N° d'ordre : 142

N° attribué par la bibliothèque : 00ENSL0142

THÈSE

présentée devant

L'ÉCOLE NORMALE SUPÉRIEURE DE LYON

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique
au titre de l'école doctorale MathIf

par **Vincent Lefèvre**

MOYENS ARITHMÉTIQUES POUR UN CALCUL FIABLE

Présentée et soutenue publiquement le 10 janvier 2000

Après avis de : Monsieur Paul Zimmermann
Monsieur Jean-Marie Chesneaux

Devant la commission d'examen formée de :

Monsieur Paul Zimmermann
Monsieur Jean-Marie Chesneaux
Madame Brigitte Vallée
Monsieur Peter Kornerup
Monsieur Jean-Michel Muller (directeur de thèse)

Thèse préparée à l'École Normale Supérieure de Lyon
au sein du Laboratoire de l'Informatique du Parallélisme.

Remerciements

Je tiens tout d'abord à remercier les membres du jury :

> Paul Zimmermann, Directeur de Recherches à l'INRIA, et Jean-Marie Chesneaux, Professeur à l'Université Paris 6, pour avoir accepté de rapporter cette thèse et pour leurs commentaires sur mon manuscrit.
>
> Brigitte Vallée, Professeur à l'Université de Caen, et Peter Kornerup, Professeur à l'Université d'Odense (Danemark), qui ont bien voulu participer au jury de ma thèse.
>
> Et mon directeur de thèse Jean-Michel Muller, Directeur de Recherches au CNRS, qui m'a beaucoup aidé pendant toute cette thèse.

Je tiens également à remercier Jean-Paul Allouche et Valérie Berthé pour les discussions que nous avons eues à propos du théorème des trois distances, ainsi que Dominique Michelucci, Helymar Balza-Gomez et Jean-Michel Moral à propos de la minoration de la distance d'une droite/courbe à $\mathbb{Z}^2$.

Je remercie tous les membres du laboratoire, notamment pour m'avoir laissé utiliser leur machine pour mes calculs, qui ont tourné depuis plusieurs années pendant mon stage de DEA et ma thèse, et plus particulièrement les membres de l'équipe Arénaire : Claire, Arnaud, Philippe, Marc, et encore une fois Jean-Michel.

Je voudrais aussi remercier tous ceux qui m'ont aidé dans les groupes de discussion sur Usenet.

Merci également à tous les développeurs de logiciels libres que j'ai utilisés pour cette thèse, en particulier Perl, zsh, screen, gcc, et le merveilleux éditeur Zap (sur Acorn / RISC OS).

Merci aussi à ma famille, à Sébastien Blondeel pour m'avoir fait connaître Perl et zsh, et à tous ceux que j'ai oubliés et qui m'ont aidé à la réalisation de cette thèse.

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

Introduction

Les programmeurs ayant besoin d'effectuer des calculs (additions, multiplications, divisions, logarithmes, sinus, etc.), en particulier pour des logiciels scientifiques, s'attendent à un certain nombre de qualités :

– rapidité des calculs ;
– résultats précis et cohérents ;
– portabilité des programmes et reproductibilité des calculs : les programmeurs espèrent qu'un même calcul exécuté sur deux machines différentes donnera le même résultat.

Pour avoir des résultats exacts ou très précis, le programmeur peut effectuer du calcul symbolique, utiliser des arithmétiques exactes, ou faire de la multiprécision, mais ces méthodes sont souvent beaucoup trop lentes. Les systèmes à virgule flottante sont presque toujours utilisés pour représenter les nombres réels dans les ordinateurs, car ils sont un bon compromis entre la rapidité des calculs, la précision, l'amplitude des valeurs représentables et la simplicité des algorithmes. Une étude sur l'arithmétique à virgule flottante est par exemple présentée dans [21].

Un *système à virgule flottante* est défini par sa base b (en général, $b = 2$, mais certaines machines utilisent la base 16, et les calculatrices travaillent en base 10, ainsi que le logiciel Maple), une longueur de mantisse n et une plage d'exposants de $E_{\min}$ à $E_{\max}$. Dans un tel système, un nombre x, appelé *nombre machine*, est représenté par une *mantisse* $m_x = x_0.x_1x_2...x_{n-1}$, qui est un nombre en base b à n chiffres vérifiant $0 \leqslant m_x < b$, un *signe* $s_x = \pm 1$ et un *exposant* E_x, entier compris entre $E_{\min}$ et $E_{\max}$:

$$x = s_x \times m_x \times b^{E_x}.$$

En pratique, la représentation doit être *normalisée* : la mantisse d'un nombre non nul doit toujours être supérieure ou égale à 1. Lorsque la base b est 2, le premier chiffre de la mantisse est donc toujours égal à 1, et n'a pas besoin d'être représenté en mémoire. Par conséquent, une représentation spéciale doit être choisie pour zéro. Le système peut également contenir des nombres dits *dénormalisés* : il s'agit des nombres ayant pour exposant $E_{\min} - 1$ et dont le premier chiffre de la mantisse, représenté, peut être égal à 0 ; nous ne reparlerons pas de ces nombres par la suite.

De telles représentations sont spécifiées par une norme, la norme IEEE-754, définie en 1985 et maintenant implantée sur la plupart des ordinateurs du marché ; en particulier, $b = 2$, $n = 24$, $E_{\min} = -126$ et $E_{\max} = +127$ pour la simple précision, et $b = 2$, $n = 53$, $E_{\min} = -1022$ et $E_{\max} = +1023$ pour la double précision. Une autre norme, la norme IEEE-854, généralise les concepts de la norme IEEE-754 à d'autres bases que 2 (en particulier la base 10, très utilisée dans les calculettes). Mais le format de représentation ne suffit pas à lui seul à spécifier complètement l'arithmétique. En effet, la somme, le produit, le quotient de deux nombres machine, ou le logarithme, le sinus, etc. d'un nombre machine n'est en général pas un nombre machine : le résultat de l'opération doit être *arrondi*. La norme IEEE-754 définit quatre modes d'arrondi (les trois premiers sont appelés *modes d'arrondi dirigés*) :

– arrondi vers $-\infty$: $\triangledown(x)$ est le plus grand nombre machine inférieur ou égal à x ;
– arrondi vers $+\infty$: $\vartriangle(x)$ est le plus petit nombre machine supérieur ou égal à x ;
– arrondi vers 0 : $\mathcal{Z}(x)$ vaut $\triangledown(x)$ si $x \geqslant 0$, et $\vartriangle(x)$ si $x < 0$.
– arrondi au plus près : $\mathcal{N}(x)$ est le nombre machine le plus proche de x (avec une convention spéciale si x est exactement entre deux nombres machine).

Lorsque le programmeur veut effectuer une opération, il choisit un de ces quatre modes d'arrondi, qui est alors appelé « mode d'arrondi actif ». Notons-le $\diamond$. Soient x et y deux nombres machine. La norme IEEE-754 exige que pour l'opération $x \star y$ (où $\star$ est $+$, $-$, $\times$ ou $\div$) ou $\sqrt{x}$, le résultat obtenu soit toujours $\diamond(x \star y)$ ou $\diamond(\sqrt{x})$, comme si le résultat avait été calculé exactement, et ensuite arrondi. L'arrondi *exact* d'un nombre réel positif de mantisse $1.b_1b_2b_3...$ (où le nombre de bits nuls est infini) 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 $-\infty$	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^{n-1} à cette mantisse (ce qui peut nécessiter un changement d'exposant, ou même provoquer un *overflow* si le nombre obtenu n'est pas représentable).

Les opérations vérifiant cette propriété sont dites *exactement arrondies* (ou *correctement arrondies*). Outre la précision, une telle propriété a des conséquences très intéressantes :

– elle assure une compatibilité totale (le résultat d'une opération ne dépend pas de l'algorithme utilisé) : le même programme donnera exactement les mêmes résultats sur des ordinateurs différents, tout du moins si l'ordre des opérations du code exécutable est le même ;

– de nombreux algorithmes peuvent utiliser cette propriété (et être prouvés en utilisant cette propriété), par exemple pour avoir une précision arbitraire [42], pour obtenir l'arrondi exact d'une somme de plusieurs nombres à virgule flottante [41, 24], ou pour prendre des décisions en géométrie algorithmique ;
– on peut facilement implémenter une arithmétique d'intervalles [28, 29], ou plus généralement on peut obtenir une borne inférieure ou supérieure d'un résultat exact d'une séquence d'opérations arithmétiques.

Par exemple, en utilisant les propriétés de l'arrondi exact, W. Kahan a montré que le calcul de

$$z = \frac{x}{\sqrt{x^2 + y^2}}$$

en arrondi au plus près sur une machine conforme à la norme donne toujours un résultat compris entre -1 et 1 (cela est également vrai pour l'arrondi vers $+\infty$, mais cela ne l'est plus pour l'arrondi vers $-\infty$ ou vers 0 — les preuves sont données en annexe, section A.1). Cette propriété est importante, car elle est vraie en arithmétique exacte, et un programmeur peut ainsi être tenté de l'utiliser dans son programme. Mais si elle n'était pas vérifiée, le programme pourrait se retrouver dans un état totalement incohérent.

Malheureusement, la norme IEEE-754 ne spécifie pas le comportement des fonctions élémentaires ($\exp$, $\log$, $\sin$, $\cos$, $\tan$, $\arctan$), car le problème est bien plus complexe, et on a longtemps cru qu'une telle exigence était impossible à satisfaire. Notre but est de montrer que l'on peut calculer ces fonctions avec arrondi exact avec un coût raisonnable (en temps et en mémoire).

Lorsque l'on veut évaluer une fonction mathématique en un point donné, avec un arrondi exact, on calcule d'abord une approximation du résultat sur m chiffres (où m est un entier choisi par celui qui implémente l'algorithme), puis on arrondit ce résultat au format virgule flottante désiré. Tout le problème est de savoir si en arrondissant cette approximation, on obtiendra la valeur que l'on aurait en arrondissant le résultat exact. Ce problème est connu sous le nom du *dilemme du fabricant de tables* (en anglais, *Table Maker's Dilemma* — TMD), car il s'est posé à l'origine aux éditeurs de tables de valeurs de fonctions numériques.

Le dilemme du fabricant de tables est indépendant de la représentation choisie (en particulier, de la base b du système à virgule flottante utilisé[1]). Pour bien comprendre cela, faisons d'abord abstraction de la représentation en base 2, et considérons le problème d'un point de vue géométrique.

Soit un nombre machine x, et $y = f(x)$, où f est une fonction mathématique quelconque ($\exp$, $\log$, $\sin$, etc.). On cherche à arrondir exactement y en un nombre machine, suivant l'un des quatre modes d'arrondi, fixé ici. Nous savons approcher $f(x)$ avec une certaine erreur maximale (que l'on a choisie),

[1]Mais les valeurs pour lesquelles il se produit dépendent de b et de n. De plus, effectuer des changements de représentation intermédiaires ne résoudrait pas le problème.

et nous voulons savoir à quelle condition, en fonction de cette erreur, le TMD peut se produire.

Considérons l'ensemble des nombres machine x_k, et les intervalles I_k contenant les nombres réels ayant le même arrondi. Supposons que la distance entre y et une frontière de ces intervalles soit égale à un nombre ε, et que l'on approche y avec une erreur inférieure ou égale à ε'. Cette approximation doit être à une distance de la frontière d'au plus ε' pour que l'on puisse être sûr de pouvoir arrondir correctement (en effet, on ne sait pas *a priori* où se situe la vraie valeur). Pour être sûr de pouvoir arrondir correctement dans tous les cas, il faut avoir $\varepsilon \geqslant 2\varepsilon'$. Alors il faudra calculer y avec une erreur inférieure à $\varepsilon/2$ (qui peut être très petit, comparé à la distance entre deux nombres représentables), sinon il est possible qu'on ne sache pas comment arrondir.

Dans l'exemple suivant, on choisit l'arrondi au plus proche. Le point y représente la valeur *réelle* et le point y' une valeur *calculée* possible (sa position dépend de l'algorithme de calcul utilisé). Sur la figure 1 ci-dessous, on voit que le réel y n'a pas été calculé à une précision suffisante : on ne sait pas s'il faut arrondir à x_{k+2} ou x_{k+3}.

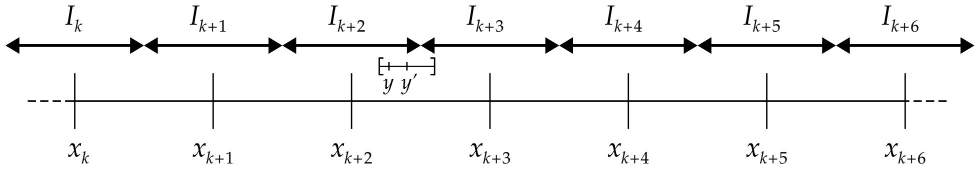

FIG. 1: Dilemme du fabricant de tables — x_j *: nombres machine ;* I_j *: intervalles des réels s'arrondissant en* x_j *(arrondi au plus près) ; y : valeur réelle* $f(x)$ *; y' : valeur calculée, approchant* $f(x)$*.*

Prenons l'exemple de la base 10, avec 5 chiffres de mantisse, l'arrondi actif étant l'arrondi vers $-\infty$. Avec la fonction cosinus et le nombre machine 1.6406, on obtient : $\cos 1.6406 = -0.069747000000219...$ L'arrondi du résultat exact dans le système choisi est donc : -0.069748. Mais si l'approximation calculée est -0.069746999999, on obtient en arrondissant cette approximation : -0.069747, ce qui est incorrect.

