TP3_correction

December 4, 2017

Correction du TP 3
Activité 1
Q1

```
In [1]: a=0.1
        print("%1.30f" % a)
```

```
0.100000000000000005551115123126
```

```
In [2]: a=0.5
        print("%1.30f" % a)
```

```
0.500000000000000000000000000000
```

```
In [3]: a=0.750
        print("%1.30f" % a)
```

```
0.750000000000000000000000000000
```

```
In [4]: a=10.87
        print("%1.30f" % a)
```

```
10.869999999999999218402990663890
```

```
In [5]: a=10.875
        print("%1.30f" % a)
```

```
10.875000000000000000000000000000
```

Q2
Nous remarquons que $0,5$, $0,875$, $0,750$ sont exactes. Cela s'explique car :
$0,5 = (0.10000000...)_2$ puisque $0.5 = \frac{1}{2}$ $0,875 = (0.1001000...)_2$ puisque $0,875 = \frac{7}{8} = \frac{2^2+2+1}{2^4} = \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4}$ $10,875 = (101.1001000...)_2$ puisque $10,875 = \frac{87}{8} = \frac{2^6+2^4+2^2+2+1}{2^4} = 2^2+1+\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}$.

Leur représentation machine avec la norme IEEE7574 est donc exacte même si l'on tronque la mantisse au 52 ème bit !

Activité 2

On vérifie la non associativité de la somme. Le phénomène mis en évidence est le phénomène d'absorption. Par exemple, si l'on saisit :

```
In [6]: 1+2**(-53)

Out[6]: 1.0
```

On voit que le $2^(-54)$ se fait absorber dans le calcul. Ceci peut s'expliquer car 2^{-54} est exacte en IEEE754 (exposant correspondant à -54 et mantisse égale à 1 !), alors que $1 + 2^(-53) = (1.000000.....01000000)_2$ avec le 1 en virgule à la 53ème position donc qui va disparaître en représentation machine (double précision) puisque la mantisse ne prendra en compte que les 52 premiers bits !

En revanche :

```
In [7]: 1+2**(-52)

Out[7]: 1.0000000000000002
```

prend bien en compte le 2^{-52} !

Plus généralement, le phénomène d'absorption apparaît lorsqu'on somme deux nombres d'odres de grandeurs très différents (comme 1 et 2^{-53}).

Activité 3 :

Q2

```
In [10]: def harm(n):
             S=1
             for k in range(2,n+1):
                 S=S+1/k
             return S


In [12]: harm(2)

Out[12]: 1.5
```

Q3

```
In [13]: def harm2(n):
             S=1/n
             for k in range(1,n):
                 S=S+1/(n-k)
             return S

In [14]: harm2(2)

Out[14]: 1.5
```

Q4

```
In [15]: harm(10**(5)),harm2(10**5)

Out[15]: (12.090146129863335, 12.090146129863408)

In [20]: harm(10**(7)),harm2(10**7)

Out[20]: (16.695311365857272, 16.695311365859965)

In [21]: harm(10**(8)),harm2(10**8)

Out[21]: (18.997896413852555, 18.997896413853447)
```

En regardant de près, on est en mesure de constater que la deuxième mesure est plus précise. En effet, dans la première mesure comme on sait que cette somme tend vers l'infini (résultat mathématique), pour le calcul de la S de l'étape n à n+1, on fait S=S+1/(n+1). Or si S est déjà très grand, 1/(n+1) est très petit et on somme donc deux nombres d'ordres de grandeurs différents, ce qui fait apparaître des erreurs dues au phénomène d'absorption.

Activité 5

```
In [39]: for k in range(2,20):
             a,b=2**(-5),2**(-5)+2**(-5+k)
             print((b-a)/a)

4.0
8.0
16.0
32.0
64.0
128.0
256.0
512.0
1024.0
2048.0
4096.0
8192.0
16384.0
32768.0
65536.0
131072.0
262144.0
524288.0
```

Ici, les nombres a et b sont de plus en plus proches et leurs erreurs erreurs relatives est de plus en plus grande. Il s'agit du phénomène d'élimination qui intervient lorsqu'on fait la différence de deux nombres du même ordre de grandeur.

