INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Arithmétique flottante*

Vincent Lefèvre — Paul Zimmermann

**N° 5105**

Janvier 2004

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--5105--FR

# Arithmétique flottante

Vincent Lefèvre , Paul Zimmermann

Thème 2 — Génie logiciel
et calcul symbolique
Projet Spaces

Rapport de recherche n° 5105 — Janvier 2004 — 60 pages

**Résumé :** Ce document rassemble des notes d'un cours donné en 2003 dans la filière « Algorithmique Numérique et Symbolique » du DEA d'Informatique de l'Université Henri Poincaré Nancy 1. Ces notes sont basées en grande partie sur le livre *Elementary Functions. Algorithms and Implementation* de Jean-Michel Muller.

**Mots-clés :** nombre flottant, précision fixe, précision arbitraire, arrondi correct, norme IEEE 754

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

# Floating-point Arithmetic

**Abstract:** This document releases lecture notes given during 2003 at University Henri Poincaré (Nancy, France). These notes are mainly based on the book "Elementary Functions. Algorithms and Implementation" by Jean-Michel Muller.

**Key-words:** floating-point number, fixed precision, arbitrary precision, correct rounding, IEEE 754 standard

## 1. Introduction

1.1. **Représentation.** Les nombres à *virgule flottante* — plus couramment appelés nombres *flottants* — sont de la forme :

$$x = s \cdot m \cdot \beta^e$$

où $s$ est le *signe* de $x$ ($s = \pm 1$), $\beta$ est la *base* (usuellement 2 ou 10), $m$ la *mantisse*, et $e$ l'*exposant* de $x$. La mantisse $m$ comprend en général au plus $p$ chiffres en base $\beta$ ; on dit alors que $p$ est la *précision* de $x$. L'exposant $e$ est compris entre deux bornes : $e_{\min} \leqslant e \leqslant e_{\max}$.

Cette représentation n'est pas unique : par exemple, le flottant 3.1416 en base 10 et précision 5 peut s'écrire aussi $.31416 \cdot 10^1$, ou encore $31416. \cdot 10^{-4}$. La notion d'exposant n'est donc pas intrinsèque, et est relative à la convention choisie pour la mantisse. Une fois choisie une telle convention (par exemple, virgule après le premier chiffre non nul, ou devant celui-ci, ou encore après le $p^{\mathrm{e}}$ chiffre de la mantisse, ce qui revient à une mantisse entière), on peut calculer l'intervalle de nombres flottants correspondant à $[e_{\min}, e_{\max}]$.

Lorsqu'on convient d'une mantisse entière, on a alors $0 \leqslant m < \beta^p$, et on voit que les opérations sur les flottants se ramènent à des calculs entiers sur les mantisses. Cependant du fait de la plage d'exposants valides, ces entiers peuvent être très grands. Par exemple, que se passe-t-il quand on ajoute 1.0 et $1.0 \cdot 2^{100}$ ? Si on se ramène à un calcul entier, il faut ajouter 1 et $2^{100}$. On constate d'une part qu'il faut effectuer des calculs intermédiaires sur 100 bits, alors que le résultat final n'est voulu qu'avec 2 bits. De plus le résultat exact $1 + 2^{100}$ n'est pas représentable exactement sur 2 bits. Il faudra donc l'*arrondir*.

Le programme ci-dessous permet de déterminer automatiquement la base $\beta$ de votre ordinateur ou calculette (à condition qu'il ou elle ne simplifie pas `((A + 1.0) - A) - 1.0` en 0 comme le fait Maple, pour lequel il faut remplacer `A + 1.0` par `evalf(A + 1.0)`, et `A + B` par `evalf(A + B)`) [28] :

```
A := 1.0;
B := 1.0;
while ((A + 1.0) - A) - 1.0 = 0.0 do A := 2 * A od;
while ((A + B) - A) - B <> 0.0 do B := B + 1.0 od;
B;
```

1.1.1. *Virgule fixe.* À l'opposé, un nombre en virgule fixe s'écrit $x = d_{n-1}...d_1 d_0.d_{-1}...d_{-m}$ où $m$ est fixé, et $n \leqslant n_0$ (ce qui revient à considérer $n = n_0$, en autorisant des chiffres initiaux nuls), et $0 \leqslant d_i < \beta$. On voit que cela revient plus ou moins à fixer $e = e_{\min} = e_{\max} = -m$ avec une mantisse entière $d_{n-1}...d_1 d_0 d_{-1}...d_{-m}$, et donc une précision de $p = n + m$ chiffres en base $\beta$.

1.1.2. *Représentation redondante.* Parfois, une représentation redondante est utilisée pour la mantisse pour accélérer les calculs. Chaque chiffre peut alors prendre plus de $\beta$ valeurs, par exemple, $\beta + 1$ valeurs pour $[0, ..., \beta]$, ou $2\beta - 1$ valeurs pour $[-(\beta - 1), ..., \beta - 1]$. Pour $\beta = 2$, cette dernière représentation est appelée codage de Booth, avec des chiffres $-1, 0, 1$.

L'intérêt d'une représentation redondante est de pouvoir retarder les propagations de retenue dans une addition ou une soustraction. Par exemple, avec les chiffres $0, 1, 2$ en base

2, on peut calculer sans propagation la somme de deux nombres dont les chiffres sont 0 ou 1 : $1011 + 0111 = 1122$. L'addition peut alors s'effectuer en parallèle, ou indifféremment de la gauche vers la droite ou de la droite vers la gauche (calcul en ligne).

## 2. La norme IEEE 754

La norme IEEE 754 (1985) [22] définit 4 formats de flottants en base 2 : simple précision (correspondant généralement au type `float` en C), simple précision étendue (obsolète), double précision (correspondant généralement au type `double` en C), et double précision étendue (*extended* en anglais). Les exposants ci-dessous sont relatifs à une mantisse $1 \leqslant m < 2$.

| format | taille | précision | $e_{\min}$ | $e_{\max}$ | valeur max. |
|---|---|---|---|---|---|
| simple | 32 | 23+1 bits | -126 | +127 | $3.403...10^{38}$ |
| double | 64 | 52+1 bits | -1022 | +1023 | $1.798...10^{308}$ |
| *extended* | $\geqslant 79$ | $\geqslant 64$ | $\leqslant -16382$ | $\geqslant 16383$ | $1.190...10^{4932}$ |

Le format double précision étendue (64 bits de mantisse sur un total d'au moins 79 bits, ou 80 bits sans bit *implicite* — cf. section 2.3) est utilisé en interne dans les processeurs x86, et la révision en cours de la norme IEEE 754 (cf. `http://www.validlab.com/754R/`) définit le format quadruple précision (*quad*), avec 113 bits de mantisse sur un total de 128 bits.

2.1. **Nombres spéciaux et exceptions.** En plus des nombres « usuels » vus ci-dessus, la norme IEEE 754 définit trois nombres spéciaux : $-\infty$, $+\infty$ et NaN (Not-a-Number). De plus, le zéro est signé (il y a un $+0$ et un $-0$, qui sont égaux). Ces nombres doivent être traités aussi bien en entrée qu'en sortie par les fonctions arithmétiques. Par exemple, $1/(-0)$ donne $-\infty$, $1/(+0)$ donne $+\infty$, et $(+\infty) - (+\infty)$ ou $\sqrt{-1}$ donne NaN.

En plus de ces valeurs spéciales, la norme IEEE 754 impose de positionner certains drapeaux (*flags* en anglais) dans les cas suivants :

– drapeau inexact (*inexact*) : lorsque le résultat d'une opération ne peut être représenté exactement, et doit donc être arrondi :
– drapeau opération invalide (*invalid*) : en cas d'opération donnant comme résultat NaN comme $\sqrt{-1}$ ou $\infty/\infty$ ;
– drapeau division par zéro : en cas de calcul de $x/0$ ($x$ n'étant ni NaN, ni un zéro) ;
– drapeau débordement vers l'infini (*overflow*) : lorsque le résultat d'une opération est trop grand en valeur absolue à cause de la borne supérieure de la plage des exposants représentables ;
– drapeau débordement vers zéro (*underflow*) : lorsque le résultat d'une opération trop petit en valeur absolue à cause de la borne inférieure de la plage des exposants représentables ;

Une fois ces drapeaux positionnés, ils le restent jusqu'à ce que l'utilisateur les remette en position initiale (non levés). Lorsqu'un drapeau a été levé, il n'est donc pas possible de savoir quelle instruction a provoqué l'exception correspondante, sauf à tester le drapeau après chaque instruction.

L'utilisateur peut aussi demander à ce que chaque type d'exception déclenche une erreur à l'exécution (*trap* en anglais). C'est le comportement par défaut dans certains cas (certaines versions de FreeBSD par exemple).

L'importance du traitement des exceptions est parfaitement illustré par le crash du vol 501 d'Ariane en 1996 [2], qui a coûté quelques 500 millions de dollars. L'origine du problème s'est avérée être une erreur de programmation dans le système de référence inertielle. Plus précisément, un flottant 64 bits donnant la vitesse horizontale de la fusée était converti en entier signé sur 16 bits. Or l'entier obtenu étant plus grand que 32767, le plus grand entier représentable sur 16 bits, la conversion échouait, déclenchant une exception non traitée. Le plus drôle — si l'on peut dire — dans cette histoire est que tous les tests logiciels avaient été réussis, mais ils avaient été effectués avec les données d'Ariane 4, fusée moins puissante et donc moins rapide qu'Ariane 5, pour laquelle la vitesse horizontale restait inférieure au maximum de 32767 !

2.2. **Notion d'arrondi.** La norme IEEE 754 définit 4 modes d'arrondi : vers $-\infty$, vers $+\infty$, vers 0, et au plus proche. Un autre mode intéressant est l'arrondi à l'opposé de 0 (*away* en anglais), qui est le symétrique de l'arrondi vers 0, comme l'arrondi vers $+\infty$ est le symétrique de celui vers $-\infty$. Les arrondis vers $\pm\infty$ et vers 0 sont appelés *arrondis dirigés* car ils sont dirigés dans une direction donnée.

Soit $x$ la valeur exacte (i.e. en précision infinie) d'une opération, et $x^-, x^+$ les deux nombres flottants entourant $x$ (qui existent toujours, car $-\infty \leqslant x \leqslant \infty$, mais on peut avoir $x^- = x^+$ lorsque $x$ est représentable exactement) :

$$x^- \leqslant x \leqslant x^+.$$

L'arrondi vers $-\infty$ de $x$ est $x^-$, l'arrondi vers $+\infty$ est $x^+$, l'arrondi vers 0 est $x^-$ si $x \geqslant 0$, et $x^+$ si $x < 0$. L'arrondi au plus proche est quant à lui défini comme le nombre parmi $x^-$ et $x^+$ qui est le plus proche de $x$ ; en cas d'égalité, ce qui arrive quand $x$ est au milieu de deux nombres flottants, la norme IEEE impose de choisir celui dont la mantisse est paire, i.e. finit par un zéro (en base différente de 2, les deux définitions ne sont pas identiques). Par abus de langage, on appelle cette convention « l'arrondi pair ».

2.2.1. *Arrondi correct.* La conséquence la plus importante est que, une fois un mode d'arrondi choisi, le résultat d'une opération est parfaitement spécifié, en particulier il y a un seul résultat possible. On parle alors d'« arrondi correct ».

La norme IEEE 754 impose l'arrondi correct pour les 4 opérations de base (addition, soustraction, multiplication, division) ainsi que pour la racine carrée. Ainsi, un programme utilisant ces 5 opérations se comporte de manière identique sur toute configuration (processeur, système et compilateur) respectant la norme IEEE 754, sous réserve qu'il n'y ait pas de précision intermédiaire étendue (ou que celle-ci soit désactivée) et que le langage de programmation ne permette pas au compilateur de changer l'ordre des opérations si cela peut produire un résultat différent.

La propriété d'arrondi correct permet de construire des algorithmes prouvés utilisant ces 5 opérations de base (cf. TwoSum, FastTwoSum, Sterbenz, Dekker, sections 10.1, 10.4 et 10.5).

L'arrondi correct vers $-\infty$ et $+\infty$ permet d'effectuer de l'arithmétique d'intervalles, i.e. de calculer à chaque opération un encadrement du résultat exact. On n'a pas vraiment besoin d'un arrondi correct pour cela, le respect de la direction d'arrondi suffit, mais l'arrondi correct donne le meilleur résultat possible, et par conséquent la portabilité en découle.

Par contre, la norme IEEE 754 n'impose rien pour les autres opérations mathématiques (puissance, exponentielle, logarithme, sinus, cosinus, etc.) pour lesquelles aucune exigence n'est imposée.

On parle quelquefois d'arrondi *fidèle* (*faithful* en anglais) quand le résultat d'une opération est l'un des deux nombres $x^-$ et $x^+$ entourant la valeur exacte $x$. L'erreur est alors au plus d'un ulp, comme pour les arrondis dirigés, mais on ne sait pas quelle est la direction de l'erreur.

L'importance de l'arrondi est illustré par l'échec du missile Patriot en 1991 [2]. Pendant la guerre du Golfe, un anti-missile Patriot tiré de Dahran (Arabie Saoudite) ayant manqué l'interception d'un missile irakien Scud, ce dernier a tué 28 soldats et blessé quelque 100 autres personnes. L'erreur était à une imprécision dans le calcul de la date de l'anti-missile Patriot. Celui-ci dispose en effet d'un processeur interne, qui calcule l'heure en multiples de dixièmes de secondes. Le nombre de dixièmes de secondes depuis le démarrage du processeur est stocké dans un registre entier, puis multiplié par une approximation sur 24 bits de $1/10$ pour obtenir le temps en secondes. L'approximation sur 24 bits effectivement stockée était $209715 \cdot 2^{-21}$, soit une erreur d'environ $10^{-7}$. Le processeur du missile ayant été démarré une centaine d'heures auparavant, l'erreur totale était donc de $0,34$ seconde, temps pendant lequel le missile Scud parcourt plus de 500 mètres, d'où l'échec de l'interception.

Si la valeur stockée avait été l'arrondi au plus proche de $1/10$ sur 24 bits, soit $13421773 \cdot 2^{-27}$, alors l'erreur totale n'aurait été que de $0,005$ seconde, temps pendant lequel le missile Scud parcourt moins de 10 mètres, et il aurait donc été certainement détruit. Mais avec des si...

2.2.2. *Le problème du double arrondi.* Soit $x$ un nombre réel, $y$ l'arrondi en précision $p$ de $x$, et $z$ l'arrondi en précision $q < p$ de $y$. Alors $z$ n'est pas toujours l'arrondi en précision $q$ de $x$. Par exemple, en arrondi au plus proche, $x = 1.0110100000001$ arrondi à 9 bits donne $y = 1.01101000$, qui arrondi à 5 bits donne $z = 1.0110$ par la règle de l'arrondi pair, alors que l'arrondi direct de $x$ à 5 bits donne $z' = 1.0111$. On parle alors de problème de « double arrondi ». On peut montrer que ce problème n'arrive que pour l'arrondi au plus proche :

> Supposons l'arrondi vers zéro. Alors $z \leqslant y \leqslant x$, par conséquent $z$ est du bon côté par rapport à $x$. Il suffit donc de montrer qu'il ne peut pas exister d'autre flottant $z'$ de même précision $q$ que $z$ qui soit strictement entre $z$ et $x$. Si tel était le cas, on aurait $z' \leqslant y \leqslant x$ car $z'$ est aussi un nombre représentable dans la précision $p$ de $y$. Par conséquent, $z$ étant l'arrondi correct de $y$ en précision $q$, on ne peut pas avoir $z < z'$.

Le problème du double arrondi survient sur les processeurs x86 : par défaut, les calculs internes sont effectués en double précision étendue, puis le résultat final est stocké en double précision. Il est possible via une instruction spéciale d'imposer que les calculs internes soient faits en double précision, pour un respect de la norme IEEE 754 (à condition de ne pas utiliser de variables en simple précision).

2.3. **Nombres normalisés et dénormalisés.** Même en fixant la place de la virgule dans la mantisse, le même nombre peut avoir plusieurs représentations. Ainsi en base $\beta = 10$, avec une précision $p = 4$, les deux flottants 3.140 et $0.314 \cdot 10^1$ sont identiques. On parle de nombre *normalisé* lorsque le chiffre de poids fort de la mantisse est non nul, et pour tout réel représentable non nul, la représentation devient alors unique.

Une conséquence intéressante en base $\beta = 2$ est que pour un nombre normalisé, le premier chiffre (bit) de la mantisse est toujours 1. On peut alors décider de ne pas le stocker physiquement en mémoire, et on parle alors de bit de poids fort implicite (*implicit leading bit* en anglais).

Ainsi en double précision, le plus petit nombre normalisé positif est $x = 1.0 \cdot 2^{-1022} \approx 2.225 \cdot 10^{-308}$. Soit $y$ le nombre normalisé suivant $x$ : $y = (1+2^{-52}) \cdot 2^{-1022}$. Si on calcule $y-x$, la valeur exacte $2^{-1074}$ n'est pas représentable par un flottant normalisé, et est donc arrondie à zéro (en mode d'arrondi au plus proche). Il s'agit d'un débordement vers zéro (*underflow*). Le problème que cela pose est le suivant. Soit un programme contenant l'instruction suivante :

```
if x <> y then
   z = 1.0 / (x - y);
```

Le test `x <> y` est vérifié puisque $x$ et $y$ diffèrent en effet de $2^{-1074}$ ; par contre le résultat de $x-y$ en arrondi au plus proche est 0, donc la division $1/(x-y)$ va soit échouer si l'exception « division par zéro » déclenche une erreur, soit donner un résultat infini.

Les nombres *dénormalisés* permettent de régler ce problème. L'idée est de réserver une valeur spéciale de l'exposant (celle qui correspondrait à $e_{\min}-1$) pour représenter ces nombres. Il n'y a alors plus de « 1 » implicite en bit de poids fort, et la valeur stockée de la mantisse correspond à $0.d_1d_2...d_{p-1}$ au lieu de $1.d_1d_2...d_{p-1}$. En prenant $d_1 = ... = d_{p-2} = 0$ et $d_{p-1} = 1$, cela permet de représenter des nombres aussi petits que $2^{e_{\min}-(p-1)}$, soit $2^{-1074}$ en double précision. On peut montrer alors que lorsque $x \neq y$, l'arrondi de $x - y$ ne peut pas être nul :

> Si $x \neq y$, alors $|x - y|$ vaut au moins $\varepsilon = 2^{e_{\min}-(p-1)}$. Or $\varepsilon$ est exactement le plus petit nombre dénormalisé. Par conséquent on a soit $x - y \leqslant -\varepsilon$, soit $\varepsilon \leqslant x-y$, et l'arrondi de $x-y$ — quel que soit le mode d'arrondi — ne peut pas être zéro.

## 3. Calcul d'erreur

Le problème majeur du calcul flottant est la propagation des erreurs d'arrondi. Du fait du problème de cancellation (cf. section 3.4), le résultat final d'un calcul comprenant plusieurs opérations peut être éloigné de la valeur exacte, voire de signe contraire !

Un exemple classique est le « polynôme » de Rump :

$$R(x, y) = \frac{1335y^6}{4} + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11y^8}{2} + \frac{x}{2y}.$$

Le problème est de déterminer le signe de $R(77617, 33096)$.

Comme il n'est pas possible d'éviter la propagation des erreurs d'arrondi, le mieux que l'on puisse faire est de borner l'erreur finale. Pour cela, plusieurs méthodes existent : on peut soit borner l'erreur absolue, ou bien l'erreur relative, ou bien borner l'erreur en ulps.

3.1. **Erreur absolue.** On utilise généralement l'erreur absolue quand on connaît une majoration a priori des valeurs intermédiaires calculées. Par exemple, quand on calcule une somme $s = t_1 + t_2 + \cdots + t_n$, et que l'on sait que les sommes intermédiaires sont bornées en valeur absolue par $T$, alors on sait que chaque étape donne une erreur d'arrondi d'au plus $\frac{1}{2}\mathrm{ulp}(T)$ en arrondi au plus proche, et donc l'erreur finale est d'au plus $\frac{n-1}{2}\mathrm{ulp}(T)$.

3.2. **Erreur relative.** On utilise a contrario l'erreur relative quand on ne connaît pas de borne a priori sur les valeurs intermédiaires calculées. Soit une opération $c = \circ(a \diamond b)$, où $\diamond$ est une opération mathématique, et $\circ(x)$ donne l'arrondi correct de $x$. On sait par la propriété d'arrondi correct que $|c - (a \diamond b)| \leqslant \mathrm{ulp}(c)$ (on considère un arrondi dirigé ici, ajouter un facteur 1/2 pour l'arrondi au plus proche). Comme on peut borner $\mathrm{ulp}(c)$ par $2^{1-p}|c|$ (cf. ci-dessous), on en déduit $|c - (a \diamond b)| \leqslant 2^{1-p}|c|$, d'où

$$a \diamond b = (1 + \varepsilon)c,$$

avec $|\varepsilon| \leqslant 2^{1-p}$.

En fait, on peut montrer qu'on a aussi $|c - (a \diamond b)| \leqslant \mathrm{ulp}(a \diamond b)$, ce qui donne

$$c = (1 + \varepsilon')(a \diamond b),$$

avec $|\varepsilon'| \leqslant 2^{1-p}$.

Higham propose d'écrire pour chaque opération $c = (1 + \theta)(a \diamond b)$. En supposant que $a = (1+\theta_a)^n\tilde{a}$ où $\tilde{a}$ est la valeur exacte dont $a$ est une approximation, et que $b = (1+\theta_b)^m\tilde{b}$, on voit qu'on obtient une borne en $(1+\theta_c)^{1+n+m}$ pour la multiplication, et $(1+\theta_c)^{1+\max(n,m)}$ pour l'addition (à condition que $a$ et $b$ soient de même signe). Il suffit donc à chaque opération de compter l'exposant de $1 + \theta$, et on obtient une erreur relative finale en $(1 + \theta)^n - 1$.

3.3. **Unit in last place (ulp).** Un ulp (abréviation d'*unit in last place* en anglais) est le poids du plus petit bit significatif de la mantisse d'un flottant. Si $x = \pm 1.b_{-1}...b_{1-p} \cdot 2^e$, alors $\mathrm{ulp}(x) = 2^{e+1-p}$. Si la définition de l'exposant n'est pas intrinsèque et dépend du choix de position de la virgule, celle d'ulp est intrinsèque. On peut étendre cette définition à un réel $x$ quelconque, en disant que $\mathrm{ulp}(x)$ est l'ulp du flottant le plus proche de $x$ en direction de zéro, ou encore $\mathrm{ulp}(x) = 2^{\lfloor \log_2 |x| \rfloor + 1 - p}$.

Parfois il est plus agréable de calculer en termes d'ulp. Il faut savoir en effet qu'on peut perdre un facteur 2 en passant de l'erreur en ulp à l'erreur relative et réciproquement : en effet, une erreur d'un ulp peut être aussi grande que $2^{1-p}|x|$ (lorsque $x$ est une puissance de 2), et une erreur de $2^{1-p}|x|$ peut être presque aussi grande que 2 ulps (pour $x$ juste en-dessous d'une puissance de 2).

3.4. **Cancellation.** Il s'agit ici du problème principal du calcul flottant. Lorsque deux nombres sont très proches, la partie significative de leur différence après arrondi diminue d'autant. En d'autres termes, la valeur du résultat étant petite devant les opérandes, l'erreur commise sur ceux-ci lors des opérations antérieures en est multipliée d'autant.

