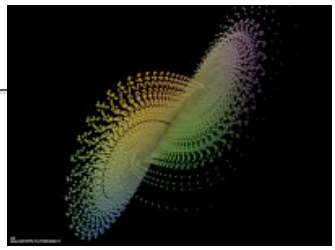
Un ordinateur est-il une parfaite machine à calculer ?

Jean-François Colonna

Un ordinateur est-il une parfaite machine à calculer ?



Le 28 avril 2010, par **Jean-François Colonna**
CMAP (Centre de Mathématiques Appliquées), Ecole Polytechnique, CNRS (**page web**)

LES ordinateurs sont partout et lorsque l'on voit leurs performances, par exemple, dans le domaine des jeux vidéos on a trop tendance à oublier que fondamentalement ils ne « savent » faire que les quatre opérations arithmétiques élémentaires (addition, soustraction, multiplication et division). Face donc à leur omniprésence, il est essentiel de savoir s'ils effectuent correctement ces calculs de base auxquels tout doit se ramener.

Toutes les valeurs numériques que nous utilisons, en particulier dans la vie courante, sont exprimées en base 10. Rappelons que, par exemple, l'écriture $5039{,}28$ signifie :

$$5 \times 10^3 + 0 \times 10^2 + 3 \times 10^1 + 9 \times 10^0 + 2 \times 10^{-1} + 8 \times 10^{-2}$$

Malheureusement, la base 10 ne peut être utilisée dans un ordinateur car pour ce faire il faudrait pouvoir disposer de systèmes physiques à dix états d'équilibre or ceux-ci seraient très difficiles à concevoir et à réaliser. Par contre, nombreux sont les systèmes à deux états d'équilibre et par exemple un interrupteur électrique qui est soit ouvert soit fermé. C'est donc la base 2 qui sera utilisée dans les ordinateurs.

Pour entrer un nombre décimal (base 10) dans un ordinateur, il faudra donc le convertir en binaire (base 2) et inversement pour sortir une valeur binaire d'un ordinateur, il conviendra de la convertir en décimal. A l'intérieur d'un ordinateur, il y a deux types principaux de nombres :

- Les nombres entiers $\{\ldots, -2, -1, 0, +1, +2, \ldots\}$.
- Les nombres flottants dont le principe, pour simplifier, consiste à ramener la valeur absolue de tout nombre dans $[1, 10[$ grâce à une multiplication par une puissance de 10 appropriée (négative, nulle ou positive). Ainsi, par exemple : $-5039{,}28 = -5{,}03928 x 10^3$. Un nombre flottant sera donc défini par un petit nombre décimal (appelé *mantisse*) et un exposant (respectivement $-5{,}03928$ et $+3$ dans l'exemple précédent).

La mémoire des ordinateurs contiendra donc des chiffres binaires 0 et 1 (ou bits contraction de l'anglais binary digits). Leur capacité pourra être énorme (par exemple mille milliards de bits), mais, et cela est très important à retenir, elle sera toujours finie.

En général, les nombres entiers sont définis à l'aide de 32 bits : un bit pour le signe (0 et 1 signifiant respectivement '+' et '-') et 31 bits pour la valeur absolue. Ainsi, seuls les nombres entiers de -2147483648 à $+2147483647$ seront représentables et manipulables. La représentation binaire des entiers est exacte ; ainsi, par exemple le nombre 25 s'écrira avec 32 bits :

$$00000000000000000000000000011001$$

car :

$$25 = 16 + 8 + 1 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 11001$$

En ce qui concerne les **nombres flottants**, 64 bits leurs sont réservés (il convient d'ajouter qu'il existe aussi une représentation n'utilisant que 32 bits, mais que celle-ci sera ignorée par soucis de simplicité). L'inventaire exhaustif des nombres alors accessibles est délicat à réaliser mais de toute évidence leur nombre est fini. A titre d'exemple le nombre pi (dont 2.699.999.990.000 décimales sont connues à la date de rédaction de cet article) ne peut donc être représenté exactement dans un ordinateur. La représentation flottante des nombres décimaux sera plus compliquée que celle des entiers, mais surtout elle sera en général approximative ; ainsi, par exemple le nombre décimal 0.3 s'écrira avec 64 bits (sans plus d'explications...) :

$$0011001100110011001100110011001100111111110100110011001100110011$$

dont la valeur est très proche de 0.3 sans toute fois l'égaler car, en effet, la somme d'un nombre fini d'inverses de

puissances de 2 ne peut être égale à 0.3 !

Intéressons-nous à quelques conséquences malheureuses de cela lors de calculs utilisant les nombres flottants.

