Erreurs en arithmétique des ordinateurs

Vincent Lefèvre, Jean-Michel Muller

▶ **To cite this version:**

Vincent Lefèvre, Jean-Michel Muller. Erreurs en arithmétique des ordinateurs. Images des Mathématiques, 2009. hal-00598554

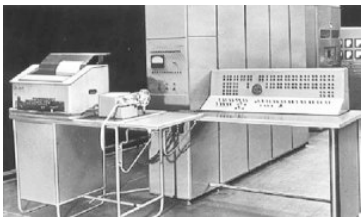
HAL Id: hal-00598554
https://hal.science/hal-00598554

Submitted on 6 Jun 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Erreurs en arithmétique des ordinateurs



Le 20 juin 2009, par **Vincent Lefèvre** et **Jean-Michel Muller**

Il est très difficile d'écrire un très gros programme informatique sans laisser au moins une erreur à l'intérieur. Heureusement, les erreurs sont souvent soit facilement détectables (et donc vite corrigées), soit peu dangereuses. Il arrive hélas parfois qu'une erreur pouvant avoir des conséquences graves passe inaperçue lors des tests du programme. Ceci peut être particulièrement préoccupant lorsque l'erreur est localisée dans l'un des programmes ou des circuits qui effectuent les opérations arithmétiques en machine : en effet, ces programmes ou ces circuits sont utilisés très souvent, par de nombreux logiciels, et parfois dans ce qu'on appelle des applications « critiques », comme le contrôle d'un véhicule ou d'un processus potentiellement dangereux. Bien entendu, les erreurs dangereuses ne sont pas toutes arithmétiques, mais c'est sur celles-ci que nous nous concentrerons.

Donnons quelques exemples particulièrement frappants d'erreurs récentes en arithmétique des ordinateurs.

- Dans la version 6.0 du système de calcul formel Maple, si on entrait :

2147483648041364781964379 4

la « quantité » affichée et mémorisée était :

413647819643790)+'—.(—.(

- L'opérateur de division de la première version du processeur Pentium d'Intel donnait parfois des résultats incorrects. Par exemple le calcul de

8391667/12582905

donnait 0.666869... au lieu de 0.666910... Ce problème a fait couler beaucoup d'encre en 1994. Il ne semble toutefois pas qu'il ait eu des conséquences graves… autrement que, peut-être, pour la carrière de l'ingénieur qui a commis la bévue.

- En 1998, à bord du navire lance-missiles américain USS Yorktown, un membre de l'équipage a par erreur tapé un « zéro » sur un clavier, au lieu d'une valeur non nulle. Ceci a provoqué une division par zéro. C'est d'habitude un problème bénin, mais visiblement le programmeur de l'application utilisée n'avait pas songé que ce problème pourrait arriver. Il s'en est suivi une cascade d'erreurs qui a fini par entraîner l'arrêt du système de propulsion du bateau.

Il y a de nombreux autres exemples de systèmes donnant des résultats incorrects à cause d'une mauvaise implantation de l'arithmétique. Ne paniquons pas : de tels problèmes restent rares. Ils ne le sont toutefois pas assez pour que l'on puisse complètement les négliger.

Il arrive que des erreurs proviennent d'une mauvaise spécification : les diverses équipes travaillant sur un même projet, ou encore le concepteur et l'utilisateur d'un programme ne se sont pas suffisamment mis d'accord sur ce que doit faire exactement le programme. Par exemple, la sonde Mars Climate Orbiter s'est écrasée sur Mars en septembre 1999 à cause d'une étourderie ahurissante : une partie des concepteurs des logiciels supposait que l'unité de mesure était le mètre, et l'autre partie que c'était le pied (l'unité anglaise).

Même lorsque le système informatique n'est pas en cause, certains calculs sont intrinsèquement difficiles et conduiront à des erreurs, même sur des machines irréprochables.