Le « polynôme » de Rump est un exemple flagrant de ce problème, puisqu'il faut pas moins de 122 bits de précision (même avec arrondi correct) pour obtenir le signe du résultat.

Pour s'en prémunir, il faut soit réordonner les calculs pour éviter les cancellations, soit maîtriser l'erreur commise lors de ces cancellations.

## 4. Approximations polynomiales et rationnelles

Pour approcher une fonction donnée sur un intervalle donné (pas trop grand) et avec une précision fixée *a priori* (pas trop grande non plus), le plus simple est d'utiliser une approximation polynomiale ou rationnelle. Les polynômes sont en effet les objets les plus simples que l'on puisse calculer, puisqu'ils requièrent juste des additions et des multiplications. Quand on autorise aussi des divisions, on obtient des fractions rationnelles.

On distingue deux types d'approximation : celles qui minimisent l'erreur « moyenne » (par exemple, approximations par moindres carrés) et celles qui minimisent l'erreur maximale (approximations *minimax*).

Pour fixer les notations, on considère que la fonction est $f$, l'intervalle $[a, b]$, et qu'on cherche un polynôme $p$ de degré inférieur ou égal à $n$.

4.1. **Approximations par moindres carrés.** On veut ici minimiser la distance

$$||p - f||_2 = \sqrt{\int_a^b w(x)[p(x) - f(x)]^2 dx}$$

où $w$ est une fonction de poids.

Soit le produit scalaire défini par $\langle f, g\rangle := \int_a^b w(x)f(x)g(x)dx$, et une famille $(T_i)_{i\geqslant 0}$ de polynômes orthogonaux pour ce produit scalaire, c'est-à-dire $\langle T_i, T_j\rangle = 0$ pour $i \neq j$. On peut obtenir un polynôme optimal pour le poids $w(x)$ en calculant les coefficients

$$a_i = \frac{\langle f, T_i\rangle}{\langle T_i, T_i\rangle},$$

et en prenant comme polynôme

$$p^* = \sum_{i=0}^{n} a_i T_i. \tag{1}$$

Pour certaines valeurs du poids $w(x)$ et de l'intervalle $[a, b]$, des familles classiques de polynômes orthogonaux sont connues.

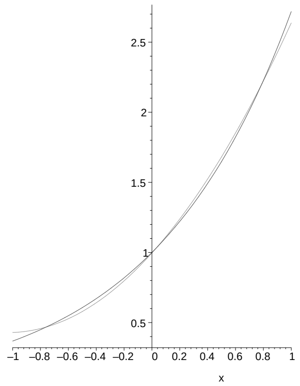
4.1.1. *Polynômes de Legendre.* Ils sont orthogonaux pour le poids uniforme $w(x) = 1$, sur l'intervalle $[-1, 1]$. Ces polynômes sont définis par $T_0(x) = 1$, $T_1(x) = x$,

$$T_n(x) = \frac{2n-1}{n} x T_{n-1}(x) - \frac{n-1}{n} T_{n-2}(x).$$

Par exemple, pour $f(x) = \exp(x)$ et $n = 2$, on obtient :

$$\begin{aligned} p^*(x) &= \frac{-3e^1 + 33e^{-1}}{4} + 3e^{-1}x + \frac{15e^1 - 105e^{-1}}{4}x^2 \\ &= 0.996294019 + 1.103638324x + 0.536721528x^2, \end{aligned}$$

ce qui donne $||p^* - \exp||_2 \approx 0.0014$, et le graphe suivant :




Le maximum de $|p^*(x) - \exp(x)|$ est atteint en $x = 1$, avec une valeur de 0.082.

4.1.2. *Polynômes de Chebyshev.* Ils sont orthogonaux pour le poids $w(x) = \frac{1}{\sqrt{1-x^2}}$, sur l'intervalle $[-1, 1]$. Ces polynômes sont définis par $T_0(x) = 1$, $T_1(x) = x$,

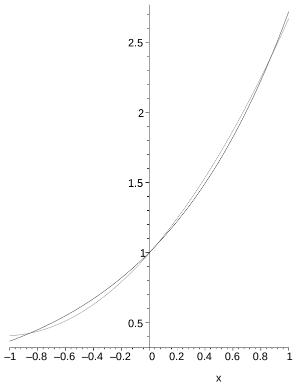
$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x),$$

ou plus simplement par $\cos(nx) = T_n(\cos x)$.

Toujours pour $f(x) = \exp(x)$ et $n = 2$, on obtient :

$$p^*(x) = 0.9945705384 + 1.130318208x + 0.5429906792x^2,$$

ce qui donne $||p^* - \exp||_2 \approx 0.0031$, et le graphe suivant :




Le maximum de $|p^*(x) - \exp(x)|$ est aussi atteint en $x = 1$, avec une valeur de 0.050.

4.2. **Approximations minimax.** On veut ici minimiser la distance

$$||p - f||_\infty = \max_{a \leqslant x \leqslant b} |f(x) - p(x)|.$$

Un polynôme $p^*$ de degré inférieur ou égal à $n$ réalisant le minimum de cette distance parmi tous les polynômes de degré $\leqslant n$ est appelé polynôme *minimax* de degré $n$ de $f$ sur $[a, b]$. Weierstrass a montré le résultat suivant en 1885 :

**Théorème 1.** *Soit $f$ une fonction continue sur $[a, b]$. Pour tout $\varepsilon > 0$, il existe un polynôme $p$ tel que $||p - f||_\infty \leqslant \varepsilon$.*

Un autre résultat, dû à Chebyshev, est le suivant :

**Théorème 2.** *Il existe un unique polynôme $p^*$ de degré inférieur ou égal à $n$ sur $[a, b]$ qui minimise $||p - f||_\infty$. Il est appelé polynôme* minimax *de degré $n$ de $f$ sur $[a, b]$. De plus, il existe $N \geqslant n + 2$ points $a \leqslant x_0 < x_1 < ... < x_{N-1} \leqslant b$ tels que*

$$p^*(x_i) - f(x_i) = (-1)^i [p^*(x_0) - f(x_0)] = \pm ||p^* - f||_\infty.$$

En d'autres termes, $p^* - f$ atteint son maximum en valeur absolue $\geqslant n + 2$ fois sur $[a, b]$, alternativement positivement et négativement.

Un algorithme dû à Remez [34] permet de calculer le polynôme *minimax* de degré $n$ de $f$ sur $[a, b]$. Voici un programme Maple simplifié effectuant ce calcul :

```
remez := proc(f, x, n, a, b) local pts, i, sys, eps, p, c, q, oldq;
   p := add(c[i]*x^i, i=0..n);
   pts := {seq(evalf((a+b)/2+(b-a)/2*cos(Pi*i/(n+1))), i=0..n+1)};
   while q<>oldq do
      sys := {seq(evalf(subs(x=op(i+1,pts),p-f))=(-1)^i*eps, i=0..n+1)};
      sys := solve(sys, {seq(c[i], i=0..n), eps});
      oldq := q;
      q := subs(sys, p);
      lprint(q);
      pts := sort([a, solve(diff(q-f,x), x), b]);
```

```
   od;
   q
end:

> remez(exp(x), x, 2, -1, 1);
.9891410756+1.130864333*x+.5539395590*x^2
.9890394441+1.130184090*x+.5540411905*x^2
.9890397287+1.130183805*x+.5540409059*x^2
.9890397284+1.130183805*x+.5540409062*x^2
.9890397284+1.130183805*x+.5540409062*x^2

                                                                2
                  .9890397284 + 1.130183805 x + .5540409062 x
```

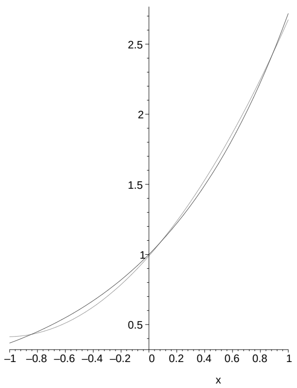
Cet algorithme est implanté dans le module `numapprox` de Maple :

```
> p:=numapprox[minimax](exp(x), x=-1..1, [2,0], 1, 'err'):
> expand(p), err;
                                                         2
         0.9890365552 + 1.130258690 x + 0.5540440796 x , 0.04505715466
```

Par exemple, pour $\exp(x)$ et $n = 2$, on obtient :

$$p^*(x) = 0.9890365552 + 1.130258690x + 0.5540440796x^2,$$

avec une distance maximale $|p^* - f|$ de 0.045 environ, et le graphe suivant :




Remarque 1. En général, une approximation de Taylor est bien moins bonne que les approximations en norme euclidienne ou minimax. Par exemple, pour $\exp(x)$ sur $[-1, 1]$, l'approximation de Taylor de degré 2 en $x = 0$, soit $1 + x + x^2/2$, donne une erreur d'environ 0.218 en $x = 1$, contre 0.082 pour l'approximation via les polynômes de Legendre, 0.050 pour ceux de Chebyshev, et 0.045 pour l'approximation minimax.

Remarque 2. L'approximation en norme euclidienne avec les polynômes de Chebyshev est souvent proche de l'approximation minimax. Ceci peut s'expliquer de la façon suivante : les coefficients $a_i = \frac{\langle f, T_i \rangle}{\langle T_i, T_i \rangle}$ décroissant rapidement, $a_{n+1}T_{n+1}$ est une bonne approximation de $f - p_n^*$. Or $T_{n+1}(x) = \cos((n+1)\cos^{-1} x)$ est une fonction qui prend $n+2$ fois les valeurs $\pm 1$ pour $-1 \leqslant x \leqslant 1$.

4.3. **Erreur relative.** En prenant comme poids $w(x) = |f(x)|^{-1}$, le polynôme minimax optimise la distance *relative* avec $f$, au lieu de la distance absolue avec $w = 1$. On obtient alors pour la fonction exponentielle sur $[-1, 1]$ avec $n = 2$ :

$$p^*(x) = 1.027030791 + 1.113889894x + 0.4693555168x^2,$$

avec une erreur relative d'au plus 0.040 environ :

```
> p:=numapprox[minimax](exp(x), x=-1..1, [2,0], 1/exp(x), 'err'):
> expand(p), err;
                                                           2
         1.027030791 + 1.113889894 x + 0.4693555168 x , 0.03974574998
```

Cependant, pour certaines fonctions comme la racine carrée, les approximations polynomiales, même minimax, sont de piètre qualité. Par exemple, sur $[1/4, 1]$, l'approximation minimax de degré 10 donne une erreur de $7.0 \cdot 10^{-8}$, alors qu'une approximation rationnelle de numérateur et dénominateur de degré 5 donne une erreur de $5.2 \cdot 10^{-12}$. On utilise alors des approximations rationnelles, ou on réduit la largeur de l'intervalle, en utilisant par exemple des tables de polynômes (cf. section 5).

4.4. **Approximations rationnelles.** On cherche ici une fraction rationnelle de la forme $r = p/q$, où $p$ est de degré $n$ et $q$ de degré $m$. Comme on peut fixer le coefficient constant de $q$ — à 1 par exemple —, on a donc $n + m + 1$ coefficients à fixer. Le cas $m = 0$ correspond à une approximation polynomiale.

Il existe dans ce cas une variante de l'algorithme de Remez, et on peut définir l'approximation rationnelle minimax, qui atteint le maximum de $|f - r|$ en $n + m + 2$ points, avec alternance des signes.

```
> p:=numapprox[minimax](exp(x), x=-1..1, [1,1], 1, 'err'):
> expand(p), err;
                 1.017021247          0.5175411565 x
            ------------------ + ------------------, 0.02097256305
            1 - 0.4397882839 x   1 - 0.4397882839 x
```

Un inconvénient des approximations rationnelles est qu'elles nécessitent une division finale, opération qui est souvent plus lente sur les processeurs actuels. Cependant la meilleure précision de ces approximations peut compenser cet inconvénient :

```
foo := proc(n) numapprox[minimax](exp(x), x=-1..1, [2*n,0], 1, 'e1');
numapprox[minimax](exp(x), x=-1..1, [n,n], 1, 'e2'); e1, e2 end:
> foo(1);
                             0.04505715466, 0.02097256305
> foo(2);
                           0.0005471825736, 0.00008692582114
> foo(3);
```

```
                                   -5                  -6
                     0.3213305387 10   , 0.1553032727 10
> Digits:=12: foo(4);
                                   -7                    -9
                   0.110714404311 10   , 0.162192037578 10
```

4.5. **Approximations avec coefficients fixés.** Un problème pratique des approximations polynomiales ou rationnelles est que les coefficients donnés par l'algorithme de Remez ne sont pas en général représentables exactement en machine. Par conséquent, non seulement on effectue des erreurs d'arrondi en évaluant $p(x)$ ou $r(x)$, mais en plus les valeurs utilisées des coefficients de ces polynômes sont fausses !

Une solution pour parer à ce problème est de déterminer de proche en proche les coefficients avec l'algorithme de Remez, en les arrondissant au plus proche à chaque fois.

Supposons par exemple qu'on veuille calculer un polynôme d'approximation de $2^x$ de degré 3 sur $[0, 1/8]$, avec des coefficients sur 24 bits :

```
> numapprox[minimax](2^x, x=0..1/8, [3,0], 1);
     0.9999999806 + (0.6931520763 + (0.2400319493 + 0.05797081006 x) x) x
```

On détermine alors le flottant sur 24 bits le plus proche de 0.9999999806, soit $a_0 = 1$. On cherche maintenant un polynôme de la forme $a_0 + x(a_1 + a_2x + a_3x^2)$, donc $a_1 + a_2x + a_3x^2$ doit approcher $\frac{2^x - a_0}{x}$ avec comme poids $w(x) = x$ :

```
> numapprox[minimax]((2^x-1)/x, x=0..1/8, [2,0], x);
                 0.6931507544 + (0.2400538055 + 0.05786898601 x) x
```

Le coefficient sur 24 bits le plus proche de 0.6931507544 est $a_1 = 2907285/2^{22}$, et ainsi de suite.

## 5. Méthodes à base de tables

Évaluer une fonction sur un grand intervalle avec une erreur bornée requiert un polynôme de grand degré, ou une fraction rationnelle avec un numérateur et un dénominateur de grands degrés. Par exemple, pour l'exponentielle sur $[0, \frac{\log 2}{2}]$, en double précision, il faudrait un polynôme de degré 9 au moins :

```
> numapprox[minimax](exp(x), x=0..log(2)/2, 9, 1/exp(x), 'err'):
> err;
                                                    -16
                        0.13183037374386121299 10
```

Une technique classique consiste à découper l'intervalle donné en plusieurs sous-intervalles, et à utiliser sur chacun de ces sous-intervalles un polynôme différent, stocké dans une table. Par exemple, pour la fonction exponentielle, en découpant l'intervalle $[0, \frac{\log 2}{2}]$ en 8 sous-intervalles, des polynômes de degré 6 suffisent pour obtenir une erreur de l'ordre de celle ci-dessus :

```
> numapprox[minimax](exp(x), x=0..log(2)/16, 6, 1/exp(x), 'err'):
> err;
                                                -17
                          0.693605083458 10
```

On constate sur cet exemple que découper en deux un intervalle permet de diminuer le degré d'une unité.

Cette méthode est d'autant plus intéressante avec les processeurs actuels, que la mémoire est de moins en moins onéreuse. David Defour a montré dans sa thèse [15] que la taille optimale d'une telle table ne devait pas dépasser la taille d'une voie du cache, soit 4 Ko sur les processeurs actuels.

Il n'est pas toujours nécessaire de stocker tous les polynômes. Par exemple, pour la fonction $\exp(x)$, si les sous-intervalles sont de la forme $[kh,(k+1)h]$, il suffit de calculer le polynôme pour l'intervalle $[0,h]$ — soit $k=0$ — et pour $kh \leqslant x \leqslant (k+1)h$, on approche $\exp(x)$ par $\exp(kh)\exp(x-kh)$, où seul $\exp(kh)$ est tabulé pour l'intervalle $k$, et $\exp(x-kh)$ est obtenu en se ramenant à l'intervalle $[0,h]$, puisque $0 \leqslant x-kh \leq h$.

On distingue trois types de tables :
- les tables classiques, avec des sous-intervalles de même longueur, dont les bornes $a_k$ sont régulièrement espacées. Ces tables nécessitent une précision interne plus grande que la précision finale voulue. En effet, en supposant qu'on tabule $f(a_k)$ dans la précision finale, l'erreur peut aller jusqu'à $\frac{1}{2}$ulp, ce qui laisse peu d'espoir d'obtenir une erreur finale bornée par $\frac{1}{2}$ulp. Ces tables conviennent par exemple aux processeurs x86, en utilisant en interne la précision étendue (64 bits de précision) pour une précision finale de 53 bits (double précision). La méthode de Tang utilise de telles tables ;
- les « tables précises », qui utilisent des points irrégulièrement espacés, en lesquels la valeur de la fonction est très proche d'un nombre machine. Avec un peu de soin, elles peuvent être implantées en faisant tous les calculs avec la précision finale. La méthode de Gal utilise de telles tables ;
- enfin, d'autres méthodes utilisent des tables en cascade, et des opérateurs hardware dédiés. Les algorithmes de Wong et Goto en sont un exemple.

5.1. **Méthode de Tang.** La méthode de Tang comprend 3 étapes :

(1) *réduction* : à partir du point initial $x$, après une éventuelle réduction d'argument (cf. section 8), on déduit un point $y$ tel que $f(x)$ se déduit facilement de $f(y)$, ou de $g(y)$ pour une autre fonction $g$ ;

(2) *approximation* : $f(y)$, ou $g(y)$, est approché par un polynôme de petit degré ;

(3) *reconstruction* : $f(x)$ est déduit de $f(y)$, ou de $g(y)$.

Voici quelques exemples.

5.1.1. *La fonction exponentielle.* Réduction : Tang suggère de mettre l'argument $x$ sous la forme

$$x = (32m+j)\frac{\log 2}{32} + r,$$

avec $|r| \leqslant \frac{\log 2}{64}$ et $0 \leqslant j < 32$ ; puis d'approcher $\exp(r)-1$ par un polynôme $p(r)$, et enfin de reconstruire $\exp(x)$ par la formule

$$\exp(x) = 2^m 2^{j/32}(1+p(r)).$$

L'argument réduit $r$ est mis sous la forme $r = r_1 + r_2$, où $r_1$ et $r_2$ sont deux nombres flottants double précision tels que $|r_2| \leqslant \frac{1}{2}\mathrm{ulp}(r_1)$, de telle sorte que $r_1 + r_2$ approche $x - (32m + j)\frac{\log 2}{32}$ en double-double précision (cf. section 10.6). Pour ce faire, Tang détermine l'entier $N$ le plus proche de $x\Lambda$, où $\Lambda$ est $\frac{32}{\log 2}$ arrondi au plus proche. On a alors $j = N \bmod 32$ et $m = (N - j)/32$. Les valeurs $r_1$ et $r_2$ sont obtenues par

$$r_1 = x - NL_{\mathrm{hi}}, \quad r_2 = -NL_{\mathrm{lo}},$$

où $L_{\mathrm{hi}} + L_{\mathrm{lo}}$ approche $\frac{\log 2}{32}$, $L_{\mathrm{hi}}$ ayant quelques bits de poids faible nuls, de telle sorte que $NL_{\mathrm{hi}}$ est exact, et par conséquent $x - NL_{\mathrm{hi}}$ l'est aussi.[1]

Approximation : $p(r)$ est calculé de la façon suivante. On calcule d'abord

$$Q = r^2(a_1 + r(a_2 + r(a_3 + r(a_4 + ra_5)))),$$

où les $a_i$ sont les coefficients d'une approximation minimax, puis on en déduit

$$p(r) = r_1 + (r_2 + Q).$$

Reconstruction : les valeurs $s^j = 2^{j/32}$ sont tabulées pour $0 \leqslant j < 32$ en double-double précision : $s^j \approx s^j_{\mathrm{hi}} + s^j_{\mathrm{lo}}$, de sorte que les 6 bits de poids faible de $s^j_{\mathrm{hi}}$ sont nuls ; $s^j_{\mathrm{hi}} + s^j_{\mathrm{lo}}$ est donc une approximation à environ 100 bits de $s^j$. On calcule :

$$\exp x \approx 2^m(s^j_{\mathrm{hi}} + (s^j_{\mathrm{lo}} + (s^j_{\mathrm{hi}}p(r) + s^j_{\mathrm{lo}}p(r)))).$$

5.2. **Méthode de Gal.** Cette méthode utilise des points *presque* régulièrement espacés, au contraire de celle de Tang. Soit à approcher par exemple la fonction $\exp x$ sur $[1/2, 1]$, avec des approximations de 4 chiffres décimaux, et 5 intervalles.

La méthode classique consiste à découper $[1/2, 1]$ en 5 intervalles de même largeur, et à prendre les approximations de $\exp x$ au milieu de ces intervalles : $\exp 0.55 \approx \mathbf{1.733}253$, $\exp 0.65 \approx \mathbf{1.915}541$, $\exp 0.75 \approx \mathbf{2.117}000$, $\exp 0.85 \approx \mathbf{2.339}647$, $\exp 0.95 \approx \mathbf{2.585}710$. L'erreur maximale est de 0.46 ulp (pour 0.65) si on arrondit au plus proche.

La méthode de Gal consiste à chercher des nombres représentables sur 4 chiffres décimaux, dont l'exponentielle est proche d'un nombre décimal de 4 chiffres. On peut trouver de tels nombres par recherche exhaustive :

```
l:=NULL;
for x from 5000 to 10^4 do
   a:=evalf(exp(x/10^4), 20); b:=evalf(a, 4);
   if abs(a-b)<2e-6 then l := l, [x/1e4, b] fi
od: l;
[0.5038, 1.655], [0.5487, 1.731], [0.5579, 1.747], [0.5619, 1.754],

    [0.5738, 1.775], [0.6065, 1.834], [0.6195, 1.858], [0.6408, 1.898],
```

[1]Comme $\exp x$ provoque un overflow pour $x > \log(2^{1024})$, on a $N \leqslant (1024 \log 2)\frac{32}{\log 2} = 2^{15}$, donc on peut stocker dans $L_{\mathrm{hi}}$ les $53 - 15 = 38$ bits de poids fort de $\frac{\log 2}{32}$, et les 53 bits suivants dans $L_{\mathrm{lo}}$, soit une approximation à 91 bits (voire 92 en arrondissant $L_{\mathrm{hi}}$ au plus proche).

```
[0.7500, 2.117], [0.7807, 2.183], [0.7948, 2.214], [0.8211, 2.273],

[0.8612, 2.366], [0.8671, 2.380], [0.8738, 2.396], [0.8817, 2.415],

[0.9582, 2.607], [0.9647, 2.624], [0.9704, 2.639]
```

Si l'on prend comme points 0.5487, 0.6408, 0.7500, 0.8612, 0.9582, l'erreur maximale diminue à $1.8 \cdot 10^{-6}$, soit 0.0018 ulp. On a gagné un facteur d'environ 250 sur l'erreur maximale.

Par exemple, pour évaluer $\sin x$ pour $|x| \leqslant \pi/4$, Gal et Bachelis proposent l'algorithme suivant : pour $x \leqslant 83/512$, ils utilisent une approximation polynomiale à l'aide de tables classiques ; sinon, ils utilisent des points $X_i$ proches de $i/256$, pour $i \leqslant 201$, tels que $\sin X_i$ et $\cos X_i$ contiennent simultanément 11 zéros après le 53e bit. Si $x \approx X_i + z$, on obtient une approximation de $\sin x$ via :

$$\sin(x) \approx \sin X_i \cos z + \cos X_i \sin z,$$

où $\sin z$ et $\cos z$ sont approchés par des approximations polynomiales de petit degré (4 ou 5). Ils obtiennent ainsi un arrondi exact dans 99.9% des cas.