Maintenant, repassons en base 2. Soit m un entier supérieur à n. Supposons que l'on veuille $f(x)$ arrondi à n bits, sachant qu'on peut calculer sa valeur avec une erreur inférieure à 2^{-m} sur la mantisse du résultat. L'arrondi exact ne sera pas toujours garanti si la mantisse de y est à une distance d'une frontière entre deux arrondis inférieure à 2^{1-m}, i.e. si y a une mantisse de la forme :

– avec arrondi vers $+\infty$, $-\infty$ ou 0 (arrondi dirigé) :

$$\overbrace{\underbrace{1.xxxx...xx}_{n\text{ bits}}0000...00}^{m\text{ bits}}xxx...$$

ou

$$\overbrace{\underbrace{1.xxxx...xx}_{n\text{ bits}}1111...11}^{m\text{ bits}}xxx...$$

– avec arrondi au plus proche :

$$\overbrace{\underbrace{1.xxxx...xx}_{n\text{ bits}}1000...00}^{m\text{ bits}}xxx...$$

ou

$$\overbrace{\underbrace{1.xxxx...xx}_{n\text{ bits}}0111...11}^{m\text{ bits}}xxx...$$

c'est-à-dire si y est suffisamment proche d'un nombre machine (cas de l'arrondi dirigé) ou du milieu de deux nombres machine consécutifs (cas de l'arrondi au plus proche). Les algorithmes devant être valables pour les deux types de mode d'arrondi, on peut regrouper les deux cas, quitte à ce qu'ils soient distingués au moment voulu, et cela revient à dire que y est suffisamment proche d'un nombre machine sur $n+1$ bits de mantisse.

Notre principal problème est de savoir pour quelles valeurs de m ce problème se produit, s'il existe une valeur maximale de m pour lequel il se produit (quel que soit le nombre machine x), et dans ce cas, d'estimer cette valeur maximale. Si cette valeur n'est pas trop grande, alors on pourra arrondir la fonction f exactement.

Voici un exemple pour la double précision et arrondi dirigé, où m est très grand : pour

$$\begin{aligned} x &= 0.011111111001110110011101110011100111010000111101101101 \\ &= \frac{8980155785351021}{18014398509481984}, \end{aligned}$$

$\sin x$ est égal à :

$$0.011110100110010101000001110011000011000100011010010101\;1\,1^{65}\;0000...$$

où 1^{65} représente 65 chiffres « 1 » consécutifs. En considérant la réciproque, nous obtenons un autre exemple : pour

$$\begin{aligned} x &= 0.011110100110010101000001110011000011000100011010010110 \\ &= \frac{4306410053968715}{9007199254740992}, \end{aligned}$$

$\arcsin x$ est égal à :

$$0.011111111001110110011101110011100111010000111101101101\;0\,0^{64}\;1000...$$

Le dilemme du fabricant de tables intervient aussi pour les opérations arithmétiques ($+$, $-$, $\times$ ou $\div$) et la racine carrée, mais ces cas sont relativement simples, et arrondir exactement ne pose pas de problème, notamment dans le cas de l'addition et de la soustraction. Considérons par exemple le cas de la division : $z = x/y$, où x et y sont des nombres machine positifs, et z le résultat exact (nombre réel). Supposons que le mode d'arrondi actif soit un mode d'arrondi dirigé (l'arrondi au plus près est à peu près équivalent). On peut écrire $z = z_0 + z_1$, où z_0 est l'arrondi de z (en particulier, un nombre machine). Puisque la mantisse du résultat ne dépend pas des exposants, on peut supposer que y et z_0 sont d'exposant $n - 1$, i.e. ce sont des entiers tels que $2^{n-1} \leqslant y, z_0 < 2^n$; x est alors aussi un entier. On a : $x - yz_0 = yz_1$, où $x - yz_0$ est un entier. Supposons que le quotient réel ne soit pas un nombre machine. Alors $|x - yz_0| \geqslant 1$. Puisque $y < 2^n$, alors $|z_1| > 2^{-n}$, i.e. $m \approx 2n$ suffit pour arrondir exactement. Une étude détaillée a été effectuée par Iordache et Matula [23]. D'autre part, puisque la borne sur m est connue, le cas où le quotient réel est un nombre machine peut être traité avec la même méthode : s'il y a plus de m_0 bits identiques consécutifs après la mantisse de n bits du résultat alors que le maximum (pour un quotient non exact) est m_0, alors le quotient réel est forcément un nombre machine. Le cas de la racine carrée est similaire. W. Kahan a proposé une méthode pour vérifier si pour une implémentation, la racine carrée est arrondie exactement ; cette méthode consiste à générer des pires cas en calculant des racines carrées dyadiques de petits entiers [25]. M. Parks a détaillé cette méthode pour la multiplication, la division et la racine carrée [40].

En 1882, Lindemann a démontré que l'exponentielle d'un nombre algébrique (éventuellement complexe) non nul n'est pas algébrique [5]. Or les nombres machine sont rationnels, donc algébriques. Par conséquent, l'exponentielle, le sinus, le cosinus et l'arctangente d'un nombre machine différent de 0 n'est pas un nombre machine ou le milieu de deux nombres machine consécutifs (nombres réels pour lesquels le TMD se produit quel que soit m), et le logarithme d'un nombre machine différent de 1 n'est pas un nombre machine ou le milieu de deux nombres machine consécutifs. Dans ces cas (on ne considère pas ici les cas triviaux où les entrées valent 0 ou 1), pour un argument x fixé, il existe donc toujours une valeur de m assez grande pour laquelle le TMD ne peut pas se produire. De plus, puisqu'il y a un nombre fini de nombres machine, la valeur de m pour laquelle le TMD peut se produire est en fait majorée. Hélas, ce raisonnement ne donne aucune idée de l'ordre de grandeur d'un majorant. De plus, un tel majorant dépend entièrement du système à virgule flottante considéré.

Remarque : pour certaines fonctions, comme 2^x, $\log_2 x$ et x^y, le résultat exact est parfois un nombre machine (ou le milieu de deux nombres machine consécutifs), et pour ces valeurs, le TMD se produit quel que soit m. Mais on peut chercher à déterminer ces valeurs. Par exemple :

– 2^x (où x est un nombre machine) ne peut être un nombre machine (ou le milieu de deux nombres machine consécutifs) que pour x entier (preuve donnée en annexe, section A.2), et dans ce cas, l'arrondi exact est facile à obtenir ;

– $\log_2 x$ (où x est un nombre machine) ne peut être un nombre machine (ou le milieu de deux nombres machine consécutifs) que si x est une puissance de 2, et dans ce cas, l'arrondi exact est également facile à obtenir ;

– le cas de x^y est plus difficile. Par exemple, $(25/16)^{3/2} = 125/64$ est un nombre machine et pose donc un problème.

Remarquons que même si la valeur de m peut être grande, il n'est pas nécessaire de toujours évaluer $f(x)$ avec une grande précision. En effet, si le TMD se produit pour une certaine valeur, on peut facilement détecter ce problème, et recommencer le calcul avec une précision plus grande. Une méthode fondée sur cette remarque a d'abord été suggérée par Gal et Bachelis [20], puis approfondie par Ziv [53] (cf section 4.2).

Schulte et Swartzlander [43, 44] ont proposé des algorithmes pour évaluer les fonctions $1/x$, $\sqrt{x}$, 2^x et $\log_2 x$ en simple précision avec arrondi exact. Pour trouver la borne exacte de m, ils ont effectué une recherche exhaustive pour $n = 16$ et $n = 24$. Pour $n = 16$, ils ont trouvé $m = 35$ pour $\log_2 x$ et $m = 29$ pour 2^x, et pour $n = 24$, ils ont trouvé $m = 51$ pour $\log_2 x$ et $m = 48$ pour 2^x. On remarque que $m \approx 2n$.

Ces valeurs de m peuvent s'expliquer par le fait qu'après les chiffres de la mantisse, les bits de $f(x)$ se comportent comme une suite aléatoire de 0 et de 1 (avec même probabilité)[2], sauf dans des domaines particuliers ou pour des fonctions particulières ; ceci peut être vu comme une conséquence des résultats de Feldstein et Goodman [18] sur la distribution des bits de mantisse des variables lorsqu'une quantité « suffisante » de calculs a été effectuée. On suppose de plus que les suites associées à deux arguments différents x_1 et x_2 peuvent être considérées indépendantes (ce n'est pas exactement le cas pour les fonctions 2^x et $\log_2 x$, car 2^{x+1} a la même mantisse que 2^x, mais cela ne change pas grand chose).

Ces hypothèses probabilistes ont été confirmées par J.-M. Muller et A. Tisserand dans [38] à l'aide de tests sur les autres fonctions élémentaires : tests exhaustifs pour de petites valeurs de n et en simple précision, et tests sur des arguments aléatoires pour la double et quadruple précision. Les valeurs maximales de m correspondant aux fonctions exponentielle, logarithme, sinus, cosinus et arctangente en simple précision ($n = 24$), pour les nombres machine d'exposant 0 et en arrondi au plus près, sont données dans le tableau 1 page suivante.

À l'aide de ces hypothèses, on peut en déduire que si n_e exposants différents sont considérés pour les arguments, alors nous obtenons $N = n_e \times 2^{n-1}$

[2] En effet, si on considère une séquence de chiffres dont la position et la longueur sont fixées, et qu'un nombre machine x est choisi au hasard, alors on peut remarquer que les différentes séquences possibles apparaissent avec à peu près la même probabilité que des séquences choisies aléatoirement, cette probabilité étant d'autant plus précise que le nombre de nombres machine est grand.

$f(x)$	m
$\exp x$	49
$\log x$	53
$\sin x$	49
$\cos x$	50
$\arctan x$	49

TAB. 1: *Valeurs maximales de m pour quelques fonctions élémentaires en simple précision (exposant 0, arrondi au plus près).*

arguments, donc m est proche de $n + \log_2(N) = 2n + \log_2(n_e) - 1$ avec une grande probabilité. Des études probabilistes similaires avaient auparavant été effectuées par Dunham [17], et par Gal et Bachelis [20].

Malheureusement, ces études probabilistes ne constituent pas une preuve. Elles donnent seulement une estimation de la précision des calculs intermédiaires nécessaire pour obtenir des résultats arrondis exactement. Des résultats de théorie des nombres peuvent être utilisés pour obtenir un majorant garanti de m, mais de tels majorants sont très grands, et en pratique toujours bien plus grands que la valeur maximale de m. Les meilleurs majorants que l'on ait obtenus proviennent de l'application d'un théorème de Nesterenko et Waldschmidt [39], qui donne des bornes de l'ordre de plusieurs millions ou milliards de bits. Bien que le calcul de fonctions sur plusieurs millions ou milliards de bits soit possible avec les machines actuelles (en utilisant les algorithmes de Brent [10, 9] pour les fonctions transcendantes, et l'étude de Zuras [54] sur la multiplication), cela peut demander plusieurs dizaines de minutes de calcul et utiliser beaucoup de mémoire, et ce n'est pas une solution satisfaisante, même si de tels calculs n'ont quasiment aucune chance de s'avérer nécessaires si la méthode de Ziv est utilisée. D'autre part, connaître la vraie valeur maximale de m nous permettrait de concevoir des algorithmes très efficaces, ne faisant pas de calculs inutiles. C'est pourquoi il est intéressant de faire une recherche exhaustive des pires cas, pour lesquels m est supérieur ou égal à une valeur m_0 fixée. Nous allons donc considérer les deux approches suivantes :

– Tests exhaustifs. Cette approche demande beaucoup de temps de calcul pour tester tous les éléments, même s'ils ne sont pas réellement tous testés (cf chapitre 2). Une telle approche est possible en simple précision. Elle est également possible en double précision si les fonctions sont suffisamment « régulières » (cf chapitre 1), en utilisant un gros réseau de stations de travail. Mais dans certains intervalles en double précision, ou en quadruple précision, les calculs demanderaient beaucoup trop de temps, même en tenant compte de l'évolution de la puissance des machines. C'est pourquoi une seconde approche est nécessaire pour ces cas-là.
– Calculs en très grande précision (jusqu'à plusieurs millions de bits). Même s'ils n'ont quasiment aucune chance d'être effectués quels que

> soient les calculs de fonction demandés par l'utilisateur, il faut en toute rigueur prévoir le cas. Ces routines ne devront pas être trop lentes, ni utiliser beaucoup de mémoire. Elles devront également être portables et *la précision des résultats devra être garantie*. Il existe déjà des bibliothèques de calcul en multiprécision, mais elles ne vérifient pas toutes les conditions requises.

Dans le chapitre 1, nous présenterons le principe des tests exhaustifs, avec pour objectif d'obtenir des résultats garantis (à l'aide de majorations d'erreurs) et d'avoir des algorithmes très rapides en pratique. Pour cela, nous reviendrons d'abord en détail sur les hypothèses probabilistes mentionnées ci-dessus. Les tests se feront à l'aide d'approximations des fonctions élémentaires par des polynômes. Nous montrerons alors comment évaluer rapidement les valeurs successives d'un polynôme ; évidemment, les calculs seront d'autant plus rapides que le degré des polynômes est petit. Mais plus le degré est petit, plus l'intervalle dans lequel l'approximation est « valide » est petit, et l'approximation elle-même (calcul des coefficients des polynômes) peut alors prendre *a priori* beaucoup de temps. Nous montrerons donc comment approcher efficacement les fonctions par des polynômes de petit degré, en tenant compte du fait que les arguments testés sont espacés de manière régulière. Le problème revient en fait à trouver les sommets d'une grille régulière qui sont « assez proches » des points d'une courbe ; cette remarque nous permettra de tester simultanément une fonction et sa réciproque, et de choisir entre ces deux fonctions (suivant le domaine considéré) celle pour laquelle les tests seront les plus rapides. Enfin, nous présenterons un algorithme permettant de générer du code pour effectuer rapidement une multiplication entière par une constante, un tel algorithme pouvant être utilisé pour notre problème, lors de l'approximation d'une fonction par des polynômes de petit degré.