Considérons la suite de nombres 3/2, 5/3, 9/5, 17/9, etc., où chaque nouveau terme se calcule comme suit :

- on appelle x l'avant-dernière valeur calculée, et y la dernière ;
- le nouveau terme vaut

$$2003 - \frac{6002}{y} + \frac{4000}{xy}.$$

Par exemple, si on veut calculer le terme qui vient après $17/9$, on pose $x = 9/5$, $y = 17/9$, et on trouve aisément

$$2003 - \frac{6002}{y} + \frac{4000}{xy} = 33/17.$$

Cette suite de nombres *tend* vers 2 (ce qui veut dire que les termes u_n s'approchent de plus en plus de 2 lorsque n croît). Pourtant si on calcule ces termes sur n'importe quel ordinateur (sauf avec un système qui fait du calcul exact avec des nombres rationnels), on aura l'impression qu'elle tend vers 2000. Par exemple, la table suivante montre ce qu'on obtient avec une arithmétique virgule flottante de base 10 avec une précision de 10 chiffres (comme celle d'une calculette, par exemple).

n	Valeur calculée	Valeur exacte
2	1,800001	1,800000000
3	1,890000	1,888888889
4	3,116924	1,941176471
5	756,3870306	1,969696970
6	1996,761549	1,984615385
7	1999,996781	1,992248062
8	1999,999997	1,996108949
9	2000,000000	1,998050682
10	2000,000000	1,999024390

En annexe, le lecteur matheux trouvera une explication de ce comportement étrange.

La virgule flottante

La plupart des systèmes informatiques représentent les nombres réels dans le format « virgule flottante ». Ce format est très proche de la « notation scientifique » des calculatrices. L'idée est la suivante : on utilisera une base B (qui vaut 2 sur la plupart des ordinateurs [1]), et plutôt que de représenter un très petit nombre avec beaucoup de zéros à gauche ou un très grand nombre avec beaucoup de zéros à droite, on préfère représenter les nombres à l'aide d'un *signe*, d'une *mantisse* (qui est un nombre réel compris entre 1 et B) et un *exposant* (qui est un nombre entier). Le nombre représenté est égal à la mantisse fois B puissance l'exposant. Par exemple en base 10 la masse de l'électron vaut $9,109381 \times 10^{-31}$ Kg. Le signe est positif, la mantisse est 9,109381 et l'exposant est –31. Cette représentation est bien plus commode que

0,000000000000000000000000000009109381

En base 2, le nombre π s'écrit

$$1,10010010000111110\cdots \times 2^1$$

La mantisse est $1,10010010000111110\cdots$ et l'exposant est 1.

Comme nous l'avons dit auparavant, nos ordinateurs, à de rares exceptions près, représentent les nombres en base 2 (les seuls chiffres utilisés sont 0 et 1). La plupart des calculatrices de poche (ainsi que le logiciel Maple) utilisent toutefois la base 10, et il a même existé une machine russe, la machine SETUN (voir photo [2]), construite à une cinquantaine d'exemplaires dans les années 60, qui utilisait la base 3. Des informations sur cette machine peuvent être trouvées sur **Internet**.

Le lecteur qui voudrait connaître la base utilisée par sa machine peut le faire en utilisant le programme décrit ci-dessous.

Programme de calcul de la base utilisée de manière interne par un système. Ce programme, inventé par Malcolm, nécessite que les variables A et B soient représentées en virgule flottante (le type real ou float des langages de programmation courants). Le symbole « $\leftarrow$ » représente l'affectation (le « = » du langage C, ou le « := » de Pascal ou Maple).

$A \leftarrow 1$;

$B \leftarrow 1$;

Tant que $((A + 1) - A) - 1 = 0$ faire $A \leftarrow 2 * A$;

Tant que $((A + B) - A) - B \neq 0$ faire $B \leftarrow B + 1$;

Imprimer B.

Le lecteur devinera-t-il comment ce programme fonctionne ?

Une piste pour le lecteur matheux qui voudrait prouver ce programme : si β est la base du système virgule flottante utilisé et p le nombre de chiffres de mantisse, montrer qu'à la fin de la première boucle « tant que », A vérifie $\beta^p \leq A < \beta^{p+1}$.