Cette méthode a cependant des limitations. Si l'on veut trouver un point $X_i$ où $k$ fonctions sont simultanément à distance $< 2^{-p}$ulp d'un nombre machine, il faut essayer de l'ordre de $2^{kp}$ points. De plus si l'on veut les $X_i$ quasi-régulièrement espacés, il faut que $2^{kp}$ ne soit pas trop grand, ce qui limite la précision $p$ que l'on peut atteindre.

5.3. **Méthode de Wong et Goto.** Les méthodes vues précédemment s'implantent aussi bien en software qu'en hardware. La méthode de Wong et Goto, quant à elle, utilise spécifiquement le hardware. En l'occurrence, ils se servent d'un multiplieur rectangulaire $16 \times 56$, qui tronque le résultat à 56 bits, en un peu plus de la moitié du temps qu'il faut pour une multiplication $53 \times 53$.

Pour approcher $\log x$, on écrit

$$x = m2^E,$$

avec $E \neq -1$, $1 \leqslant m < 2$ pour $E \neq 0$, et $1/2 \leqslant m < 2$ pour $E = 0$.[2] On calcule $\log m$, auquel on ajoute $E \log 2$.

Pour approcher $\log m$, l'idée de base consiste à trouver un petit entier $K_1$ tel que $K_1 m \approx 1$, la multiplication $K_1 \times m$ pouvant se faire via un multiplieur rectangulaire :

$$\log m = \log(K_1 m) - \log K_1,$$

où $\log K_1$ est tabulé. On continue, en cherchant un petit entier $K_2$ tel que $K_1 K_2 m$ est encore plus proche de 1 :

$$\log m = \log(K_1 K_2 m) - \log K_1 - \log K_2.$$

On s'arrête quand $K_1 ... K_k m$ est si proche de 1 qu'un développement de Taylor de petit degré (ici 3) suffit.

[2] On évite ainsi une possible cancellation pour $E = -1$ et $m$ proche de 2.

Pour $\log x$, l'algorithme proposé par Wong et Goto utilise trois multiplications rectangulaires par $K_1$, $K_2$, $K_3$ (après recherche dans une table pour chaque), une multiplication double précision pour le terme d'ordre 2 de $\log(K_1K_2K_3m)$, et une recherche en table pour celui d'ordre 3. Ils obtiennent ainsi une erreur d'au plus 0.5 ulp sur le format interne (56 bits), soit au plus 1 ulp après l'arrondi final en 53 bits.

5.4. **Tables bipartites et multipartites.** Les tables bipartites et multipartites sont utilisées pour stocker les approximations d'une fonction $f$ avec $n$ bits en entrée et en sortie. Elles sont utilisées en pratique pour stocker par exemple une approximation initiale de $f(x)$ pour l'itération de Newton (par exemple, $1/x$, $\sqrt{x}$, ...). Comme il y a $2^n$ sorties possibles sur $n$ bits, une table classique aurait une taille de l'ordre de $n2^n$ bits.

L'idée des tables bipartites est de découper les bits de l'argument $x$ en trois parties à peu près égales $x_0$, $x_1$, $x_2$ de $n/3$ bits chacune, et d'utiliser deux tables, l'une indexée par $x_0$ et $x_1$, l'autre par $x_0$ et $x_2$. On obtient alors l'approximation voulue par

$$t_1[x_0, x_1] + t_2[x_0, x_2].$$

Ainsi, au lieu d'avoir une seule table de taille $n2^n$, on a deux tables, chacune ayant une taille de l'ordre de $n2^{2n/3}$ bits.

Si on écrit $x = x_0 + x_1 + x_2$, la formule de MacLaurin à l'ordre 2 donne :

$$f(x) = f(x_0 + x_1) + x_2 f'(x_0 + x_1) + x_2^2 f''(\theta), \tag{2}$$

pour $\theta \in [x_0 + x_1, x]$. Une table bipartite revient à une approximation :

$$f(x) \approx f(x_0 + x_1) + x_2 f'(x_0),$$

et l'erreur est de l'ordre de $2^{-n}$. En effet, le développement (2) donne $f(x) = f(x_0 + x_1) + x_2 f'(x_0 + x_1) + O(x_2^2)$ ; or de même $f'(x_0 + x_1) = f'(x_0) + O(x_1)$, d'où $f(x) = f(x_0 + x_1) + x_2 f'(x_0) + O(x_1 x_2 + x_2^2)$.

Les tables multipartites sont une généralisation des tables bipartites : au lieu de découper en 3 parties, avec 2 tables, on découpe en $k+1$ parties, avec $k$ tables, et une taille totale de l'ordre de $kn2^{2n/(k+1)}$ bits. On peut aussi avoir des découpages différents pour les tables $t_1$ et $t_2$.

## 6. Méthodes à base d'additions et de décalages

6.1. **Une petite histoire.** Au début du 17e siècle, il y eut une terrible pénurie de nourriture. Le roi décida alors d'imposer une taxe sur la consommation de pain, proportionnelle à l'exponentielle du poids du pain ! Comment calculer rapidement cette taxe ? Un vieux mathématicien, Briggs, trouva une solution convenable. Il dit au roi : « Pour calculer la taxe, j'ai besoin d'une balance à plateaux, des poids et d'une lime. » Surpris, le roi fit apporter le matériel à Briggs.

Briggs commença par limer les poids, puis demanda au roi un morceau de pain. Il pesa le pain et trouva un poids *apparent* (les poids ayant été limés) égal à 0,572 kg. Puis il dit :

« J'écris 0,572 ; je remplace le 0 par un 1 ; ceci donne 1,572. Maintenant je calcule le produit des deux premières décimales ($5 \times 7$), que je divise par 1000 ; ceci donne 0,035. Je calcule le produit de la première et de la troisième décimale ($5 \times 2$), que je divise par 10 000 ; ceci donne 0,001. J'ajoute 0,035 et 0,001 à 1,572, et j'obtiens l'exponentielle du poids : 1,608. »

Le roi était plutôt sceptique et demanda à ses serviteurs de peser le pain (0,475 kg) et aux mathématiciens de calculer l'exponentielle. Après de longs calculs, il trouvèrent le résultat : 1,608014... L'estimation de Briggs n'était pas si mauvaise !

Expliquons maintenant la méthode de Briggs. Les poids étaient limés de façon à ce qu'un poids de $x$ kg pesait finalement $\log(1+x)$ kg. Par conséquent, si le poids apparent était de $0{,}x_1x_2x_3$ kg, le poids réel était, en kg, de

$$\log\left(1+\frac{x_1}{10}\right)+\log\left(1+\frac{x_2}{100}\right)+\log\left(1+\frac{x_3}{1000}\right),$$

dont l'exponentielle est :

$$\left(1+\frac{x_1}{10}\right)\left(1+\frac{x_2}{100}\right)\left(1+\frac{x_3}{1000}\right)\approx 1+\frac{x_1}{10}+\frac{x_2}{100}+\frac{x_3}{1000}+\frac{x_1x_2}{1000}+\frac{x_1x_3}{10\,000}.$$

Bien que cette histoire soit de la pure fiction (elle a été inventée par Xavier Merrheim pour sa soutenance de thèse), Henry Briggs (1561-1631) a réellement existé et a conçu les premiers algorithmes pratiques pour calculer les logarithmes ; il a publié des tables de logarithmes avec 15 chiffres de précision.

6.2. **La forme générale de ces algorithmes.** On se restreint à des opérations très simples : ces algorithmes sont destinés à être implémentés en hardware, en utilisant très peu de ressources. En particulier : additions, soustractions, décalages (i.e. multiplications par une puissance de la base), multiplications par un seul chiffre (triviales si la base est 2), lectures dans de très petites tables.

Dans la suite, nous supposons que la base est 2.

6.3. **Un premier algorithme (calcul de l'exponentielle).** Exposons-en d'abord le principe. L'algorithme se basera sur la propriété suivante : $e^{u+v}=e^u\,e^v$ (et pourra être généralisé pour calculer $a^x$). Algorithme itératif : calculer deux suites $(x_k)$ et $(y_k)$ telles que $y_k=e^{x_k}$ pour tout $k$ et la suite $(x_k)$ converge vers $x$, de façon à ce que la suite $(y_k)$ converge vers $y=e^x$. Plus précisément :

$$\left\{\begin{array}{lcl} x_{k+1} & = & x_k \text{ ou } x_k+t_k \\ y_{k+1} & = & y_k \text{ ou } y_k\,e^{t_k} \end{array}\right.$$

où les $t_k$ et $e^{t_k}$ sont fixés et tabulés.

Rappelons que le but est de n'utiliser que des opérations très simples. Ici, le seul problème concerne les multiplications. Mais nous pouvons choisir les $t_k$ de manière à ce que ces multiplications soient très simples ! Si nous choisissons $e^{t_k}=1+2^{-k}$, une multiplication correspondra (après développement) à un décalage et une addition.

Algorithme (très proche de celui de Briggs) : calculer les suites suivantes définies par :

$$\begin{array}{lll} x_0 & = & 0 \\ y_0 & = & 1 \\ x_{k+1} & = & x_k + \log(1 + d_k\, 2^{-k}) = x_k + d_k\, \log(1 + 2^{-k}) \\ y_{k+1} & = & y_k(1 + d_k\, 2^{-k}) = y_k + d_k\, y_k\, 2^{-k} \\ d_k & = & \left\{ \begin{array}{ll} 1 & \text{si } x_k + \log(1 + 2^{-k}) \leqslant x \\ 0 & \text{sinon.} \end{array} \right. \end{array}$$

Si $(x_k)$ converge vers $x$, alors $(y_k)$ converge vers $y = e^x$. Mais quelles sont les valeurs de $x$ pour lesquelles la suite $(x_k)$ converge vers $x$ ?

**Théorème 3** (algorithme de décomposition restaurant)**.** *Soit $(w_k)$ une suite de réels positifs décroissante telle que la série $\sum_{k=0}^{\infty} w_k$ converge vers un réel $w$ et pour tout $n$,*

$$w_n \leqslant \sum_{k=n+1}^{\infty} w_k$$

*(une telle suite est appelée base discrète). Alors pour tout $x \in [0, w]$, les suites $(x_k)$ et $(d_k)$ définies par*

$$\begin{array}{lll} x_0 & = & 0 \\ x_{k+1} & = & x_k + d_k\, w_k \\ d_k & = & \left\{ \begin{array}{ll} 1 & si\ x_k + w_k \leqslant x \\ 0 & sinon \end{array} \right. \end{array}$$

*vérifient*

$$x = \sum_{k=0}^{\infty} d_k\, w_k = \lim_{n \to \infty} x_k.$$

Preuve : montrer par récurrence sur $n$ que

$$0 \leqslant x - x_n \leqslant \sum_{k=n}^{\infty} w_k$$

en distinguant les cas $d_n = 0$ et $d_n = 1$.

Conséquence : en admettant que la suite des $\log(1 + 2^{-k})$ soit une base discrète, l'algorithme calculant l'exponentielle est applicable pour tout $x$ compris entre 0 et $\sum_{k=0}^{\infty} \log(1 + 2^{-k}) \approx 1{,}562$.

En pratique, les calculs ne sont pas faits exactement ; d'autre part, on doit s'arrêter à une certaine itération et renvoyer une valeur $y_n$. Il y aura donc deux types d'erreur : des erreurs d'arrondi, qui peuvent être évaluées de manière classique (il faudrait également tenir compte du fait qu'une mauvaise valeur de $d_k$ peut être choisie), et l'erreur due au reste. Cette erreur correspond à une erreur sur $x$ inférieure à :

$$\sum_{k=n}^{\infty} \log(1 + 2^{-k}) \leqslant \sum_{k=n}^{\infty} 2^{-k} = 2^{-n+1},$$

qui donne l'erreur relative sur $e^x$ :

$$\left|\frac{e^x - y_n}{e^x}\right| = 1 - e^{x_n - x} \leqslant 1 - e^{-2^{-n+1}} \leqslant 2^{-n+1},$$

i.e. on obtient environ $n-1$ bits significatifs, sans tenir compte des erreurs d'arrondi.

6.4. **Calcul du logarithme.** Il suffit de reprendre le même algorithme, mais en changeant la condition sur $x$ (dont la valeur est maintenant inconnue) en une condition sur $y$ (dont la valeur est maintenant connue) :

$$d_k = \begin{cases} 1 & \text{si } y_k(1+2^{-k}) \leqslant y \\ 0 & \text{sinon.} \end{cases}$$

Cet algorithme permet alors de calculer $x = \log y$ pour tout $y$ compris entre 1 et $\prod_{k=0}^{\infty}(1 + 2^{-k}) \approx 4{,}768$.

6.5. **CORDIC.** Algorithme introduit par Volder en 1959 pour calculer des sinus, cosinus et arctangente (ainsi que des produits et des quotients), et généralisé par Walther en 1971 pour calculer des logarithmes, exponentielles et racines carrées. Cet algorithme a été implémenté dans de nombreuses calculatrices, pour sa simplicité.

L'idée de départ, pour calculer un sinus et un cosinus (coordonnées d'un point sur un cercle unité), est de se rapprocher du point voulu en effectuant des rotations successives :

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} \cos(d_k\, w_k) & -\sin(d_k\, w_k) \\ \sin(d_k\, w_k) & \cos(d_k\, w_k) \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix}$$

qui peut se simplifier en :

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \cos(d_k\, w_k) \begin{pmatrix} 1 & -\tan(d_k\, w_k) \\ \tan(d_k\, w_k) & 1 \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix}.$$

Problème : on ne peut plus choisir $d_k = 0$ ou 1 à cause du facteur $\cos(d_k\, w_k)$. Mais si on se restreint à $d_k$ valant $-1$ ou $+1$ (algorithme de décomposition *non* restaurant), le problème est quasiment résolu : $\cos(d_k\, w_k) = \cos(w_k)$, et on peut mettre en facteur le produit de ces $\cos(w_k)$, qui est une constante (i.e. qui ne dépend pas des valeurs des $d_k$, et donc de la valeur d'entrée).

De manière à simplifier le produit par les $\tan(w_k)$, nous choisissons $w_k = \arctan 2^{-k}$, ce qui donne comme itération (similitude et non plus une rotation) :

$$\begin{pmatrix} x_{k+1} \\ y_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & -d_k\, 2^{-k} \\ d_k\, 2^{-k} & 1 \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix}.$$

Choix de $d_k$ pour effectuer une rotation d'angle $\theta$ :

$$\begin{aligned} z_0 &= \theta \\ z_{k+1} &= z_k - d_k\, w_k \\ d_k &= \begin{cases} 1 & \text{si } z_k \geqslant 0 \\ -1 & \text{sinon.} \end{cases} \end{aligned}$$

En résumé, l'algorithme CORDIC est basé sur l'itération

$$\begin{cases} x_{k+1} &= x_k - d_k\, y_k\, 2^{-k} \\ y_{k+1} &= y_k + d_k\, x_k\, 2^{-k} \\ z_{k+1} &= z_k - d_k \arctan 2^{-k} \end{cases}$$

où les termes $\arctan 2^{-k}$ sont précalculés et tabulés, et où les $d_k$ sont égaux à $+1$ ou $-1$. Le rapport $K$ de la similitude est égal à :

$$K = \prod_{k=0}^{\infty} \frac{1}{\cos(\arctan 2^{-k})} = \prod_{k=0}^{\infty} \sqrt{1 + 2^{-2k}} = 1{,}646760258...$$

Comme avec une itération, on pouvait calculer soit une exponentielle, soit un logarithme, on a ici aussi deux modes possibles.

6.5.1. *Rotation mode.* Pour un angle $\theta$ tel que $|\theta| \leqslant \sum_{k=0}^{\infty} \arctan 2^{-k} = 1{,}743...$, si nous prenons

$$\begin{cases} x_0 &= 1/K &= 0{,}607252935... \\ y_0 &= 0 \\ z_0 &= \theta \end{cases}$$

les suites $(x_k)$ et $(y_k)$ convergent respectivement vers $\cos\theta$ et $\sin\theta$.

6.5.2. *Vectoring mode.* Ce mode est utilisé pour calculer des arctangentes, plus précisément $\theta = \arctan(y_0/x_0)$. Si on part de $(x_0, y_0)$ quelconque et que l'on fait la similitude d'angle $z_0 = -\theta$ (i.e. l'opération inverse, en quelque sorte) et de rapport $K$ en itérant, on obtient :

$$\begin{cases} x_k &\to K\sqrt{x_0^2 + y_0^2} \\ y_k &\to 0 \\ z_k &\to 0. \end{cases}$$

Cependant $\theta$ n'est pas connu initialement. Notons que $z_k$ n'est utilisé que pour déterminer $d_k$. Mais comme le but est de ramener le point sur l'axe des $x$ par similitudes successives, nous pouvons donc remplacer la condition sur $z_k$ par : $y_k \leqslant 0$, (i.e. $d_k = 1$ si $y_k \leqslant 0$, $-1$ sinon). Nous obtenons alors, en partant de $z_0 = 0$ :

$$\begin{cases} x_k &\to K\sqrt{x_0^2 + y_0^2} \\ y_k &\to 0 \\ z_k &\to \arctan(y_0/x_0). \end{cases}$$

6.5.3. *Généralisation par Walther.* Puisque les fonctions trigonométriques et les fonctions hyperboliques sont assez semblables, il n'est pas étonnant de pouvoir modifier cet algorithme pour calculer les fonctions hyperboliques. C'est ce qu'a fait John Walther en 1971 :

$$\begin{cases} x_{k+1} &= x_k - m\, d_k\, y_k\, 2^{-\sigma(k)} \\ y_{k+1} &= y_k + d_k\, x_k\, 2^{-\sigma(k)} \\ z_{k+1} &= z_k - d_k\, w_{\sigma(k)} \end{cases}$$

avec :

– Circulaire : $m = 1$, $w_k = \arctan 2^{-k}$, $\sigma(k) = k$.

| Rotation mode | Vectoring mode |
|---|---|
| $x_k \rightarrow K(x_0 \cos z_0 - y_0 \sin z_0)$ | $x_k \rightarrow K\sqrt{x_0^2 + y_0^2}$ |
| $y_k \rightarrow K(y_0 \cos z_0 + x_0 \sin z_0)$ | $y_k \rightarrow 0$ |
| $z_k \rightarrow 0$ | $z_k \rightarrow z_0 + \arctan(y_0/x_0)$ |

– Linéaire : $m = 0$, $w_k = 2^{-k}$, $\sigma(k) = k$.

| Rotation mode | Vectoring mode |
|---|---|
| $x_k \rightarrow x_0$ | $x_k \rightarrow x_0$ |
| $y_k \rightarrow y_0 + x_0 z_0$ | $y_k \rightarrow 0$ |
| $z_k \rightarrow 0$ | $z_k \rightarrow z_0 - y_0/x_0$ |

– Hyperbolique : $m = -1$, $w_k = \tanh^{-1} 2^{-k}$, $\sigma(k) = k - \ell$, où $\ell$ est le plus grand entier tel que $3^{\ell+1} + 2\ell - 1 \leqslant 2k$.

| Rotation mode | Vectoring mode |
|---|---|
| $x_k \rightarrow K'(x_1 \cosh z_1 + y_1 \sinh z_1)$ | $x_k \rightarrow K'\sqrt{x_1^2 - y_1^2}$ |
| $y_k \rightarrow K'(y_1 \cosh z_1 + x_1 \sinh z_1)$ | $y_k \rightarrow 0$ |
| $z_k \rightarrow 0$ | $z_k \rightarrow z_1 - \tanh^{-1}(y_1/x_1)$ |

où $K' = \prod_{k=1}^{\infty} \sqrt{1 - 2^{-2\sigma(k)}} = 0{,}828159...$

À partir de ces fonctions de base, on peut calculer :

$$e^x = \cosh x + \sinh x$$

$$\log x = 2 \tanh^{-1}\left(\frac{x-1}{x+1}\right)$$

$$\sqrt{x} = K'\sqrt{\left(x + \frac{1}{4K'^2}\right)^2 - \left(x - \frac{1}{4K'^2}\right)^2}.$$

## 7. Algorithmes à convergence quadratique (Newton, Goldschmidt)

7.1. **Itération de Newton.** Soit $f \in \mathcal{C}^1$, ayant une racine simple $\alpha$ non nulle. Considérons l'itération $x_{n+1} = x_n - f(x_n)/f'(x_n)$. Si $x_0$ est suffisamment proche de $\alpha$, alors $(x_n)$ converge vers $\alpha$, et la convergence est quadratique (la précision double à chaque itération).

7.1.1. *Application à la division.* Pour calculer $a/b$, il suffit de calculer $1/b$ par la méthode de Newton et de multiplier le résultat par $a$. On pose $f(x) = 1/x - b$. L'itération de Newton s'écrit : $x_{n+1} = x_n(2 - bx_n)$, qui converge si et seulement si $x_0 \in ]0, 2/b[$. Une valeur approchée de $1/b$ pour $x_0$ peut être obtenue en lisant dans une table à partir des premiers bits de la mantisse de $b$.

7.1.2. *Choix des valeurs initiales pour l'itération de Newton.* Supposons qu'on veuille approcher une fonction $f(x)$ pour $x \in [a, b]$ par un nombre fixé à l'avance d'itérations de Newton. Quel est le meilleur point de départ, si l'on veut minimiser l'erreur maximale sur $[a, b]$ ? Peter Kornerup et Jean-Michel Muller [25] ont montré que pour l'inverse, la racine carrée ou la racine carrée inverse, on pouvait gagner jusqu'à un facteur 400 avec un choix judicieux de valeur initiale.

Par exemple, pour l'inverse, la meilleure valeur initiale $\beta_i$ pour $i$ itérations est $\beta_0 = \frac{1}{2}(\frac{1}{a} + \frac{1}{b})$, $\beta_1 = (ab)^{-1/2}$, $\beta_\infty = \frac{2}{a+b}$.

Pour la racine carrée — avec l'itération directe $x_{n+1} = \frac{1}{2}(x_n + y/x_n)$ — on trouve $\beta_0 = \frac{1}{2}(\sqrt{a} + \sqrt{b})$, et $\beta_\infty \approx \frac{a^{1/4}+b^{1/4}}{a^{-1/4}+b^{-1/4}}$.

7.2. **Algorithme de Goldschmidt.** Cet algorithme est implanté dans l'IBM 360.

On cherche à calculer $a/b$, avec $1/2 \leqslant b < 1$. But : chercher une suite $(r_n)$ telle que $b\, r_0\, r_1 \ldots r_n$ converge vers 1 le plus rapidement possible. Ainsi, $a\, r_0\, r_1 \ldots r_n$ convergera vers $a/b$ à la même vitesse.