Avec les algorithmes présentés dans le chapitre 1, il faut en moyenne une addition et une comparaison pour chaque argument testé, en approchant la fonction par des polynômes de degré 1 (en négligeant le temps pris pour trouver les approximations). Le problème revient en fait à trouver s'il y a des points d'une grille qui sont suffisamment proches d'un segment de droite (approchant la courbe de la fonction). Dans le chapitre 2, nous présenterons un algorithme rapide calculant un minorant de la distance entre un segment de droite et $\mathbb{Z}^2$ (points du plan à coordonnées entières). Cet algorithme sera utilisé comme filtre pour éliminer rapidement les intervalles où il n'y a pas de sommet de la grille « très proche » de la courbe.

Dans le chapitre 3, nous présenterons comment les algorithmes décrits dans les chapitres 1 et 2 ont été implémentés, pour la double précision (53 bits de mantisse). Nous présenterons également la parallélisation sur réseau de stations de travail et expliquerons les choix qui ont été faits pour éviter un certain nombre de problèmes rencontrés en pratique. Nous donnerons les temps de calcul mesurés et les résultats obtenus.

Les chapitres 1, 2 et 3 concernaient les tests exhaustifs, permettant de trouver des valeurs qui seront utilisées lors de l'implémentation des fonctions élémentaires avec arrondi exact. Dans le chapitre 4, nous parlerons de l'implémentation proprement dite. Nous décrirons d'abord plus en détail la stratégie de Ziv, permettant d'avoir des algorithmes très rapides en moyenne. Puis nous présenterons des algorithmes de calcul en multiprécision, utiles dans les cas où aucun majorant de m (précision maximale des calculs intermédiaires pour pouvoir arrondir exactement) n'est connu, en particulier différents algorithmes de multiplication (base de tous les autres algorithmes). Nous montrerons comment les résultats des tests exhaustifs peuvent être utilisés de manière efficace. Enfin, nous donnerons des mesures (pour ce qui a été implémenté) et des évaluations des temps de calcul.

Ce travail sur le dilemme du fabricant de tables a fait l'objet de publications : [34, 35, 36].

Chapitre 1

Tests exhaustifs, principes

1.1 Introduction

Une approche pour garantir l'arrondi exact d'une fonction élémentaire f sur n bits de mantisse est de chercher, à l'aide de tests exhaustifs, tous les arguments x pour lesquels le dilemme du fabricant de tables (TMD) peut se produire si $f(x)$ est évalué avec une erreur (absolue) d'au plus 2^{-m} sur sa mantisse, m étant un entier supérieur à n. Ces tests seront effectués *une fois pour toutes* (le système à virgule flottante étant fixé), et les résultats seront utilisés pour concevoir les algorithmes de calcul avec arrondi exact.

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 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, comme expliqué plus bas.

Nous cherchons principalement à trouver les pires cas pour la double précision (nombres machine de $n = 53$ bits de mantisse). Cependant, les méthodes que nous décrirons dans ce chapitre seront valables pour toute valeur de n. Elles ne seront pas appliquées dans tous les intervalles. Par exemple, pour l'exponentielle, si dans l'intervalle testé, l'argument x est suffisamment petit (inférieur à 2^{-n}), un raisonnement rapide basé sur la formule de Taylor permet de conclure ; et si x est suffisamment grand, l'exponentielle donne un *overflow*. Dans les intervalles pour lesquels la différence entre deux nombres machine consécutifs est grande (arguments x ayant un grand exposant), les fonctions trigonométriques sinus et cosinus ne sont pas régulières du point de vue numérique (elles le sont, bien entendu, du point de vue mathématique), et nos méthodes s'appliqueront trop mal pour être efficaces. D'autre part, même lorsque ces méthodes s'appliquent bien, les tests exhaustifs ne seront pas toujours possibles en temps raisonnable, par exemple pour $n = 113$ (quadruple précision) ; ils peuvent être également restreints à des intervalles, et les résultats pourront par exemple être utilisés pour concevoir des algorithmes de calcul de f avec la méthode de Gal [19], qui nécessite la connaissance de nombres machine dont l'image par f est presque un nombre machine, ou pour tester des implémentations existantes.

Notons que si l'on a des résultats complets dans un certain intervalle, on ne peut généralement pas en déduire les résultats dans d'autres intervalles, même s'il existe une relation mathématique, du style $e^{x+y} = e^x e^y$ pour l'exponentielle. En revanche, cela peut être possible pour certaines fonctions *alternatives*, comme 2^x, $\log_2(x)$, $\cos(\pi x)$ et $\sin(\pi x)$. Par exemple, si n est entier, alors 2^{n+x} et 2^x ont la même mantisse, et on peut se ramener dans l'intervalle $[-\frac{1}{2}, +\frac{1}{2}]$.

Supposons que l'on veuille tester tous les arguments ayant un signe et un exposant fixés (bien sûr, en pratique on testera plusieurs exposants, mais cela donne déjà une idée de la quantité de calculs). Il y a 2^{n-1} mantisses possibles (le premier bit de la mantisse étant toujours 1), 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} (l'algorithme de test sera donné plus loin). Évaluons le temps total de calcul à l'aide des algorithmes classiques de calcul de fonctions, en supposant que $n = 53$ (double précision), $m \approx 90$, et que les calculs se font sur une machine à 500 MHz et demandent 200 cycles d'horloge par argument. Il faudra environ :

$$\frac{2^{52} \times 200}{500 \times 10^6 \times 86400} \text{ jours} \approx 20850 \text{ jours},$$

soit 57 ans ! Et ceci, à multiplier par le nombre d'exposants (plusieurs dizaines ou centaines) et de fonctions à tester. Même en parallélisant les calculs, cela ne peut pas se faire en temps raisonnable dans ces conditions. Nous cherchons donc à avoir des algorithmes de test les plus rapides possible. Pour cela, nous utiliserons des *filtres* : lors d'une première étape, nous chercherons à effectuer les calculs avec une précision pas trop grande, de manière à éliminer la plupart des cas, les cas restants étant suffisamment peu nombreux (disons, quelques millions) pour être testés avec un algorithme beaucoup plus lent lors d'une seconde étape (d'autres étapes peuvent être utilisées éventuellement) ; il s'agit d'une méthode analogue à celle de Ziv (cf section 4.2). De plus, nous tiendrons compte du fait que dans la première étape, les arguments sont « très proches » les uns des autres et espacés régulièrement.

Nous allons décrire des méthodes pour effectuer très rapidement les tests. Ces méthodes seront assez générales, et nous ne nous occuperons pas pour l'instant des détails liés à l'implémentation, qui ne sera abordée qu'au chapitre 3.

1.2 Test sur un ensemble d'arguments

Nous allons montrer précisément comment les arguments seront testés, et à l'aide des hypothèses probabilistes, nous pourrons déduire une idée du nombre d'arguments qui échoueront au test (pires cas), ce qui permettra d'ajuster les paramètres (précision de calcul lors des différentes étapes) en conséquence. Notons que si dans un certain domaine, les hypothèses probabilistes ne sont pas vérifiées, par exemple si le nombre de pires cas est beaucoup plus grand que prévu, ce n'est pas trop grave : cela n'impliquera pas des résultats faux. Au pire, la mémoire sera saturée, et les résultats seront incomplets ; si cela se produit, tout problème pourra évidemment être détecté. Ceci dit, il est très peu probable d'obtenir une mauvaise estimation, i.e. beaucoup plus (ou moins) de pires cas que prévu, et si jamais c'est le cas, il est important de savoir pourquoi, cela pouvant notamment être signe d'un « bug » dans l'implémentation.

Un test pour une fonction f et un argument x s'effectuera en gros de la manière suivante. La mantisse de $f(x)$ est calculée sur m bits : $y_0.y_1y_2y_3...y_{m-1}$ avec une erreur d'au plus 2^{1-m} (poids du dernier bit calculé). Les n bits de la mantisse dans la précision cible sont y_0, y_1, ..., y_{n-1} ; ils n'ont aucune importance pour le TMD et ne seront donc pas testés. Le bit y_n est le bit d'arrondi ; comme on considère les deux types de mode d'arrondi (dirigé et au plus près) en même temps, il peut être quelconque au moment des tests : nous voulons obtenir les pires cas pour les deux types de mode d'arrondi. Nous testerons donc les bits à partir de y_{n+1}, et jusqu'au bit y_{m-2}. Si tous ces bits sont égaux, l'argument sera mémorisé, et il faudra le retester lors de l'étape suivante. Mais s'ils sont différents, alors les bits y_{n+1} à y_{m-1} de la mantisse de la valeur exacte de $f(x)$ ne peuvent pas tous être égaux ; lors de la première étape, un tel argument sera rejeté.

Considérons une séquence de t bits testés (ici, $t = m - n - 2$). D'après les hypothèses probabilistes, chaque bit a la probabilité un demi d'apparaître, et puisque les bits sont indépendants les uns des autres, la probabilité d'avoir t bits égaux est de 2^{1-t}. Si on considère 2^r arguments, il y aura de l'ordre de 2^{r+1-t} pires cas ; il s'agit juste d'un ordre de grandeur, car c'est seulement une estimation.

Notons que ceci est valable lors des différentes étapes, le nombre d'arguments considérés étant le nombre initial d'arguments (avant la première étape). Lors de certaines étapes, on pourra tester plus de bits, mais ne considérer que t bits pour décider s'il s'agit d'un pire cas (à garder) ou d'un nombre à rejeter. Lors de la première étape, le nombre t sera choisi de manière à avoir des tests rapides ; t devra donc être relativement petit, mais pas trop, de manière à ne pas avoir trop de pires cas à tester lors de l'étape suivante (où les tests sont beaucoup plus lents). Lors des étapes suivantes, le nombre t sera plutôt choisi en fonction du nombre de pires cas que l'on veut garder ; par exemple, si on veut de l'ordre de 2^p pires cas, on choisira $t = r + 1 - p$.

Nous avons décrit comment tester les arguments, une fois une approximation de la mantisse du résultat calculée. Dans les sections suivantes, nous allons montrer comment calculer une telle approximation.

1.3 Approximation d'une fonction par un polynôme

Pour calculer les mantisses correspondant aux différents arguments à tester, nous avons choisi d'approcher la fonction par un polynôme. Ce choix se justifie par le fait qu'on peut rapidement calculer les valeurs d'un polynôme (la méthode sera décrite en section 1.4). Évidemment, plus le degré du polynôme est petit, plus les calculs seront rapides. Nous découperons donc l'intervalle à tester en petits sous-intervalles, dans lesquels nous approcherons la fonction par des polynômes de petit degré.

Nous avons un (petit) intervalle, une fonction élémentaire et un réel ε strictement positif (et inférieur à 2^{-m}, majorant de l'erreur finale). Nous voulons approcher cette fonction dans l'intervalle par un polynôme avec une erreur inférieure à ε et garantir cette majoration d'erreur.

Il existe diverses méthodes pour approcher une fonction par un polynôme, mais le principal problème est de majorer l'erreur automatiquement. Bien que ne donnant pas une excellente approximation, la formule de Taylor permet de calculer facilement les coefficients du polynôme ainsi qu'une majoration de l'erreur effectuée pour les quelques fonctions qui nous intéressent. Les coefficients doivent pouvoir s'exprimer de manière formelle en fonction du point d'origine. Ils pourront alors être calculés à l'aide d'un logiciel de calcul formel avec arithmétique d'intervalles ; nous choisirons Maple avec le *package* d'arithmétique d'intervalles intpak (bien que Maple travaille entièrement en base 10, ce qui posera quelques problèmes d'implémentation). L'erreur totale effectuée pourra être majorée par la somme des majorations de deux termes :

- l'erreur due à l'approximation de Taylor, en supposant les coefficients du polynôme exacts — une majoration de cette erreur sera notée E ;
- l'erreur due à l'arrondi des coefficients du polynôme.

Notons que l'on ne connaît pas à l'avance le degré du polynôme ; il sera calculé en fonction de l'erreur maximale autorisée ε par essais successifs. Par conséquent, il est préférable que les coefficients s'expriment en fonction du point d'origine et du degré du monôme associé à l'aide d'une formule assez simple.

Voici quelques exemples d'expression des coefficients et de majoration d'erreur due à la formule de Taylor ; il s'agit d'expressions symboliques, nous ne nous préoccupons pas encore de l'arrondi. Soit f la fonction considérée, x_0 le point d'origine, T la borne supérieure de l'intervalle d'approximation (i.e. on s'intéresse à l'approximation de $f(x_0 + t)$ avec $0 \leqslant t \leqslant T$), d le degré du polynôme, a_i le coefficient de degré i, et E une majoration de l'erreur. Une condition sur T peut être exigée, et elle sera vérifiée lors des tests.

– $f(x) = e^x$: si $T \leqslant 1$, alors $a_i = \dfrac{e^{x_0}}{i!}$. L'erreur peut être majorée par :

$$e^{x_0} \sum_{i=d+1}^{\infty} \frac{T^i}{i!} \leqslant e^{x_0} \frac{T^{d+1}}{(d+1)!} \left(1 + \sum_{i=1}^{\infty} \frac{1}{(d+2)^i}\right) = e^{x_0} \frac{T^{d+1}}{(d+1)!} \frac{d+2}{d+1}.$$

On peut donc prendre :

$$E = e^{x_0} \frac{T^{d+1}(d+2)}{(d+1)!\,(d+1)}.$$

Notons qu'en pratique, T sera beaucoup plus petit que 1, et que nous pourrions donc trouver un meilleur majorant de l'erreur. Mais puisque le majorant est supérieur à $e^{x_0} \dfrac{T^{d+1}}{(d+1)!}$, le gain ne serait pas très grand.

