ERREURS D'ARRONDI (ROUND-OFF ERROR)

GUIRAUD Jean - LABORDE Corentin

UNIVERSITÉ TOULOUSE III PAUL SABATIER

iut Occitanie TOULOUSE AUCH-CASTRES UNIVERSITÉ TOULOUSE III

Sommaire

I. Introduction

Les erreurs d'arrondi ont été fréquentes depuis que l'informatique existe. En effet, il est complexe de coder sans qu'une erreur ne soit introduite dans le programme. Néanmoins si ces erreurs sont généralement bénignes et facilement détectables et par conséquent résolvables, elles peuvent parfois être bien plus problématiques.

A. Définition

Pour commencer, **qu'est ce qu'une erreur d'arrondi ?** Une erreur d'arrondi est la différence entre la valeur approchée calculée d'un nombre et sa valeur mathématique exacte. Des erreurs d'arrondi peuvent survenir lorsque des nombres exacts sont représentés dans un système incapable de les exprimer exactement. Les erreurs d'arrondi se propagent au cours des calculs avec des valeurs approchées ce qui peut augmenter l'erreur du résultat final. Dans le système décimal des erreurs d'arrondi surviennent, lorsqu'avec une troncature, un grand nombre (peut-être une infinité) de décimales ne sont pas prises en considération. Ce processus d'arrondi apporte des gains de temps de calcul au détriment de la précision (Définition Wikipédia).

Après une brève définition, on est amené naturellement à se poser ces questions :

Comment une erreur d'arrondi se produit-elle ? Dans quel cas concret nous trouvons des erreurs d'arrondi ?

B. Cas concret

Nous retrouvons l'un des cas les plus connus lorsqu'en 1991 pendant la guerre du Golfe, 28 soldats américains furent tués à cause d'un problème de ce type. En effet, le système antimissile américain n'est pas parvenu à intercepter un missile irakien à cause d'une accumulation d'erreurs d'arrondi créant un décalage de 0,34 secondes sur le calcul de la trajectoire de ce missile. Il a suffit de cette très minime erreur de temps pour que l'antimissile rate sa cible de 500 mètres.

En 1982, la Bourse de Vancouver (Vancouver Stock Exchange) a institué un nouvel indice initialisé à une valeur de 1 000. L'indice était mis à jour après chaque transaction. 29 mois plus tard, il était tombé à 520. La cause était que la valeur

actualisée était tronquée plutôt que arrondie. Le calcul arrondi a donné une valeur de 1098,892.

II. Différents langages

A. Format des nombres usuels

Nous utilisons trois langages différents (Python, C, Java) pour effectuer nos tests numériques afin de démontrer différentes erreurs d'arrondi dans ces langages.

Que ce soit en C, en Python ou en Java, les nombres usuels sont stockés sous différents formats ayant une taille (en octets) et une plage de valeur acceptées pour chacune d'entre elles.

Ensemble des types en C sont :

- **Char :** Il représente les plus petites unités adressables de la machine pouvant contenir un jeu de caractères de base (Taille : 8 bits) (%c).

- Signed char : De la même taille que le char, mais avec la garantie d'être signé. Capable de contenir au moins les nombres [-127, +127] (Taille : 8 bits) (%c ou %hhi).

- Unsigned char : De la même taille que le char, mais non signé. Contient au moins les nombres [0, 255] (Taille : 8 bits) (%c ou %hhu).

- Short, short int, signed short et signed short int : Il représente les types d'entiers courts signés. Capable de contenir au moins la plage [-32 767, +32 767] (Taille : 16 bits) (%hi ou %hd)

- Unsigned short, unsigned short int : Il représente les types d'entiers courts non signés. Ils contiennent au moins la plage [0, 65 535] (Taille : 16 bits) (%hu).

- Int, signed, signed int : Il représente les types d'entiers signés. Capable de contenir au moins la plage [-32 767, +32 767] (Taille : 16 bits) (%i ou % d).

- Unsigned, unsigned int : Type d'entier non signé de base. Ils contiennent au moins la plage [0, 65 535] (Taille : 16 bits) (%u).

- Long, long int, signed long, signed long int : Type d'entier signé long. Capable de contenir au moins la plage [-2 147 483 647, +2 147 483 647] (Taille : 32 bits) (%li ou %ld).