Le choix des $r_n$ se fait à partir de l'identité : $(1 - x)(1 + x) = 1 - x^2$. Soit $\varepsilon = 1 - b$. Ainsi, on part de $b = 1 - \varepsilon$. On va alors prendre $r_0 = 1 + \varepsilon$ pour obtenir $b\, r_0 = 1 - \varepsilon^2$, puis $r_1 = 1 + \varepsilon^2$ pour obtenir $b\, r_0\, r_1 = 1 - \varepsilon^4$, et ainsi de suite, $r_n = 1 + \varepsilon^{2^n}$ pour obtenir $b\, r_0\, r_1 \ldots r_n = 1 - \varepsilon^{2^{n+1}}$. On calcule donc :

$$\begin{cases} q_0 & = & a \\ \varepsilon_0 & = & 1 - b \\ q_{n+1} & = & q_n(1 + \varepsilon_n) \\ \varepsilon_{n+1} & = & \varepsilon_n^2. \end{cases}$$

Note : si dans l'itération de Newton, on pose $\varepsilon_n = 1 - bx_n$, alors on a $x_{n+1} = x_n(1 + \varepsilon_n)$ et $\varepsilon_{n+1} = 1 - bx_{n+1} = 1 - bx_n(2 - bx_n) = \varepsilon_n^2$.

Remarque : l'algorithme de Goldschmidt converge de façon quadratique comme l'itération de Newton : chaque étape double le nombre de bits corrects. Par contre l'algorithme de Goldschmidt n'est pas *auto-correcteur* comme l'itération de Newton, c'est-à-dire que pour obtenir $n$ bits corrects, il faut effectuer *toutes* les itérations avec $n$ bits. Cela provient du fait que l'entrée $b$ n'est utilisée qu'au départ dans le schéma de Goldschmidt, alors qu'elle est réintroduite à chaque itération de Newton. Pour des calculs de complexité polynomiale, la complexité asymptotique de l'algorithme de Goldschmidt est donc plus élevée d'un facteur $\log n$ que celle de l'itération de Newton.

## 8. Réduction d'argument

Les algorithmes vus jusqu'à présent supposent que l'on cherche à évaluer une fonction $f$ sur un « petit » intervalle. Quand l'argument $x$ est dans un « grand » intervalle — comme $[-1.798...10^{308}, 1.798...10^{308}]$ pour les flottants double précision — il faut trouver une transformation simple qui permette de déduire $f(x)$ d'une valeur $g(x^*)$, où $x^*$ est lui-même dans un petit intervalle (on peut avoir $g = f$) ; $x^*$ est appelé l'*argument réduit*. L'étape de *reconstruction* consiste ensuite à retrouver $f(x)$ en fonction de $g(x^*)$.

On distingue deux grandes classes de réduction d'argument ($k$ désigne un entier et $C$ une constante) :

– la réduction *additive*, où $x^* = x - kC$, par exemple $C = \pi/4$ pour les fonctions trigonométriques, $C = \log 2$ pour la fonction exponentielle, $C = 1$ pour la fonction $2^x$ ;
– la réduction *multiplicative*, où $x^* = x/C^k$, par exemple $C = 2$ ou $C = e$ pour le logarithme.

Exemple 1. [Réduction additive] On suppose qu'on sait calculer le sinus et le cosinus pour $x \in [-\pi/4, \pi/4]$. Pour calculer $\cos x$ pour $x$ quelconque, on commence par déterminer $k$ tel que $x^* = x - k\pi/2 \in [-\pi/4, \pi/4]$, puis on a $\cos x = \pm \cos x^*$ ou $\pm \sin x^*$ suivant la valeur de $k \bmod 4$. Ici la reconstruction est triviale, et n'introduit pas d'erreur.

Exemple 2. [Réduction multiplicative] On suppose qu'on sait calculer le logarithme pour $x \in [1, e]$. On commence par déterminer $k$ tel que $x^* = x/e^k \in [1, e]$, on évalue $\log x^*$, et on reconstruit $\log x$ via la formule $\log x = k + \log x^*$. Ici la reconstruction n'est pas triviale, et peut même engendrer un phénomène de cancellation pour $k = -1$ et $x^*$ proche de $e$, c'est-à-dire $x$ proche de 1 (cf. méthode de Wong et Goto, section 5.3).

Pour une fonction donnée, plusieurs méthodes de réduction d'argument peuvent exister. Par exemple, pour le logarithme, on peut aussi utiliser $x^* = x/2^k$, ce qui donne $\log x = \log x^* + k \log 2$. La réduction est plus facile (en base 2) puisqu'on n'a pas besoin de stocker la constante $e$, par contre la reconstruction nécessite la constante $\log 2$.

Il faut aussi garder à l'esprit que l'intervalle réel pour lequel on a besoin de réduire l'argument n'est pas forcément tout l'intervalle des nombres flottants possibles. Par exemple, pour l'exponentielle en double précision, il y a débordement (*overflow*) pour $x > \log(2^{1024}) \approx 709.8$.

En pratique, la réduction multiplicative se produit uniquement pour les fonctions de type logarithme et les racines (carrée et cubique). Dans tous ces cas, on peut prendre pour $C$ une puissance de la base interne ($\beta = 2$ dans notre cas), et par conséquent le calcul $x/C^k$ peut se faire sans erreur. On s'intéresse donc dans la suite uniquement au cas de la réduction additive.

Remarque. En général, l'argument réduit $x^*$ n'est pas représenté dans le même format que l'argument initial $x$. En effet, pour obtenir un arrondi correct sur $f(x)$, en tenant compte des erreurs lors de la réduction et de la reconstruction, il faut calculer $g(x^*)$ avec une plus grande précision. On peut par exemple représenter $x^*$ par deux flottants de même précision que $x$ (expansion flottante) : $x^* = hi + lo$ (voir section 10.6).

Exemple. Un exemple « classique » [30] est le calcul de $\sin 10^{22}$ : $10^{22}$ est la plus grande puissance de 10 exactement représentable en double précision, car $5^{22} < 2^{53} < 5^{23}$. On obtient sur Athlon XP avec gcc 3.2 sous Mandrake 9.0 :

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main (int argc, char *argv[])
{
```

```
  double x = atof (argv[1]);
  printf ("sin(%1.20e)=%1.20e\n", x, sin (x));
  return 0;
}
```

```
% gcc e.c -lm ; ./a.out 10000000000000000000000.0
sin(1.00000000000000000000e+22)=4.62613040764601746169e-01
```

alors que la valeur exacte est :

```
> evalf(sin(10^22), 20);
                          -.85220084976718880177
```

On peut d'ailleurs deviner l'algorithme utilisé ci-dessus. La fonction `sin` de la bibliothèque mathématique (plus précisément, du processeur) a en mémoire une approximation sur 64 bits (format double précision étendue) de $\pi$, et la réduction d'argument se fait par rapport à *cette valeur approchée*, et non par rapport à la valeur réelle de $\pi$ :

```
> fpi:=approx(evalf(Pi), 64, 0);
                                   14488038916154245685
                         fpi := --------------------
                                   4611686018427387904
> x:=10^22: k:=round(x/fpi);
                         k := 3183098861837906715327
> evalf((-1)^k*sin(x-k*fpi));
              .46261304076460176119008297523756774002009543800596
```

Le problème est ici une cancellation lors de la réduction d'argument : $10^{22}$ contient 74 bits devant le point binaire, alors que `x-k*fpi` n'en contient plus aucun.

Exemple. [Cody, 1982] On peut obtenir de pires résultats encore lorsque $x - k\pi$ est très petit. Au phénomène de cancellation s'ajoute le fait qu'on perd des bits significatifs supplémentaires dans la valeur de $x - k\pi$. Il suffit par exemple de prendre pour $x$ le numérateur d'une réduite $p/q$ du développement en fraction continue de $\pi$. On aura alors $k = q$, et on sait que $|p/q - \pi|$ est de l'ordre de $1/q^2$. Par conséquent $|p - q\pi|$ est de l'ordre de $1/q$, et donc le nombre de bits perdus est environ $\log_2(pq)$.

Le tableau ci-dessous indique pour différentes valeurs de $x$, l'erreur en ulps effectuée sur le calcul de $\sin x$ sur un Athlon XP, avec gcc 3.2, sous Mandrake 9.0.

| $x$ | 355 | 103993 | 104348 | 208341 | 312689 | 833719 | 1146408 |
|---|---|---|---|---|---|---|---|
| erreur (ulps) | 135 | 39516 | 79300 | 158330 | 950519 | 2534357 | 13939505 |

Le pire cas est obtenu pour la 29$^{e}$ réduite du développement en fraction continue de $\pi$, dont le numérateur est exactement représentable en double précision :

```
> Digits:=40: convert(evalf(Pi), confrac, 'l'): l[29];
                                6134899525417045
                                ----------------
                                1952799169684491
```

et on obtient avec le même programme C que ci-dessus :

```
% ./a.out 6134899525417045.0
sin(6.13489952541704500000e+15)=-7.89815970288233026159e-06
```

alors que le résultat exact est $\approx 0.950 \cdot 10^{-16}$ !

Nous allons voir une première méthode de réduction additive due à Cody et Waite, peu coûteuse mais limitée à de petits arguments, puis une méthode plus générale due à Payne et Hanek.

8.1. **Méthode de Cody et Waite.** La méthode de Cody et Waite pour estimer $x - kC$ est la suivante :

(1) on décompose tout d'abord $C$ en $C_1 + C_2$, où $C_2$ est petit devant $C_1$ ;
(2) on calcule $x' = x - kC_1$, qui est exact si le nombre de bits significatifs de $C_1$ plus celui de $k$ est inférieur ou égal à la précision courante ;
(3) puis on calcule l'arrondi de $x' - kC_2$.

En fait, on simule ainsi le calcul de $x - kC$ avec la précision de $C_1 + C_2$.

Cette méthode s'implante très facilement, par exemple en prenant pour $C_1$ l'approximation au plus proche sur 27 bits de $\pi$, et $C_2$ l'approximation au plus proche sur 53 bits de $\pi - C_1$ :

```
double cw_sin (double x)
{
  double C1 = 3.1415926516056060791;
  double C2 = 0.19841871593610808504e-8;
  long k = rint (x / C1);
  double y = x - k * C1;
  y = y - k * C2;
  return (k % 2) ? -sin (y) : sin (y);
}
```

Cela donne les résultats suivants :

| $x$ | 355 | 103993 | 104348 | 208341 | 312689 | 833719 | 1146408 |
|---|---|---|---|---|---|---|---|
| erreur (ulps) | 0 | 1 | 1 | 1 | 8 | 20 | 113 |

8.2. **Calcul de pires cas avec fractions continues.** Soit un nombre flottant $x$ pour lequel on veut effectuer une réduction additive $x^* = x - kC$.

**Théorème 4.** *Soit $p_i/q_i$ une réduite du développement en fraction continue d'un nombre irrationnel $\alpha$, pour $i \geqslant 2$. Alors pour tout rationnel $\frac{p}{q} \neq \frac{p_i}{q_i}$, tel que $\frac{p}{q} \neq \frac{p_{i+1}}{q_{i+1}}$, et $q \neq q_{i+1}$, on a :*

$$|\alpha - \frac{p_i}{q_i}| < \frac{q}{q_i}|\alpha - \frac{p}{q}|.$$

Par conséquent, $p_i/q_i$ est la meilleure approximation de $\alpha$, parmi toutes les approximations rationnelles de dénominateur $q \leqslant q_i$.

Mettons $x$ sous la forme $x = M\beta^e$ avec $M$ entier d'au plus $n$ chiffres en base $\beta$ : $|M| < \beta^n$. Un pire cas pour la réduction d'argument est un $x$ tel que $x - kC$ est très petit, à savoir $M\beta^e$ proche de $kC$, ou encore $\frac{\beta^e}{C}$ proche du rationnel $k/M$. Le pire cas est donc obtenu en prenant la dernière réduite de $\frac{\beta^e}{C}$ dont le dénominateur est inférieur à $\beta^n$. Soit $p/q$ cette réduite : on a alors $p/q \approx \frac{\beta^e}{C}$, soit $pC \approx q\beta^e$.

L'algorithme ci-dessous (ici en Maple) détermine le dénominateur $q \leqslant q_{\max}$ donnant la meilleure approximation rationnelle du réel $\alpha$ (il n'est pas nécessaire de calculer le numérateur, qui se retrouve simplement par $p = \lfloor q\alpha \rceil$ avec ici $\alpha = \frac{\beta^e}{C}$) :

```
find_worst_case := proc (alpha, qmax) local a, q, k;
   a := alpha; k := floor (a); q := [0, 1];
   while q[2] <= qmax do
      a := 1 / (a - k); k := floor (a); q := [q[2], k*q[2]+q[1]];
   od;
   q[1]
end:

> Digits:=20: find_worst_case (2^0/Pi, 2^10);
                                      355

> Digits:=20: find_worst_case (2^0/Pi, 2^17);
                                    104348
```

Il est facile d'en déduire un algorithme trouvant le flottant $x = qr^e$ le plus proche d'un multiple $pC$, pour une précision $n$ donnée, et $|x| < r^{e_{\max}}$ (on a noté ici $r = \beta$) :

```
find_all := proc (r, n, C0, emax)
local C, e, alpha, qmax, d, dmin, p, q, qmin;
   C := evalf (C0); qmax := r^n;
   alpha := 1/C/qmax; dmin := infinity;
   for e from -n+1 to emax-n do
      alpha:=alpha*r; q:=find_worst_case (alpha, qmax);
      p:=round(q * alpha); d:=abs(q*r^e - p*C);
      if d < dmin then dmin := d; qmin:=q, e fi
   od;
   evalf(dmin, 3), qmin
end:

> Digits:=60: find_all (2, 24, Pi/2, 127);
                                   -8
                            .161 10   , 16367173, 72

> C:=Pi/2: x:=16367173*2^72: p:=round(x/C): evalf(x-p*C);
                                                        -8
                         .16147697982476211883054 10
```

**Théorème 5.** *Les pires cas pour $|q\alpha - p|$ sont obtenus* exactement *pour les réduites $p/q$ du développement en fractions continues de $\alpha$. Par contre une telle réduite donne un pire cas de $|\alpha - p/q|$, mais pas réciproquement.*

Par exemple, 13/4 est un pire cas pour la distance $\pi - p/q$, mais pas pour $q\pi - p$ :

```
> dmin := infinity:
> for q to 1000 do
     p := round(q*Pi);
```

```
     d := abs(evalf(p-q*Pi));
     if d<dmin then dmin:=d; lprint(p/q, d) fi
  od:
3, .141592654
22/7, .885142e-2
333/106, .88213e-2
355/113, .301e-4