Les nombres flottants ne sont pas des nombres au sens mathématique du terme car, en effet, les propriétés d'associativité (de l'addition et de la multiplication) et de distributivité de la multiplication par rapport à l'addition sont perdues. Rappelons qu'elles signifient respectivement :

$$(AB)C = A(BC)$$

$$(A + B) + C = A + (B + C)$$

$$A(B + C) = AB + AC$$

pour tout triplet de nombres A, B et C. Vérifions-le pour l'associativité en essayant les deux petits programmes suivant écrits dans le langage C (ces programmes élémentaires sont communiqués au lecteur afin de permettre à tout un chacun de reproduire le phénomène [1]) :

```
double    addition(x,y)
double    x;
double    y;
          {
          return(x+y);
          }
main()
          {
          double    a=1.1;
          double    b=3.7;
          double    c=5.5;

          double    x1,x2;

          x1 = addition(addition(a,b),c);
          x2 = addition(a,addition(b,c));

          printf("(%.6f + %.6f) + %.6f = %.15f\n",a,b,c,x1);
          printf("%.6f + (%.6f + %.6f) = %.15f\n",a,b,c,x2);
          }
```

```
(1.100000 + 3.700000) + 5.500000 = 10.300000000000001
1.100000 + (3.700000 + 5.500000) = 10.299999999999999
```

```
double    multiplication(x,y)
double    x;
double    y;
          {
          return(x*y);
          }
 main()
          {
          double    a=1.5;
          double    b=2.3;
          double    c=3.7;

          double    x1,x2;

          x1 =

          x2 =

          printf("(%.6f * %.6f) * %.6f
= %.15f\n",a,b,c,x1);
          printf("%.6f * (%.6f * %.6f)
= %.15f\n",a,b,c,x2);

          }
```

```
(1.500000 * 2.300000) * 3.700000 =
12.764999999999999
1.500000 * (2.300000 * 3.700000) =
12.765000000000001
```

Les résultats obtenus sont différents mais malgré tout proches l'un de l'autre. Mais en est-il toujours ainsi ? Exploitons cet autre petit programme [2] dans lequel nous allons calculer successivement les valeurs de x_0, x_1, etc qui dépendent les unes des autres, puis imprimer [3] celles-ci :

```
main()
          {
          double    A,B,x0,x1,x2,x3,x4,x5,x6,x7;

          B=4095.1;
          A=B+1;

          x0 = 1;
```

```
          x1 = (A*x0) - B;
          x2 = (A*x1) - B;
          x3 = (A*x2) - B;
          x4 = (A*x3) - B;
          x5 = (A*x4) - B;
          x6 = (A*x5) - B;
          x7 = (A*x6) - B;

          printf("x0=%+.16f\n",x0);
          printf("x1=%+.16f\n",x1);
          printf("x2=%+.16f\n",x2);
          printf("x3=%+.16f\n",x3);
          printf("x4=%+.16f\n",x4);
          printf("x5=%+.16f\n",x5);
          printf("x6=%+.16f\n",x6);
          printf("x7=%+.16f\n",x7);
          }
```

```
x0=          +1.0000000000000000
x1=          +1.0000000000004547
x2=          +1.0000000018631452
x3=          +1.0000076316294826
x4=          +1.0312599175240718
x5=        +129.0437481703507956
x6=     +524480.9968805739190429
x7=+2148322516.2225189208984375
```

Notons au préalable que ce programme très simple ne contient pas (ne peut pas contenir...) d'erreur de conception, ne fait pas appel à des méthodes d'approximation [4] et qu'enfin les réponses attendues (=1) sont connues a priori (ce qui est exceptionnel !). En effet, la propriété suivante est vraie :

$$A = B + 1 ==> A - B = 1 ==> x7 = x6 = x5 = x4 = x3 = x2 = x1 = x0 = 1$$

Malheureusement ce programme ne donne pas du tout des valeurs égales à 1 (sauf évidemment la première). Où est le problème ? En fait, $A - B$ n'est pas égale à 1 ; $A - B$ est égale à 1 plus ou moins ϵ (un simple bit), tout simplement parce que 4095.1 et 4096.1 ne sont pas représentables exactement dans un ordinateur en utilisant la représentation flottante ! Evidemment, il est possible d'imaginer (et il en existe) d'autres façons de représenter les nombres qui permettraient de résoudre ce problème particulier : on pourrait, par exemple, manipuler 4095.1 et 4096.1 comme deux fractions $40951/10$ et $40961/10$ et travailler ainsi uniquement avec des nombres entiers. On pourrait aussi imaginer que le compilateur (programme traduisant en instructions élémentaires les instructions données par l'utilisateur) se rende compte, par des manipulations « formelles », que tous les x_i sont égaux exactement à 1 et remplace alors les expressions $x_{i+1} = (Ax_i) - B$ par $x_{i+1} = 1$. Mais cela ne serait que répondre à des cas particuliers et ne résoudrait pas le problème général qui encore une fois vient de la capacité finie des ordinateurs.

