ISM MPSI/PCSI, Informatique
Année 2024/2025

Corrigé du TP Informatique 16

Exercice 1

1. On saisit :

```
>>> 2.**1024
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    2.**1024
OverflowError: (34, 'Result too large')
>>> 2**1024
179769313486231590772930519078902473361797697894230657273430081157732675
...
```

2. La première saisie est faite avec virgule et l'interpréteur traite donc le nombre au format flottant. L'exposant qui suit est hors du champ de validité du format flottant d'où l'erreur `Overflow`. La deuxième saisie est faite sans virgule donc l'interpréteur traite le nombre au format `int`, format sans limitation pour les entiers relatifs d'où le résultat.

Exercice 2

1. On saisit :

```
>>> 1+2**(-10)-1==2**(-10)
True
>>> 1+2**(-100)-1==2**(-100)
False
```

2. Cette expérimentation illustre le phénomène d'absorption : bien que python sache traiter 2^{-100} seul, la limite de précision est dépassée lors du calcul de $1 + 2^{-100}$ qui est donc traité comme 1 d'où le résultat observé.

3. On complète :

```
def seuil():
    n=1
    while 1+2**(-n)-1==2**(-n):
        n+=1
    return n-1
```

On obtient :

```
>>> seuil()
52
```

On trouve la taille de la mantisse ce qui était prévisible : c'est elle qui détermine la précision du codage d'un flottant.

4. On saisit :

```
def seuil1(deb,fin):
    c=(deb+fin)//2
    while deb<=fin:
        if 1+2**(-c)-1==2**(-c):
            deb=c+1
        else:
           fin=c-1
        c=(fin+deb)//2
    return c
```

Exercice 3

1. On saisit :

```
>>> .1+.2
0.30000000000000004
>>> .3+.6
0.8999999999999999
>>> .8+.9
1.7000000000000002
```

Ces calculs très simples montrent les limites de codage du format flottant.

2. On obtient :

```
0.10000000000000000555
0.20000000000000001110
...
0.50000000000000000000
0.59999999999999997780
...
```

Les nombres dont la saisie est fidèle au codage sont ceux dont la partie fractionnaire est constituée de puissances négatives de $2 : 0$ et $0.5 = \frac{1}{2}$. Pour tous les autres, le codage n'est pas conforme à la saisie de l'utilisateur. Il s'agit d'un défaut structurel du format `float`, défaut présent dans tous les langages de programmation utilisant ce format.

3. On obtient :

```
0.01000000000000000021
0.02000000000000000042
...
0.23999999999999999112
0.25000000000000000000
0.26000000000000000888
...
```

On constate le même phénomène avec $0.75 = \frac{1}{2} + \frac{1}{4}$ et $0.25 = \frac{1}{4}$ et un codage non conforme pour les autres.

Exercice 4

1. Le calcul donne $\quad P = X^2 - 0.4X + 0.04 \quad$ et $\quad Q = X^2 - X + 0.25$

2. On saisit :

```
def discriminant(a,b,c):
    return b**2-4*a*c==0
```

puis

```
>>> discriminant(1,-.4,.04)
False
>>> discriminant(1,-1,.25)
True
```

3. Dans les deux cas, les polynômes sont à racines doubles mais le test utilisé n'en détecte qu'un sur les deux, celui dont les coefficients sont constitués exactement de puissances (négatives) de deux. Pour un tel test, on s'expose immanquablement à des erreurs de codage.