> dmin := infinity:
> for q to 1000 do
     p := round(q*Pi);
     d := abs(evalf(p/q-Pi));
     if d<dmin then dmin:=d; lprint(p/q, d) fi
  od:
3, .141592654
13/4, .108407346
16/5, .58407346e-1
19/6, .25074013e-1
22/7, .1264489e-2
179/57, .1241777e-2
201/64, .967654e-3
223/71, .747584e-3
245/78, .567013e-3
267/85, .416183e-3
289/92, .288306e-3
311/99, .178513e-3
333/106, .83220e-4
355/113, .266e-6
```

8.3. **Méthode de Payne et Hanek.** On suppose toujours qu'on veut effectuer une réduction additive de la forme $x^* = x - kC$. On a donc $k = \lfloor x/C \rceil$, puis $x^* = C(x/C - k)$. Supposons maintenant que l'on dispose d'une fonction $g(y)$ calculant $f(Cy)$. Il suffit donc de déterminer $y^* = \mathrm{frac}(x/C)$, puis d'évaluer $g(y^*)$. A priori si $x$ vaut de l'ordre de $2^e$, $C$ est proche de 1, et que l'on veut $n$ bits significatifs de $y^*$, il faut considérer $e + n$ bits de $x$ et de $1/C$.

L'idée de base est qu'on n'a pas besoin de calculer les bits de poids fort de $x/C$, puisqu'on ne s'intéresse qu'à la partie fractionnaire.

Supposons

$$x = X \cdot 2^e$$

avec $X$ entier de $n$ bits, $2^{n-1} \leqslant X < 2^n$. En supposant $1 \leqslant C < 2$, on a $1/2 < 1/C \leqslant 1$, et on veut les bits d'exposant $-1$ à $-n$ de $x/C$, c'est-à-dire les bits d'exposant $-e-1$ à $-e-n$ de $X/C$.

Décomposons $1/C$ de la façon suivante :

$$1/C = C_0 2^{-e} + C_1 2^{-e-2n} + C_2 2^{-e-2n},$$

avec $C_0$ et $C_1$ entiers, et $|C_2| < 1$. Alors $XC_0 2^{-e}$ est multiple entier de $2^{-e}$, donc n'intervient pas dans $\mathrm{frac}(x/C)$. De même, $XC_2 2^{-e-2n}$ est borné par $2^{-e-n}$, donc n'intervient pas dans les bits d'exposant $-e-1$ à $-e-n$ de $X/C$. Il reste donc à considérer seulement le produit $XC_1$, où $X$ a $n$ bits significatifs, et $C_1$ de l'ordre de $2n$ bits significatifs. La partie intéressante pour les bits d'exposant $-1$ à $-n$ de $x/C$ consiste en les $n$ bits du milieu de ce produit $n \times (2n)$, aussi appelé « produit médian »[20].

## 9. Arrondi final

**9.1. Arrondi d'une valeur (exacte).** Supposons que l'on veuille arrondir une valeur $y$ (non entachée d'erreur), de mantisse $1.b_1b_2b_3...$ (canonique, i.e. où le nombre de bits nuls est infini) dans un système à virgule flottante avec $n$ bits de mantisse. L'arrondi *exact* de $y$ peut se définir comme suit, suivant le mode d'arrondi actif et les valeurs des bits $r = b_n$ (appelé *round bit*, ou *bit d'arrondi*) et $s = b_{n+1} \vee b_{n+2} \vee ...$ (appelé *sticky bit*).

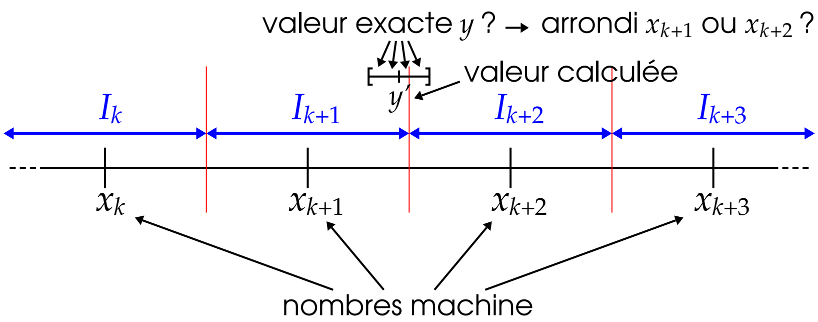
| $r$ / $s$ | vers 0 | vers $\infty$ | au plus près |
|---|---|---|---|
| 0 / 0 | $-$ | $-$ | $-$ |
| 0 / 1 | $-$ | $+$ | $-$ |
| 1 / 0 | $-$ | $+$ | $-$ / $+$ |
| 1 / 1 | $-$ | $+$ | $+$ |

Un $-$ dans le tableau ci-dessus signifie que la mantisse de l'arrondi est $1.b_1b_2...b_{n-1}$, et un $+$ signifie que l'on doit ajouter $2^{1-n}$ à cette mantisse (ce qui peut nécessiter un changement d'exposant, ou même provoquer un *overflow* si le nombre obtenu n'est pas représentable). La case $-$ / $+$ dans le cas $r = 1, s = 0$ pour l'arrondi au plus près correspond à la règle de l'arrondi pair (section 2.2).

**9.2. Le *dilemme du fabricant de tables*.** Considérons maintenant le cas de l'évaluation d'une fonction mathématique $f$ quelconque (exp, log, sin, etc.) en un point $x$ (nombre machine). On cherche à arrondir exactement $y = f(x)$ dans le mode d'arrondi actif.

Si on se fixe une erreur maximale $\varepsilon > 0$, on sait calculer $y$ à $\varepsilon$ près. Notons $y'$ cette valeur approchée de $y$. On sait également arrondir exactement $y'$. Mais obtient-on ainsi toujours le même résultat que si on avait arrondi $y$ ?

Exemple dans le cas de l'arrondi au plus près :



Dans cet exemple, la valeur exacte $y$ peut se situer d'un côté ou de l'autre de la frontière entre les deux arrondis vers $x_{k+1}$ (intervalle $I_{k+1}$) et vers $x_{k+2}$ (intervalle $I_{k+2}$). On ne peut donc pas décider de l'arrondi exact (qui est soit $x_{k+1}$, soit $x_{k+2}$).

Reprenons le cas général... Notons $\varepsilon'$ la distance entre $y'$ et la plus proche frontière entre deux arrondis (ces frontières sont les nombres machine dans le cas d'un arrondi dirigé, et les milieux de deux nombres machine consécutifs dans le cas de l'arrondi au plus près). Si $\varepsilon' \leqslant \varepsilon$, il est impossible de garantir l'arrondi exact. Ce problème est connu sous le nom de *dilemme du fabricant de tables* (en anglais, *Table Maker's Dilemma* – TMD), car il s'est posé à l'origine (en base 10) aux éditeurs de tables de valeurs numériques.

En base 2, le problème s'exprime de la façon suivante : si $y$ est calculée avec une erreur de $2^{-m}$ sur sa mantisse, alors l'arrondi exact ne pourra pas toujours être garanti si la mantisse de $y$ est de la forme :

– en arrondi au plus près,

$$\overbrace{\underbrace{1.xxxxx...xxx}_{n \text{ bits}}100000...0000}^{m \text{ bits}}xxx... \quad \text{ou} \quad \overbrace{\underbrace{1.xxxxx...xxx}_{n \text{ bits}}011111...1111}^{m \text{ bits}}xxx...$$

– avec les autres modes d'arrondi (arrondi *dirigé*),

$$\overbrace{\underbrace{1.xxxxx...xxx}_{n \text{ bits}}000000...0000}^{m \text{ bits}}xxx... \quad \text{ou} \quad \overbrace{\underbrace{1.xxxxx...xxx}_{n \text{ bits}}111111...1111}^{m \text{ bits}}xxx...$$

Attention, il s'agit ici d'une condition sur la valeur exacte $y$ (donc indépendante de l'algorithme de calcul) et non plus sur la valeur approchée $y'$ !

9.3. **Étude probabiliste.** Nous pouvons faire les hypothèses probabilistes suivantes :
– quand $x$ est un nombre machine, les bits de $f(x)$ après la $n$-ième position peuvent être vus comme des *suites aléatoires de* 0 *et de* 1, « 0 » et « 1 » étant équiprobables ;
– ces suites peuvent être considérées « indépendantes » pour deux nombres machine différents.

Note : ces hypothèses probabilistes ne sont pas valables dans certains domaines, par exemple pour $\sin x$ avec $x$ suffisamment proche de 0. Mais ce sont aussi des cas où la fonction peut être calculée simplement et le TMD peut être facilement résolu.

Alors la probabilité pour que $k$ bits donnés soient identiques est de $2^{1-k}$. La probabilité pour que le TMD se produise diminue donc rapidement avec la précision. En particulier, puisqu'il y a de l'ordre de $2^n$ nombres machine (le nombre d'exposants possibles de $x$ étant relativement faible, si on ignore les cas particuliers, comme pour $\exp x$ avec $x$ suffisamment grand, qui donne un overflow), pour un calcul intermédiaire avec un peu plus de $2n$ bits de précision, il y a de grandes chances pour que le TMD ne se produise jamais.

9.4. **La stratégie en peau d'oignon de Ziv.** Évaluer directement $f(x)$ à grande précision (pour éviter que le TMD ne se produise) prendrait beaucoup de temps. La méthode de Ziv permet d'obtenir un temps de calcul moyen raisonnable, sensiblement supérieur à celui nécessaire pour obtenir un résultat à 1 ulp près. Elle consiste à évaluer $f(x)$, à l'aide d'une méthode classique (cf. sections précédentes), avec tout d'abord une précision intermédiaire $m$ un peu plus grande que $n$ (précision cible), et en cas d'échec, prendre une valeur de $m$ plus grande, et ainsi de suite. Par exemple, on peut commencer avec $m = n + 20$ ; la probabilité pour que le TMD se produise et qu'on ne puisse pas garantir l'arrondi exact à cette étape est très faible : environ une chance sur 500 000. Dans ce cas, on peut alors réévaluer $f(x)$ avec $m = n + 40$. La probabilité d'échec est encore très faible : environ une chance sur un million (une chance sur 500 milliards en cumulant). Et ainsi de suite... Cette stratégie est illustrée figure 1.

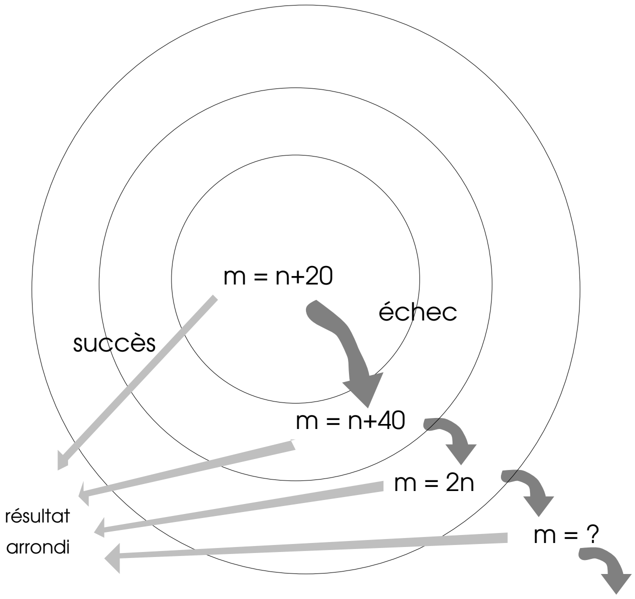



FIG. 1. La stratégie en peau d'oignon de Ziv.

Question : jusqu'à quelle valeur de $m$ peut-on aller ? Les hypothèses probabilistes suggèrent une valeur un peu supérieure à $2n$. Mais peut-on le prouver ?

9.5. **Étude mathématique.** Lindemann a montré en 1882 [4] que l'exponentielle d'un nombre algébrique non nul n'est pas algébrique. Or les nombres machine sont algébriques (car rationnels). Par conséquent, $\exp x$, $\sin x$, $\cos x$, $\arctan x$ pour $x \neq 0$, et $\log x$ pour $x \neq 1$ ne peuvent pas avoir une infinité de 0 ou 1 consécutifs dans leur écriture binaire.

Donc pour tout $x$ (hors cas particulier), il existe une valeur $m_x$ à partir de laquelle le TMD ne peut pas se produire. Comme les nombres machine de format fixé sont en nombre fini, il existe une valeur de $m$ à partir de laquelle pour tout $x$, le TMD ne peut pas se produire.

Nous savons maintenant qu'il existe une borne finie sur $m$, mais nous ne savons toujours pas quelle est cette borne.

Le meilleur résultat actuellement connu en théorie des nombres est un théorème de Nesterenko et Waldschmidt [29], qui donne un minorant de la valeur de $\left|e^{\theta} - \alpha\right| + |\theta - \beta|$, où $\alpha$ et $\beta$ sont des nombres algébriques, et $\theta$ un nombre complexe non nul arbitraire.

Ce théorème permet d'obtenir des bornes sur $m$ pour exp et log, mais aussi pour les fonctions trigonométriques et hyperboliques. Malheureusement, comme on pouvait s'y attendre, ces bornes sont très grandes : de plusieurs millions à plusieurs milliards, suivant la fonction et le domaine considérés. Même si de tels calculs sont faisables sur les machines de bureau d'aujourd'hui, ils prendraient beaucoup de temps et beaucoup de mémoire.

La seule solution actuelle pour obtenir des bornes raisonnables, de l'ordre de celles prévues par les hypothèses probabilistes (en fait, les véritables bornes) est d'effectuer des tests exhaustifs, une fois pour toutes.

### 9.6. **Recherche exhaustive de pires cas.**

9.6.1. *Introduction.* Nous ne sommes pas uniquement intéressés par la plus grande valeur de $m$ pour laquelle le TMD peut se produire. En effet, il est possible que pour certaines fonctions, cette valeur maximale $m_0$ soit très grande à cause d'un certain argument $x_0$, appelé *pire cas*, mais qu'à part cet argument, les valeurs de $m$ pour lesquelles le TMD se produit sont beaucoup plus petites. Dans ce cas, il sera facile de traiter le pire cas lors de la conception de l'algorithme de calcul de la fonction $f$ (avec arrondi exact), et les calculs pourront se faire à une précision plus petite que $m_0$. De même, il est intéressant de trouver tout un ensemble de pires cas (arguments $x$ pour lesquels le TMD se produit, $m$ étant fixé), qui pourront être traités rapidement lors du calcul de $f(x)$, à l'aide de tables construites lors de la conception de l'algorithme.

Connaissant l'ordre de grandeur du nombre de pires cas que l'on souhaite obtenir, la valeur de $m$ peut être fixée à l'aide des hypothèses probabilistes. Par exemple, si on souhaite tester $2^r$ arguments et que l'on veuille de l'ordre de $2^p$ pires cas, on fixera $m = n+2+r-p$.

Supposons que l'on veuille tester tous les arguments ayant un signe et un exposant fixés. Il y a $2^{n-1}$ mantisses possibles, donc $2^{n-1}$ arguments à tester. Si on veut savoir si le TMD se produit pour $m$ bits de précision, il faudra calculer la mantisse du résultat avec une erreur d'au plus $2^{-m}$. Évaluons le temps total des tests à l'aide des algorithmes classiques de calcul de fonctions, en supposant que $n = 53$ (double précision), $m \approx 90$, que les calculs se font sur une machine à 3 GHz et demandent 200 cycles d'horloge par argument. Il faudra environ :

$$\frac{2^{52} \times 200}{3 \times 10^9 \times 86400} \text{ jours} \approx 3475 \text{ jours},$$

soit 9 ans et demi ! Et ceci, à multiplier par le nombre d'exposants (plusieurs dizaines ou centaines) et de fonctions à tester.

La solution : concevoir des algorithmes très rapides en tenant compte des spécificités du problème (en particulier, les nombres machine sont espacés régulièrement). Les tests sont effectués en deux étapes :

(1) Filtre : algorithme très rapide (basse précision) qui sélectionne un ensemble $S$ contenant tous les pires cas, mais bien plus petit que l'ensemble de départ.

(2) On teste ensuite chaque nombre machine de $S$ avec un algorithme plus précis, mais qui peut être beaucoup plus lent.

Le tout peut être parallélisé.

Note : au lieu d'effectuer des tests à la fois sur la fonction $f$ et la réciproque $f^{-1}$, on peut obtenir les résultats sur ces deux fonctions simultanément en n'en testant qu'une seule en précision $n+1$ (et seulement en arrondi dirigé, i.e. le bit d'arrondi doit être le même que les bits suivants) au lieu de $n$ ; on pourrait penser que cela ne sert à rien, puisque le nombre d'arguments à tester double, mais en choisissant la fonction ayant la plus forte pente relative, ou en considérant des algorithmes de test à complexité sous-linéaire, on y gagne.

9.6.2. *Calcul des valeurs successives d'un polynôme.* On peut calculer les valeurs successives d'un polynôme de degré $d$ à l'aide de seulement $d$ additions par valeur (figure 2). Avantages : cette méthode est rapide et on peut faire les calculs modulo $2^{-n}$ (seuls les bits situés après la mantisse nous intéressent).

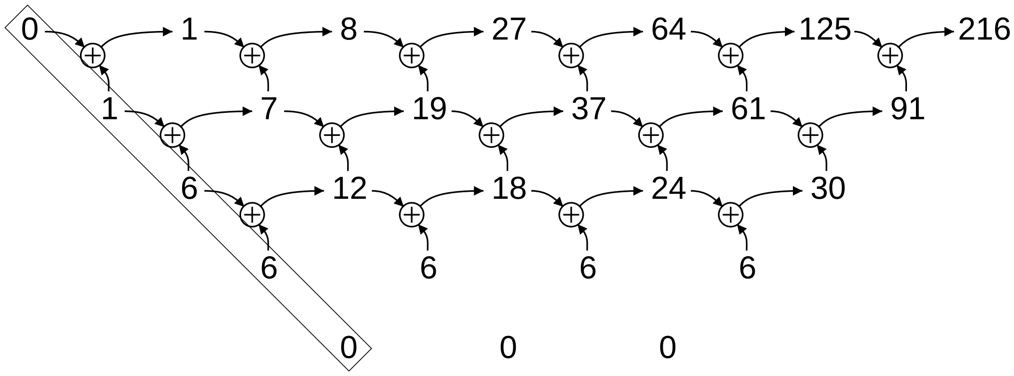



FIG. 2. Table des différences de $P(X) = X^3$, permettant de calculer les valeurs successives $P(0)$, $P(1)$, $P(2)$, $P(3)$, etc.

9.6.3. *Approximations hiérarchiques.* Pour que la méthode décrite précédemment soit rapide, il est préférable que le degré $d$ soit petit (ceci sera aussi nécessaire par la suite, pour d'autres algorithmes, mentionnés dans les sections suivantes) ; mais plus le degré est petit, plus l'intervalle dans lequel l'approximation a une erreur inférieure à une valeur fixée est petit, ce qui signifie qu'il faudra calculer beaucoup d'approximations. Pour gagner du temps sur ces approximations, une solution est de les faire hiérarchiquement. Un exemple, généralisable, est donné figure 3.

9.6.4. *Minoration de la distance d'un segment de droite à $\mathbb{Z}^2$.* On peut tenir compte du fait qu'en général, il n'y a pas de pire cas dans un petit intervalle testé : une minoration de la distance entre les valeurs $f(x)$ et les nombres machine suffit alors. Lorsque le degré $d$ du polynôme est égal à 1 (i.e. on passe d'un point au suivant en ajoutant une constante), on

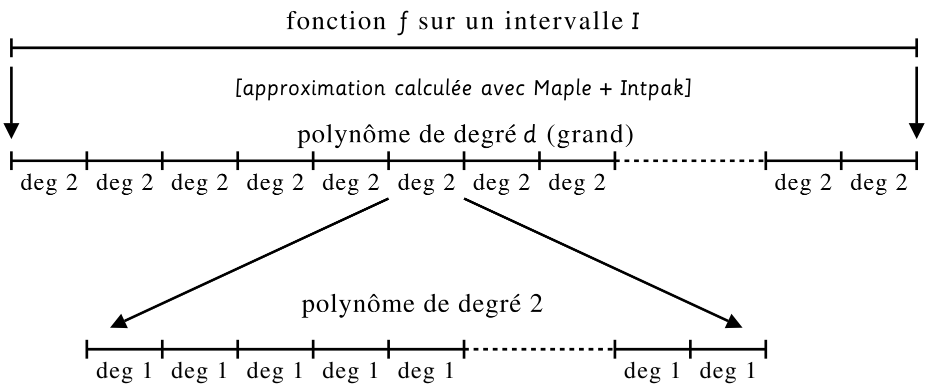



FIG. 3. Approximations hiérarchiques d'une fonction $f$ par des polynômes.

peut trouver une telle minoration rapidement, avec la même complexité que l'algorithme d'Euclide.

Ajouter une constante modulo $2^{-n}$ revient à faire une rotation d'un angle constant $\alpha$ sur un cercle. Si on construit ainsi des points $x_0$, $x_1$, $x_2$, ..., $x_k$ sur un cercle, alors à tout moment, l'angle entre deux points voisins sur le cercle peut prendre au plus 3 valeurs possibles (*théorème des trois distances*). Ces valeurs dépendent de $k$ et $\alpha$. De plus, il y a une infinité de valeurs de $k$ pour lesquelles l'angle ne prend que 2 valeurs possibles. L'algorithme suivant [26] se base sur ces configurations pour calculer une minoration de $\{b - ax\}$, où $x \in [\![0, N-1]\!]$.

**Initialisation :** $x = \{a\}$ ; $y = 1 - \{a\}$ ; $d = \{b\}$ ; $u = v = 1$;

**Boucle infinie :**

- **si** $(d < x)$
  - **tant que** $(x < y)$
    - **si** $(u + v \geqslant N)$ **fin**
    - $y = y - x$; $u = u + v$;
  - **si** $(u + v \geqslant N)$ **fin**
  - $x = x - y$; $v = v + u$;
- **sinon**
  - $d = d - x$;
  - **tant que** $(y < x)$
    - **si** $(u + v \geqslant N)$ **fin**
    - $x = x - y$; $v = v + u$;
  - **si** $(u + v \geqslant N)$ **fin**
  - $y = y - x$; $u = u + v$;

**Valeur de retour :** $d$

Notez la ressemblance avec l'algorithme d'Euclide...

Une modification de cet algorithme permet également de trouver les pires cas (et non plus seulement une minoration).

9.6.5. *Algorithme SLZ.* Et qu'en est-il des degrés supérieurs à 1 ? On a aussi un algorithme sous-linéaire [36], en $O(N^{(d+1)/(2d+1)+\varepsilon})$, basé sur les travaux de Coppersmith, utilisant l'algorithme LLL de réduction des réseaux. Note : le cas $d = 1$ correspond à l'algorithme de Lefèvre, mais l'algorithme donné ci-dessus est bien plus rapide que la forme générale de SLZ pour $d = 1$.

L'idée est la suivante. Coppersmith utilise l'algorithme LLL pour calculer rapidement les petites racines d'un polynôme multivarié modulo un entier $A$. Nous pouvons ici choisir $Q(x, y) = P(x) + y$. Dans notre problème, les valeurs de $x$ et de $y$ sont bornées par la taille de l'intervalle et par le choix de $m$ respectivement.

9.6.6. *Résultats après plusieurs mois de calculs.* Voici deux exemples de pires cas trouvés par tests exhaustifs.

Pour $x = .0111111110011101100111011100111001110100001111011011 01$, $\sin x$ vaut :

$$.01111010011001010100000111001100001100010001101001010 1\ 1 \underbrace{1111...1111}_{\text{65 bits}}\ 00...$$

ce qui donne $m = 53 + 1 + 65 = 119$ pour les modes d'arrondi dirigé.

Pour $x = 1.1110000100101101011001100111010001001111111110000001 \times 2^{429}$, $\log_{10} x$ vaut :

$$10000001.011010100111101010011011100101001111001000100\ 1 \underbrace{0000...0000}_{\text{68 bits}}\ 10...$$

ce qui donne $m = 53 + 1 + 68 = 122$ pour le mode d'arrondi au plus près.

## 10. Calcul exact avec des nombres flottants

L'arrondi exact permet de prouver des propriétés liées au système à virgule flottante et de concevoir des algorithmes permettant de faire du calcul exact en utilisant les flottants comme briques de base. On considère ici un système en base 2, avec $p$ bits de mantisse, en représentation signe-magnitude, avec arrondi exact au plus près, même si certains résultats sont généralisables dans toute base ou dans des systèmes plus généraux.

Note : certains calculs ne donnent pas le bon résultat sous certaines conditions spéciales (*overflow*, *underflow*, passage aux dénormalisés). Ces conditions ne seront pas toujours données dans la suite, le but de ce document étant essentiellement la connaissance des diverses techniques utilisées.

On note $\oplus$, $\ominus$ et $\otimes$ l'addition, la soustraction et la multiplication dans le système flottant (i.e. avec un résultat arrondi).

10.1. **Théorème de Sterbenz.** Ce théorème est un exemple de nombreuses propriétés élémentaires que l'on peut prouver sur des systèmes flottants. Ce théorème est valable dans n'importe quelle base $\beta$, à condition que le système supporte les dénormalisés (ou qu'il ne se produise pas d'*underflow*).

**Théorème 6** (Sterbenz)**.** *Si $x$ et $y$ sont deux flottants vérifiant $y/2 \leqslant x \leqslant 2y$ (ou de manière équivalente $x/2 \leqslant y \leqslant 2x$), alors $x - y$ est exactement représentable.*

*Preuve.* Si $x = 0$ ou $y = 0$, alors $x = y = 0$ et $x - y = 0$ est représentable. De même si $x = y$. Supposons maintenant que $x > y$. Alors $x - y \leqslant 2y - y = y$. L'exposant de $x \ominus y$ est donc inférieur ou égal à celui de $y$ (en particulier, il ne peut pas y avoir d'*overflow*) ; donc si $u$ dénote l'exposant du bit de poids faible (ulp) d'un flottant, alors $u(x \ominus y) \leqslant u(y)$. D'autre part, l'exposant du dernier bit 1 de $x - y$ est supérieur ou égal à $u(y)$, donc supérieur ou égal à $u(x \ominus y)$. La borne $u(y)$, resp. $u(x \ominus y)$, assure que $x - y$ est représentable si c'est un dénormalisé, resp. normalisé. Si $x < y$, la preuve reste la même par symétrie.

Note : il existe des généralisations de ce théorème pour des systèmes flottants exotiques [5].

Conséquence sur un système avec arrondi exact : puisque l'arrondi d'un flottant est ce flottant, la soustraction est exacte.

10.2. **Découpage d'un flottant en deux.** Une des opérations élémentaires est le découpage d'un flottant en deux : une partie haute et une partie basse (de même taille ou non). La manière de le faire précisément dépend du contexte, mais l'idée générale est de générer, à partir du flottant initial, un flottant ayant un plus gros exposant (de façon à perdre des chiffres de poids faible), puis de faire l'opération inverse exacte, qui donnerait 0 s'il n'y avait pas d'arrondi ; mais à cause de l'arrondi de la première opération, on obtient la partie de poids fort. La partie de poids faible s'obtient en soustrayant cette partie du flottant initial. Deux exemples sont donnés ci-dessous.

10.2.1. *Découpage sur une position absolue.* Cette méthode est utilisée quand on veut découper le flottant au niveau d'un exposant fixé. Si $s$ est un entier et $x$ est un flottant normalisé positif inférieur à $2^s$ :

$$\begin{aligned} y &= x \oplus 2^s \\ x_h &= y \ominus 2^s \\ x_\ell &= x \ominus x_h \end{aligned}$$

L'exposant de $y$ étant $s$, l'ulp de $y$ a pour exposant $s - p + 1$. La deuxième opération se fait exactement et on obtient les bits de $x$ de poids supérieur ou égal à $2^{s-p+1}$. La troisième opération se fait aussi exactement, et on obtient les bits de $x$ de poids inférieur ou égal à $2^{s-p}$.

Cette méthode telle quelle ne fonctionne plus aussi bien si le signe de $x$ n'est pas connu, car $y$ aurait un exposant $s$ pour $x$ positif et un exposant inférieur à $s$ pour $x$ négatif. Une solution : remplacer $2^s$ par $3 \times 2^{s-1}$ par exemple...

10.2.2. *Découpage sur une position relative.* Cette méthode est utilisée quand on n'a pas de connaissance particulière sur les exposants : on veut simplement découper un flottant en deux parties de taille fixée. Pour $s$ entier positif :

$$\begin{aligned} x' &= x \otimes (2^s + 1) \\ x_h &= (x \ominus x') \oplus x' \\ x_\ell &= x \ominus x_h \end{aligned}$$

Ici, $x_h$ reçoit les $n - s$ bits de poids fort de $x$, et $x_l$ les $s$ bits de poids faible.

Justification rapide : supposons $x$ entier pour simplifier, c'est-à-dire $\text{ulp}(x) = 1$. Soit $x = h2^s + l$, avec $h, l$ entiers, $|l| < 2^s$. Alors $x(2^s + 1) = (x + h)2^s + l$. Supposons que l'arrondi est $x' = (x + h)2^s$, alors $x - x' = l - x2^s$ s'arrondit en $-x2^s$, et après ajout de $x'$ on obtient $h2^s$.

10.3. **Application à l'implémentation de la fonction `rint`.** La fonction `rint` arrondit un flottant $x$ en un entier dans le mode d'arrondi courant. On suppose qu'on est en double précision, et on définit `TWO52` comme étant $2^{52}$.