Acitivité 6

Q2

```
In [52]: from math import factorial
         import numpy as np

         def expo(x,n):
             S=1
             for k in range(1,n+1):
                 S=S+(x)**(k)/factorial(k)
             return S
```

Q3

```
In [53]: for k in range(10,60,10):
             a=expo(-10,k)
             print(a)
```

```
1342.5873015873017
13.396865995695713
0.0009703415796020374
4.540234176855547e-05
4.5399929434148165e-05
```

```
In [54]: from math import exp
         b=exp(-10)
         print(b)
```

```
4.5399929762484854e-05
```

```
In [55]: (b-a)/a
```

```
Out[55]: 7.232096905333341e-09
```

Q4

```
In [56]: for k in range(10,60,10):
             a2=1/expo(10,k)
             print(a2)
```

```
7.78676406667942e-05
4.547215139175042e-05
4.539993338712231e-05
4.5399929762492925e-05
4.539992976248486e-05
```

```
In [57]: print((a2-a)/a)
```

```
7.23209705459048e-09
```

Q6

On constate une meilleure précision pour la deuxième méthode. En effet dans la première méthode, on somme des termes de même ordre de grandeur (des nombres très petits) avec une alternance de signe $(-1)^k$. On se retrouve donc en présence de différence de termes de même ordre de grandeur, d'où un phénomène d'élimination qui a tendance à augmenter l'erreur.

Activité 7

Q3

```
In [58]: from math import sqrt

         def approx1(k):
             s=sqrt(2)/2
             for l in range(3,k+1):
                 s=sqrt((1-sqrt(1-s*s))/2)
             return 2**(k)*s

In [59]: approx1(20)

Out[59]: 3.1415965537048196

In [60]: from math import pi
         pi

Out[60]: 3.141592653589793

In [73]: approx1(25),approx1(26),approx1(27),approx1(28),approx1(29),approx1(30)

Out[73]: (3.142451272494134,
          3.1622776601683795,
          3.1622776601683795,
          3.4641016151377544,
          4.0,
          0.0)
```

On constate que l'on s'éloigne de plus en plus de la valeur exacte jusqu'à même obtenir 0 pour $N = 30$!

Q5

Si l'on regarde de plus près l'algorithme, la valeur de s correspondant à l'angle est de plus en plus petite. Ainsi, 1 et $\sqrt{1-s^2}$ sont de plus en plus proche. On est en présence de la différence de deux nombres du même ordre de grandeur, d'où des erreurs qui sont de plus en plus grandes.

Q6

Il nous faut éviter cette différence, pour ceci, on remarque que :

$1-\sqrt{1-s^2} = \frac{s^2}{1+\sqrt{1-s^2}}$ (multiplication par l'expression conjuguée au numérateur et au dénominateur).

D'où le code ci-dessous :

```
In [74]: def approx2(k):
             s=sqrt(2)/2
             for l in range(3,k+1):
                 s=sqrt(s*s/(2*(1+sqrt(1-s*s))))
             return 2**(k)*s
```

```
In [77]: approx2(25),approx2(26),approx2(27),approx2(28),approx2(29),approx2(30)

Out[77]: (3.14159265358979,
          3.1415926535897936,
          3.1415926535897944,
          3.141592653589795,
          3.141592653589795,
          3.141592653589795)

In [78]: pi

Out[78]: 3.141592653589793
```

Ce qui donne une approximation nettement plus satisfaisante.
Activité 4

```
In [81]: def kharm(n):
             s=0
             c=0
             for k in range(1,n+1):
                 y=1/k-c
                 t=s+y
                 c=(t-s)-y
                 s=t

             return s

In [84]: kharm(10**(8))

Out[84]: 18.997896413853898

In [86]: harm2(10**(8))

Out[86]: 18.997896413853447
```

On constate donc une précision supplémentaire pour Kahan en comparant avec la valeur de référence de l'activité 3, ce qui s'analyse par les explications fournies en commentaires à côté du pseudo-code donné dans l'activité 4.

```
In [ ]:
```