On ne dispose en pratique que d'un nombre fini de chiffres pour représenter les mantisses et les exposants. Un nombre x est alors écrit en virgule flottante, en base 2 et avec n chiffres de mantisse lorsqu'il est représenté par un signe s (qui vaut +1 ou –1), une mantisse M qui s'écrit $m_0, m_1 m_2 \cdots m_{n-1}$ (il y a n chiffres de base 2) et un exposant E qui est un nombre entier tels que

$$x = s \times M \times 2^E .$$

Les premiers systèmes implantant l'arithmétique virgule flottante étaient très divers et en général très mauvais. Il n'était pas rare qu'en une seule opération arithmétique, on perde plusieurs chiffres de précision. Sur certaines machines, une multiplication par 1 pouvait conduire à un dépassement de capacité. Un standard datant de 1985, la norme IEEE 754, est venu mettre bon ordre dans cette pagaille. Il doit beaucoup à l'influence de William Kahan, professeur à l'Université de Californie à Berkeley, qui a obtenu la médaille Turing pour ses travaux sur l'arithmétique virgule flottante. La norme IEEE 754 a été révisée en 2008. Nous utiliserons ici le vocabulaire de la version de 1985 de la norme, qui est mieux connu, pour l'instant, des utilisateurs.

Les deux principaux formats de nombres virgule flottante définis par la norme IEEE 754 sont le format *simple précision* (pour lequel $n = 24$) et le format *double précision* (pour lequel $n = 53$). En comptant les chiffres binaires (appelés bits) utilisés pour représenter le signe et l'exposant, un nombre simple précision est représenté sur 32 bits et un nombre double précision sur 64 bits. Par la suite, un format étant choisi, on appellera nombre machine un nombre exactement représentable dans ce format.

L'arrondi correct

La norme IEEE 754 spécifie en particulier comment doivent être faits les arrondis. En effet, lorsque l'on effectue une opération portant sur des nombres machine (par exemple, une multiplication), il est rare que le résultat exact de l'opération soit égal à un nombre machine. On doit alors l'arrondir, c'est-à-dire fournir un nombre machine proche du résultat exact. Plusieurs stratégies sont alors possibles : on peut chercher à fournir le nombre machine le plus proche du résultat exact (on parle alors d'*arrondi au plus près*), ou le nombre machine immédiatement inférieur au résultat exact (on parle alors d'*arrondi par défaut*), ou le nombre machine immédiatement supérieur au résultat exact (on parle alors d'*arrondi par excès*). Chacune de ces stratégies s'appelle un mode d'arrondi. La norme IEEE 754 propose 4 modes d'arrondis différents (les trois que nous venons de citer, plus un *arrondi vers zéro*). L'utilisateur doit en début de calcul en choisir un [3], qui sera appelé *mode d'arrondi actif*. La norme exige que pour les quatre opérations arithmétiques (addition, soustraction, multiplication et division) et la racine carrée, le système se comporte comme si le résultat d'une opération était calculé exactement, puis ensuite arrondi suivant le mode d'arrondi actif.

Cette propriété est appelée exigence d'arrondi correct. Elle présente de nombreux avantages :

- le comportement d'un programme n'utilisant que les opérations $+$, $-$, $\times$, $\div$, et $\sqrt{}$ est parfaitement prévisible. On peut élaborer des preuves, qui permettent de prédire de manière certaine des propriétés. Un exemple classique (et très utile en pratique) est le *lemme de Sterbenz* : si X et Y sont des nombres machine positifs tels que $X/2 \leq Y \leq 2X$, alors le calcul de $X - Y$ s'effectue exactement (ce qui signifie que le nombre $X - Y$ est un nombre machine : il n'est pas nécessaire de l'arrondir). Ce résultat est vrai dans n'importe quelle base, dès que l'arithmétique est avec arrondi correct. Plusieurs chercheurs (dont John Harrison d'Intel, et Sylvie Boldo et Guillaume Melquiond, de l'INRIA) travaillent activement sur la preuve formelle de programmes utilisant l'arithmétique virgule flottante. Ce travail pionnier devrait permettre de garantir le bon comportement de parties critiques d'un programme numérique. La preuve formelle est également le seul moyen de garantir qu'un opérateur arithmétique (par exemple un diviseur) fait bien ce que l'on souhaite.
- La définition des opérations arithmétiques étant indépendante de la machine utilisée, si deux machines différentes effectuent le même calcul sans changer l'ordre des opérations, elles obtiendront le même résultat. Ceci facilite grandement ce qu'on appelle la portabilité des programmes (qui est le problème de faire fonctionner sur une machine donnée un programme conçu sur une autre machine).