```
if (fabs (x) < TWO52)
  {
    if (x > 0.0)
      {
        x += TWO52;
        x -= TWO52;
        if (x == 0.0)
          x = 0.0;
      }
    else if (x < 0.0)
      {
        x -= TWO52;
        x += TWO52;
        if (x == 0.0)
          x = -0.0;
      }
  }
return x;
```

10.4. **Algorithmes TwoSum et FastTwoSum.** L'algorithme suivant [24] permet de calculer l'erreur $e_2$ sur une somme $x + y$ arrondie ($s$).

Algorithme TwoSum

$$
\begin{array}{lcl}
s & = & x \oplus y \\
t & = & x \ominus s \\
e_1 & = & y \oplus t \\
u & = & s \oplus t \\
v & = & x \ominus u \\
e_2 & = & e_1 + v
\end{array}
$$

Mais si $|x| \geqslant |y|$, alors cette erreur est donnée par $e_1$ [17], ce qui fait gagner 3 opérations. En fait, il suffit que l'exposant de $x$ soit supérieur ou égal à l'exposant de $y$ [14], ou bien que le poids du dernier bit 1 de $x$ soit supérieur ou égal à l'ulp de $y$ [11].

Algorithme FastTwoSum

$$\begin{array}{lcl} s & = & x \oplus y \\ t & = & x \ominus s \\ e_1 & = & y \oplus t \end{array}$$

Note : en base 10, FastTwoSum ne fonctionne pas toujours ; par exemple, avec une précision de $p = 3$ chiffres, pour $x = 998$ et $y = 997$ :

$$\begin{array}{lclclcl} s & = & 998 \oplus 997 & = & \circ(1995) & = & 2000 \\ t & = & 998 \ominus 2000 & = & \circ(-1002) & = & -1000 \\ e_1 & = & 997 \oplus -1000 & = & \circ(-3) & = & -3 \end{array}$$

au lieu de $-5$.

10.5. **Algorithme de Dekker (multiplication exacte).** L'algorithme de Dekker permet de calculer $xy$ exactement : il renvoie le résultat arrondi $r_h = x \otimes y$ et le terme d'erreur $r_\ell = xy - x \otimes y$.

Soit $s = \lceil p/2 \rceil$. Par exemple, pour la double précision ($p = 53$), $s = 27$.

$$\begin{array}{lcl} x' & = & x \otimes (2^s + 1) \\ x_h & = & (x \ominus x') \oplus x' \\ x_\ell & = & x \ominus x_h \\[4pt] y' & = & y \otimes (2^s + 1) \\ y_h & = & (y \ominus y') \oplus y' \\ y_\ell & = & y \ominus y_h \\[4pt] r_h & = & x \otimes y \\ r_\ell & = & (((x_h \otimes y_h \ominus r_h) \oplus x_h \otimes y_\ell) \oplus x_\ell \otimes y_h) \oplus x_\ell \otimes y_\ell \end{array}$$

En résumé, cet algorithme fonctionne de la manière suivante : la première étape consiste à découper $x$ et $y$ en deux flottants de taille moitié (une partie de l'information se trouve dans le signe, grâce à l'arrondi au plus près). Ensuite, on part du résultat arrondi $r_h$, et on enlève le résultat exact, formé à partir des 4 sous-produits (exacts), par une succession de soustractions (ou additions) exactes.

10.6. **Expansions de nombres flottants.** Une expansion de nombres flottants est une représentation d'un flottant en multiprécision à l'aide d'une somme de nombres flottants d'un système à précision fixée, avec quelques contraintes. Cette représentation a été introduite par T. J. Dekker [17], et étudiée par [31], J. R. Shewchuk [35], M. Daumas et C. Finot-Moreau [12, 13] (cette partie est principalement basée sur la thèse de cette dernière [18]).

Plus précisément, une expansion de nombres flottants est la représentation d'un nombre $x$ par un $n$-uplet $(x_0, x_1, ..., x_{n-1})$ de nombres flottants tels que :

(1) $x$ est égal à la somme *exacte* des $x_i$ ;

(2) les composantes $x_i$ non nulles sont triées par ordre décroissant de valeur absolue ;

(3) les composantes $x_i$ non nulles ne se recouvrent pas : $x_i$ et $x_{i+1}$ ne doivent pas avoir de bits significatifs de même poids.

Certaines contraintes peuvent être relâchées. Par exemple, on peut autoriser deux composantes consécutives à se recouvrir partiellement (mais pas $x_i$ et $x_{i+2}$) ; on parle de pseudo-expansions. De telles représentations conviennent mieux pour le calcul en ligne.

10.6.1. *Addition d'expansions.* L'addition d'expansions de nombres flottants peut se faire intuitivement à l'aide de FastTwoSum, en additionnant soit les deux composantes de poids fort, soit les deux composantes de poids faible, parmi celles des arguments et résultats intermédiaires. La version poids faible en tête produit bien une expansion, mais avec beaucoup de composantes : le résultat aurait besoin d'être normalisé. La version poids fort en tête ne produit pas une expansion, car certaines composantes du résultat peuvent se recouvrir.

10.6.2. *Addition de pseudo-expansions.* L'additionneur se base sur un opérateur $\Sigma_3$ prenant trois flottants en entrée et sortant une pseudo-expansion de taille au plus 3 (il s'agit en quelque sorte d'une généralisation de FastTwoSum à 3 entrées et 3 sorties). Cet opérateur utilise un accumulateur contenant deux composantes. Les composantes des deux arguments sont triées poids fort en tête et fournies une par une à $\Sigma_3$ (les deux autres entrées provenant de l'accumulateur). La composante de poids fort produite par $\Sigma_3$ sera la composante suivante du résultat final et les deux autres composantes forment l'accumulateur.

10.6.3. *Opérateur* $\Sigma_3$. Si $a$, $b$ et $c$ sont triés par exposant décroissant, alors l'algorithme suivant calcule une pseudo-expansion de longueur 3.

$$\begin{aligned}(u, v) &= \mathrm{FastTwoSum}(b, c)\\ (a', w) &= \mathrm{FastTwoSum}(a, u)\\ (b', c') &= \mathrm{FastTwoSum}(w, v)\end{aligned}$$

10.6.4. *Multiplication.* Pour effectuer une multiplication de pseudo-expansions, l'idée est la suivante :
- les produits partiels sont calculés comme dans l'algorithme de Dekker ;
- ces produits partiels sont triés par ordre décroissant d'exposant (de manière similaire à l'addition) ;
- les produits partiels sont additionnés par $\Sigma_3$ (comme dans l'addition de pseudo-expansions) ou par un opérateur $\Sigma_5$.

10.7. **Multiplication en multiprécision basée sur les flottants.** Une autre façon de faire de la multiprécision avec les flottants comme briques de base est d'écrire les nombres dans une grande base, chaque flottant représentant un chiffre (entier). Par exemple, les flottants en double précision peuvent stocker des chiffres dans la base $2^{50}$, les trois bits supplémentaires pouvant servir à accumuler des retenues (ainsi on a en quelque sorte une représentation redondante).

Voici un exemple de code de multiplication (avec accumulation, i.e. calcul de $xy$, que l'on ajoute à la valeur courante de $z$). Dans les commentaires, la notation `[E:N1|N2|...|Nk]` représente l'ensemble suivant :

$$2^{E\times B_Q}\left(\sum_{i=1}^{k}\left[-2^{N_i}, 2^{N_i}\right]\right).$$

```
#define BQ 25
#define SRB (1<<BQ)
#define ROUND(x,c) ((x + c) - c)
#define C0 6755399441055744.0  /* 3*2^51 */
#define C1 (C0 * SRB)
#define C2 (C1 * SRB)
#define CR (1.0 / SRB)
#define CS (CR / SRB)

void mac(double *z, double *x, double *y, int n)
{
  int i, j;
  double xx[2*MAXN];

  assert(n <= MAXN);

  for (i = 0; i < n; i++)   /* splits the digits of x */
    {
      double x0, x1;

      x0 = x[i]; x1 = ROUND(x0,C1); x0 -= x1;
      xx[2*i] = x0; xx[2*i+1] = x1;
    }

  for (j = 0; j < n; j++)
    {
      double *p;
      double y0, y1, z1 = 0.0;

      p = z + j;
      y0 = y[j]; y1 = ROUND(y0,C1); y0 -= y1;

      for (i = 0; i < n; i++)
        {                               /*    z1: [0:2BQ|BQ+2]  */
          double x0, x1, z0, zr;
          x0 = xx[2*i];                 /* x0,y0: [0:BQ]        */
          x1 = xx[2*i+1];               /* x1,y1: [1:BQ]        */
          z0 = x0*y1 + x1*y0;           /*    z0: [1:2BQ+1]     */
          zr = ROUND(z0,C2);            /*    zr: [2:BQ+1]      */
          z0 = z1 + (z0-zr) + x0*y0 + *p;  /*    z0: [0:2BQ+3]     */
          z1 = x1*y1 + zr;              /*    z1: [2:2BQ|BQ+1]  */
          zr = ROUND(z0,C2);            /*    zr: [2:3]         */
```

```
        *p++ = z0-zr;                     /* z0-zr: [0:2BQ]        */
        z1 = (z1+zr) * CS;                /*    z1: [0:2BQ|BQ+2]   */
      }

    while (z1 != 0)
      {                                   /*    z1: [0:2BQ|BQ+2]   */
        double z0;
        z0 = z1 + *p;                     /*    z0: [0:2BQ+2]      */
        z1 = ROUND(z0,C2);                /*    z1: [2:2]          */
        *p++ = z0-z1;                     /* z0-z1: [0:2BQ]        */
      }
  }
}
```

## 11. Calcul en précision arbitraire

Quand on travaille en précision arbitraire, certains algorithmes vus en précision fixe ne sont plus adaptés. Par exemple, les méthodes à base de tables ne sont plus utilisables, puisque ces tables dépendent de la précision requise, et on ne peut pas calculer des tables de taille quelconque. Les approximations minimax ne sont plus utilisables non plus, car elles sont relativement coûteuses : si on les calculait à la volée, le gain par rapport à un simple développement de Taylor serait sans doute nul. Il faut donc repenser la plupart des algorithmes. Cette section décrit brièvement les principaux algorithmes utilisés en précision arbitraire.

11.1. **Développements de Taylor ou asymptotiques.** Les développements de Taylor ne sont pas les meilleures approximations à degré fixé (cf. approximations minimax), mais ils offrent l'avantage de la simplicité : seules des additions et des multiplications sont nécessaires. De plus pour la plupart des fonctions, les développements de Taylor en 0 sont connus de manière formelle [1].

Prenons l'exemple de la fonction d'erreur, définie par

$$\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.$$

Un développement à l'origine de la fonction d'erreur est le suivant :

$$\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \sum_{i=0}^{\infty} \frac{(-1)^i}{i!(2i+1)} x^{2i+1}.$$

Le principe général de l'évaluation par série formelle (de Taylor ou développement asymptotique à l'infini) d'une fonction $f$ en un point $x$ est le suivant :

(1) pour $k$ fixé, déterminer une borne $R_k$ sur l'erreur commise en sommant la série jusqu'à l'ordre $k$ ;
(2) en fonction de la précision relative voulue $n$, et éventuellement du point $x$, déterminer une « bonne » valeur de $k$ ;

(3) évaluer une approximation $S_k$ de la série jusqu'à l'ordre $k$, une borne $\varepsilon_k$ sur l'erreur d'arrondi sur $S_k$, et déterminer si $S_k$, en tenant compte des erreurs $R_k$ et $\varepsilon_k$, permet de déterminer l'arrondi correct de $f(x)$ ;

(4) sinon, augmenter $k$ et la précision, et retourner à l'étape (3).

$R_k$ est l'erreur « mathématique » ; elle ne dépend pas de la méthode choisie pour sommer la série à l'étape (3). $\varepsilon_k$ est l'erreur « d'arrondi » ; elle dépend au contraire fortement de la méthode choisie pour sommer la série.

Pour la fonction erf, la série étant alternée et décroissante en valeur absolue pour $k \geqslant x^2$, la somme de $k+1$ à l'infini est bornée en valeur absolue par son premier terme, et on a donc, en utilisant le fait que $k! \geqslant (k/e)^k\sqrt{2\pi k}$, et $k \geqslant x^2$ :

$$R_k \leqslant \frac{|x|^{2k+3}}{(k+1)!(2k+3)} \leqslant \frac{1}{5}\left(\frac{x^2 e}{k}\right)^k.$$

Si on veut $R_k \leqslant 2^{-n}$, on voit qu'il suffit de prendre $k$ l'entier immédiatement supérieur à la solution de $k\log_2\frac{k}{x^2e} = n$.

Remarque. Cette solution s'exprime aussi symboliquement à l'aide de la fonction $W$ de Lambert : on a en effet $k = \frac{n\log 2}{W(\frac{n\log 2}{x^2e})}$. Cependant on peut tout aussi bien déterminer une estimation fine de $k$ en partant de $k=n$, et en itérant $k \leftarrow \frac{n}{\log_2(kx^{-2}e^{-1})}$ jusqu'à ce que $\lceil k\rceil$ ne change plus. Par exemple, pour $x=1$, $n=1000$, on obtient successivement 117.3, 184.1, 164.4, 168.9, 167.8, 168.1, 168.0.

Il nous faut maintenant calculer une approximation de la série tronquée $\sum_{i=0}^{k}\frac{(-1)^i}{i!(2i+1)}x^{2i}$, qu'on multipliera ensuite par $\frac{2x}{\sqrt{\pi}}$. Il faut pour cela expliciter la méthode de calcul :

$t \leftarrow 1$
$s \leftarrow 1$
$y \leftarrow \circ(x^2)$
**for** $i$ **from** 1 **to** $k$ **do**
    $t \leftarrow \circ(ty)$
    $t \leftarrow \circ(t/i)$
    $u \leftarrow \circ(t/(2i+1))$
    $s \leftarrow \circ(s+(-1)^i u)$

On suppose ici que toutes les variables ont la même « précision de travail » $p$, supérieure à la précision finale $n$ à laquelle on veut évaluer erf $x$. En utilisant la méthode de Higham, on peut écrire qu'après $i$ itérations, on a $t = \frac{x^{2k}}{k!}\prod_{j=1}^{i}(1+\theta_j)(1+\theta_y)(1+\theta'_j)$, où $\theta_y$ est l'erreur relative sur $y$, $\theta_j$ celle sur l'arrondi de $ty$ à l'étape $j$, et $\theta'_j$ celle sur l'arrondi de $t/k$. L'erreur relative sur $t$ est donc bornée par $(1+\theta)^{3i}-1$ à l'étape $i$.

Une autre façon d'analyser l'erreur est d'utiliser les ulps. L'erreur sur $y$ est d'au plus 1 ulp, et si on arrondit vers l'infini, l'erreur d'une multiplication est bornée par 1 ulp plus les erreurs sur les opérandes, soit $3i$ ulps à l'étape $i$ (on suppose ici qu'il n'y a pas d'erreur sur $i$ dans la division $t/i$). L'erreur sur $u$ est donc d'au plus $3i+1$ ulps.

Borner l'erreur sur $s$ n'est pas aussi facile car il peut y avoir cancellation lors de la soustraction $s-u$ pour $i$ impair. D'autre part il faudrait exprimer l'erreur de $(3i+1)\,\mathrm{ulp}(u)$ en fonction d'ulp$(s)$. Une façon de contourner ce problème est de borner non pas l'erreur relative sur $s$, mais l'erreur absolue. Soit $B$ une borne en valeur absolue sur $u$ et $s$ au cours du calcul. L'erreur absolue sur $s$ due à l'étape $i$ est bornée par $\mathrm{ulp}(s)+(3i+1)\mathrm{ulp}(u) \leqslant (3i+2)\mathrm{ulp}(B)$. L'erreur finale sur $s$ est donc au plus $\sum_{i=1}^{k}(3i+2) = \frac{1}{2}k(3k+7)$ en termes d'ulp$(B)$. Il suffit ensuite d'ajouter l'erreur d'arrondi commise lors de la multiplication par $\frac{2x}{\sqrt{\pi}}$ pour obtenir une approximation $S_k$ de la série tronquée, avec une erreur bornée par $\varepsilon_k$.

Pour décider si on peut calculer l'arrondi correct de $\operatorname{erf} x$ à partir de $S_k$, comme on sait que $S_k-(R_k+\varepsilon_k) \leqslant \operatorname{erf} x \leqslant S_k+(R_k+\varepsilon_k)$, il suffit de s'assurer que l'arrondi de $S_k-(R_k+\varepsilon_k)$ égale celui de $S_k+(R_k+\varepsilon_k)$. Pour cela, une condition nécessaire est bien sûr que $R_k$ et $\varepsilon_k$ soient inférieurs à $\mathrm{ulp}_n(S_k)$, où $n$ est la précision finale voulue.

On a vu sur cet exemple quelques difficultés que l'on peut rencontrer pour déterminer l'arrondi correct de $f(x)$ en précision arbitraire. On peut aussi avoir une série convergente mais non alternée, pour laquelle borner le reste n'est pas si facile. On voit aussi sur cet exemple que le calcul de $f(x)$ peut faire intervenir des constantes comme $\pi$ ou $\log 2$ : il est donc important de savoir calculer ces constantes de manière efficace, éventuellement en gardant en mémoire la valeur la plus précise calculée.

11.2. **Développement asymptotique.** Pour $x$ tendant vers l'infini, on a $\operatorname{erf} x = 1-\operatorname{erfc} x$, avec

$$\operatorname{erfc} x = \frac{1}{\sqrt{\pi}e^{x^2}} \sum_{k=0}^{\infty} (-1)^k \frac{(2k)!}{k!2^{2k}} x^{-2k-1}.$$

Attention, ce développement asymptotique n'est pas convergent ! En effet, le rapport de deux termes consécutifs est de l'ordre de $k/x^2$, et tend vers l'infini avec $k$, $x$ étant fixé.

Si on se contente du premier terme du développement asymptotique, cela donne $\operatorname{erf} x \approx 1-\frac{e^{-x^2}}{\sqrt{\pi}x}$. Lorsque $\frac{e^{-x^2}}{\sqrt{\pi}x} \leqslant 2^{-n}$, l'arrondi de $\operatorname{erf} x$ est donc soit 1, soit $1^-$ (le nombre flottant immédiatement inférieur à 1) suivant le mode d'arrondi.

11.3. **Méthode de Brent et Kung.** La méthode vue ci-dessus pour l'évaluation de la série de Taylor de $\operatorname{erf} x$ nécessite $k$ itérations avec $k = O(n)$, où $n$ est la précision finale voulue. Chaque itération nécessite quant à elle une multiplication $(ty)$ entre deux flottants de $n$ bits, ce qui coûte $M(n)$. Le coût total est donc $O(nM(n))$, soit $O(n^3)$ avec une multiplication naïve.

La méthode de Brent et Kung [9] permet d'évaluer une telle série en temps $O(n^{1/2}M(n))$. L'idée de base est de diviser les $k = O(n)$ termes à sommer en $\sqrt{k}$ morceaux de $\sqrt{k}$ termes. Reprenons l'exemple de la fonction erf. On a en notant $y = x^2$ :

$$\frac{\sqrt{\pi}}{x}\operatorname{erf} x = 2-\frac{2}{3}y+\frac{1}{5}y^2-\frac{1}{21}y^3+\frac{1}{108}y^4-\frac{1}{660}y^5+\frac{1}{4680}y^6-\frac{1}{37800}y^7+\frac{1}{342720}y^8+O(y^9),$$

ce qui s'écrit aussi :

$$\begin{array}{rcccc} \frac{\sqrt{\pi}}{x}\operatorname{erf} x & \approx & 2 & -\frac{2}{3}y & +\frac{1}{5}y^2 \\ +\frac{y^3}{3} & (-\frac{1}{7} & +\frac{1}{38}y & -\frac{1}{220}y^2 \\ +\frac{y^3}{120} & (\frac{1}{13} & -\frac{1}{105}y & +\frac{1}{952}y^2)). \end{array}$$

Une fois calculés $y^2$ et $y^3$, il suffit de deux multiplications entre flottants de $n$ bits pour évaluer cette dernière formule, soit en tout 4 telles multiplications ; toutes les autres opérations sont soit des multiplications ou divisions entre un flottant de $n$ bits et un petit entier, soit des additions ou soustractions, opérations qui coûtent $O(n)$. À titre de comparaison, la première formule nécessite 7 multiplications entre flottants de $n$ bits, pour calculer $y^2$ à $y^8$.

La méthode de Brent et Kung consiste à récrire la série en termes de $X = x^l$ où $l \approx \sqrt{k}$ :

$$\sum_{i=0}^{k} a_i x^i = \sum_{j=0}^{l} b_j(x) X^j.$$

Chaque sous-série $b_j(x)$ est de degré inférieur à $l$, et s'évalue uniquement via des opérations en $O(n)$, après précalcul de $1, x, x^2, ..., x^{l-1}$. Quant à l'évaluation de $\sum_{j=0}^{l} b_j(x)X^j$, elle se fait via le schéma de Horner, en $O(l)$ multiplications. Cette méthode utilise donc en tout environ $2\sqrt{k}$ multiplications de flottants de $n$ bits.

11.4. **Approximation de Padé.** Au lieu d'utiliser un développement de Taylor, on peut aussi utiliser une approximation rationnelle (Padé). En général, un développement de Padé de numérateur et dénominateur d'ordre $k$ équivaut à un développement de Taylor d'ordre $2k$. On peut gagner ainsi a priori un facteur 2 sur le nombre de multiplications entre flottants de $n$ bits, au prix d'une division finale supplémentaire. Cependant les coefficients de l'approximation de Padé peuvent s'avérer plus compliqués, par exemple toujours pour $\frac{\sqrt{\pi}}{x}\operatorname{erf} x$ :

$$\frac{2 + \frac{2227367513}{9804764229}x^2 + \frac{2295626419}{32682547430}x^4 + \frac{912396623}{524843261670}x^6 + \frac{121749556141}{1039189658106600}x^8}{1 + \frac{2921292333}{6536509486}x^2 + \frac{5496758043}{65365094860}x^4 + \frac{80154641}{9997014508}x^6 + \frac{2491500345}{7477766851984}x^8}.$$

11.5. **Fractions continues.** Les fractions continues sont un autre moyen d'évaluer une fonction ayant un développement de Taylor. On peut facilement calculer l'un à partir de l'autre, par exemple en Maple :

```
> series(exp(x), x, 6);
                            2         3          4           5      6
            1 + x + 1/2 x  + 1/6 x  + 1/24 x  + 1/120 x  + O(x )