L'intérêt de ce programme est donc d'une part de révéler de façon tout à fait violente un problème général d'exactitude des calculs dans un ordinateur. D'autre part il montre qu'une erreur infime (le simple bit qui faisait que A-B n'était pas égal à 1) peut s'amplifier de manière « explosive » ! Enfin, il fait référence à un processus appelé calcul itératif omniprésent, en particulier, en physique mathématique où une grandeur est transformée et retransformée suivant certaines lois. Cela sera le cas dans le dernier exemple avec les coordonnées tridimensionnelles des quatre corps (deux étoiles et deux planètes). En effet, ces dernières seront données a priori à l'instant initial puis transformées ensuite d'instants en instants selon les lois de Newton de la Physique classique.

Ces deux expériences « simplistes » démontrent qu'un ordinateur, quel qu'il soit, n'est pas une machine à calculer parfaite. La cause en est donc l'impossibilité de représenter exactement tous les nombres dont nous avons besoin.

Malheureusement, la plupart des calculs « utiles » qui sont effectués dans les ordinateurs font référence aux nombres réels. D'après ce qui précède, ces derniers ne pourront pas être ni représentés, ni manipulés exactement dans un ordinateur (sauf cas très particuliers, comme les petits nombres entiers...).

Rappelons d'abord qu'un ordinateur est fait de matériel (le *hardware*) et de programmes -ou logiciels- (le *software*). En général deux ordinateurs quelconques ne seront pas strictement identiques ; il en sera ainsi, en particulier, en ce qui

concerne l'interprétation des expressions mathématiques avec les programmes appelés *compilateurs*. Par exemple, l'expression suivante :

$$(A + B)(C + D)$$

pourra être « comprise » de plusieurs façons différentes :

