TD n°02 : Limites du calcul informatique

1 heure

Rédigé par Pascal Delahaye

11 octobre 2015

Le fait que les nombres soient codés sur un nombre fini de bits est source d'erreurs potentielles dont il faut connaître l'existence et les conséquences possibles.

1 Dépassement de capacité

1.1 Dans le calcul avec des entiers

En CAML, les entiers sont codés sur 31 bits dont un bit pour le signe.

1. le plus grand entier qui peut être codé est donc $max1 = 2^{30} - 1 = 1073741823$
2. le plus petit entier qui peut être codé est donc $min1 = -2^{30} = -1073741824$.

En PYTHON, les entiers sont codés sur 32 bits dont un bit pour le signe.

1. le plus grand entier qui peut être codé est donc $max2 = 2^{31} - 1 = 2147483647$
2. le plus petit entier qui peut être codé est donc $min2 = -2^{31} = -2147483648$.

Exercice : 1

Taper les instructions suivantes et commentez les résultats obtenus :

Sous PYTHON :

```
>>> max1 = 2147483647
>>> max1 + 1
>>> min1 = -2147483648
>>> min1 - 1
```

Sous CAML :

```
let max2 = 1073741823;;
max2 + 1;;
let min2 = -1073741824;;
min2 - 1;;
```

Remarque 1. Python est muni d'une *sécurité* lui permettant de manipuler des *entiers aussi grands que l'on veut* sans erreur (dans la limite de la mémoire allouée à l'exécution du programme par l'ordinateur) !

Attention, comme c'est le cas avec CAML, ce n'est pas le cas de tous les langages de programmation.
Un dépassement de capacité peut donc entraîner des aberrations !

1.2 Dans le calcul avec des nombres flottants

Nous avons vu que les nombres flottants étaient codés sur un nombre de bits limité (64 en Python).
Comme pour les entiers, il existe une limite qu'il vaut mieux ne pas dépasser !

`Exemple 1.` Sous Python, la commande `10.**100.**100.` génère un dépassement de capacité.

Exercice : 2

1. Sous Python, déterminer pour quelle puissance de 10, on obtient un dépassement de capacité.
   Attention : pour travailler avec des nombres flottants, il vous faudra taper `10.**n` et non `10**n`.
2. Vérifier si cette limite est la même sous CAML.
3. Que pouvez-vous en déduire ?

Remarque 2. Les dépassements de capacité ne sont pas à prendre à la légère. C'est en effet un dépassement de capacité qui est à l'origine d'un dysfonctionnement dans le module de pilotage lors du premier vol d'Ariane 5, obligeant les ingénieurs à détruire leur fusée en vol.

2 Erreurs dans les calculs avec les nombres flottants

2.1 Origines des erreurs d'approximation

`Exemple 2.` Taper sous spyder le calcul suivant : $12 \times (\frac{1}{3} - \frac{1}{4}) - 1$. Que constatez-vous ?

a) Nombres réels simples non flottants :

Certains nombres simples ne sont pas reconnus à leur juste valeur par l'ordinateur.

`Exemple 3.` Soit $x = 0.35$.
Une vérification simple montre que le codage en binaire de x est un nombre comportant une infinité de décimales : un ordinateur codant les nombres sur un nombre de bits fini en utilisant leur représentation binaire, le nombre 0.35 ne peut être codé de façon exacte.
On peut facilement vérifier cela en tapant l'instruction suivante qui donne la valeur de 0.35 à 25 chiffres après la virgule :

```
format(0.35, '.25f')
```

Vous constaterez que la réponse donnée par Python est : 0.3499999999999999777955395

`Exemple 4.` Demandez la valeur de 0.25 à 25 chiffres après la virgule avec l'instruction `format(0.25, '.25f')`.
Python donne pour réponse : 0.2500000000000000000000000
Rien d'étonnant à cela puisque $0.25 = \overline{0.01}^{(2)}$.

b) Nombres réels avec plus de 16 chiffres significatifs :

Python fonctionne en double précision, c'est à dire que les nombres flottants sont codés sur 64 bits. La mantisse étant quant à elle codée sur 53 bits, la manipulation des nombres flottants entraîne systématiquement une erreur de $2^{-53} \sim 10^{-16}$ sur la mantisse, c'est à dire sur le 17ième chiffre significatif du nombre.

Ainsi : dès qu'un nombre flottant admet plus de 16 chiffres significatifs, celui est automatiquement approximé par l'ordinateur au niveau du 17ième chiffre.

`Exemple 5.` Sous python, tapez la commande `x = 123456789012345678.`, puis la commande `format(x, '.25f')`. Vous constaterez que la valeur est arrondie au niveau du 17ème chiffre significatif.

2.2 Perte d'associativité

Si en mathématiques, la multiplication et l'addition sont deux lois associatives dans $\mathbb{R}$, observons cependant ce qui se produit lors de calculs sur ordinateur...

Exercice : 3

1. Sous PYTHON, calculer les deux expressions : `(3.11*2.3)*1.5` et `3.11*(2.3*1.5)`.
   Que constatez-vous ?

2. Sous PYTHON, calculer les deux expressions : `1.11+(3.33+5.50)` et `(1.11+3.33)+5.50`.
   Que constatez-vous ?

Remarque 3. Les erreurs d'arrondi sont à l'origine de la perte d'associativité de la multiplication et de l'addition.

2.3 Propagation des erreurs

Nous avons vu que, en Python, les erreurs d'arrondis portent sur le 17ème chiffre significatif des nombres flottants. Cependant, lors de calculs itératifs, ces erreurs peuvent se cumuler et devenir beaucoup plus significatives.

`Exemple 6.` Lorsqu'on tape la commande `exp(log(1000)-log(10))` sous Python, le résultat obtenu est :

`99.999999999999957!!`

Cette fois, l'erreur ne porte plus sur le 17ème chiffre significatif, mais sur le 16ème.
On peut donc penser que la succession de calculs peut entraîner une augmentation des erreurs....

Exercice : 4

Comparez les résultats obtenus par différents programmes obtenu en modifiant l'expression `f(X)`.
Vous pourrez prendre tour à tour les expressions théoriquement équivalentes suivantes :

1. `f(X) = (R+1)*X-R*(X*X)`
2. `f(X) = R*X*(1-X)+X`
3. `f(X) = X*(R+1)-(R*X)*X`

```
R=3.
X=0.5
for i in range(1,500): X=f(X)
print(X)
```

Il ne faudra donc pas s'étonner si des programmes théoriquement justes donnent des résultats très éloignés de la réalité !

Remarque 4. Les erreurs d'arrondi ne sont pas non plus à prendre à la légère. C'est en effet une accumulation d'erreurs d'arrondi dans l'horloge d'un sytème de missiles Patriot qui est à l'origine d'un dysfonctionnement qui a causé 28 morts dns le camp US lors de la première guerre du Golfe.

2.4 Conséquence sur la terminaison d'une boucle

Sous Python, que pensez-vous du programme suivant :

```
A = 1.
while ((A + 1.) - A) - 1. == 0:
       A = 2*A
print(A)
```

Lancez-le ! Comment interpréter le résultat obtenu ?

Remarque 5. De façon générale, dans les boucles "while" ou dans les structures de contrôle "if...then...else", il ne faudra jamais effectuer un test d'égalité entre deux flottants.