Le dilemme du fabricant de tables

Dans la version de 1985 de la norme IEEE 754, cette exigence d'arrondi correct n'est pas formulée en ce qui concerne les autres fonctions que $+$, $-$, $\times$, $\div$, et $\sqrt{}$. Une conséquence de l'absence de spécification de ces fonctions est que parfois, certains systèmes donnent des résultats étranges lorsqu'on les calcule.

Si ces fonctions n'étaient pas spécifiées, c'est à cause d'une difficulté connue sous le nom de Dilemme du Fabricant de Tables. Essayons de décrire sommairement ce problème. On s'intéresse à l'évaluation des fonctions dites « élémentaires » (sinus, exponentielle, logarithme, tangente, etc. : ce sont des fonctions très importantes en mathématiques mais il n'est pas nécessaire de les connaître pour comprendre la suite). Cette évaluation ne peut s'effectuer qu'en calculant une *approximation* du résultat (à l'exception de cas particuliers comme $\log(1) = 0$, on ne peut calculer la valeur exacte). Tout le problème est alors de savoir si en arrondissant l'approximation, on obtiendra le même résultat que celui qu'on aurait en arrondissant la valeur exacte. Imaginons par exemple une machine fonctionnant en base dix ($B = 10$), avec des mantisses de 6 chiffres, et supposons que l'on veuille arrondir au plus près. On cherche le sinus du nombre machine $2,18998$. Ce sinus vaut

$$0,81435250000019\cdots$$

Il est donc presque égal au milieu des nombres machine $x = 0,814352$ et $y = 0,814353$. Pour savoir si on doit retourner x ou y comme résultat du calcul, il faut calculer ce sinus avec une précision d'environ 13 chiffres.

Considérons également l'exemple suivant, en base 2 avec une mantisse sur 53 chiffres (ceci correspond à la double précision de la norme IEEE 754).

Le sinus du nombre machine x qui s'écrit

$$0,01111111100111011001110111001110011101000011110110110 1$$

en base 2 est égal à

$$0,01111010011001010100000111001100001100010001101001010111111111111111111111111111111111111111111111111111111111111111111111111111111111$$

dans cette dernière écriture, le chiffre « 1 » apparaît de manière consécutive 66 fois après le 53ème chiffre. Ceci signifie que cette valeur est très proche du nombre machine

$$B = 0,01111010011001010100000111001100001100010001101001011 0$$

Pour pouvoir calculer $\sin x$ en arrondi par défaut sans risquer de se tromper, il faudra faire les calculs intermédiaires avec au moins 120 chiffres dans ce cas-là. Si les calculs intermédiaires ne se font pas avec suffisamment de précision, l'erreur effectuée lors de l'approximation ne permet pas de déterminer si le résultat exact est inférieur ou supérieur à B . C'est ce problème qu'on appelle le *Dilemme du Fabricant de Tables*, car à l'origine, il s'est posé aux éditeurs de tables de valeurs de fonctions numériques.

Afin de pouvoir écrire un programme efficace de calcul d'une fonction f avec arrondi correct, il faut donc déterminer la précision à laquelle on doit faire les calculs intermédiaires pour être sûr que le Dilemme du Fabricant de Tables ne se produira jamais. Pour les fonctions les plus simples (addition, soustraction, multiplication, division et racine carrée) ce problème se résout facilement : c'est pour cela que la version de 1985 de la norme IEEE 754 impose qu'elles doivent être arrondies correctement. Mais ce n'est pas le cas des autres fonctions, plus « compliquées ». Pour ces fonctions, les meilleurs résultats connus ne sont pas utilisables aisément : en appliquant un théorème de théorie des nombres récent, dû à Yuri Nesterenko (de l'Université de Moscou) et Michel Waldschmidt (de l'Université Paris 6), on peut prouver que pour arrondir correctement l'exponentielle, le logarithme, le sinus ou le cosinus dans le format « double précision », il « suffit » de faire les calculs intermédiaires avec une précision allant de plusieurs millions à plusieurs milliards de chiffres. Ceci n'est pas totalement impossible (voir l'annexe), mais le temps de calcul et la consommation mémoire sont évidemment très importants. Et pourtant des arguments probabilistes nous permettent d'être presque certains qu'un peu plus d'une centaine de chiffres suffisent.