- Unsigned long, unsigned long int : Type d'entier long non signé. Capable de contenir au moins la plage [0, 4 294 967 295] (Taille : 32 bits) (%lu).

- Long long, long long int, signed long long, signed long long int : Type d'entier long signé. Capable de contenir au moins la plage [-9 223 372 036 854 775 807, +9 223 372 036 854 775 807] (Taille : 64 bits) (%lli ou %lld).

- Unsigned long long, unsigned long long int : Type d'entier long non signé. Il contient au moins la plage [0, +18 446 744 073 709 551 615] (Taille : 64 bits) (%llu).

- Float : Type à virgule flottante réel, généralement appelé type à virgule flottante de simple précision (Taille : 32 bits) (%f ou %F ou %g ou %G ou %e ou %E ou %a ou %A).

- Double : Type à virgule flottante réelle, généralement appelé type à virgule flottante à double précision (Taille : 64 bits) (%lf ou %lF ou %lg ou %lG ou %le ou %lE ou %la ou %lA).

- Long double : Type à virgule flottante réel, généralement appelé type à virgule flottante de quadruple précision (Taille : 128 bits) (%Lf ou %LF ou %Lg ou %LG ou %Le ou %LE ou %La ou %LA).

Les principaux types en python sont :

.

- Type int (entier) : Ce type est utilisé pour stocker un entier.

- Type float (flottant) : Ce type est utilisé pour stocker des nombres à virgule flottante, désignés en anglais par l’expression floating point numbers..

- Type str (chaîne de caractères) : Sous Python, une donnée de type str est une suite quelconque de caractères délimités soit par des apostrophes (simple quotes), soit par des guillemets (double quotes). str est l’abréviation de string.

- Type bool (booléen) : Le type bool est utilisé pour les booléens. Un booléen peut prendre les valeurs True ou False.

- Type list (liste) : Sous Python, on peut définir une liste comme une collection d'éléments séparés par des virgules, l'ensemble étant enfermé dans des crochets.

- Type complex (complexe) : Python possède nativement un type pour manipuler les nombres complexes. Sauf que la partie imaginaire est indiquée par la lettre "j" car en informatique la lettre "i" est souvent utilisée dans des boucles.

Le Java est comparable au Python au niveau des types, nous avons choisi de ne pas les lister encore une fois pour ce langage.

B. Cas type

Il existe différents cas où les erreurs d'arrondi peuvent influer sur le résultat, notamment :

- lorsque que les nombres possèdent de nombreux chiffres différents de 0, la mantisse est arrondie pour ne pas saturer la mémoire (Pi par exemple),

- quand il y a une perte de chiffres significatifs par arrondi lorsque les nombres sont grands,

- lors d'un calcul long avec de nombreuses opérations, les erreurs d'arrondi s'accumulent et peuvent mener à un résultat approximatif (erreur complémentaire),

- si le résultat d'un calcul est très petit (x - y avec x presque égal à y), l'erreur peut-être conséquente et le résultat complètement faux,

- lors d'un calcul demandant des ressources supérieures à celles à disposition de l'ordinateur, il est possible que certaines machines réduisent le nombre de chiffres significatifs pour simplifier le calcul, ce qui engendre des erreurs.

Ces erreurs peuvent avoir de graves conséquences, notamment lors d'importants calculs en temps réel. Par exemple, dans les systèmes embarqués d'une voiture autonome (tesla), des calculs prenant en compte de nombreux paramètres (vitesse, distance par rapport à un obstacle, adhérence etc …) sont effectués en permanence. Si les erreurs d'arrondi sont trop importantes, l'ordinateur de bord pourrait par exemple décider d'une accélération alors que la vitesse est suffisante, et donc éventuellement provoquer un accident. Il est donc essentiel aux développeurs de trouver des solutions pour atténuer ces erreurs d'arrondi.

III. Démonstration de la précision machine

A. Définition

Erreur aproximation max causé par arrondi.

La précision machine correspond au nombre de chiffres significatifs qui peuvent être stockés. Cela correspond aux nombres que le machine peut traiter et enregistrer fidèlement. Plus le nombre de chiffres significatifs est élevé, plus les calculs qui seront effectués pourront être précis à leurs tours (plus grande précision machine).