– $f(x) = 2^x$: il s'agit en fait de $e^{x.\log 2}$. Si $T \leqslant \dfrac{1}{\log 2}$, alors $a_i = \dfrac{2^{x_0}.\log(2)^i}{i!}$ et $E = 2^{x_0} \dfrac{(T.\log 2)^{d+1}(d+2)}{(d+1)!\,(d+1)}$.

– $f(x) = \log x$: si $T \leqslant x_0$, alors $a_0 = \log x_0$, pour $i \geqslant 1$, $a_i = \dfrac{-1}{i.(-x_0)^i}$, et $E = \dfrac{1}{d+1} \left(\dfrac{T}{x_0}\right)^{d+1}$.

– $f(x) = \log_2 x$: il s'agit en fait de $\dfrac{\log x}{\log 2}$. Si $T \leqslant x_0$, alors $a_0 = \log_2 x_0$, pour $i \geqslant 1$, $a_i = \dfrac{-1}{i.(-x_0)^i.\log 2}$, et $E = \dfrac{1}{(d+1).\log 2} \left(\dfrac{T}{x_0}\right)^{d+1}$.

– $f(x) = \cos x$: si $T \leqslant 1$, alors $a_{2i} = \dfrac{(-1)^i}{(2i)!}.\cos x_0$, $a_{2i-1} = \dfrac{(-1)^i}{(2i-1)!}.\sin x_0$, et $E = \dfrac{T^{d+1}(d+2)}{(d+1)!\,(d+1)}$.

– $f(x) = \sin x$: si $T \leqslant 1$, alors $a_{2i} = \dfrac{(-1)^i}{(2i)!}.\sin x_0$, $a_{2i+1} = \dfrac{(-1)^i}{(2i+1)!}.\cos x_0$, et $E = \dfrac{T^{d+1}(d+2)}{(d+1)!\,(d+1)}$.

1.4 Calcul des valeurs successives d'un polynôme

Nous avons approché la fonction à tester par un polynôme, et nous voulons calculer efficacement la valeur de ce polynôme en différents points. En utilisant le fait que les arguments sont espacés régulièrement, i.e. en progression arithmétique (sauf quand l'exposant change, mais en pratique, le changement d'exposant coïncidera avec une frontière entre deux intervalles), nous pouvons calculer les valeurs très rapidement en utilisant la méthode des différences finies. Cette méthode, décrite dans [27] (mais déjà connue du mathématicien chinois Zhu Shijié, vers l'an 1300), permet de calculer les valeurs successives d'un polynôme de degré d à l'aide de seulement d additions par valeur. Nous allons expliquer le principe sur un exemple simple : $P(x) = x^3$.

Notation : si P est un polynôme, on note ΔP le polynôme tel que

$$\Delta P(X) = P(X+1) - P(X),$$

et on note Δ^i la composition i fois de l'opérateur Δ.

On calcule d'abord $P(0)$, $P(1)$, $P(2)$, ..., $P(d)$ (ici, $d = 3$) à l'aide d'une méthode quelconque. Puis on calcule les différences finies : en-dessous de deux éléments consécutifs x et y, on écrit $y - x$, i.e. on calcule $\Delta P(0)$, $\Delta P(1)$, ..., $\Delta P(d-1)$, et on recommence pour chaque ligne (calcul des valeurs de $\Delta^2 P$, $\Delta^3 P$, ..., $\Delta^d P$). Ensuite, on peut calculer successivement les autres valeurs $P(d+1)$, $P(d+2)$, etc. à l'aide d'additions, comme indiqué sur la figure 1.1.

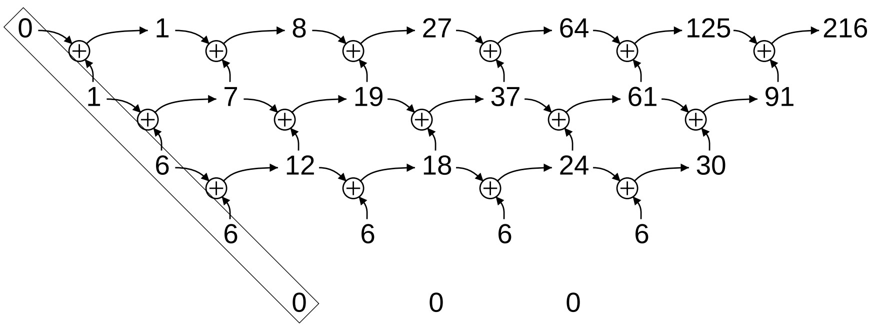

FIG. 1.1: *Table des différences finies du polynôme $P(X) = X^3$.*

Les éléments situés à gauche dans chaque ligne (encadrés sur la figure) sont les coefficients du polynôme dans la base

$$\left\{1, X, \frac{X(X-1)}{2}, \frac{X(X-1)(X-2)}{3!}, \frac{X(X-1)(X-2)(X-3)}{4!}, \dots\right\}$$

(le calcul de ces éléments par différences finies à partir des premières valeurs du polynôme constitue la formule d'interpolation de Newton). Le polynôme peut aussi bien être donné directement dans cette base.

Cette méthode a l'avantage de n'utiliser que des opérations très simples : des additions. D'autre part, cette méthode reste valable modulo n'importe quel réel positif. Comme nous n'avons pas besoin des premiers bits de la mantisse pour les tests (le premier bit testé est y_{n+1}), nous pouvons utiliser cette méthode modulo le poids de y_n. Ce n'est possible que si le poids de y_n reste constant dans l'intervalle testé. Cette condition est vérifiée sauf si l'exposant du résultat change. Ce cas peut être facilement résolu ; il sera abordé en section 1.6, et sera traité plus précisément en section 3.1.

1.5 Évaluation approchée des valeurs successives d'une fonction

1.5.1 Introduction

Nous pouvons approcher une fonction f par des polynômes de très petit degré, l'idéal étant d'obtenir des polynômes de degré 1 pour minimiser les calculs des valeurs successives. Mais plus le degré est petit, plus l'intervalle dans lequel l'approximation est valable est petit, et donc plus il y a d'approximations de f par un polynôme à effectuer. Dès que le degré des polynômes est très petit (1 ou 2), le temps de calcul de ces approximations n'est pas du tout négligeable si on utilise la méthode décrite dans la section 1.3. L'idée est de subdiviser hiérarchiquement les intervalles en sous-intervalles de même longueur, et de faire à chaque fois des approximations par des polynômes de plus petit degré. Comme les sous-intervalles sont de même longueur, on peut utiliser la méthode des différences finies (mais avec quelques calculs supplémentaires) pour trouver l'approximation dans le sous-intervalle suivant.

Reprenons le problème. Étant donnés une fonction f et un entier positif K, nous cherchons à évaluer les K valeurs successives $f(0)$, $f(1)$, $f(2)$, ..., $f(K-1)$, avec une erreur inférieure à un seuil ε fixé.

Pour cela, on commence par approcher la fonction f par un polynôme P avec une erreur inférieure à ε_0 (inférieure à ε). Comme cette approximation est valable dans l'intervalle entier, le degré de P peut être très grand. On va donc subdiviser l'intervalle $[0, K-1]$ en petits sous-intervalles contenant chacun k valeurs[1], et approcher le polynôme P par des polynômes de plus petit degré dans chacun de ces sous-intervalles, avec une erreur inférieure à $\varepsilon - \varepsilon_0$. Cette opération pourra être appliquée récursivement.

1.5.2 Approximation par des polynômes de plus petit degré

Le polynôme P de degré d étant fixé, et k étant également fixé, on définit les polynômes P_n (de degré d) de la façon suivante :

$$P(kn + m) = P_n(m).$$

[1]On s'arrangera pour que K soit divisible par k, éventuellement en ajoutant des valeurs à calculer, i.e. en augmentant K. En pratique, K et k seront des puissances de deux.

Dans chacun des sous-intervalles ($0 \leqslant m < k$), P_n est approché par un polynôme P'_n de degré d'.

L'évaluation des valeurs successives dans un sous-intervalle ne pose pas de problème : c'est toujours la méthode des différences finies. Le problème est le passage d'un polynôme au polynôme suivant (passage d'un sous-intervalle au sous-intervalle suivant). Nous cherchons également à utiliser la méthode des différences finies, mais nous ne sommes pas dans la bonne base. Nous allons considérer deux méthodes différentes pour remédier à ce problème :

– la mise à jour des coefficients, en tenant compte des calculs qui n'ont pas été effectués ;
– l'utilisation de la méthode des différences finies en considérant que chaque coefficient de P_n est la valeur d'un polynôme en n et en écrivant chacun de ces polynômes dans la bonne base.

1.5.2.1 Méthode 1

Au départ, nous avons un polynôme de degré d, avec ses coefficients a_0, a_1, a_2, ..., a_d dans la base des

$$\binom{X}{i} = \frac{X(X-1)(X-2)...(X-i+1)}{i!}.$$

Mais ce polynôme est approché par un polynôme de degré d' (avec $d' < d$) de la manière suivante : les calculs ne se font que sur les $d'+1$ premiers coefficients (du coefficient de degré 0 jusqu'au coefficient de degré d'). Pour le passage au sous-intervalle suivant, nous cherchons à ce que tous les coefficients a_i soient mis à jour, comme si les calculs avaient été effectués sur tous les coefficients. Ceci revient à ajouter à certains coefficients une combinaison linéaire d'autres coefficients (multiplication du vecteur des coefficients par une matrice constante...).

Sans perte de généralité, nous pouvons considérer que $d' = d-1$ pour simplifier : nous pouvons découper les intervalles récursivement, et représenter les approximations sous forme d'un arbre. Un tel découpage est même préférable, car pouvant mener à une évaluation plus rapide en moyenne.

La mise à jour peut se faire soit sur les coefficients initiaux (au début du sous-intervalle, avant les calculs sur le polynôme de degré d'), soit sur les coefficients finaux (à la fin du sous-intervalle, après les calculs sur le polynôme de degré d'). Le second choix est ici le meilleur, car il permet de faire moins de calculs : il suffit d'ajouter $\binom{k}{i} a_d$ à a_{d-i} (pour $1 \leqslant i \leqslant d$).

Cette méthode demande d'effectuer des multiplications en multiprécision par des constantes (certaines constantes pouvant prendre plusieurs mots) : au moins des multiplications par les d coefficients $\binom{k}{i}$, comme indiqué ci-dessus, ces coefficients étant précalculés. Pour donner un ordre de grandeur, k peut être de l'ordre de plusieurs milliers, voire beaucoup plus, et d peut être égal à 20, par exemple. Une telle multiplication peut prendre du temps avec certains processeurs (par exemple, Sparc), qui ont une multiplication entière assez lente. Mais il est possible de tenir compte du fait que

- l'on multiplie par des constantes un grand nombre de fois,
- ces constantes ont de gros diviseurs communs,
- on peut choisir k de manière à avoir des multiplications simples à effectuer (par exemple, quelques décalages et additions),
- plus d est petit, plus les constantes sont petites ; par conséquent, les plus gros calculs se produisent moins souvent.

Nous étudierons des algorithmes de multiplication par une constante en section 1.7.

1.5.2.2 Méthode 2

Reprenons la notation de la section 1.4 : si P est un polynôme, on note ΔP le polynôme tel que

$$\Delta P(X) = P(X+1) - P(X),$$

et on note Δ^i la composition i fois de l'opérateur Δ.

Les coefficients des polynômes P'_n de degré d' peuvent être eux-mêmes considérés comme des polynômes en n (cf ci-dessous). La mise à jour de ces coefficients peut donc se faire par la méthode des différences finies.

On rappelle que P_n est défini par :

$$P(kn+m) = P_n(m).$$

Le polynôme P_n est approché par le polynôme P'_n de degré d' tel que

$$P'_n(X) = \sum_{i=0}^{d'} a_i(n).\binom{X}{i}$$

où les $a_i(n)$ sont définis par

$$P(kn+X) = \sum_{i=0}^{d} a_i(n).\binom{X}{i}$$

(coefficients de P translaté de kn).

Les coefficients $a_i(n)$ peuvent se déterminer de la manière suivante :

$$a_i(n) = \Delta^i P(kn),$$

ce qui prouve que les a_i sont des polynômes en n de degré $d-i$. On pose :

$$a_{i,j}(n) = \Delta^j a_i(n),$$

coefficients de $a_i(X+n)$ dans la base des $\binom{X}{j}$. L'initialisation consiste à calculer les $a_{i,j}(0)$ à partir des premières valeurs des a_i, et ensuite, les $a_i(n)$ pourront être calculés à chaque passage au sous-intervalle suivant par la méthode des différences finies. On pose maintenant :

$$a_i(n,m) = \Delta^i P'_n(m),$$

coefficients de $P'_n(X+m)$ dans la base des $\binom{X}{i}$. Par définition, on a :

$$a_i(n,0) = a_i(n) = a_{i,0}(n).$$

Les calculs s'effectueront comme suit : d'abord une étape effectuée dans un langage de haut niveau (par exemple script Perl et utilisation de Maple avec *package* d'arithmétique d'intervalles) pour précalculer les valeurs initiales, générer un source C/assembleur et compiler ce source, puis une seconde étape effectuée par ce programme C/assembleur pour faire les calculs eux-mêmes.