Explorer 18446744073709551616 cas

La seule solution connue actuellement pour trouver la précision minimale nécessaire aux calculs intermédiaires consiste à effectuer une recherche exhaustive pour chaque fonction et chaque précision cible. Il faut chercher les « pires cas », c'est-à-dire les nombres machine pour lesquels l'évaluation de la fonction demandera la plus grande précision intermédiaire. Le cas de la simple précision étant suffisamment simple (il y a « seulement » $2^{32} = 4294967296$ valeurs à tester par fonction, ce qui prend au plus quelques jours), nous nous sommes principalement intéressés à la double précision, d'autant plus que c'est le format le plus utilisé actuellement. Il y a $2^{64} = 18446744073709551616$ nombres machine dans ce format, et tester ces nombres un par un pour chacune des fonctions usuelles (sinus, cosinus, tangente, arctangente, logarithme, exponentielle) demanderait trop de temps (quelques siècles sur un gros réseau de machines actuelles).

L'un d'entre nous, Vincent Lefèvre, a conçu un algorithme pour tester globalement un ensemble de nombres machine sur un petit intervalle, en approchant efficacement la courbe de la fonction à tester par un segment de droite et en cherchant une minoration de la distance de ce segment aux sommets d'une grille régulière. Au bout de quelques années (!) de calcul sur des machines de l'École Normale Supérieure de Lyon, cela nous a permis d'obtenir un certain nombre de résultats pour la double précision, dont le « pire cas » donné plus haut, avec notamment des résultats complets pour certaines fonctions (exponentielle et logarithme, 2^x et $\log_2(x)$).

Les programmes sont actuellement en train d'être améliorés de façon à pouvoir compléter nos résultats en double précision, voire obtenir certains résultats dans le format dit « double précision étendue » (80 bits), pour lequel il y a $1208925819614629174706176$ nombres machine possibles.

Damien Stehlé (du CNRS, actuellement à la Macquarie University et à l'Université de Sydney, en Australie) et Paul Zimmermann (de l'INRIA), ont récemment mis au point un algorithme basé sur des travaux de Coppersmith, qui sont ordinairement utilisés dans le domaine de la cryptographie. Cet algorithme permet d'obtenir une meilleure complexité théorique que celui de Vincent Lefèvre, ce qui signifie qu'il sera d'autant plus intéressant que la précision est grande. Les résultats déjà obtenus sont prometteurs. Malgré ceci, sauf avancée théorique majeure en théorie des nombres, le cas de la quadruple précision (nombres de 128 bits) semble toujours inaccessible, à cause du trop gros nombre de valeurs à tester. Les meilleurs algorithmes connus demanderaient, sur une machine actuelle, plusieurs milliards d'années.

Conclusion

L'arithmétique de nos machines évolue : bientôt toutes les fonctions usuelles (et plus seulement les quatre opérations arithmétiques et la racine carrée) pourront être avec « arrondi correct », tout au moins dans les formats « simple précision » et « double précision ». D'ailleurs, en partie suite à nos travaux, la révision de la norme IEEE 754, qui date d'août 2008, recommande maintenant (sans l'imposer toutefois) l'arrondi correct des principales fonctions mathématiques. Nous espérons qu'il en résultera une meilleure qualité et une meilleure portabilité des programmes numériques. Et pourtant… les programmes numériques sont de plus en plus gros, conçus par des équipes de plus en plus nombreuses : la probabilité qu'une erreur se glisse quelque part ne peut que croître, sauf peut-être si l'utilisation d'outils de preuve formelle, pour valider des « parties critiques » bien isolées, se généralise. La communauté informatique (et même de manière plus générale une bonne partie de la communauté scientifique) est maintenant confrontée au formidable défi de la complexité : comment comprendre le fonctionnement, comment garantir le bon comportement, d'un dispositif, ou d'un programme, ou d'une machine composé de milliers (si ce n'est de millions) de parties interagissantes ?