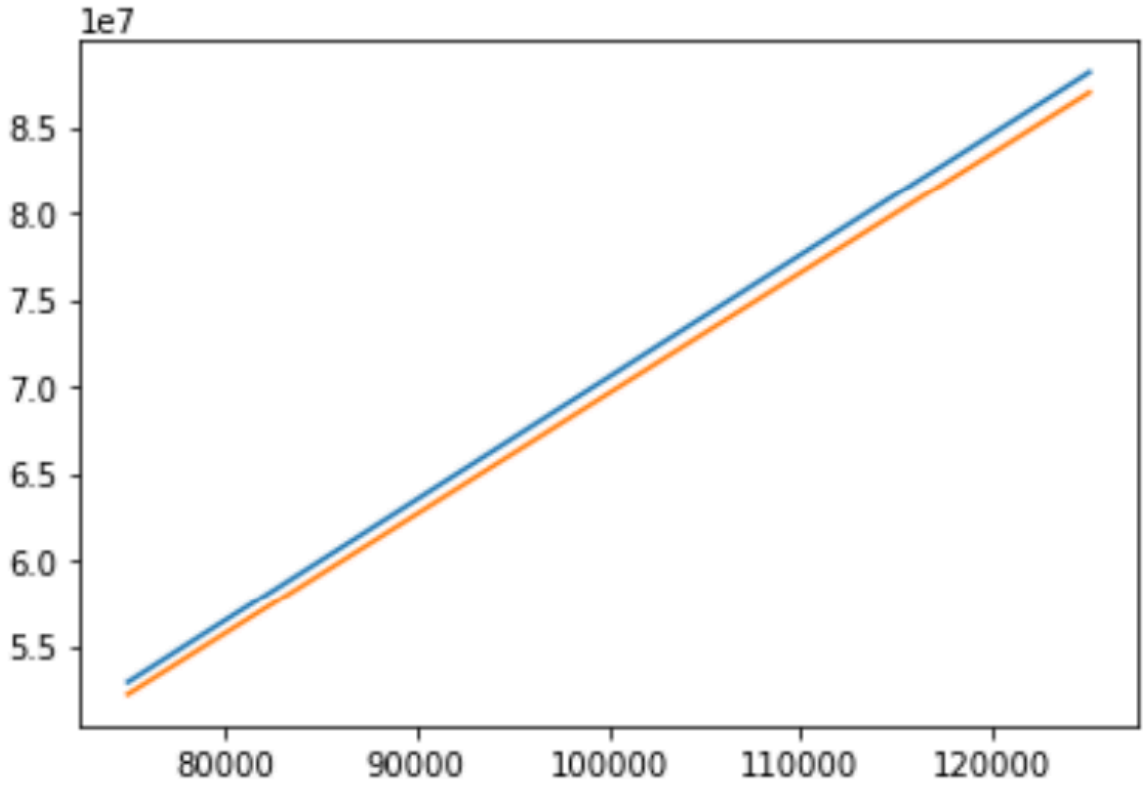
B. Démonstration

Dans le premier programme détaillé dans le sujet, on obtient le nombre de chiffres significatifs qui peuvent être stockés (pour les nombres flottants) en Python :

```
Le résultat est de :  1.1102230246251565e-16
```

Ce résultat montre la précision machine d'un double (plus petite valeur précise sans exposant) que Python peut contenir et afficher. On l'appelle epsilon noté ε.

Pour le second programme, ce graphe est un moyen de comparaison afin de démontrer la précision machine, à l'aide de deux valeurs dont une est infinie et l'autre n'est que son arrondi aux centième près.



Dans notre cas, la courbe bleue représente la valeur Pi (nombre irrationnel) qui s'accroît avec une valeur aléatoire. La courbe orange représente une valeur approchée de Pi (dans notre cas 3.14), qui elle-même s'accroît avec la même valeur que la précédente. En étant moins précis les deux courbes seraient juxtaposées et seraient peu distinguables l'une de l'autre, mais avec plus de précision, comme ici, nous nous apercevons d'une différence notoire entre les deux courbes. Cet exemple illustre bien la précision machine en raison de la précision des courbes et du calcul et permet, par la même occasion, de montrer comment une erreur d'arrondi se produit.