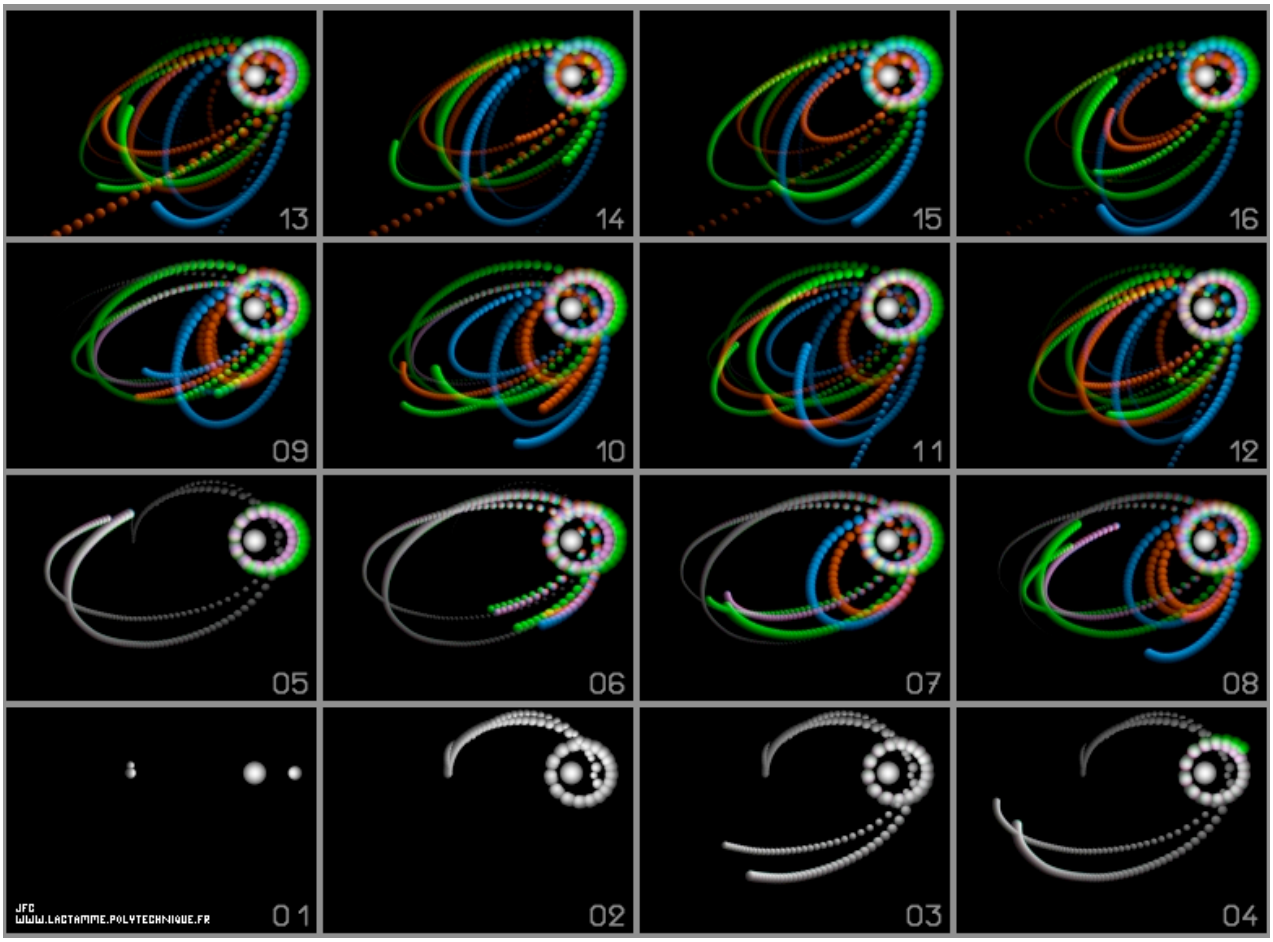
$$(A + B)(C + D)$$

$$A(C + D) + B(C + D)$$

$$AC + AD + BC + BD$$

$$etc...$$

qui sont équivalentes mathématiquement parlant. Mais ceci n'est plus vrai dans un ordinateur, à cause de la perte des propriétés d'associativité et de distributivité. Alors, dans ces conditions, un programme unique pourra produire des résultats non identiques s'il est exécuté sur plusieurs ordinateurs différents.



Cette expérience montre le calcul des trajectoires de deux planètes en orbite autour d'une étoile binaire. Le calcul est effectué sur trois ordinateurs différents à l'aide d'un seul et même programme aussi bien en ce qui concerne les instructions que les conditions initiales. Les résultats du premier sont coloriés en rouge, ceux du second en vert et enfin ceux du troisième en bleu. L'état initial est représenté dans l'image numérotée 01. Au début du calcul (jusqu'à environ l'image 04) les trois ordinateurs donnent pratiquement les mêmes résultats qui se superposent donc et apparaissent en blanc (puisque, suivant le principe de la synthèse additive des couleurs utilisé pour les écrans d'ordinateur, un point qui possède le même niveau de rouge, de vert et de bleu apparait gris). Au-dela de l'image 04, les trajectoires blanches semblent se subdiviser en trois, visualisant ainsi la divergence entre les trois machines qui sont alors bien loin d'être d'accord entre-elles et évidemment toutes les trois se trompent ! Une remarque s'impose alors : le calcul ici présenté repose sur les principes de la mécanique newtonienne et des méthodes dites *d'intégration numérique*, mais cela importe peu : en effet, ce qui est mis en évidence c'est qu'un programme peut donner des résultats différents suivant la machine sur laquelle il s'exécute.

Fort heureusement, tous les programmes ne sont pas sensibles à ce phénomène !

On pourra trouver ici d'autres exemples d'anomalies concernant, par exemple, l'influence du style du programmeur ou encore la difficulté de pratiquer le calcul dit *parallèle* sur des systèmes hétérogènes (c'est-à-dire non strictement identiques aussi bien au niveau du matériel que du logiciel) ou enfin la pérennité et la reproductibilité des résultats numériques.

Il est incontestable que l'ordinateur est une machine aux possibilités quasiment infinies, aussi bien dans la vie courante que dans la recherche scientifique la plus fondamentale. Mais il convient de na pas oublier qu'il n'est pas infaillible et que, comme tout outil, il n'est pas neutre. Connaître et maîtriser ses limites c'est pouvoir en tirer le meilleur parti, mais tout en restant vigilant.

Notes

[▲1] Dans ces programmes deux fonctions $addition(x, y)$ et $multiplication(x, y)$ sont définies afin de faire respectivement l'addition et la multiplication de deux nombres flottants 64 bits (dits *double*) x et y. Ces deux fonctions sont destinées à imposer l'ordre des opérations dans le programme principal (ou *main*) : puisque l'on veut comparer les valeurs de $(AB)C$ et de $A(BC)$ il faut être sûr que l'on calcule bien ces deux expressions. Sinon, imposer l'ordre des opérations est très difficile, voire impossible, même à l'aide de parenthèses, lorsqu'il y a plusieurs possibilités mathématiquement équivalentes.

[▲2] Pour des raisons de simplicité et de compréhension, la notion d'*itération* n'y est pas utilisée, alors qu'il pourrait être évidemment rédigé de façon beaucoup plus compacte

[▲3] à l'aide la fonction $printf$.

[▲4] Contrairement à l'exemple du calcul de trajectoires décrit à la fin de cet article.

Crédits images

Pour citer cet article : **Jean-François Colonna**, **Un ordinateur est-il une parfaite machine à calculer ?**. *Images des Mathématiques*, CNRS, 2010. En ligne, URL : **http://images.math.cnrs.fr/Un-ordinateur-est-il-une-parfaite.html**