Annexe 1 : Comment calculer des fonctions sur plusieurs millions de chiffres ? (piste rouge)

Lorsque l'on désire calculer un sinus, un logarithme, une exponentielle, etc., avec une précision allant de quelques dizaines à quelques milliers de chiffres, la solution la plus couramment retenue consiste à approcher la fonction désirée par un polynôme. Pour obtenir de bien plus grandes précisions à un coût raisonnable, on utilise la moyenne arithmético-géométrique de Gauss-Legendre. Si on part de deux valeurs a_0 et b_0, et que l'on construit des suites de valeurs a_n et b_n à l'aide des relations

$$\begin{aligned} a_{n+1} &= \frac{a_n + b_n}{2} \\ b_{n+1} &= \sqrt{a_n b_n} \end{aligned}$$

on peut montrer que les deux suites a_n et b_n convergent très vite (c'est-à-dire s'approchent très vite) d'une valeur $A(a_0, b_0)$, qui est appelée moyenne arithmético-géométrique de a_0 et b_0. La table ci-dessous donne les premières étapes du calcul de $A(1, 2)$.

i	a_i	b_i
0	1	2
1	1.5	1.4142135623730
2	1.4571067811865	1.4564753152197
3	1.4567910481542	1.4567910139395
4	1.4567910310469	1.4567910310469

En utilisant (pour un x grand) la relation

$$\frac{\pi}{2A(1, 4/x)} = \log(x) + \frac{4\log(x) - 4}{x^2} + \frac{36\log(x) - 42}{x^4} + \cdots,$$

on se ramène au calcul d'une moyenne arithmético-géométrique bien choisie pour calculer des logarithmes avec une très grande précision.

Annexe 2 : la suite qui tend vers 2, mais à l'air de tendre vers 1000 en machine (piste rouge)

Cherchons à voir pourquoi la suite de nombres donnée en exemple dans l'introduction de cet article a un comportement aussi étrange. Dans cette partie, je supposerai que le lecteur possède un bagage mathématique de niveau L1 ou classe prépa.

La suite discutée plus haut est la suite de nombres $u_0, u_1, u_2, \ldots$ définie par :

$$\begin{cases} u_0 &= 3/2 = 1.5 \\ u_1 &= 5/3 = 1.666666\cdots \\ u_{n+1} &= 2003 - \dfrac{6002}{u_n} + \dfrac{4000}{u_n u_{n-1}} \end{cases}$$

Il n'est pas très difficile de montrer (par exemple par récurrence), qu'une suite v_n satisfaisant la relation

$$v_{n+1} = 2003 - \frac{6002}{v_n} + \frac{4000}{v_n v_{n-1}}$$

est de la forme

$$v_n = \frac{\alpha + \beta \cdot 2^{n+1} + \gamma \cdot 2000^{n+1}}{\alpha + \beta \cdot 2^n + \gamma \cdot 2000^n},$$

où α, β et γ dépendent des « points de départ » v_0 et v_1. On en déduit aisément que si γ est non nul, alors la suite tend vers 2000, et que si γ est nul, mais pas β, elle tend vers 2 (si β et γ sont nuls, la suite perd quelque peu de son intérêt).

Les points de départs $u_0 = 3/2$ et $u_1 = 5/3$ ont été choisis pour que γ soit nul. La suite tend donc vers 2. Mais numériquement, dès que l'on a commis des erreurs d'arrondi (aussi infimes soient-elles), on se retrouve sur une « trajectoire » correspondant à une valeur de γ certes extrêmement petite mais non nulle, ce qui suffit à faire tendre les termes calculés vers 2000.

Notes

[▲1] Certes, ceux-ci nous affichent leurs résultats en base 10, mais avant de le faire ils effectuent une conversion de base. Ce sont les calculs *internes* qui sont effectués en base 2.

[▲2] La photo illustrant l'article représente l'ordinateur ternaire « Setun » développé à l'Université de Moscou (1958). Cet ordinateur a été fabriqué par l'usine mathématique de Kazan. 50 ordinateurs ont été fabriqués, dont 30 ont été exploités dans les Universités d'Union Soviétique. **Source**.

[▲3] S'il ne choisit rien, c'est l'arrondi au plus près qui est pris par défaut.

Pour citer cet article : **Vincent Lefèvre** et **Jean-Michel Muller**, **« Erreurs en arithmétique des ordinateurs »** — *Images des Mathématiques*, CNRS, 2009. En ligne, URL : **http://images.math.cnrs.fr/Erreurs-en-arithmetique-des.html**