casas decimais corretas) que a primeira implementação do exercício 2. Através do cálculo do erro relativo, uma vez que sabemos que a série aproxima π, podemos concluir que para o valor que consideramos ser exato, a nossa aproximação, atinge um rigor máximo para erros inferiores a $5x10^{-11}$, daí podemos assumir que o valor de π calculado pela biblioteca *numpy* de Python, é um valor com erro 10^{-10}, ou seja, a partir do elemento n=16, a nossa série aproxima com exatidão o valor de π representado pelo computador.

5.2. Implementação de Exercício 3 para Valor de Série Conhecido

Para este exercício voltamos a deparar-nos com o mesmo problema, o programa não termina, não chegando à solução para o erro 10^{-8}. A *Figura 10* e *Figura 11* apresentam os código e a tabela respetivamente.

```
#Ex 3 in 4
def Ex3n4():
    exact=mt.pi
    a=1
    n=0
    sum=a
    for i in range(8,16):
        e = error(i)
        print "Erro "+str(e)+":"
        while(abs(exact-4*sum)>e):
            n+=1
            a=((-1)**n)/(2*n+1)
            sum+=a
        print "      |-n: "+str(n)
        print "      |-Sn: "+str(dm.Decimal(4*sum))
        print "      |-|S-Sn|: "+str(dm.Decimal(abs(exact-4*sum)))
    return
```

Figura 10-Implementação em Python 3.7

ε	n	S_n	$\lvert\pi - S_n\rvert$
10^{-8}	Não conseguimos obter resultados por excesso de tempo		

Figura 10-Tabela