En effet, plus les valeurs du graphe augmentent, plus les courbes tendent à se distancier. Par le biais de l'accumulation des nombres décimaux de Pi, il y a une réelle différence qui se creuse entre Pi et son arrondi, justifiant la précision machine.

IV. Influence des erreurs d'arrondi

En mathématique, nous pouvons écrire une infinité de nombres car il suffit simplement d'ajouter un chiffre pour qu'il se concrétise.

A l'inverse, en informatique la capacité de stockage est limitée, alors que les nombres, eux, sont par définition infinis. Par conséquent, les nombres décimaux sont stockés sous la forme de nombre à virgule flottante (float). L'informatique fonctionnant grâce à l'électronique, tout est calculé en binaire, et la représentation dispose d'un certain nombre de bits pour la mantisse, ainsi que d'un certain nombre de bits pour l'exposant.
Cette répartition va donc dépendre du type choisi : 8, 16, 32 ou 64 bits.
Dès lors, certains nombres, lors de la conversion en binaire pour être enregistrés, vont être arrondis (manque de bits), et la valeur exacte n'a donc pas de réelle existence pour un ordinateur.

A. Erreurs d'arrondi en Python

En Python, il suffit d'un simple calcul avec des nombres flottants pour expérimenter une erreur d'arrondi. Dans un premier cas je réalise le calcul suivant :

0.1 + 0.2

Le résultat mathématique est égal à 0.3, mais le compilateur de Spyder, lui, va me donner un tout autre résultat :

```
Erreur simple d'arrondi (Addition de deux nombres flottants 0.1 + 0.2) :
Résultat attendu = 0.3, résultat obtenu = 0.30000000000000004
```

Mieux encore, si je réalise le calcul suivant :

(0.1 + 0.1 + 0.1) - 0.3

Ici on s'attend logiquement à obtenir 0, mais à nouveau le compilateur va donner un autre résultat qui paraît complètement absurde :

```
In [3]: c = (0.1 + 0.1 + 0.1) - 0.3
   ...: print("Résultat attendu = 0.0, résultat obtenu =", c)
Résultat attendu = 0.0, résultat obtenu = 5.551115123125783e-17
```

Quelle est la raison d'un résultat aussi aberrant ? Et comment résoudre ces erreurs d'arrondi ?

Les ordinateurs représentent donc les nombres en binaire, pour notre cas, c'est la partie décimale qui pose problème. Par exemple, le nombre 0.1 n'a qu'une seule décimale en base 10, par contre il en a une infinité en base 2 (il existe aussi des cas inverses, comme par exemple 2/3 qui n'a qu'une seule décimale en base 3 où il vaut 0.2 alors qu'il va en avoir une infinité en base 10 où il vaut 0.66666666666...). Le système peut garder, seulement, 53 chiffres binaires. Une fois converti en base 10, le nombre le plus proche de 0.1 est :

0.1000000000000000055511151231257827021181583404541015625

Dans le premier calcul, le résultat réel pour l'ordinateur (donc 0.3) passant du binaire en base 10 est :

0.3000000000000000366453525910037569701671600341796875

Lors de la compilation le print a décidé d'arrondir ce résultat, ce qui donne le résultat obtenu ci-dessus.

Dans le second calcul, le problème est équivalent, les nombres étant approximatifs le calcul devient complètement faux lors des opérations arithmétiques.

Pour résoudre ces erreurs, il faudrait tout multiplier par 100 000 puis tout diviser par 100 000, pour être bien certain du résultat. Mais en Python il existe différents

modules pour résoudre ce type de problème (exemple : decimal, fractions, operator, etc…). Par exemple pour le second calcul, j'ai rajouté le module "decimal" qui va résoudre les problèmes des nombres flottants :

```
Résultat attendu = 0.0, résultat obtenu = 0.0
```

On peut aussi démontrer l'accumulation d'erreurs d'arrondi grâce à l'itération d'une variable dans une boucle.
Dans l'exemple des missiles américains, c'est l'accumulation d'erreurs d'arrondi qui est dangereuse. Pour démontrer cet effet là, j'ai réalisé un programme qui va réitérer le même calcul 50 fois. Lors des différentes itérations, on réalise un simple calcul : i = (e^1 - n (incrément) * i.