Première étape. On calcule :

- $p_{n,m} = P(kn+m)$ pour $0 \leqslant m \leqslant d'$ et $0 \leqslant n \leqslant d$.
- $a_i(n) = \displaystyle\sum_{m=0}^{i} (-1)^{i-m} \binom{i}{m} p_{n,m}$ pour $0 \leqslant i \leqslant d'$ et $0 \leqslant n \leqslant d-i$.
- $a_{i,j}(0) = \displaystyle\sum_{n=0}^{j} (-1)^{j-n} \binom{j}{n} a_i(n)$ pour $0 \leqslant i \leqslant d'$ et $0 \leqslant j \leqslant d-i$.

Seconde étape. Le passage d'un sous-intervalle n au sous-intervalle suivant $n+1$ se fait par :

$$a_{i,j}(n+1) = a_{i,j}(n) + a_{i,j+1}(n)$$

pour $0 \leqslant i \leqslant d'$ et $0 \leqslant j \leqslant d-i-1$, $a_{i,d-i}$ étant constant.

On rappelle que les valeurs initiales d'un sous-intervalle sont données par $a_i(n,0) = a_{i,0}(n)$. Le passage d'une valeur à la suivante se fait par :

$$a_i(n,m+1) = a_i(n,m) + a_{i+1}(n,m)$$

pour $0 \leqslant i \leqslant d'-1$, $a_{d'}(n,m)$ ne dépendant pas de m.

Majoration de l'erreur. L'erreur ε_1 due à l'approximation de P par P' au point (n,m) est égale à

$$\left|P(kn+m) - P'_n(m)\right| = \left|\sum_{i=d'+1}^{d} a_i(n) \binom{m}{i}\right|.$$

On majore alors cette erreur indépendamment de m par :

$$\varepsilon_1 \leqslant \sum_{i=d'+1}^{d} |a_i(n)| \binom{k-1}{i}.$$

Le polynôme $\Delta^i P$ s'écrit dans la base des $\binom{X}{j}$:

$$\Delta^i P(X) = \sum_{j=0}^{d-i} \Delta^{i+j} P(0). \binom{X}{j},$$

et puisque $a_i(n) = \Delta^i P(kn)$, on a alors :

$$a_i(n) = \sum_{j=i}^{d} a_j(0) \binom{kn}{j-i}.$$

Par conséquent,

$$\varepsilon_1 \leqslant \sum_{i=d'+1}^{d} \sum_{j=i}^{d} |a_j(0)| \binom{K-k}{j-i} \binom{k-1}{i}$$

i.e.

$$\boxed{\varepsilon_1 \leqslant \sum_{j=d'+1}^{d} |a_j(0)| \sum_{i=d'+1}^{j} \binom{K-k}{j-i} \binom{k-1}{i}}.$$

Note : cette majoration est optimale si on a $a_j(0) \geqslant 0$ pour tout $j \geqslant d' + 1$.

1.5.3 Approximation des valeurs successives d'un polynôme

1.5.3.1 Introduction

Nous pouvons supposer que les coefficients du polynôme P ont été choisis de manière à ce que les calculs correspondants (passage d'un sous-intervalle au suivant) puissent se faire exactement. Par exemple, on a un entier ν bien choisi, et toutes les valeurs sont dans $2^{-\nu}\mathbb{Z}$. Ces calculs n'étant pas effectués très souvent, on doit pouvoir se permettre de les faire exactement, ou alors on peut utiliser la méthode ci-dessous puisque le passage d'un sous-intervalle au suivant consiste justement à calculer des valeurs successives d'un polynôme (méthode 2).

Passons aux calculs effectués à l'intérieur d'un sous-intervalle. Il s'agit de la boucle interne ; il faut donc l'optimiser au maximum. Ainsi, les calculs se feront de manière approchée, ce qui donnera une erreur ε_2.

1.5.3.2 Problème

Soit $Q(X) = \sum_{i=0}^{\delta} a_i \binom{X}{i}$ un polynôme de degré δ à coefficients a_i réels dans la base

$$\left\{1, X, \frac{X(X-1)}{2}, \frac{X(X-1)(X-2)}{6}, \ldots\right\},$$

choisie pour pouvoir calculer les k valeurs successives $Q(0)$, $Q(1)$, $Q(2)$, ..., $Q(k-1)$. Les coefficients a_i seront approchés par des éléments $\hat{a}_i \in 2^{-n_i}\mathbb{Z}$ avec une erreur inférieure à 2^{-n_i}. Il est naturel de supposer la suite (n_i) croissante.

La méthode des différences finies consiste à effectuer successivement les δ additions suivantes, en boucle :

$$\begin{aligned}
&\hat{a}_0 = \hat{a}_0 + \hat{a}_1;\\
&\hat{a}_1 = \hat{a}_1 + \hat{a}_2;\\
&\quad\vdots\\
&\hat{a}_{\delta-1} = \hat{a}_{\delta-1} + \hat{a}_\delta;
\end{aligned}$$

(cf section 1.4). À chaque itération (δ additions), on obtient la valeur suivante du polynôme Q dans $\hat{a}_0$.

Puisque $n_i \leqslant n_{i+1}$, les additions ne se feront pas exactement : la valeur de la variable à additionner est tronquée de manière à avoir la même précision que la variable de destination. Cherchons alors une majoration de l'erreur en fonction des n_i.

1.5.3.3 Majoration de l'erreur

Une partie de l'erreur est due à l'approximation initiale des a_i par les $\hat{a}_i$: pour tout $0 \leqslant m \leqslant k-1$,

$$\left|\sum_{i=0}^{\delta}(a_i - \hat{a}_i)\binom{m}{i}\right| \leqslant \sum_{i=0}^{\delta} 2^{-n_i}\binom{k-1}{i}.$$

Une autre partie de l'erreur est due aux arrondis lors des additions. Soit $\epsilon_i(m)$ une majoration de l'erreur sur $\hat{a}_i$ à l'itération m. Avant la première itération, il n'y a pas d'erreur : $\epsilon_i(0) = 0$ pour tout i. $\hat{a}_\delta$ est constant, donc $\epsilon_\delta(m) = 0$ pour tout m. Concernant l'erreur effectuée lors de chaque addition, on a pour tout m et pour tout $i < \delta$:

$$\epsilon_i(m) = \epsilon_i(m-1) + \epsilon_{i+1}(m-1) + 2^{-n_i}.$$

On a alors :

$$\epsilon_i(m) = \sum_{j=i}^{\delta-1} 2^{-n_j}\binom{m}{j-i+1}.$$

En effet, pour $m = 0$ ou $i = \delta$, on a bien $\epsilon_i(m) = 0$; et pour $m > 0$ et $i < \delta$:

$$
\begin{aligned}
&2^{-n_i} + \epsilon_i(m-1) + \epsilon_{i+1}(m-1) \\
&\quad = 2^{-n_i} + \sum_{j=i+1}^{\delta-1} 2^{-n_j} \binom{m-1}{j-i} + \sum_{j=i}^{\delta-1} 2^{-n_j} \binom{m-1}{j-i+1} \\
&\quad = \sum_{j=i}^{\delta-1} 2^{-n_j} \left[\binom{m-1}{j-i} + \binom{m-1}{j-i+1} \right] \\
&\quad = \sum_{j=i}^{\delta-1} 2^{-n_j} \binom{m}{j-i+1} = \epsilon_i(m)
\end{aligned}
$$

L'erreur ε_2 maximale est majorée par :

$$
\begin{aligned}
&\sum_{i=0}^{\delta} 2^{-n_i} \binom{k-1}{i} + \epsilon_0(k-1) \\
&\quad = \sum_{i=0}^{\delta-1} 2^{-n_i} \left[\binom{k-1}{i} + \binom{k-1}{i+1} \right] + 2^{-n_\delta} \binom{k-1}{\delta} \\
&\quad = \sum_{i=0}^{\delta-1} 2^{-n_i} \binom{k}{i+1} + 2^{-n_\delta} \binom{k-1}{\delta}.
\end{aligned}
$$

En fait, quelle que soit la valeur de n_δ, on n'a besoin de stocker que les bits correspondant à $n_{\delta-1}$. Par conséquent, on peut considérer que $n_\delta = +\infty$. On a donc finalement :

$$
\boxed{\varepsilon_2 \leqslant \sum_{i=0}^{\delta-1} 2^{-n_i} \binom{k}{i+1}}.
$$

1.6 Test simultané d'une fonction et de sa réciproque

Nous allons montrer comment avoir une première étape plus rapide si on considère simultanément la fonction et sa réciproque, en supposant que la fonction réciproque est bien définie et qu'elle nous intéresse dans le domaine testé.

1.6.1 Méthode choisie

Nous supposons dans cette section que nous sommes restreints à un domaine dans lequel tous les arguments x ont le même exposant e et que toutes les valeurs $f(x)$ ont le même exposant e'.

Le problème revient à trouver les sommets d'une grille régulière qui sont assez proches de la courbe de la fonction à tester. Puisque la courbe est la même pour la fonction et pour la réciproque, il y a une certaine symétrie entre le test de la fonction et celui de la réciproque. Encore faut-il que la grille soit la même dans les deux cas. Dans le cas du test de la fonction, la grille, notée G, est définie comme suit : les abscisses des sommets de G sont les nombres machine (sur n bits de mantisse) situés dans un certain intervalle, et les ordonnées des sommets de G sont les nombres machine et les milieux de deux nombres machine consécutifs, c'est-à-dire les nombres sur $n + 1$ bits. Dans le cas du test de la réciproque, sans changer de repère, c'est l'inverse ; la grille, notée G', est définie comme suit : les ordonnées des sommets de G' sont les nombres sur n bits situés dans le domaine considéré, et les abscisses sont les nombres sur $n + 1$ bits situés dans le domaine considéré. Nous allons donc considérer la grille régulière engendrée par G et G' (grille dont les sommets ont pour abscisses et pour ordonnées les nombres sur $n+1$ bits dans le domaine considéré), et chercher les sommets de cette grille qui sont suffisamment proches de la courbe. Ainsi, nous trouverons les pires cas de la fonction et de sa réciproque. Notons que certains sommets (ceux dont l'abscisse et l'ordonnée sont tous deux des nombres impairs sur $n + 1$ bits) ne font partie d'aucune des deux grilles de départ ; ils ne sont donc pas intéressants, et seront rejetés lors de la seconde étape.

Un exemple (dans un domaine beaucoup plus petit que ceux qui seront considérés dans la pratique) est donné sur la figure 1.2 page ci-contre. Les points noirs représentent les sommets de la grille engendrée par G et G' : points dont les coordonnées sont des nombres sur $n + 1$ bits. De plus, si le point appartient à la grille G, c'est-à-dire si l'abscisse est un nombre machine sur n bits, ou encore si le bit de poids faible de l'abscisse est nul, alors on ajoute un petit segment vertical ; et si le point appartient à la grille G', c'est-à-dire si l'ordonnée est un nombre machine sur n bits, ou encore si le bit de poids faible de l'ordonnée est nul, alors on ajoute un petit segment horizontal.

Ces petits segments ont en fait une signification particulière : s'ils représentent les intervalles pour lesquels le TMD se produit, alors les pires cas correspondent aux intersections de la courbe avec ces segments.

Sur l'exemple de la figure 1.2, si on veut tester la fonction et la réciproque séparément dans le domaine choisi, cela demande de tester 9 arguments pour la fonction et 3 pour la réciproque, soit au total 12 arguments. Mais si on teste la réciproque uniquement, en considérant des arguments sur $n+1$ bits comme expliqué ci-dessus, il n'y a que 6 arguments à tester. Notons que le fait qu'il y ait deux fois moins d'arguments à tester ne signifie pas que le test de ce domaine sera deux fois plus rapide, car les fonctions testées (f et f^{-1}) ne sont pas les mêmes, et par exemple, l'une peut s'approcher mieux que l'autre. D'autre part, un algorithme de test plus rapide sera décrit dans le chapitre 2. La solution la meilleure dépend beaucoup de l'implémentation et du domaine en question, mais le test simultané de la fonction et de la réciproque est en général très intéressant.

En général, le nombre d'arguments à tester n'est pas le même suivant que

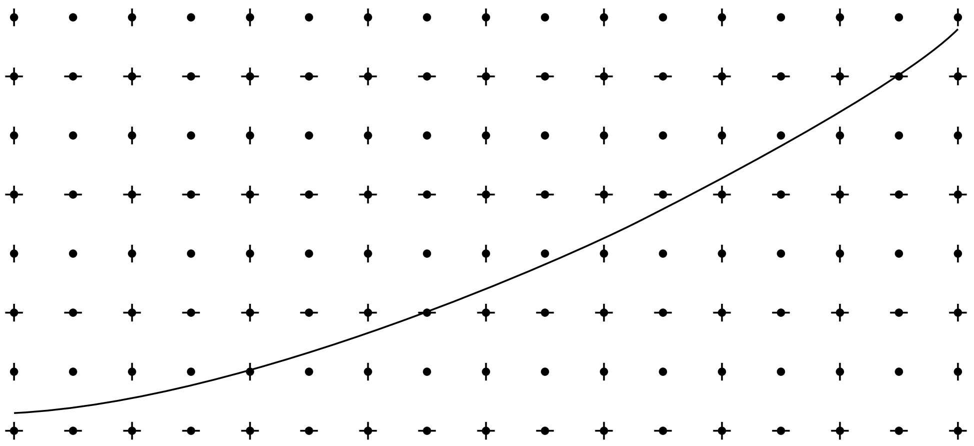

FIG. 1.2: *Exemple de fonction et la grille régulière considérée (points noirs) pour le test simultané de la fonction et de sa réciproque. Les points avec deux segments (+) sont ceux dont les coordonnées sont des nombres machine. Les pires cas correspondent aux intersections de la courbe de la fonction avec les segments.*

l'on considère la fonction ou sa réciproque. Dans l'exemple ci-dessus, si on teste la fonction uniquement (au lieu de la réciproque), il y a 17 arguments à tester. Le problème est donc de savoir s'il faut tester la fonction ou la réciproque.