> convert(%, confrac);
                                   x
                  1 + -----------------------
                                     x
                      1 + -------------------
                                       x
                          -2 + --------------
                                         x
                               -3 + ---------
                                    2 + 1/5 x
```

Par rapport au schéma de Horner pour l'évaluation d'une série de Taylor, où l'opération de base est $y \to xy + b$, ici l'opération de base est $y \to x/y + b$. Il est donc important d'avoir une division efficace.

11.6. **Méthode de Newton (rappel).** Comme vu lors du paragraphe sur les algorithmes à convergence quadratique (section 7), la méthode de Newton permet de trouver les racines d'une équation implicite $f(x, y(x)) = 0$. Elle consiste à itérer :

$$y_{k+1} = y_k - \frac{f(x, y_k)}{f'_y(x, y_k)}.$$

Cette méthode comporte deux propriétés importantes :

(1) la convergence est quadratique, c'est-à-dire que le nombre de bits corrects de $y_k$ double à chaque étape. Ainsi, pour obtenir $n$ bits corrects, il suffit de $\log_2 n$ étapes.

(2) la méthode de Newton est auto-correctrice, c'est-à-dire que pour obtenir $n$ bits corrects à l'étape $k$, il suffit d'avoir $n/2$ bits corrects à l'étape $k - 1$.

Une conséquence importante de ces deux propriétés est que toute fonction algébrique se calcule en temps $O(M(n))$ via la méthode de Newton. En effet, une fonction algébrique est définie par un polynôme $P(x, y) = 0$ : $P(x, y)$ et $P'_y(x, y)$ s'évaluent en $O(M(n))$, ainsi que leur quotient, donc le coût $C(n)$ du calcul de $y(x)$ vérifie $C(n) = O(M(n)) + C(n/2)$, soit $C(n) = O(M(n))$ pour $M(n) \sim n^\alpha$ avec $\alpha \geqslant 1$.

Une application importante de la méthode de Newton est le calcul de fonction inverse. Supposons qu'on sache calculer efficacement une fonction $f(x)$. La méthode de Newton permet alors de calculer la fonction inverse $g(x)$ — telle que $f(g(x)) = x$. En effet, la fonction $g(x)$ est solution de $f(y) - x = 0$. On en déduit l'itération de Newton :

$$y_{k+1} = y_k - \frac{f(y_k) - x}{f'(y_k)}.$$

Si l'on dispose d'une méthode de calcul de $f$ de complexité $C(n)$, on en déduit une méthode de calcul de $g$ de même complexité, à une constante près. Par exemple, pour $f = \log$ et $g = \exp$, on obtient [7] :

$$y_{k+1} = y_k - y_k(\log y_k - x).$$

Ainsi, le logarithme se calculant en $O(M(n) \log n)$ via la moyenne arithmético-géométrique (cf. ci-dessous), on en déduit un algorithme de même complexité pour l'exponentielle.

11.7. **Moyenne arithmético-géométrique.** La moyenne arithmético-géométrique (AGM en anglais) de deux réels $0 < a \leqslant b$ est la limite commune des deux suites adjacentes :

$$u_0 = a, u_{n+1} = \frac{u_n + v_n}{2}, \qquad v_0 = b, v_{n+1} = \sqrt{u_n v_n}.$$

Cette limite est notée $\mathrm{AGM}(a, b)$. De nombreuses fonctions transcendantes peuvent se calculer efficacement grâce à la moyenne arithmético-géométrique, par exemple [7] :

$$\log x = \frac{\pi}{2\ \mathrm{AGM}(\frac{4}{2^m x}, 1)} - m \log 2 + o(2^{-n} \log x),$$

où $n$ est la précision finale voulue, et $x2^m > 2^{n/2}$.

11.8. **Le scindage binaire.** Le scindage binaire (*binary splitting* en anglais) consiste à ramener le calcul d'un flottant en grande précision à une approximation rationnelle $p/q$, avec $p$ et $q$ deux grands entiers, qui sont eux-mêmes calculés par multiplications d'entiers de taille moitié, et ainsi de suite. On utilise ainsi de manière optimale les algorithmes de multiplication rapide (Karatsuba, Toom-Cook, transformée de Fourier rapide). Les algorithmes obtenus sont en général de complexité $O(M(n)\log n)$ ou $O(M(n)\log^2 n)$.

L'exemple classique pour comprendre cette méthode est le calcul de $n!$. La méthode naïve consiste à calculer successivement 1!, 2!, 3!, ..., $(n-1)!$, $n!$, avec à chaque fois multiplication d'un grand entier par un petit entier. L'idée fondamentale du scindage binaire est d'équilibrer les multiplications. Ainsi $n!$ sera calculé par FastFactorial$(0, n)$, où FastFactorial$(a, b)$ calcule $(a+1)(a+2)...(b-1)b$ :

Algorithme FastFactorial$(a, b)$.
**if** $a + 1 = b$ **then** return $b$
$c \leftarrow \lfloor \frac{a+b}{2} \rceil$
$f \leftarrow$ FastFactorial$(a, c)$
$g \leftarrow$ FastFactorial$(c, b)$
Return $fg$.

Cette idée peut être étendue au calcul de nombreuses fonctions, donnant des algorithmes de calcul en $O(M(n)\log^2 n)$ si une relation fonctionnelle simple existe entre $f(x+y)$ et $f(x), f(y)$ ; sinon, on obtient du $O(M(n)log^3 n)$. Ces résultats sont dus aux frères Borwein, à Haible et Papanikolaou, et aux frères Chudnovsky [6, 10, 19].

Voici par exemple comment on peut calculer l'exponentielle en temps $O(M(n)\log^2 n)$. Cet algorithme est dû à Brent [8]. On commence par supposer que $|x| < 1/2$, après éventuelle réduction d'argument multiplicative, utilisant $\exp x = (\exp \frac{x}{2})^2$. On décompose ensuite $x$ de la manière suivante :

$$x = 0.0\,\underbrace{b_2}_{x_1}\,\underbrace{b_3 b_4}_{x_2}\,\underbrace{b_5 b_6 b_7 b_8}_{x_3}...,$$

ce qui revient à écrire $x = \sum_{i \geqslant 1} \frac{r_i}{2^{2^i}}$, avec $0 \leqslant r_i < 2^{2^{i-1}}$. Par conséquent on a :

$$\exp x = \prod_{i \geqslant 1} \exp \frac{r_i}{2^{2^i}}.$$

L'exponentielle de chaque $\frac{r_i}{2^{2^i}}$ se calcule par scindage binaire en temps $O(M(n)\log n)$. Comme il y a $O(\log n)$ valeurs non nulles de $r_i$, cela donne une complexité totale en $O(M(n)\log^2 n)$.

## 12. Hardware (et redondance)

Nous nous intéressons ici à l'implémentation en matériel d'opérateurs, en mettant principalement l'accent sur la complexité en temps, mais aussi en surface.

12.1. **Le théorème de Winograd.** Une première question qui se pose naturellement : étant donnée une fonction $f$ sur des données de taille finie, combien de temps au moins faut-il pour la calculer ?

Si on considère une représentation binaire classique, le problème revient, du point de vue de la complexité en temps, à calculer chaque bit du résultat en parallèle en fonction d'un sous-ensemble des bits d'entrée : pour chaque bit de rang $i$,

$$f_i \,:\, \{0,1\}^{n_i} \to \{0,1\}.$$

On cherche à calculer $f_i$ par un circuit logique ayant des portes de *fan-in* $r$ (i.e. avec au plus $r$ entrées — par exemple $r = 2$ si on ne considère que les opérations logiques de base : `NOT`, `AND`, `OR`, `NAND`, `NOR`) et de temps de réponse $\tau$. On suppose de plus que $n_i$ est minimal, i.e. que chaque bit d'entrée a une influence sur le résultat : pour tout $1 \leqslant k \leqslant n_i$, il existe une valeur des autres entrées telle que $f_i(...0_k...) \neq f_i(...1_k...)$. Le circuit devra alors avoir au moins $n_i$ entrées, et son temps de réponse sera supérieur ou égal à $\lceil \log_r n_i \rceil \tau$, car un circuit de $h$ étages ne peut pas avoir plus de $r^h$ entrées.

Le temps nécessaire pour calculer la fonction $f$ est donc d'au moins $\max_i \lceil \log_r n_i \rceil \tau$ (hauteur de l'arbre associé au circuit).

12.2. **Additionneurs.** Nous considérons ici l'addition d'entiers, à laquelle se ramènent de nombreuses opérations.

12.2.1. *Additionneurs séquentiels simples.* Les entiers étant écrits en binaire, la cellule de base est le *full adder* (FA) ; elle additionne 3 bits (2 bits des 2 opérandes et un bit de retenue) et renvoie un bit de résultat et un bit de retenue : $x_i + y_i + c_i = 2c_{i+1} + s_i$.

$$\begin{cases} s_i &= x_i \oplus y_i \oplus c_i \\ c_{i+1} &= \mathrm{Maj}(x_i, y_i, c_i) \end{cases}$$

Problème : la propagation de la retenue donne un temps linéaire.

12.2.2. *Additionneurs parallèles.* Si $x_i = y_i$, l'additionneur peut générer la bonne retenue immédiatement, ce qui donne un temps moyen en $\log_2 n$, mais le temps de calcul au pire reste linéaire.

12.2.3. *Additionneurs à retenue conditionnelle.* On reprend ici plus ou moins l'idée de l'additionneur parallèle : si on ne peut pas deviner la retenue, on peut toujours effectuer en parallèle le calcul avec une retenue à 0 et le calcul avec une retenue à 1 pour anticiper, et garder le « bon » résultat une fois la retenue récupérée.

Ainsi, un additionneur de taille $2n$ s'obtient avec 3 additionneurs de taille $n$. On écrit : $x = x_{\mathrm{hi}} + x_{\mathrm{lo}}$ et $y = y_{\mathrm{hi}} + y_{\mathrm{lo}}$ ; on calcule en parallèle :

$$\begin{cases} (c_{\mathrm{lo}}, s_{\mathrm{lo}}) &= x_{\mathrm{lo}} + y_{\mathrm{lo}} \\ s_{\mathrm{hi},0} &= x_{\mathrm{hi}} + y_{\mathrm{hi}} \\ s_{\mathrm{hi},1} &= x_{\mathrm{hi}} + y_{\mathrm{hi}} + 1 \end{cases}$$

et on multiplexe $s_{\mathrm{hi},0}$ et $s_{\mathrm{hi},1}$ avec la valeur de $c_{\mathrm{lo}}$.

En appliquant cette idée récursivement, on obtient un temps en $O(\log n)$, mais cela occupe beaucoup de surface ! Cette méthode n'est donc pas utilisée en pratique.

12.2.4. *Additionneurs « carry look-ahead ».* On note le « ou logique » additivement et le « et logique » multiplicativement.

On remarque que :

– si $x_i = y_i$, alors $c_{i+1} = x_i$ $(= y_i)$ : génération (ou non) de retenue ;
– si $x_i \neq y_i$, alors $c_{i+1} = c_i$ : propagation de retenue.

On définit alors : $\left\{ \begin{array}{lll} p_i & = & x_i \oplus y_i \quad \text{(aptitude à propager une retenue)} \\ g_i & = & x_i y_i \quad \text{(aptitude à générer une retenue).} \end{array} \right.$

L'addition pourra alors s'effectuer de la façon suivante :

(1) On construit en parallèle les $p_i$ et les $g_i$.

(2) On calcule le plus vite possible les $c_i$ (Carry Look-ahead Unit).

(3) On calcule en parallèle les $s_i = p_i \oplus c_i$.

Carry Look-ahead Unit (CLU) : on a $c_{i+1} = g_i + p_i c_i$ (soit la retenue est générée, soit elle est propagée), et en déroulant :

$$c_{i+1} = g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + \ldots + p_i p_{i-1} \ldots p_1 g_0 + p_i \ldots p_1 p_0 c_0$$

i.e. les $c_i$ se calculent à l'aide de deux portes logiques seulement, c'est donc très rapide, mais le nombre d'entrées de ces portes (fan-in) augmente avec la taille des nombres à additionner. Cette méthode est donc réservée à de petits additionneurs (jusqu'à 8 bits).

12.2.5. *Additionneurs « carry look-ahead » cascadés.* On peut mettre des CLU en cascade. Si on utilise des CLU de 4 bits et on a des nombres de $n = 4k$ bits, on pose :

$$\left\{ \begin{array}{lll} p^* & = & p_3 p_2 p_1 p_0 \\ g^* & = & g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 \end{array} \right.$$

et on a toujours : $c^*_{i+1} = g^*_i + p^*_i c^*_i$ avec $c^*_i = c_{4i}$.

Une addition sur $4^2 = 16$ bits consiste à :

(1) calculer les $p^*_i$ et $g^*_i$ ;

(2) calculer les $c^*_i$ ;

(3) calculer les $c_i$.

Pour une addition sur $4^r$ bits, avec $r$ étages de CLU, on obtient un temps de calcul proportionnel au nombre d'étages, donc logarithmique.

12.2.6. *Additionneurs « carry skip » (Babbage, 1840).* Cette méthode consiste à faire un découpage en blocs de $m$ bits : $n = km$. Chaque bloc calcule le produit de ses $p_i$. Deux cas peuvent se produire :

– la retenue entrante $c_{jm}$ est propagée si le produit des $p_i$ vaut 1 ;
– la retenue sortante ne dépend pas de la retenue entrante dans le cas contraire, et son temps de calcul est donc celui du bloc.

Soient $\alpha$ le temps de traversée d'une cellule d'un bloc et $\beta$ le temps pour sauter par dessus un bloc. Le cas le pire est le suivant : une retenue apparaît dans le premier bloc, saute les $k-2$ blocs suivants et disparaît à la fin du dernier bloc. Ceci donne un temps au pire de $2\alpha n/k + (k-2)\beta$, qui est minimal pour $k = \sqrt{2\alpha n/\beta}$. Le temps au pire est alors de $2\sqrt{2\alpha\beta n} - 2\beta$, i.e. en $O(\sqrt{n})$.

Note : un bloc étant un additionneur, on peut aussi faire du carry-skip en cascades.

12.2.7. *Éviter les propagations de retenues : représentations redondantes.* Peut-on aller plus vite (au sens gagner un facteur non constant) ? D'après le théorème de Winograd (1965), non : noter que pour calculer le bit de rang $n$ du résultat, on a besoin de *tous* les bits de rang inférieur ou égal à $n$.

Le problème essentiel concernant le temps de calcul de l'addition est dû aux propagations des retenues. Une solution pour les éviter est d'utiliser une représentation redondante pour les calculs intermédiaires.

12.2.8. *Systèmes d'Avizienis (1961).* On considère une représentation en base $B$ avec les $2a+1$ chiffres signés allant de $-a$ à $+a$. Pour pouvoir représenter tous les entiers, il est nécessaire que $2a+1 \geqslant B$, et réciproquement. De plus, si $2a \geqslant B+1$ et $a \leqslant B-1$, alors on peut additionner deux entiers $x$ et $y$ par l'algorithme parallèle suivant (algorithme d'Avizienis) :

$$\left\{\begin{array}{l} t_{i+1} = \left\{\begin{array}{rl} +1 & \text{si} \quad x_i + y_i \geqslant a \\ 0 & \text{si} \quad 1-a \leqslant x_i + y_i \leqslant a-1 \\ -1 & \text{si} \quad x_i + y_i \leqslant -a \end{array}\right. \\ s_i = x_i + y_i - Bt_{i+1} + t_i \end{array}\right.$$

Note : à cause des contraintes entre $a$ et $B$, cet algorithme n'existe pas en base $B = 2$.

12.2.9. *Représentation « carry-save » (CS).* Un nombre $x$ est représenté par un couple $(a, b)$ de deux nombres écrits en binaire, tels que $x = a + b$. Une addition d'un nombre en carry-save et d'un nombre en représentation non redondante se fait à l'aide d'une chaîne de FA (le temps de calcul est donc celui d'un FA), qui réduit les 3 entrées en 2 ; il y a un décalage sur la seconde sortie, mais cela ne coûte rien en hardware. Et une addition de deux nombres en carry-save se ramène à deux additions d'un nombre en carry-save et d'un nombre en représentation binaire non redondante.

12.2.10. *Représentation « borrow-save ».* Un nombre $x$ est représenté par un couple $(x^+, x^-)$ de deux nombres écrits en binaire, tels que $x = x^+ - x^-$. La cellule de base n'est plus le full adder (FA), mais la cellule PPM :

$$\left\{\begin{array}{rcl} u & = & x \oplus y \oplus z \\ t & = & \mathrm{Maj}(x, y, \overline{z}) \end{array}\right.$$

qui calcule $x + y - z = 2t - u$. Les calculs sont très similaires à ceux du carry-save. C'est juste une histoire de complémentation.

12.3. **Multiplieurs.**

12.3.1. *Généralités concernant la complexité.* La complexité est au moins en $\log n$ (Winograd, 1967). Si on pose $X = 2^n x + 1$ et $Y = 2^n y + 1$ avec $x < 2^n$ et $y < 2^n$, on remarque que l'écriture de $XY$ fait intervenir une addition : $XY = 2^{2n}xy + 2^n(x+y) + 1$.

Brent et Kung, 1981 : si $A(n)$ et $T(n)$ sont la surface et le temps de calcul d'un multiplieur d'entiers de $n$ bits, alors il existe deux valeurs $A_0$ et $T_0$ qui ne dépendent que de la technologie, telles que :

$$\frac{A(n)}{A_0}\left(\frac{T(n)}{T_0}\right)^2 \geqslant n^2$$

et cette borne est atteinte.

Des multiplieurs basés sur la FFT permettent d'avoir $A(n).T(n)^2 = O(n^2)$ avec :

$$K.\log^2 n \leqslant T(n) \leqslant L\sqrt{n}.$$

Les multiplieurs récursifs de Luk et Vuillemin permettent d'obtenir un temps logarithmique : découpage récursif en 4 multiplications de taille $n/2$ en parallèle (3 avec la méthode de Karatsuba), et un additionneur en carry-save.

12.3.2. *Méthode de Booth.* Pour multiplier $x$ par $y$, on récrit $y$ en chiffres signés $-1$, 0, 1, avec un nombre minimal de chiffres non nuls, et la multiplication revient à faire une succession d'additions et de soustractions.

12.3.3. *Multiplication par arbre binaire.* On additionne les produits partiels $xy_i$ sous forme d'un arbre binaire pour obtenir un temps de calcul logarithmique (sous réserve que l'addition se fasse en temps constant, donc en utilisant une représentation redondante).

12.3.4. *Multiplication par réseaux cellulaires (Braun, 1963).* On a un réseau de $n^2$ cellules calculant les $x_i y_i$ (« et logique » associé à un full/half adder). Le réseau est bien régulier, mais c'est plutôt lent et ça prend de la place ; cette solution est donc destinée à disparaître. Il en existe diverses variantes.

12.3.5. *Multiplieur de Dadda.* On reprendre l'idée précédente, qui est de calculer les $x_i y_i$ et de tout additionner, mais on se débrouille pour équilibrer les additions afin d'obtenir un temps logarithmique plus une addition finale :

$$T_{\text{mult}} \approx T_{\text{add}} + (1,709\log_2 n - 1,193)T_{\text{FA}}.$$

12.4. **Retour sur les algorithmes à base d'additions et de décalages.**

12.4.1. *L'algorithme SRT et le bug du Pentium* [27]. L'algorithme de division à base d'additions (soustractions) et de décalages est celui que l'on apprend à l'école : à partir du dividende $x$ et du diviseur $y$, on détermine le prochain chiffre du quotient $q_i$ et on calcule $x' = Bx - q_i y$ (lorsque l'on pose la division, la multiplication par la base $B$ n'apparaît pas puisqu'il ne s'agit que d'un décalage) ; puis on réitère...

Lors de l'implantation en matériel, on retrouve les mêmes problèmes : la détermination de $q_i$ est compliquée (par rapport au cas moyen) lorsque l'on est proche d'un cas frontière ($x$ proche d'un multiple entier de $y$) ; idem pour la soustraction (propagation des retenues). La solution est donc d'utiliser une représentation redondante.

C'est ce qui a été choisi pour le Pentium. Il s'agit ici d'une variante d'un algorithme découvert indépendamment en 1958 par Sweeney, Robertson et Tocher, d'où le nom d'algorithme « SRT ». Le Pentium utilise la base $B = 4$ avec les chiffres $-2$, $-1$, 0, 1 et 2. Les chiffres du quotient sont choisis de telle sorte que $\left|x^{(i)}\right| \leqslant 2y/3$. Ce choix se fait à l'aide d'une table dont les entrées sont les 5 bits de poids fort du diviseur $y$ et 7 bits du dividende partiel obtenus par une conversion approchée de $x^{(i)}$ ; cf. figure 4.3 page 10 du rapport d'Intel [23], correspondant aux valeurs positives. Suivant les entrées de la table, le chiffre $q_i$ du quotient peut être 0, 1 ou 2, ce qui définit 3 zones se recouvrant partiellement (et une zone impossible, au-delà de $8y/3$). Les rectangles sont identifiés par leur sommet inférieur gauche, mais tous les points du plan hors zone impossible doivent être pris en compte dans le choix de $q_i$ ; ceci définit un certain nombre de contraintes, les valeurs indécises ayant certainement été fixées lors de la construction/optimisation du PLA (Programmable Lookup Array).

Les 5 rectangles grisés correspondent aux valeurs erronées de $q_i$ (0 au lieu de 2) ; l'ingénieur ayant implémenté la table s'est probablement dit que si le point identifiant le rectangle était dans la zone impossible, alors il s'agissait d'un cas impossible...

Note 1 : la probabilité d'erreur augmente lorsque $y$ se rapproche de la partie droite d'une colonne contenant une valeur erronée. Conséquence : les valeurs du diviseur pouvant conduire à une erreur sont une puissance de 2 (négative ou positive) multipliée par un nombre très légèrement inférieur à 3 (cas 3), à 9 (cas 1), à 15 (cas 5), à 21 (cas 2) ou à 27 (cas 4). Exemple de Thomas Nicely (octobre 1994) : 1.0/824633702441, où le diviseur a pour écriture binaire : 1011111111111111111111111011100000101001 (cas 3).

Note 2 : l'erreur est toujours assez faible. Ceci est dû au prétraitement pour garantir que $\left|x^{(0)}\right| \leqslant 2y/3$ quel que soit $y$, si bien que $1/4 \leqslant \left|x^{(0)}\right| < 1/2$, et que $\left|x^{(i)}\right|$ ne pourra pas être proche de $2y/3$ avant quelques itérations.

12.4.2. *L'algorithme BKM.* En ce qui concerne les fonctions élémentaires, il existe des variantes des algorithmes présentés précédemment utilisant une représentation redondante, avec souvent des inconvénients pour les variantes de CORDIC (facteur d'échelle non constant, itérations supplémentaires...). Une méthode intéressante est l'algorithme BKM [3].

Si $E_n$ est un nombre complexe et que l'on définit $E_{n+1} = E_n(1 + id_n 2^{-n})$, on obtient en fait une itération très proche de CORDIC, d'où l'idée d'effectuer des multiplications par des termes de la forme $1+d_n 2^{-n}$, où les $d_n$ sont des nombres complexes tels qu'une multiplication par $d_n$ se réduise à quelques additions. On considère l'itération :

$$\left\{\begin{array}{lcl} E_{n+1} & = & E_n(1 + d_n 2^{-n}) \\ L_{n+1} & = & L_n - \log(1 + d_n 2^{-n}) \end{array}\right.$$

avec $d_n \in \{0, 1, 1+i, i, i-1, -1, -1-i, -i, -i+1\}$.

Comme avec les autres algorithmes à base d'additions et de décalages, il y a deux modes :

– Mode E : si on trouve une suite $(d_n)$ telle que $L_n \to 0$, alors $E_n \to E_1 e^{L_1}$.
– Mode L : si on trouve une suite $(d_n)$ telle que $E_n \to 1$, alors $L_n \to L_1 + \log(E_1)$.

Considérons par exemple le mode E. On pose $L_n = L_n^x + iL_n^y$ et $d_n = d_n^x + id_n^y$ où $d_n^x, d_n^y \in \{-1, 0, 1\}$. On a :

$$\left\{ \begin{array}{lcl} L_{n+1}^x & = & L_n^x - \frac{1}{2}\log\left[1 + d_n^x 2^{-n+1} + \left({d_n^x}^2 + {d_n^y}^2\right)2^{-2n}\right] \\ L_{n+1}^y & = & L_n^y - d_n^y \arctan\left(\dfrac{2^{-n}}{1 + d_n^x 2^{-n}}\right). \end{array} \right.$$

En considérant l'influence respective de $d_n^x$ et de $d_n^y$ dans ces deux expressions, on choisira $d_n^x$ en fonction des premiers chiffres de $L_n^x$ et $d_n^y$ en fonction des premiers chiffres de $L_n^y$.

D'autre part, tout comme dans les autres algorithmes où le domaine de convergence était un segment, on va assurer ici la convergence dans un rectangle $R_n = [-s_n^x, r_n^x] + i[-s_n^y, r_n^y]$, où les bornes seront identifiées plus loin.

– Choix de $d_n^x$. La valeur de $L_{n+1}^x$ s'exprime en fonction de la valeur de $L_n^x$ par des segments de droite paramétrés par $d_n^x$ et $d_n^y$ (diagramme de Robertson). De plus, $d_n^x$ va être choisi de manière à rendre $\left|L_{n+1}^x\right|$ suffisamment petit (pas forcément le plus petit possible dans les zones où cela a peu d'importance, notamment pour garder de la redondance). Cependant, on ne contrôle pas ici le choix de $d_n^y$, ce qui signifie que les suites $(s_n^x)$ et $(r_n^x)$ doivent être établies de manière à ce que $L_{n+1}^x \in [-s_{n+1}^x, r_{n+1}^x]$ quelle que soit la valeur de $d_n^y$. Pour la définition de $s_n^x$, on considère $d_n^x = -1$ (pour minimiser) et $d_n^y = 1$ (pour maximiser), ce qui donne :

$$s_n^x = -\frac{1}{2}\sum_{k=n}^{\infty} \log(1 - 2^{-k+1} + 2^{-2k+1}).$$

Pour la définition de $r_n^x$, on considère $d_n^x = 1$ (pour maximiser) et $d_n^y = 0$ (pour minimiser), ce qui donne :

$$r_n^x = \sum_{k=n}^{\infty} \log(1 + 2^{-k}).$$

On définit ensuite :

$$\left\{ \begin{array}{lcl} A_n & = & -s_{n+1}^x + \frac{1}{2}\log(1 + 2^{-2n}) \\ B_n & = & r_{n+1}^x + \log(1 - 2^{-n}) \\ C_n & = & -s_{n+1}^x + \frac{1}{2}\log(1 + 2^{-n+1} + 2^{-2n+1}) \\ D_n & = & r_{n+1}^x \end{array} \right.$$

et on choisit $d_n^x$ de la façon suivante :

$$\left\{ \begin{array}{lcll} \text{si} & L_n^x < A_n & \text{alors} & d_n^x = -1 \\ \text{si} & A_n \leqslant L_n^x < B_n & \text{alors} & d_n^x = -1 \text{ ou } 0 \\ \text{si} & B_n \leqslant L_n^x < C_n & \text{alors} & d_n^x = 0 \\ \text{si} & C_n \leqslant L_n^x \leqslant D_n & \text{alors} & d_n^x = 0 \text{ ou } 1 \\ \text{si} & D_n < L_n^x & \text{alors} & d_n^x = 1. \end{array} \right.$$

– Choix de $d_n^y$. Avec des considérations similaires, on obtient :
$$s_n^y = r_n^y = \sum_{k=n}^{\infty} \arctan\left(\frac{2^{-k}}{1+2^{-k}}\right).$$
On définit :
$$\begin{cases} A'_n &= -r_{n+1}^y + \arctan\left(\frac{2^{-n}}{1-2^{-n}}\right) \\ B'_n &= r_{n+1}^y \end{cases}$$
et on choisit $d_n^y$ de la façon suivante :
$$\begin{cases} \text{si} & L_n^y < -B'_n & \text{alors} & d_n^y = -1 \\ \text{si} & -B'_n \leqslant L_n^y < -A'_n & \text{alors} & d_n^y = -1 \text{ ou } 0 \\ \text{si} & -A'_n \leqslant L_n^y < A'_n & \text{alors} & d_n^y = 0 \\ \text{si} & A'_n \leqslant L_n^y \leqslant B'_n & \text{alors} & d_n^y = 0 \text{ ou } 1 \\ \text{si} & B'_n < L_n^y & \text{alors} & d_n^y = 1. \end{cases}$$

Le domaine de convergence de l'algorithme est :
$$-0,8298... = -s_1^x \leqslant L_1^x \leqslant r_1^x = 0,8688...$$
$$-0,7497... = -s_1^y \leqslant L_1^y \leqslant r_1^y = 0,7497...$$
Note : grâce au recouvrement des domaines pour le choix de $d_n^x$ et $d_n^y$, on peut simplifier les comparaisons et ne considérer que les premiers chiffres de $L_n^x$ et $L_n^y$ pour déterminer $d_n^x$ et $d_n^y$.

## 13. Implémentation de la fonction exponentielle

Nous étudierons ici la méthode choisie par D. Defour pour sa thèse [15] et qu'il a implémentée pour la bibliothèque *crlibm* [32], avec quelques corrections. Ce n'est pas une tâche facile (pour information, cette implémentation, hors tests, représente 30 pages de description et preuves).

Un autre exemple d'implémentation de fonctions transcendantes (sans arrondi exact) est décrit dans [21].

13.1. **Introduction.** Les objectifs sont les suivants :
– toujours fournir l'arrondi correct, pour chacun des quatre modes d'arrondi ;
– borner le temps d'exécution dans le pire cas ;
– avoir un temps d'exécution en moyenne acceptable (au maximum deux fois le prix d'une évaluation relativement précise) ;
– avoir du code portable (en faisant tout de même quelques suppositions raisonnables sur l'implémentation de C : par exemple, la norme IEEE 754 doit être supportée à ce niveau, le type `double` doit correspondre à la double précision, le mode d'arrondi actif doit être le mode d'arrondi au plus près) ;
– avoir un code simple (quitte à perdre sur le plan des performances) ;
– avoir des preuves des algorithmes et de l'implémentation correspondante (d'autant plus simples que le code l'est).

La seule façon de satisfaire les premier (arrondi correct) et troisième (temps moyen acceptable) points est d'utiliser la stratégie de Ziv (cf. section 9 sur l'arrondi final). Concernant le deuxième point, il y aurait deux choix possibles :
– utiliser des théorèmes de théorie des nombres (en particulier, celui de Nesterenko et Waldschmidt), au prix d'un certain nombre de problèmes (implémentation, temps, mémoire) ;
– utiliser les résultats de tests exhaustifs, quand ils sont connus.

Pour l'exponentielle, les pires cas sont connus (obtenus par V. Lefèvre en novembre 1999, avec une seconde série de tests en 2003), donc on peut choisir la deuxième méthode.

Le calcul de l'exponentielle s'effectue de la façon suivante :

(1) Traitement des cas spéciaux.

(2) Calcul rapide d'une approximation à 68 bits près.

(3) Calcul plus précis (si le dilemme du fabricant de tables se produit).

13.2. **Remarque préliminaire.** L'exponentielle étant une fonction positive, l'arrondi vers 0 se ramène au cas de l'arrondi vers $-\infty$.

13.3. **Cas spéciaux.** Les bornes liées à l'*underflow* et à l'*overflow* sont définies de la manière suivante :

$$b_{\mathrm{u}} = \vartriangle(\log(2^{-1075})) = -745,1332...$$
$$b_{\mathrm{o}} = \triangledown(\log((1-2^{-53}).2^{1024})) = 709,7827...$$

Les différents cas traités (regroupés ici pour l'ensemble des modes d'arrondi) sont :
– si $x$ est NaN, alors $\exp(x)$ vaut NaN ($x+x$ est renvoyé pour générer une exception opération invalide) ;
– si $x = \pm\infty$, alors on renvoie $+\infty$ ou $+0$, sans générer d'exception ;
– si $x > b_{\mathrm{o}}$, alors un *overflow* est généré (implémenté par un `DBL_MAX * DBL_MAX` en arrondi au plus près, et les autres exceptions le sont de manière similaire) ;
– si $x < b_{\mathrm{u}}$, alors un *underflow* est généré ;
– en arrondi au plus près, si $|x| \leqslant 2^{-54}$, alors on renvoie 1, avec une exception *inexact* si $x \neq 0$ (tests similaires dans les autres modes d'arrondi).

Note 1 : les *overflow* et *underflow* des calculs intermédiaires devront être évités par la suite.

Note 2 : l'implémentation *crlibm* effectue certains des tests sur la représentation en mémoire des flottants en double précision (type `double`), i.e. ce sont des tests sur des entiers. Ce n'est pas forcément un bon choix, que ce soit au niveau de la portabilité (la représentation des flottants en mémoire n'est pas normalisée et la taille des types entiers peut varier d'une implémentation à une autre) ou de la rapidité (cela demande un stockage du flottant en mémoire), surtout que la précision étendue ne pose pas vraiment de problème à ce niveau.

13.4. **Calcul rapide.** Le calcul à 68 bits près s'effectue en 4 étapes :

(1) Première réduction d'argument : on calcule un entier $k$ et une approximation $r$ tels que $x \approx k.\log(2) + r$, avec $r \in [-\log(2)/2, +\log(2)/2]$.

(2) Seconde réduction d'argument, par une table indexée par les premiers bits de $r$.

(3) Évaluation polynomiale.

(4) Reconstruction.

13.4.1. *Première réduction d'argument.* Il s'agit d'une réduction d'argument de type additif : on calcule un entier $k$ et deux flottants $r_{\mathrm{hi}}$ et $r_{\mathrm{lo}}$ représentant un réel $r = r_{\mathrm{hi}} + r_{\mathrm{lo}}$ tels que

$$x = k.\log(2) + r.(1+\varepsilon)$$

avec $|r| < \frac{1}{2}\log(2)$. Il y a donc une cancellation importante pour obtenir $r$. En plus de la cancellation normale ($r$ étant généralement de l'ordre de $0,5$), la véritable cancellation peut être encore plus grande et donner une valeur de $r$ très petite par rapport à $0,5$. Mais seule l'erreur absolue sur $r$ est ici importante, puisque l'on va évaluer $e^r$, qui est de l'ordre de 1, donc avec une erreur absolue fixée.

Grâce aux bornes de l'*overflow* et de l'*underflow*, le calcul est assez simple...