4. Il faut privilégier un test de la forme `abs(delta)<eps` avec un seuil `eps` choisi par l'utilisateur.

Exercice 5

2. Théoriquement, on devrait avoir $R(h) = 0$ pour tout $h \neq 0$. Ce n'est pourtant pas ce qu'on observe avec `taux_acc(10,2**55,100)` puis en concentrant l'étude avec `taux_acc(2**50,2**53,100)` :

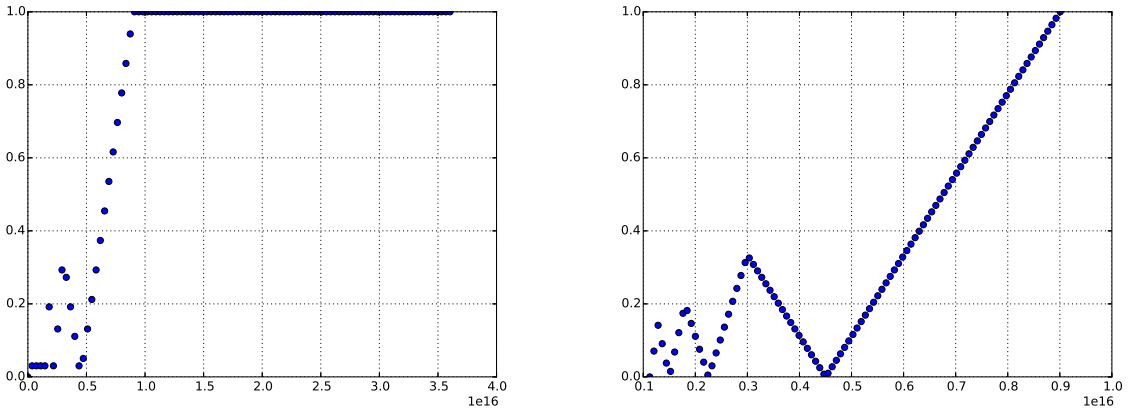

FIGURE 1 – Tracé de l'erreur relative $R(h)$ en fonction de $\frac{1}{h}$

3. On constate qu'à partir de $\frac{1}{h} \gtrsim 0.6 \times 10^{16}$, on a $R(h) = 1$ soit une erreur de 100%. On remarque que $0.6 \times 10^{16} \simeq 2^{52}$ autrement dit l'erreur devient totale au delà du seuil d'absorption. Mais avant d'atteindre ce seuil, on observe une zone d'instabilité numérique avec des résultats très médiocres : c'est le phénomène de *cancellation catastrophique* (ou élimination catastrophique) où le calcul amplifie l'erreur numérique jusqu'à un niveau pathologique.

Exercice 6

1. On saisit :

```
def mant(x,n):
    a=bin(int(2**n*x))
    return a[3:]
```

2. On saisit :

```
>>> liste=[1,1.5,1.25,1.125,1.1,1.9]
>>> for x in liste:
        print(mant(x,50))
00000000000000000000000000000000000000000000000000
10000000000000000000000000000000000000000000000000
01000000000000000000000000000000000000000000000000
00100000000000000000000000000000000000000000000000
00011001100110011001100110011001100110011001100110
11100110011001100110011001100110011001100110011001
```

3. Pour les nombres de la forme $1 + \frac{k}{2^{52}}$ avec $k \in [\![0 \,;\, 2^{52} - 1]\!]$, l'écriture binaire fractionnaire semble finie (elle l'est mais on ne peut pas vérifier expérimentalement qu'il y a une infinité de zéros). Pour les autres, l'écriture binaire fractionnaire semble au contraire infini.

4. On saisit :

```
>>> for x in liste:
        print(mant(x,100))
0000000000000000000000000000000000000000000000000000000000000000000000000000...
1000000000000000000000000000000000000000000000000000000000000000000000000000...
0100000000000000000000000000000000000000000000000000000000000000000000000000...
0010000000000000000000000000000000000000000000000000000000000000000000000000...
0001100110011001100110011001100110011001100110011001000000000000000000000000...
1110011001100110011001100110011001100110011001100110000000000000000000000000...
```

On gagne quelques chiffres significatifs mais à peine. En fait, la taille de la mantisse est de 52 bits donc au delà de 52 chiffres dans l'écriture binaire fractionnaire, c'est peine perdue. On ne peut pas espérer plus de précision que le format codé en machine.