La distance entre deux points consécutifs sur l'axe des abscisses est égale à 2^{e-n}, donc il y a environ $N_x = (x_{\max} - x_{\min}).2^{n-e}$ points sur l'axe des abscisses, dans le domaine considéré. De même, il y a environ $N_y = (y_{\max} - y_{\min}).2^{n-e'}$ points sur l'axe des ordonnées. Nous supposons que dans le domaine choisi, la fonction f est monotone, et sans perte de généralité, nous pouvons la supposer croissante. Donc, dans le domaine considéré, il y a environ $N_y = (f(x_{\max}) - f(x_{\min})).2^{n-e'}$ points sur l'axe des ordonnées. *A priori*, si $N_x < N_y$, il vaut mieux tester la fonction f, et si $N_y < N_x$, il vaut mieux tester la réciproque f^{-1}. Nous pouvons donc définir, pour le domaine D :

$$T_D = \frac{N_y}{N_x} = 2^{e-e'} \frac{f(x_{\max}) - f(x_{\min})}{x_{\max} - x_{\min}}.$$

En gros, si $T_D > 1$, c'est la fonction f qui sera testée, et si $T_D < 1$, c'est la réciproque f^{-1} qui sera testée.

Pour avoir une idée de la valeur de T_D aux alentours d'un argument x fixé, nous approchons la pente de f par la valeur $f'(x)$. Notons que nous choisirons des domaines très petits, et que cette approximation sera généralement très bonne en pratique, même si ce n'est pas très important. Nous avons donc $T_D \simeq 2^{e-e'} f'(x)$, et l'ordre de grandeur de T_D (à un facteur 2 près) est donné par la fonction $T(x)$ suivante :

$$T(x) = \left| \frac{x}{f(x)} f'(x) \right| = \left| \frac{f'(x)}{f(x)} x \right|.$$

Notons que comparer T à 1 revient à comparer la dérivée logarithmique de f à la dérivée logarithmique de la fonction identité.

1.6.2 Exemples

- Exponentielle : $T(x) = |x|$. Donc nous testerons de préférence l'exponentielle dans les domaines $]\log(2^{E_{\min}}), -1]$ et $[1, \log(2^{E_{\max}+1})[$, et nous testerons de préférence le logarithme dans le domaine $[e^{-1}, e]$.
- Logarithme : $T(x) = \dfrac{1}{|\log x|}$. Nous retrouvons évidemment les mêmes domaines que pour l'exponentielle.
- 2^x : $T(x) = |x| \log(2)$. Donc nous testerons de préférence 2^x dans des sous-domaines[2] de $]-\infty, -1/\log(2)]$ et $[1/\log(2), +\infty[$, et nous testerons de préférence $\log_2 x$ dans des sous-domaines de $[e^{-1}, e]$.
- $\log_2 x$: $T(x) = \dfrac{1}{|\log x|}$. Nous retrouvons évidemment les mêmes domaines que pour 2^x.
- Sinus : c'est une fonction impaire, donc nous nous restreignons à $\mathbb{R}_+^*$ $(x > 0)$: $T(x) = x\,|\cot x|$. Pour $x \leqslant 1$ par exemple, il vaut mieux *a priori* tester l'arcsinus. Mais la valeur de $T(x)$ est assez proche de 1 (et même très proche de 1 quand x devient « petit ») et l'approximation du sinus est beaucoup plus facile à calculer. Donc nous testerons en fait le sinus dans ce domaine.
- Cosinus : c'est une fonction paire, donc nous nous restreignons à $\mathbb{R}_+^*$ $(x > 0)$: $T(x) = x\,|\tan x|$. Quand x est petit, disons $x \leqslant \frac{1}{2}$, $T(x)$ est de l'ordre de x^2, donc la réciproque devient vite très intéressante quand x s'approche de 0. Par exemple, au lieu de tester tous les exposants négatifs pour la fonction cosinus, nous pourrions tester l'exposant -1 de l'arccosinus uniquement. Mais l'arccosinus s'approche très mal au voisinage de 1. Ensuite, tout dépend de l'implémentation...

1.6.3 Déduction des résultats pour la réciproque

Nous supposons que c'est la fonction f qui est testée (dans le cas contraire, il suffit d'échanger f et f^{-1}). Le programme de tests recherche tous les couples (x, y) sur $n+1$ bits tels que $|f(x) - y| < \varepsilon$, où ε est fixé. Le problème est d'en déduire une valeur ε' aussi grande que possible telle que tous les couples (x, y) non trouvés par le programme de tests vérifient $\left|f^{-1}(y) - x\right| \geqslant \varepsilon'$, i.e. tous les couples vérifiant $\left|f^{-1}(y) - x\right| < \varepsilon'$ doivent être trouvés.

Soit (x, y) un couple non trouvé par le programme de tests. Alors

$$|f(x) - y| \geqslant \varepsilon.$$

[2] Nous parlons de sous-domaines, car nous avons vu que l'on n'a pas besoin de tout tester pour les fonctions 2^x et $\log_2 x$.

Soit M un majorant de $|f'|$ sur l'intervalle $[f^{-1}(y), x]$ ou $[x, f^{-1}(y)]$. Alors

$$\left|f^{-1}(y) - x\right| \geqslant \frac{1}{M}\left|f(x) - y\right|.$$

D'où

$$\left|f^{-1}(y) - x\right| \geqslant \frac{\varepsilon}{M}.$$

On peut donc prendre :

$$\varepsilon' = \frac{\varepsilon}{M}.$$

Notons que ε' peut être très petit, au point que le nombre de pires cas pour la réciproque f^{-1} sera inférieur au nombre de pires cas voulu (d'après les hypothèses probabilistes). Nous pouvons donc choisir de prendre une plus grande valeur de ε, de manière à avoir un nombre acceptable de pires cas pour la réciproque. Cela a pour conséquence d'augmenter le nombre de pires cas pour la fonction f ; mais s'il y en a trop, la plupart pourront être éliminés lors de la deuxième étape. D'autre part, la valeur de ε ne peut pas être aussi grande que l'on veut (les cas extrêmes devraient donner plusieurs couples (x, y) pour x fixé). Si une trop grande valeur de ε est nécessaire, il faudra traiter un tel cas séparément (mais cela n'arrivera pas pour les fonctions et les domaines que nous considérerons).

1.6.4 Changement d'exposant

Si l'exposant des abscisses ou celui des ordonnées change, la grille n'est plus complètement régulière : l'espacement entre deux points consécutifs sur une même ligne horizontale ou verticale n'est plus constant. Or, pour que les algorithmes soient très simples et très rapides, une régularité complète est requise. Une solution sera détaillée en section 3.1 en considérant la représentation binaire des nombres manipulés, mais considérons ici le problème géométriquement, et en tenant compte du fait que la fonction et sa réciproque sont testées simultanément.

La courbe est placée de telle sorte qu'en abscisse se trouvent les arguments de la fonction effectivement testée.

Le problème pourrait se résoudre en découpant le domaine au niveau des changements d'exposant, en abscisse ou en ordonnée. Un tel découpage peut être considéré comme naturel pour les abscisses (et ne posera pas de problème lors de l'implémentation) : en effet, le domaine sera découpé en fonction de la valeur des arguments, et les arguments situés à la frontière entre deux sous-domaines seront des nombres s'exprimant sur peu de bits, c'est-à-dire dont la mantisse se termine par un grand nombre de 0 ; c'est en particulier le cas des points correspondant aux changements d'exposant (puissances de 2). Par exemple, le découpage pourra se faire aux points

$$2^E\left(1 + \frac{k}{2^{13}}\right)$$

où E et k sont des entiers, avec $0 \leqslant k \leqslant 2^{13} - 1$. Pour les ordonnées, le découpage est aussi possible et la frontière peut être calculée assez précisément, mais cela compliquerait l'implémentation.

La solution choisie est de garder un *découpage régulier* du domaine et d'ajouter des points pour rendre la grille régulière. Ceci se fait avant le test du sous-domaine en question. À l'endroit où il y a un changement d'exposant, la distance entre deux points consécutifs double, et il suffit donc d'ajouter les milieux entre deux points consécutifs, et recommencer s'il y a plusieurs changements d'exposants (cas très rare). Puisque le changement d'exposant à l'intérieur d'un sous-domaine ne peut se produire qu'en ordonnée (grâce au choix du découpage), le nombre de points en abscisse ne change pas, donc l'ajout des points ne change pas le nombre d'arguments à tester.

1.7 Multiplication par une constante

1.7.1 Introduction

Nous avons vu en section 1.5.2.1 que des multiplications par une constante peuvent intervenir dans le problème des tests exhaustifs, et qu'elles seront effectuées un grand nombre de fois. Pour que ces multiplications se fassent rapidement, nous pouvons passer un peu de temps à générer, en fonction de cette constante, du code le plus rapide possible effectuant les multiplications en question. Pour simplifier le problème, nous supposons qu'il s'agit de multiplications sur des nombres entiers positifs.

De telles multiplications interviennent dans d'autres problèmes, comme les algorithmes du style Toom-Cook pour effectuer des multiplications de grands entiers en multiprécision (cf section 4.4.3), et la génération de multiplications entières par les compilateurs (certains processeurs n'ont pas de multiplication entière ou cette instruction est relativement lente).

Nous cherchons un algorithme qui générera du code effectuant la multiplication par une constante entière n à l'aide de décalages vers la gauche (multiplications par une puissance de deux), d'additions et de soustractions. Nous supposons que la constante n peut avoir plusieurs centaines de bits. Si la constante n est suffisamment grande (plusieurs milliers de bits ?), des algorithmes rapides de multiplication de grands entiers peuvent aussi être utilisés (cf section 4.4.3).

Pour faciliter l'étude de ce problème, nous devons choisir un modèle assez simple et assez proche de la réalité. Nous pouvons supposer que toutes les opérations élémentaires (décalages d'un nombre de bits quelconque, additions et soustractions) demandent le même temps de calcul. Il s'agit d'un problème plus difficile que celui bien connu des *chaînes d'additions* [27].

Le problème de la multiplication par une constante a déjà été étudié pour les compilateurs, mais pour des constantes relativement petites (sur 32 bits au maximum, par exemple). La plupart des compilateurs implémentent un algorithme de Robert Bernstein [6] ou un algorithme similaire (dû à la spécificité du processeur cible). Mais cet algorithme est trop lent pour de très grandes constantes. Nous allons présenter un algorithme complètement différent, qui est suffisamment rapide pour des constantes de plusieurs dizaines ou centaines de bits. Mais d'abord, nous allons formuler précisément le problème et présenter l'algorithme de Bernstein.

1.7.2 Formulation du problème

Le décalage d'un nombre x de k bits vers la gauche (x multiplié par 2^k) est noté $x << k$, et dans les expressions parenthésées, le décalage est prioritaire par rapport à l'addition et la soustraction. Le temps de calcul d'un décalage ne dépend pas du nombre de bits k de décalage, appelé *taille* du décalage. C'est aujourd'hui vrai pour de nombreux processeurs. En ce qui concerne la multiprécision, les décalages se décomposent en un décalage au niveau du processeur (taille majorée par la taille des registres) et un décalage d'un nombre entier de mots. En fait, les décalages seront presque toujours associés à une autre opération (addition ou soustraction) ; en gros, nous ne manipulerons que des nombres impairs, et les décalages seront toujours différés autant que possible. Par exemple, au lieu d'effectuer $x << 3 + y << 8$, nous choisirons d'effectuer $(x + y << 5) << 3$. Cela peut être vu un peu comme les calculs en virgule flottante, où un décalage correspond à une modification de l'exposant, mais où les décalages ne sont réellement effectués que lors des additions et des soustractions.

Les opérations élémentaires seront donc en fait les additions et les soustractions où un des deux opérandes est décalé d'un nombre de bits fixé (éventuellement nul), et ces opérations élémentaires prendront le même temps de calcul. Le fait de supposer que les opérations $x+y$ et $x+y << k$ avec k non nul prennent le même temps de calcul est discutable, mais raisonnable. En effet, sur certains processeurs (par exemple, les ARM), de telles opérations prennent effectivement le même temps de calcul (1 cycle d'horloge). Et en multiprécision, le temps de calcul ne dépend pas de ce nombre de mots de décalage : ce n'est pas le fait que k est nul ou pas qui est important, mais le fait que k est un multiple ou non de la taille des registres. D'autre part, notons que ces suppositions modifient le temps de calcul d'un facteur borné ; par conséquent, une complexité du type $O(F(n))$ n'est pas affectée par ces suppositions.

Soit un entier positif impair n. Une suite finie d'entiers positifs u_0, u_1, u_2, ..., u_q est *acceptable* si elle vérifie les propriétés suivantes :

- $u_0 = 1$;
- pour tout $i > 0$, $u_i = |s_i\, u_j + 2^{c_i}\, u_k|$, avec $j < i$, $k < i$, $s_i \in \{-1, 0, 1\}$, $c_i \in \mathbb{N}$;
- $u_q = n$.

Le problème est de trouver un algorithme qui prend en entrée le nombre n et qui trouve une suite acceptable minimale $(u_i)_{0 \leqslant i \leqslant q}$. Mais ce problème est très difficile (on pense qu'il est NP-complet). Nous devons donc trouver des heuristiques.

Note : nous nous sommes restreints aux entiers positifs. Nous aurions pu changer la formulation pour accepter les entiers négatifs (en enlevant la valeur absolue et en laissant le choix entre u_j et u_k pour appliquer le signe), mais la formulation serait en fait équivalente.