On part de I = 1, c'est-à-dire I_1, et on calcule I_n ainsi que I_{n-1}. Cependant, très rapidement, le résultat devient aberrant dès I_{18} (comme le montre le résultat ci-dessous).

Comme démontré dans le premier exemple, l'erreur d'arrondi est minime, mais en avançant dans les calculs l'erreur s'impose et finit par créer un résultat inexact emportant avec elle une suite de résultats encore plus absurdes.

```
n =  15  ,I(n) =  0.1604958541638526  , nI(n-1) =  2.5577859742951925
n =  16  ,I(n) =  0.15034816183740363  , nI(n-1) =  2.5679336666216415
n =  17  ,I(n) =  0.16236307722318344  , nI(n-1) =  2.5559187512358617
n =  18  ,I(n) =  -0.2042535615582568  , nI(n-1) =  2.922535390017302
n =  19  ,I(n) =  6.599099498065924  , nI(n-1) =  -3.8808176696068792
n =  20  ,I(n) =  -129.26370813285942  , nI(n-1) =  131.98198996131848
n =  21  ,I(n) =  2717.256152618507  , nI(n-1) =  -2714.5378707900477
n =  22  ,I(n) =  -59776.91707577869  , nI(n-1) =  59779.63535760715
n =  23  ,I(n) =  1374871.8110247385  , nI(n-1) =  -1374869.09274291
n =  24  ,I(n) =  -32996920.746311896  , nI(n-1) =  32996923.464593723
n =  25  ,I(n) =  824923021.3760792  , nI(n-1) =  -824923018.6577973
n =  26  ,I(n) =  -21447998553.05978  , nI(n-1) =  21447998555.77806
n =  27  ,I(n) =  579095960935.3323  , nI(n-1) =  -579095960932.614
n =  28  ,I(n) =  -16214686906186.586  , nI(n-1) =  16214686906189.305
n =  29  ,I(n) =  470225920279413.7  , nI(n-1) =  -470225920279411.0
n =  30  ,I(n) =  -1.4106777608382408e+16  , nI(n-1) =  1.410677760838241e+16
n =  31  ,I(n) =  4.3731010585985466e+17  , nI(n-1) =  -4.3731010585985466e+17
n =  32  ,I(n) =  -1.399392338751535e+19  , nI(n-1) =  1.399392338751535e+19
```

B. Erreurs d'arrondi en C

Il est intéressant de noter que selon le compilateur, le résultat peut être différent.
En C nous avons différents compilateurs comme : gcc (Gnu C Compiler), MinGW (MiNimalist Gnu for Windows), Clang voir Visual C++.

Pour notre premier cas, nous avons réalisé une comparaison entre deux langages informatiques différents avec un compilateur différents, Spyder pour Python et GCC pour le C.
Par exemple, un programme en C qui réalise une simple multiplication avec trois nombres flottants (3.1 * (2.3 * 1.5) = 10.695) :

En C (GCC) on obtient :

```
3.100000 x (2.300000 x 1.500000) = 10.694999
```

Mais à contrario, en Python (Spyder) on obtient :

```
3.1  x ( 2.3  x  1.5 ) =  10.695
```

Avec les mêmes conditions de calcul, les deux compilateurs dans deux langages différents ne produisent pas le même résultat.

Comme en Python, avec un simple programme et une simple addition, sur trois valeurs qui devraient avoir le même résultat nous retrouvons des différences dans les calculs. Ici deux calculs basiques sont réalisés :

Le premier : (1.10 + 3.30) + 5.50 = 9.9.
Le second : 1.10 + (3.30 + 5.50) = 9.9.

En C (GCC) :

```
(1.100000 + 3.300000) + 5.500000 = 9.900000

1.100000 + (3.300000 + 5.500000) = 9.900001
```

Comme en C, cette expérience peut être réalisée dans d'autres langages comme le Python et le Java mais ils produisent les mêmes erreurs sur ce cas précis. Et toujours pour les mêmes raisons évoquées plus haut pour le python. Les nombres n'existant pas, ils sont approchés par les nombres dits flottants. Et par conséquent les nombres en apparence simple, deviennent difficilement représentables pour un ordinateur.

