Fiche n°05 : Les Limites du calcul informatique

Rédigée par Pascal Delahaye

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.

I] Dépassement de capacité

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) !
- CAML comme beaucoup d'autres langages de programmation, ne possède pas cette sécurité et renvoie des résultats aberrants en cas de dépassement de capacité.

2) Dans le calcul avec des nombres flottants

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

`Exemple 1.`

1. Montrer que le plus grand flottant codé en 64 bits est $2^{1024} - 1$.
2. Vérifier le comportement de l'ordinateur de Python et lorsqu'on tape cette valeur.
   Pour que ce nombre soit bien considéré comme flottant, vous taperez $2.^{1024} - 1$.
3. Calculer cette valeur sous la forme d'un entier, puis convertissez-la en flottant pour connaître son ordre de grandeur.

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.*

II] Erreurs dans les calculs avec les nombres flottants

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 puisque contrairement à 0.35, ce nombre a un codage fini et simple en binaire : $0.25 = \overline{0.01}^{(2)}$.

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 52 bits, la manipulation des nombres flottants entraîne systématiquement une erreur relative de $2^{-52} \approx 10^{-16}$ ce qui induit une erreur systématique sur le 17ième chiffre significatif du nombre. Ainsi, dès qu'un nombre admet plus de 16 chiffres significatifs, celui est automatiquement approximé par l'ordinateur au niveau du 17ième chiffre. Ces erreurs seront appelées des *erreurs d'arrondi*.

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

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 : 2

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. Cela signifie que selon d'ordre dans lequel on effectue les calculs, il faut s'attendre à obtenir des résultats différents.

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.` Sous Python, lorsqu'on tape la commande `(1234567891*0.1-123456789.1)*100000000`, le résultat obtenu est :

`1.4901161193847656!!` au lieu de `0`

Observons la propagation des erreurs sur l'exemple suivant :

Exercice : 3

On souhaite calculer le terme u_n de la suite (u_n) définie par : $\begin{cases} u_0 = 1/2 \\ u_{n+1} = -3u_n^2 + 4u_n = f(u_n) \end{cases}$.

Pour cela, nous allons utiliser plusieurs expressions mathématiquement égales de l'expression `f(x)`.

1. `f(x) = 4*x-3*(x*x)` 2. `g(x) = 3*x*(1-x)+x` 3. `h(x) = x*4-(3*x)*x`

Le programme python donnant la valeur de u_n est le suivant :

Python

```
f = lambda x : 4*x-3*(x*x)
g = lambda x : 3*x*(1-x)+x
h = lambda x : x*4-(3*x)*x

def u(n,f):
    X=0.5
    for i in range(1,n+1):
        X=f(X)
    return X
```

Remplissez le tableau suivant et commentez les résultats obtenus :

	u_{10}	u_{100}	u_{500}
Avec f			
Avec g			
Avec h			

Il ne faudra donc pas s'étonner si parfois, des programmes théoriquement justes donnent des résultats aberrants !

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.*

3) Conséquence sur la terminaison d'une boucle

Sous Python, que pensez-vous du programme suivant :

Python
```
def boucleinf():
    A = 1.
    N = 0
    while ((A + 1.) - A) - 1. == 0:
          A = 2*A
          N = N + 1
    print(N,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.

1 Ce qu'il faut savoir et/ou savoir faire :

1. Sur les entiers :

   (a) Savoir coder / décoder un entier naturel en binaire
   (b) Savoir programmer en python le codage et le décodage d'un entier naturel en binaire
   (c) Savoir calculer le nombre de bits nécessaires pour coder un entier en binaire
   (d) Savoir déterminer le codage machine sur n bits d'un entier naturel
   (e) Savoir déterminer le codage machine sur n bits d'un entier strictement négatif
   (f) Savoir retrouver la valeur d'un entier dont on nous donne le codage binaire en machine
   (g) Connaître les plus petits et plus grands entiers relatifs que l'on peut coder avec n bits.

2. Sur les nombres flottants :

   (a) Savoir justifier qu'un nombre flottant est un nombre décimal
   (b) Savoir coder / décoder un nombre décimal positif en binaire avec p chiffres après la virgule
   (c) Savoir programmer en python le codage et le décodage en binaire d'un réel $x \in]0,\ 1[$
   (d) Savoir déterminer le codage machine d'un nombre flottant sur 32 bits
   (e) Savoir déterminer la valeur du nombre décimal dont on nous donne le codage machine sur 32 bits
   (f) Savoir expliquer pourquoi l'ordinateur ne reconnaît pas un nombre aussi simple que 0.1
   (g) Savoir expliquer pourquoi les nombres flottants codés sur 64 bits ne comportent que 16 chiffres significatifs
   (h) Savoir retrouver les valeurs du plus petit et du plus grand flottant normalisé strictement positif que l'on peut coder avec n bits
   (i) Savoir expliquer pourquoi la plupart des nombres réels sont codés de façon approximative par l'ordinateur.
   (j) Avoir conscience des conséquences possibles des erreurs d'approximation dues au codage machine des nombres.

3. Entraînement :

Faire un programme Python de décodage d'un nombre flottant donné par son codage machine sur 32 bits.
On pourra considérer que le codage est donné sous la forme d'une liste L de longueur 32.
On pourra également commencer par concevoir 2 sous-programmes, l'un calculant l'exposant p et l'autre calculant la mantisse.