1.7.3 Méthode binaire

L'heuristique la plus simple consiste à écrire la constante n en binaire et à générer un décalage et une addition pour chaque 1 dans l'écriture en binaire (en prenant les 1 dans n'importe quel ordre) : par exemple, considérons $n = 113$, pour calculer $113x$. En binaire, $113 = 1110001_2$. Si le nombre est lu de gauche à droite, les opérations suivantes seront générées :

$$\begin{aligned} 3x &\leftarrow (x << 1) + x \\ 7x &\leftarrow (3x << 1) + x \\ 113x &\leftarrow (7x << 4) + x \end{aligned}$$

Le nombre d'opérations élémentaires est le nombre de 1 dans l'écriture de n en binaire, moins 1.

Cette méthode peut être améliorée en utilisant le recodage de Booth, qui consiste à introduire des chiffres signés (-1, noté $\overline{1}$, 0 et 1) et à effectuer autant de fois que nécessaire la transformation suivante :

$$0\underbrace{1111\ldots1111}_{k \text{ chiffres}} \rightarrow 1\underbrace{0000\ldots000}_{k-1 \text{ chiffres}}\overline{1}.$$

Cette transformation est basée sur la formule :

$$2^{k-1} + 2^{k-2} + \ldots + 2^2 + 2^1 + 2^0 = 2^k - 1.$$

Cela permet de diminuer le nombre de chiffres non nuls. Puisqu'un chiffre 0 est transformé en chiffre 1 sur la gauche, il est plus intéressant de parcourir le nombre de droite à gauche : ainsi, le 1 créé pourra être utilisé dans la séquence éventuelle de 1 suivante. Par exemple, 11011 est d'abord transformé en $1110\overline{1}$, puis en $100\overline{1}0\overline{1}$.

Avec l'exemple choisi plus haut, $113 = 100\overline{1}0001_2$. Le recodage de Booth permet de générer 2 opérations seulement, au lieu de 3 :

$$\begin{aligned} 7x &\leftarrow (x << 3) - x \\ 113x &\leftarrow (7x << 4) + x \end{aligned}$$

1.7.4 Algorithme de Bernstein

L'algorithme de Bernstein est basé sur des opérations arithmétiques ; il n'utilise pas explicitement l'écriture binaire de n. Il consiste à restreindre les opérations à $k = i - 1$ et $j = 0$ ou $i - 1$ (dans la formulation), et un des avantages est qu'il peut être utilisé avec différents coûts pour les différentes opérations (addition `add`, soustraction `sub`, et décalages `shift`). Il s'agit d'un algorithme *branch-and-bound*, utilisant les formules suivantes :

$$\begin{array}{lcl}
\text{coût}(1) & = & 0 \\
\text{coût}(n \text{ pair}) & = & \text{coût}(n/2^c \text{ impair}) + \texttt{coût_shift} \\
\text{coût}(n \text{ impair}) & = & \min \left\{ \begin{array}{l} \text{coût}(n+1) + \texttt{coût_sub} \\ \text{coût}(n-1) + \texttt{coût_add} \\ \text{coût}(n/(2^c+1)) + \texttt{coût_shift} + \texttt{coût_add} \\ \text{coût}(n/(2^c-1)) + \texttt{coût_shift} + \texttt{coût_sub} \end{array} \right.
\end{array}$$

Un autre avantage de l'algorithme de Bernstein est qu'il n'y a pas besoin de mémoire supplémentaire (registres du processeur ou RAM) pour des résultats temporaires dans le code généré. Mais un besoin de mémoire supplémentaire n'est pas toujours un problème.

1.7.5 Un algorithme basé sur des motifs dans l'écriture binaire

1.7.5.1 Algorithme

Cet algorithme est basé sur la méthode binaire : après le recodage de Booth, nous considérons le nombre n comme un vecteur de chiffres signés 0, $+1$ et -1, notés `0`, `P` et `N` (et parfois 0, 1 et $\overline{1}$). L'idée, qui sera appliquée récursivement, est la suivante : nous cherchons des motifs (pas forcément adjacents) de chiffres non nuls (`P` et `N`) qui se répètent, pour faire disparaître le plus de chiffres non nuls en une seule opération. Pour simplifier, nous recherchons seulement les motifs qui se répètent deux fois (bien qu'en fait, ils puissent très bien se répéter plus souvent). Par exemple, le nombre $20061 = 100111001011101_2$, qui est recodé en `P0P00N0P0N00N0P`, contient deux fois le motif `P000000P0N` : la première fois sous sa forme positive, et la seconde fois sous sa forme négative `N000000N0P`. Ainsi, le fait de considérer ce motif permet de faire disparaître 3 chiffres non nuls en une seule opération, et nous avons seulement besoin de calculer `P000000P0N` (motif) et le reste `00P000000000000`. Ceci peut être résumé par :

```
  P000000P0N
-      P000000P0N
+ 00P000000000000
------------------
  P0P00N0P0N00N0P
```

Pour cet exemple, nous obtenons 4 opérations (`P000000PON` est calculé avec 2 opérations à l'aide de la méthode binaire) :

$$\begin{array}{rcl} 129x & \leftarrow & (x << 7) + x \\ 515x & \leftarrow & (129x << 2) - x \\ 15965x & \leftarrow & (515x << 5) - 515x \\ 20061x & \leftarrow & (x << 12) + 15965x \end{array}$$

alors que l'algorithme de Bernstein génère 5 opérations, soit une de plus.

Maintenant, nous cherchons comment trouver assez rapidement un bon motif qui se répète. Donnons d'abord quelques définitions. Un *motif* est un sous-ensemble de chiffres non nuls d'une écriture avec chiffres signés à un décalage près (par exemple, `PONOON` et `PONOONO` constituent le même motif) ; si on inverse chaque chiffre non nul, on obtient alors le motif sous sa forme opposée (par exemple, `PONOON` devient `NOPOOP`). Nous appelons *poids* d'un motif le nombre de chiffres non nuls (`P` et `N`) de ce motif ; le poids d'un motif sera noté w (pour *weight*). Nous cherchons en fait un motif de poids maximal. Pour cela, nous prenons en compte le fait qu'en général, un motif contient très peu de chiffres non nuls par rapport aux chiffres nuls, en particulier près des feuilles de l'arbre de récursion, à cause de la relation suivante : $w(\text{père}) = w(\text{fils } 1) + 2\,w(\text{fils } 2)$. La solution est de calculer toutes les distances possibles entre deux chiffres non nuls, en faisant la distinction entre chiffres identiques et chiffres opposés. Cela donne un majorant du poids du motif associé à chaque distance. Par exemple, avec `POPOONOPONOONOP` :

distance	majorant	poids
2 (P-N / N-P)	3	2
5 (P-N / N-P)	3	3
7 (P-P / N-N)	3	2

Les distances sont triées suivant le majorant associé, puis elles sont testées l'une après l'autre jusqu'à ce que le poids maximal et un motif correspondant soient trouvés.

1.7.5.2 Comparaison avec l'algorithme de Bernstein

Cet algorithme a été comparé à celui de Bernstein et nous avons trouvé qu'en moyenne, il est un peu meilleur que celui de Bernstein pour des petites constantes. Des comparaisons avec des constantes plus grandes n'ont pas pu être effectuées, parce que l'algorithme de Bernstein est trop lent : la complexité de l'algorithme de Bernstein est exponentielle, tandis que celle de notre algorithme est polynomiale ; il semble être en $O(m^3)$ en moyenne, où m est la taille de la constante n ($m = \lfloor \log_2 n \rfloor + 1$) : $O(m^2)$ pour chaque hauteur de récursion.

Si nous considérons le nombre d'opérations générées[3] par ces algorithmes

[3] Une opération est un décalage suivi d'une addition ou d'une soustraction ; nous considérons donc la valeur de q définie dans la formulation.

pour les nombres allant de 1 à 2^{20}, la plus grande différence est obtenue pour $n = 543413$.

Opérations générées par notre algorithme :

$$\begin{array}{rrcl}
1. & 255x & \leftarrow & (x << 8) - x \\
2. & 3825x & \leftarrow & (255x << 4) - 255x \\
3. & 19125x & \leftarrow & (3825x << 2) + 3825x \\
4. & 543413x & \leftarrow & (x << 19) + 19125x
\end{array}$$

Opérations générées par l'algorithme de Bernstein :

$$\begin{array}{rrcl}
1. & 9x & \leftarrow & (x << 3) + x \\
2. & 71x & \leftarrow & (9x << 3) - x \\
3. & 283x & \leftarrow & (71x << 2) - x \\
4. & 1415x & \leftarrow & (283x << 2) + 283x \\
5. & 11321x & \leftarrow & (1415x << 3) + x \\
6. & 33963x & \leftarrow & (11321x << 2) - 11321x \\
7. & 135853x & \leftarrow & (33963x << 2) + x \\
8. & 543413x & \leftarrow & (135853x << 2) + x
\end{array}$$

1.7.5.3 Résultats sur des constantes aléatoires

Une implémentation de notre algorithme a été testée sur des nombres aléatoires (un test exhaustif aurait été trop long). Voici le nombre moyen d'opérations générées en fonction du nombre de bits de la constante n (pour laquelle le premier et le dernier bits sont 1) :

#bits	#op
32	8.0
64	14.5
128	26.3
256	47.6
512	86.5
1024	157.4
2048	289.4

Le rapport entre deux nombres consécutifs est presque constant. À partir de ces résultats, nous pouvons conjecturer que le nombre moyen d'opérations générées pour un entier de m bits est en $O(m^k)$, où $k \approx 0.85$.

1.7.5.4 Améliorations possibles

Notre algorithme peut être encore amélioré. Voici quelques idées, qui n'ont pas encore été implémentées :

- Nous pouvons chercher des sous-motifs communs à différentes branches de la récursion. Par exemple, considérons `PONONOOPONONOOOPON`, et le motif `PONON`. `PON` apparaît à la fois dans le motif `PONON` et dans le motif formé par les chiffres restants ; par conséquent, il n'a besoin d'être calculé qu'une seule fois (à condition de calculer `PONON` en faisant apparaître le `PON`). Une solution est d'arrêter la récursion quand le poids maximal est égal à 1 (à ce moment, c'est la méthode binaire qui doit être utilisée) ; la recherche de motifs communs en serait facilitée. Notons que la recherche de motifs communs doit être effectuée avant d'utiliser la méthode binaire : avec l'exemple ci-dessus, si on commence par calculer `NON` pour obtenir `PONON`, le motif commun `PON` ne pourra pas être utilisé ; il faut obligatoirement commencer par `PON`.
- Dans la recherche d'un motif qui se répète, il y a généralement plusieurs motifs possibles qui ont un poids maximal. Au lieu d'essayer avec un seul motif, nous pouvons en considérer plusieurs, les essayer un par un, et garder la suite d'opérations la plus courte. L'idéal serait de tester tous les motifs de poids maximal, mais il faut faire attention à ce que la complexité de l'algorithme ne devienne pas exponentielle.
- Nous pouvons considérer la transformation suivante, qui ne change pas le poids : `PON` $\leftrightarrow$ `OPP` (et `NOP` $\leftrightarrow$ `ONN`). Par exemple, 11010101001_2 s'encode par défaut en `PONOPOPOPOOP`, ce qui donne un poids maximal de 1, et donc 5 opérations générées ; mais le code équivalent `POONONNOPOOP` est meilleur : avec le motif `POONOON` de poids 3, seulement 3 opérations seront générées. Mais le nombre de codes équivalents est exponentiel. Il n'est donc pas envisageable de tous les tester. Nous devons alors chercher une méthode permettant de trouver rapidement les meilleures transformations.
- Au lieu de définir un motif qui apparaît deux fois de poids maximal, nous pouvons définir un nouveau « chiffre » composé de deux anciens chiffres non nuls. Par exemple, considérons $10\bar{1}0\bar{1}0010\bar{1}0\bar{1}00010\bar{1}$ et le motif $10\bar{1}0\bar{1}$. Définissons alors le chiffre correspondant au décalage effectué : $\mathrm{A} = 10000001$. Le code correspondant au nombre devient : $\mathrm{A}0\bar{\mathrm{A}}0\bar{\mathrm{A}}00010\bar{1}$. Ensuite, nous pouvons définir $\mathrm{B} = \mathrm{A}00000001$, et obtenir finalement : $\bar{\mathrm{A}}000\mathrm{B}0\bar{\mathrm{B}}$. Cela mène à 4 opérations, comme si la méthode de recherche de sous-motifs communs avait été utilisée.

1.7.6 Conclusion

Grâce à l'algorithme présenté dans cette section, nous pourrons effectuer plus rapidement des multiplications par des constantes entières, pouvant faire jusqu'à plusieurs centaines ou milliers de bits. Mais il reste à l'améliorer, à trouver le nombre moyen d'opérations générées en fonction de la taille de la constante, et à prouver tous les résultats obtenus.

Ce travail sur la multiplication par une constante a été présenté dans [32].

Chapitre 2

Minoration de la distance entre un segment de droite et $\mathbb{Z}^2$

2.1 Introduction

La première étape de la résolution du dilemme du fabricant de tables pour une fonction élémentaire f consiste à trouver tous les couples (x, y) dans le domaine considéré tels que $|f(x) - y| < \varepsilon$, où ε est un réel positif fixé, et où x et y sont des nombres sur $n + 1$ bits de mantisse (si nous voulons les résultats pour f et pour la fonction réciproque f^{-1}, comme nous l'avons montré dans la section 1.6). Si seuls les résultats concernant la fonction f nous intéressent, alors x est un nombre sur n bits de mantisse seulement. Si seuls les résultats concernant la réciproque nous intéressent, alors y est un nombre sur n bits de mantisse seulement. Dans tous les cas, nous considérons des sommets d'une grille régulière. Nous allons montrer comment le problème peut se ramener à calculer un minorant de la distance entre un segment de droite et une telle grille, et comment calculer ce minorant efficacement.