C. Erreurs d’arrondi en Java

Comme dans la plupart des autres langages, les erreurs d’arrondi en Java sont fortement conditionnées par le type des variables choisies. Par défaut, un littéral représentant une valeur décimale est de type double (15 chiffres significatifs). Mais comme les nombres sont enregistrés en binaire, le même problème qu’en Python est rencontré (0.1 + 0.2 = 0.30000000000000004).

Pour pallier cela, nous pouvons importer la classe BigDecimal, qui permet de représenter des entiers ou des nombres flottants sans les limitations de taille des types primitifs.

```
System.out.println(0.1+0.2);

BigDecimal bd1 = new BigDecimal("0.1");
BigDecimal bd2 = new BigDecimal("0.2");
System.out.println(bd1.add(bd2));
```

```
0.30000000000000004
0.3
```

Ici, nous avons construit un *BigDecimal* à partir de sa représentation sous forme de chaîne de caractères. Mais il est également possible de le construire à partir d’un double, d’un tableau de caractères, en précisant le nombre de chiffres significatifs voulus etc ...

En conclusion, les erreurs d'arrondi nous rappellent qu’un ordinateur, sauf dans le cas de petits nombres entiers, ne calculent jamais avec une précision maximale. Cependant, il est souvent possible de pallier ces erreurs grâce à certains outils fournis par les langages utilisés. Il est donc important d’avoir conscience de ces erreurs, puis de trouver l’outil le mieux adapté pour s’en prémunir.

Il est important de les garder à l’esprit avant de prendre des décisions conditionnées par des résultats mathématiques, surtout qu’aujourd’hui, beaucoup de domaines sont impactés par le calcul numérique, que ce soit dans la finance, dans l’aviation, dans l’automobile ou même encore le monde industriel.

V. Bibliographie

Partie I : Introduction :

http://images.math.cnrs.fr/Erreurs-en-arithmetique-des.html

https://tel.archives-ouvertes.fr/tel-00287209/document

https://whatis.techtarget.com/definition/rounding-error#:~:text=Rounding%20error%20is%20the%20difference,value%20and%20the%20actual%20value.&text=For%20example%2C%20the%20irrational%20number,of%20light%20in%20a%20vacuum

https://www5.in.tum.de/persons/huckle/bugse.html

https://web.ma.utexas.edu/users/arbogast/misc/disasters.html

Partie II : Différents langages :

https://femto-physique.fr/omp/erreurs-numeriques.php

http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2008/PhD2008-07.pdf

https://www.apmep.fr/IMG/pdf/AAA98012.pdf

https://developpement-informatique.com/article/208/types-de-donnees-en-c

https://en.wikipedia.org/wiki/C_data_types

https://docs.python.org/fr/3.5/library/string.html

https://courspython.com/types.html

Partie III : Démonstration de la précision machine :

https://zestedesavoir.com/tutoriels/pdf/570/introduction-a-larithmetique-flottante.pdf

https://www.math.univ-paris13.fr/~japhet/Doc/Handouts/RoundOffErrors.pdf

http://images.math.cnrs.fr/Erreurs-en-arithmetique-des.html#:~:text=En%20comptant%20les%20chiffres%20binaires,exactement%20repr%C3%A9sentable%20dans%20ce%20format.

https://en.wikipedia.org/wiki/Machine_epsilon

Partie IV : Influence des erreurs d'arrondi :

https://koor.fr/C/Tutorial/Compilation.wp

http://www.lactamme.polytechnique.fr/descripteurs/FloatingPointNumbers.01.Fra.html

https://stackoverflow.com/questions/11638698/rounding-errors-in-python

https://stackoverflow.com/questions/53472132/how-do-i-avoid-round-off-error-in-this-list

https://stackoverflow.com/questions/249467/what-is-a-simple-example-of-floating-point-rounding-error

https://www.irisa.fr/sage/jocelyne/cours/precision/precision-2016.pdf

https://www.xspdf.com/resolution/50630556.html

https://realpython.com/python-rounding/

http://homepages.ulb.ac.be/~majansen/teaching/INFO-F-205/slides02erreurs2x2nup.pdf

http://serge.mehl.free.fr/anx/acc_arr.html

http://sametmax.com/les-nombres-en-python/ ("L'éternel problème de la virgule qui part en couille")

https://miashs-www.u-ga.fr/prevert/Prog/Java/CoursJava/lesBig.html