```
/* log(2) = 1.62E42FEFA39EF35793C7673007... * 2^(-1) */
#define ln2_hi 1.62E42FEFA38P-1  /* 42 bits de poids fort de log(2) */
#define ln2_lo 1.EF35793C7673P-45        /* bits suivants de log(2) */
#define inv_ln2 1.715476533245FP0      /* approximation de 1/log(2) */

#define DOUBLE2INT(i, d) \
  { double t = ((d) + 3.P51); i = LO(t); }
/* où LO() prend la partie basse dans la représentation en mémoire */

double r_hi, r_lo, s_hi, s_lo;
int k;

DOUBLE2INT (k, x * inv_ln2);  /* arrondi au plus près */
if (k != 0)
  { s_hi = x - ln2_hi * k;  /* exact, cf. Sterbenz */
    s_lo =   - ln2_lo * k;
    Fast2SumCond (r_hi, r_lo, s_hi, s_lo); }
else
  { r_hi = x;
    r_lo = 0; }
```

On peut prouver les propriétés suivantes (on note ici $\varepsilon_k$ une quantité bornée en valeur absolue par $2^k$) :

$$\texttt{ln2\_hi} + \texttt{ln2\_lo} = \log(2).(1+\varepsilon_{-102})$$

et

$$-1075 \leqslant k \leqslant 1025$$

si bien que $k$ s'écrit sur au plus 11 bits et la multiplication `ln2_hi * k` s'effectue exactement. On peut également prouver que les conditions du théorème de Sterbenz s'appliquent pour la soustraction, si bien que celle-ci s'effectue aussi exactement. On a également :

$$\texttt{s\_lo} = -\texttt{ln2\_lo} \times k + \varepsilon_{-91}.$$

La macro `Fast2SumCond` effectue l'algorithme FastTwoSum avec un test sur les exposants des deux entrées (i.e. c'est l'équivalent de TwoSum, cf. section 10.4). On a donc :

$$\exp(x) = 2^k \exp(\texttt{r\_hi} + \texttt{r\_lo} + \varepsilon_{-91}).$$

13.4.2. *Seconde réduction d'argument.* Le nombre $r = \texttt{r\_hi} + \texttt{r\_lo}$ est toujours trop grand pour être utilisé dans une évaluation polynomiale. Une seconde réduction d'argument est donc nécessaire.

Cette réduction d'argument s'effectue à l'aide d'une table indexée par les premiers bits de $r$. Utiliser 8 bits (dont le signe) permet d'avoir une table de $\lceil 2^8 \log(2) \rceil \times 16 = 2848$ octets.

Si on note $i$ le flottant formé par les 8 premiers bits et on calcule $s = r - i$, alors $\exp(r) = \exp(i). \exp(s)$, où l'approximation de $\exp(i)$ sur 105 bits (en fait, 70 bits suffiraient) est lue dans la table et $s$, vérifiant $|s| \leqslant 2^{-9}$, sera représenté par $\texttt{s\_hi} + \texttt{s\_lo}$, obtenus par FastTwoSum. On a alors :

$$\exp(x) = 2^k.(\texttt{ex\_hi} + \texttt{ex\_lo}). \exp(\texttt{s\_hi} + \texttt{s\_lo} + \varepsilon_{-91}).(1 + \varepsilon_{-105}).$$

13.4.3. *Évaluation polynomiale.* L'évaluation de $\exp(s)$, où $s = \texttt{s\_hi} + \texttt{s\_lo}$, se fait comme suit :

$$1 + s + \frac{1}{2}s^2 + s^3.P$$

avec `P = c0 + s_hi * (c1 + s_hi * (c2 + s_hi * c3))`, correspondant à l'évaluation approchée d'un polynôme de degré 3 en `s_hi`.

Exemple de majoration d'erreurs, en notant de $P_0$ à $P_5$ les résultats intermédiaires de l'évaluation de $P$ $(= P_5)$ :

- $|P_0| \leqslant 2^{-18}$, donc $P_0 = (\texttt{s\_hi} \times \texttt{c3}) + \varepsilon_{-71}$.
- $|P_1| \leqslant 2^{-6}$, donc $P_1 = (\texttt{c2} + P_0) + \varepsilon_{-59}$.
- $|P_2| \leqslant 2^{-15}$, donc $P_2 = (\texttt{s\_hi} \times P_1) + \varepsilon_{-68}$.
- $|P_3| \leqslant 2^{-4}$, donc $P_3 = (\texttt{c1} + P_2) + \varepsilon_{-57}$.
- $|P_4| \leqslant 2^{-13}$, donc $P_4 = (\texttt{s\_hi} \times P_3) + \varepsilon_{-66}$.
- $|P_5| \leqslant 2^{-2}$, donc $P_5 = (\texttt{c0} + P_4) + \varepsilon_{-55}$.

On en déduit l'erreur sur $P$ :

$$\varepsilon_{-55} + \varepsilon_{-66} + 2^{-9}\varepsilon_{-57} + 2^{-9}\varepsilon_{-68} + 2^{-18}\varepsilon_{-59} + 2^{-18}\varepsilon_{-71}$$

que l'on peut majorer par $2^{-55} + 2^{-64}$.

13.4.4. *Reconstruction.* En résumé, $\exp(x)$ est approché par $2^k$ multiplié par :

$$(\texttt{ex\_hi} + \texttt{ex\_lo}). \left(1 + \texttt{s\_hi} + \texttt{s\_lo} + \frac{1}{2}(\texttt{s\_hi} + \texttt{s\_lo})^2 + (\texttt{s\_hi} + \texttt{s\_lo})^3.P\right)$$

avec une erreur relative $\varepsilon_{-77}$.

Dans l'expression ci-dessus, on développe et élimine les termes non significatifs par rapport à la précision finale voulue (68 bits). Dans le calcul, il faudra utiliser l'algorithme de Dekker (pour les multiplications) et l'algorithme FastTwoSum (pour les additions), sauf si on peut se permettre de perdre de la précision.

13.5. **Test si l'arrondi est possible.** À ce niveau, on a :

$$\exp(x) = 2^k(\texttt{y\_hi} + \texttt{y\_lo} + \varepsilon_{-70})$$

où `y_hi` et `y_lo` ont été obtenus par un FastTwoSum.

La multiplication par $2^k$ (avec $k$ entier) est normalement exacte, mais si le résultat final est un dénormalisé, il est possible que cette multiplication fasse perdre de la précision ! Ce cas est traité à part.

Le test de l'arrondi lui-même peut se faire avant la multiplication par $2^k$ puisque celle-ci ne change pas la mantisse du résultat. Une solution intuitive serait de considérer l'erreur minimale (incluant `y_lo`) et de l'ajouter à `y_hi` pour voir si on obtient le même résultat, et faire de même avec l'erreur maximale. Cela demande beaucoup d'opérations. Un test plus simple consiste à faire : `y_hi == y_hi + y_lo * errn` avec $\texttt{errn} = 1+2^{-15}$, en remarquant que :

– on a $\texttt{y\_hi} \geqslant 1/\sqrt{2}$ ;

– le TMD ne pourra se produire que si $|\texttt{y\_lo}|$ est suffisamment proche de $\mathrm{ulp}(\texttt{y\_hi})/2$ ;

ce qui permet de transformer l'erreur absolue $\varepsilon_{-70}$ en erreur relative à `y_lo`. En effet, le TMD ne peut se produire que lorsque $|\texttt{y\_lo}| \geqslant \frac{1}{4}\mathrm{ulp}(\texttt{y\_hi}) \geqslant 2^{-55}$, donc $2^{-70} \leqslant 2^{-15}\,|\texttt{y\_lo}|$.

13.6. **Calcul plus précis.** Il faut faire du calcul en multiprécision (les expansions de longueur 2 ne sont pas suffisantes).

13.6.1. *Précision nécessaire.* Les résultats des tests exhaustifs donnent un pire cas à 158 bits, tous modes d'arrondi confondus. À noter que les plus gros pires cas sont en fait très particuliers, liés à l'approximation $\exp(x) \approx 1 + x + \frac{1}{2}x^2$ pour $x$ suffisamment proche de 0 (en gros, $|x| < 2^{-29}$). Une solution serait de tabuler ces pires cas ou de faire un traitement particulier, de façon à calculer à une précision intermédiaire de 120 bits environ au lieu de 160. Ce n'est pas ce qui a été choisi ici.

Les algorithmes seront essentiellement basés sur des additions et des multiplications. L'implémentation utilisera la bibliothèque SCSLib (développée par D. Defour, C. Daramy et F. de Dinechin, dans le projet Arénaire [33, 16]), qui satisfait les conditions requises ultérieurement :

$$a + b = (a \oplus b).(1 + \varepsilon_{-210})$$

et

$$a \times b = (a \otimes b).(1 + \varepsilon_{-207}).$$

13.6.2. *L'algorithme.*

(1) Pas de gestion des cas spéciaux (ils ont déjà été traités).

(2) Réduction d'argument : on calcule l'argument réduit $r$ et l'entier $k$ tels que :

$$r = \frac{x - k.\log(2)}{512} \quad \text{avec} \quad \frac{-\log(2)}{1024} \leqslant r \leqslant \frac{\log(2)}{1024}.$$

(3) Évaluation polynomiale :

$$\exp(r) = P(r).(1 + \varepsilon_{-179}), \quad \text{où} \quad P(r) = 1 + r + c_2 r^2 + ... + c_{12} r^{12}.$$

(4) Reconstruction : $\exp(x) = 2^k . \exp(r)^{512} . (1 + \varepsilon_{-170})$, où l'élévation à la puissance 512 se fait en élevant 9 fois $\exp(r)$ au carré.

## Références


[1] Abramowitz, M., and Stegun, I. A. *Handbook of Mathematical Functions*. Dover, 1973.

[2] Arnold, D. N. Some disasters attributable to bad numerical computing. `http://www.ima.umn.edu/~arnold/disasters/`, 1998.

[3] Bajard, J.-C., Kla, S., and Muller, J.-M. BKM : A new hardware algorithm for complex elementary functions. *IEEE Trans. Comput. 43*, 8 (1994), 955–963.

[4] Baker, A. *Transcendental Number Theory*. Cambridge University Press, 1975.

[5] Boldo, S., and Daumas, M. Properties of the subtraction valid for any floating point system. In *7th International Workshop on Formal Methods for Industrial Critical Systems* (Málaga, Spain, 2002), pp. 137–149. `http://www.inrialpes.fr/vasy/fmics/workshop-7/proceedings.pdf`.

[6] Borwein, J. M., and Borwein, P. B. The arithmetic-geometric mean and fast computation of elementary functions. *SIAM Review 26*, 3 (July 1984), 351–366.

[7] Brent, R. P. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. In *Analytic Computational Complexity* (New York, 1975), J. F. Traub, Ed., Academic Press, pp. 151–176. `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb028.ps.gz`.

[8] Brent, R. P. The complexity of multiple-precision arithmetic. In *The Complexity of Computational Problem Solving* (1976), R. S. Anderssen and R. P. Brent, Eds., University of Queensland Press, pp. 126–165. `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Richard.Brent/rpb032.ps.gz`.

[9] Brent, R. P., and Kung, H. T. Fast Algorithms for Manipulating Formal Power Series. *J. ACM 25*, 4 (1978), 581–595.

[10] Chudnovsky, D. V., and Chudnovsky, G. V. Computer algebra in the service of mathematical physics and number theory. In *Computers in mathematics (Stanford, CA, 1986)* (New York, 1990), D. V. Chudnovsky and R. D. Jenks, Eds., Marcel Dekker, pp. 109–232.

[11] Daumas, M. Multiplications of floating point expansions. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic* (Adelaide, Australia, 1999), I. Koren and P. Kornerup, Eds., IEEE Computer Society Press, Los Alamitos, CA, pp. 250–257. `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-39.ps.Z`.

[12] Daumas, M., and Finot, C. Algorithm, proof and performances of a new division of floating-point expansions. Rapport de recherche 3771, Institut National de Recherche en Informatique et en Automatique, 1999.

[13] Daumas, M., and Finot, C. Division of floating point expansions with an application to the computation of a determinant. *Journal of Universal Computer Science 5*, 6 (1999), 323–338. `http://www.jucs.org/jucs_5_6/division_of_floating_point`.

[14] Daumas, M., Rideau, L., and Théry, L. A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. In *Theorem Proving in Higher Order Logics : 14th International Conference* (Sept. 2001), no. 2152 in LNCS, Springer-Verlag.

[15] Defour, D. *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. Thèse de doctorat, École Normale Supérieure de Lyon, Sept. 2003.

[16] Defour, D., and de Dinechin, F. Software carry-save fast multiple-precision algorithms. Rapport de recherche RR2002-08, Laboratoire de l'Informatique du Parallélisme, Lyon, Feb. 2002.

[17] Dekker, T. J. A floating-point technique for extending the available precision. *Numerische Mathematik 18*, 3 (1971), 224–242.

[18] Finot-Moreau, C. *Preuves et algorithmes utilisant l'arithmétique flottante normalisée IEEE*. Thèse de doctorat, École Normale Supérieure de Lyon, July 2001.

[19] Haible, B., and Papanikolaou, T. Fast multiprecision evaluation of series of rational numbers. Technical Report TI-7/97, Darmstadt University of Technology, 1997. `http://www.informatik.th-darmstadt.de/TI/Mitarbeiter/papanik/`.

[20] Hanrot, G., Quercia, M., and Zimmermann, P. Speeding up the division and square root of power series. Rapport de recherche 3973, Institut National de Recherche en Informatique et en Automatique, 2000. `http://www.inria.fr/RRRT/RR-3973.html`.

[21] Harrison, J., Kubaska, T., Story, S., and Tang, P. T. P. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal 1999 Q4* (1999). `http://www.intel.com/technology/itj/q41999/articles/art_5.htm`.

[22] IEEE standard for binary floating-point arithmetic. Tech. Rep. ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985 : IEEE Standards Board, approved July 26, 1985 : American National Standards Institute, 18 pages.

[23] Intel Corporation. Statistical analysis of floating point flaw in the pentium processor, Nov. 1994. `http://citeseer.nj.nec.com/366418.html`.

[24] Knuth, D. E. *The Art of Computer Programming*, vol. 2. Addison-Wesley, 1973.

[25] Kornerup, P., and Muller, J.-M. Choosing starting values for Newton-Raphson computation of reciprocals, square-roots and square-root reciprocals. Research Report 4687, Institut National de Recherche en Informatique et en Automatique, Jan. 2003.

[26] Lefèvre, V. An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. Rapport de recherche RR1997-18, Laboratoire de l'Informatique du Parallélisme, Lyon, 1997.

[27] Muller, J.-M. Algorithmes de division pour microprocesseurs : illustration à l'aide du « bug » du Pentium. *Technique et Science Informatiques 14*, 8 (1995). `ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1995/RR1995-06.ps.Z`.

[28] Muller, J.-M. *Elementary Functions. Algorithms and Implementation*. Birkhauser, 1997. 232 pages.

[29] Nesterenko, Y. V., and Waldschmidt, M. On the approximation of the values of exponential function and logarithm by algebraic numbers (in russian). *Mat. Zapiski 2* (1996), 23–42.

[30] Ng, K. C. Argument reduction for huge arguments : Good to the last bits. Technical report, SunPro, 1992. Can be obtained by sending an e-mail to the author : `kwok.ng@eng.sun.com`.

[31] Priest, D. M. Algorithms for arbitrary precision floating point arithmetic. In *Proceedings of the 10th Symposium on Computer Arithmetic* (Grenoble, France, 1991), P. Kornerup and D. Matula, Eds., IEEE Computer Society Press, pp. 132–144.

[32] Projet Arénaire. Bibliothèque crlibm. `http://perso.ens-lyon.fr/catherine.daramy/crlibm.html`.

[33] Projet Arénaire. The software carry-save multiple-precision library. `http://www.ens-lyon.fr/LIP/Arenaire/News/SCSLib/`. Version 1.4.1.

[34] Remez, E. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *Comptes-rendus de l'Académie des Sciences, Paris*, 198 (1934).

[35] Shewchuk, J. R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry* (1997), vol. 18, pp. 305–363. `http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps`.

[36] Stehlé, D., Lefèvre, V., and Zimmermann, P. Worst cases and lattice reduction. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (Arith'16)* (2003), J.-C. Bajard and M. Schulte, Eds., pp. 142–147.

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399