Quand nous parlerons de distances entre une courbe et la grille, ce seront en fait des distances suivant l'axe des ordonnées, et non la distance classique $d(A, B) = \inf\{d(a, b) : a \in A,\ b \in B\}$, où $d(a, b)$ est la distance euclidienne. Mais remarquons tout de même que nous nous placerons dans des intervalles où la courbe s'approche très bien par un segment de droite, et que ces deux distances diffèrent d'un facteur constant (cosinus de l'angle formé par le segment avec l'axe des abscisses) ; même si cette propriété ne sera pas utilisée par la suite, cela justifie le terme de « distance » employé ici.

Pour x nombre machine fixé, la distance $|f(x) - y|$ est notée $d_0(x)$, ou plus simplement d_0 (x étant fixé) ; d_0 ne dépend pas de y, car pour notre problème, nous pouvons supposer que y est choisi (du point de vue mathématique, et non par algorithme) de façon à ce que cette distance soit minimale. Le domaine considéré est découpé en très petits intervalles, de telle manière à ce qu'une approximation de f par des polynômes de degré 1 suffise, et nous recherchons, dans chaque intervalle I, l'ensemble S_I des nombres machine x pour lesquels d_0 est inférieure à un réel fixé ε_I. Dans chaque intervalle I, la fonction f est approchée par un polynôme ℓ_I de degré 1 (la courbe est donc approchée par un segment), avec une erreur inférieure ou égale à un réel ε_0. L'approximation se fait par des polynômes de degré 1 d'une part parce qu'une telle fonction est facile à évaluer, et d'autre part parce qu'elle a des propriétés intéressantes que nous utiliserons pour trouver un algorithme rapide. Nous avons montré dans la section 1.5 comment trouver les approximations très rapidement. La partie qui prend le plus de temps est pour l'instant la recherche de l'ensemble S_I et nous voulons trouver un algorithme rapide pour le déterminer.

Sur la figure 2.1 ci-dessous, la distance $|\ell_I(x) - y|$ est notée d ; y est défini de la même manière que ci-dessus, mais ce qui suit resterait valable pour tout autre nombre machine. Si $d \geqslant \varepsilon_I + \varepsilon_0$, alors $d_0 \geqslant \varepsilon_I$. Par conséquent, pour trouver les points de S_I, nous rechercherons l'ensemble S'_I dont les points vérifient $d < \varepsilon_I + \varepsilon_0$. Les intervalles I et les réels ε_I et ε_0 seront choisis assez petits de telle sorte que l'ensemble S'_I sera généralement vide (pour une raison expliquée plus tard), mais assez grands de manière à pouvoir faire les calculs avec des nombres d'assez faible précision : nous devons éviter des calculs en multiprécision très coûteux.

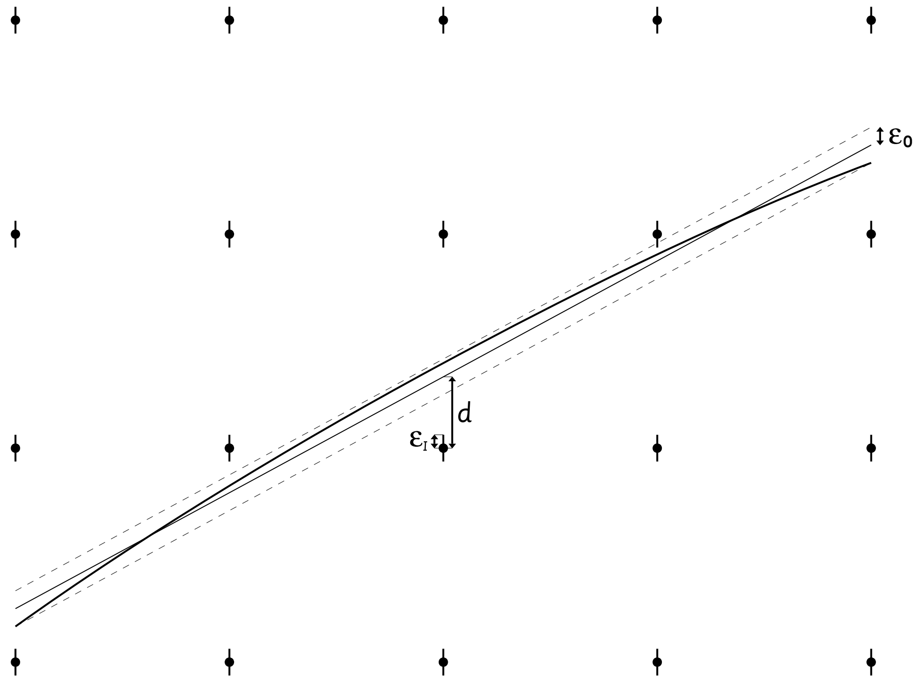

FIG. 2.1: *Courbe de la fonction approchée par un segment.*

Dans les domaines considérés après découpage, les nombres machine sont régulièrement espacés, si bien que nous pouvons multiplier les abscisses et les ordonnées par des puissances de 2 convenablement choisies pour considérer que les nombres machine correspondent en fait aux entiers ; la valeur de $\varepsilon_I + \varepsilon_0$ multipliée par la puissance de 2 correspondante est notée ε. Le problème s'énonce donc de la manière suivante : quels sont les points d'un segment donné, dont l'abscisse est un entier et dont la distance entre l'ordonnée et les entiers est inférieure à un réel ε fixé ? Nous rappelons que dans la plupart des cas, il n'y a aucun point satisfaisant ces conditions, d'après le choix de ε (le cardinal de S'_I est estimé à l'aide de la formule suivante, grâce aux hypothèses probabilistes : $N\varepsilon$, où N est le nombre d'entiers dans l'intervalle).

L'approche naïve consiste à tester chaque point dont l'abscisse est un entier : pour chaque itération, il faut une addition et une comparaison en calculant modulo 1 (c'est possible si on utilise l'arithmétique entière du processeur), et en décalant le segment de ε vers le haut : au lieu de tester si un nombre y est dans $[-\varepsilon, \varepsilon]$, on teste si $y + \varepsilon$ est dans $[0, 2\varepsilon]$. Le temps demandé par les autres opérations (approximations, incrémentation des compteurs de boucle, etc.) peut être négligé.

Nous allons montrer que si le nombre de points à tester, noté N, est assez grand (par exemple, 1000 ou plus), il existe alors une méthode plus rapide, en utilisant le fait que l'ensemble S'_I est généralement vide : nous pouvons chercher un minorant de la distance d, et si ce minorant est plus grand que ε, alors nous pouvons en déduire que S'_I est vide ; sinon, nous pouvons découper l'intervalle en sous-intervalles et utiliser cette méthode avec des paramètres différents, ou utiliser la méthode naïve (quand le nombre N de points à tester devient suffisamment petit).

Le segment a une équation de la forme $y = ax - b$, où x est restreint à un intervalle donné, par exemple $0 \leqslant x < N$. Dans la section 2.2, nous donnons quelques préliminaires et notations mathématiques. Dans la section 2.3, nous étudions la distribution des points $k.a$ modulo 1, où k est un entier vérifiant une inégalité $0 \leqslant k < n$; en particulier, nous mentionnons un théorème connu sous le nom du « théorème des trois distances » [1, 7, 47, 48, 49]. Dans la section 2.4, nous donnons l'algorithme, basé sur les propriétés décrites dans la section 2.3.

2.2 Préliminaires mathématiques, notations

$\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{N}$ sont respectivement les ensembles des nombres réels, des rationnels, des entiers relatifs, et des entiers naturels.

$\mathbb{R}/\mathbb{Z}$ est le groupe additif des réels modulo 1. Cet ensemble peut être vu comme un cercle, ou comme le segment $[0, 1]$ où on identifie les deux points 0 et 1 (i.e. les réels 0 et 1 représentent le même point). Avec cette seconde représentation, le point représenté par 0 (ou 1) peut être considéré comme une origine. Si $a \in \mathbb{R}/\mathbb{Z}$ et $k \in \mathbb{N}$, k est appelé *indice* de $k.a$ (dans le groupe engendré par a).

Si $a \in \mathbb{R}$, son image dans $\mathbb{R}/\mathbb{Z}$ sera aussi notée a, puisqu'il n'y a pas d'ambiguïté. Nous pouvons définir par partie fractionnaire positive de a : $\{x\} = x - \lfloor x \rfloor$. Cette définition reste valable si a est un élément de $\mathbb{R}/\mathbb{Z}$: c'est l'image de a dans $[0, 1[$.

$[x, y]$ représente un intervalle de réels (ouvert, si les crochets sont dans l'autre sens). $[[x, y]]$ représente un intervalle d'entiers. Si A est un ensemble fini, le cardinal de A se note $\#A$.

2.3 Propriétés de $k.a$ modulo 1

Dans cette section, nous étudions les propriétés des points $y = k.a$ modulo 1, où a est un nombre réel donné et k un entier restreint à un intervalle donné, par exemple $0 \leqslant k < N$ (où N est un entier positif donné). Les nombres a et y peuvent être considérés comme des éléments de $\mathbb{R}/\mathbb{Z}$. Pour $n \in \mathbb{N}$, définissons :

$$E_n = \{k.a \in \mathbb{R}/\mathbb{Z} : \ k \in \mathbb{N}, \ k < n\}.$$

Les ensembles E_n s'obtiennent en ajoutant à chaque fois un point, point précédent translaté de a, en partant de E_1 contenant un point unique 0 (origine).

Le passage du segment $y = ax - b$ ($0 \leqslant x < N$) aux ensembles E_n peut se résumer sur la figure 2.2 ci-dessous.

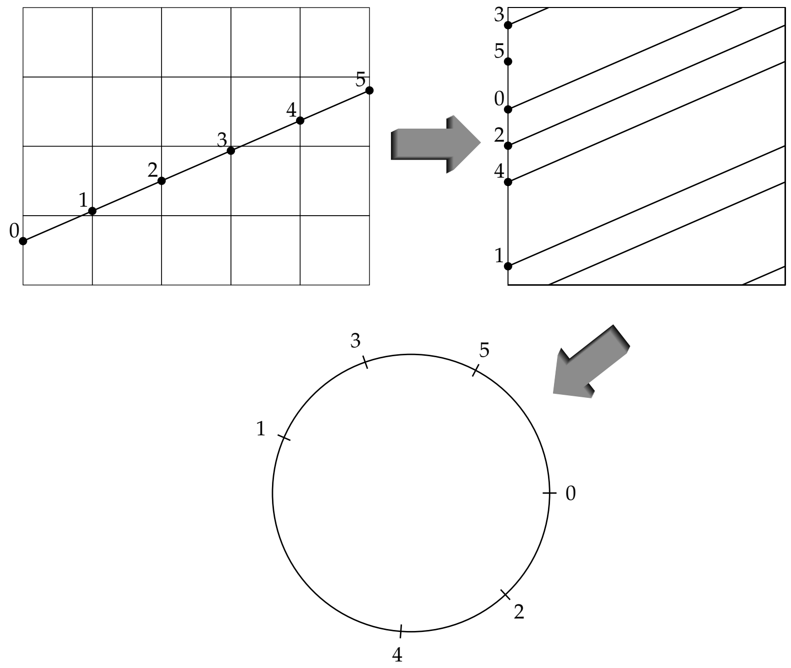

FIG. 2.2: *Passage du segment approchant la courbe de la fonction à l'ensemble E_6.*

Nous pouvons voir sur des exemples que la distribution des points de E_n possède des propriétés très intéressantes. En particulier, nous chercherons une construction de E_n basée sur les distances entre deux points de E_n adjacents sur $\mathbb{R}/\mathbb{Z}$.

L'une des propriétés remarquables est que pour chaque valeur de n, il y a au plus trois distances possibles entre deux points consécutifs [1, 7, 47, 48, 49]. Cette propriété peut se reformuler de la façon suivante. Partons d'un cercle avec un point x_0 dessus (origine), et considérons l'angle $\alpha = 2\pi a$ (modulo 2π). Construisons alors successivement les points x_1, x_2, x_3, ..., x_n sur le cercle de telle façon que x_{i+1} s'obtienne à partir de x_i en effectuant à chaque fois une rotation d'angle α (voir les figures 2.3 et 2.4) ; la rotation d'angle α correspond à ajouter a dans $\mathbb{R}/\mathbb{Z}$. Nous pouvons remarquer que quel que soit n, la distance entre deux points voisins sur le cercle ne peut prendre que trois valeurs possibles au maximum (théorème des trois distances). De plus, pour certaines valeurs de n (dépendant de α), il n'y a que deux valeurs possibles.

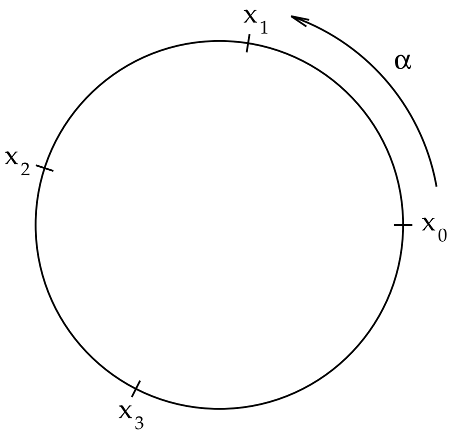

FIG. 2.3: *Théorème des 3 distances. Construction des premiers points.*

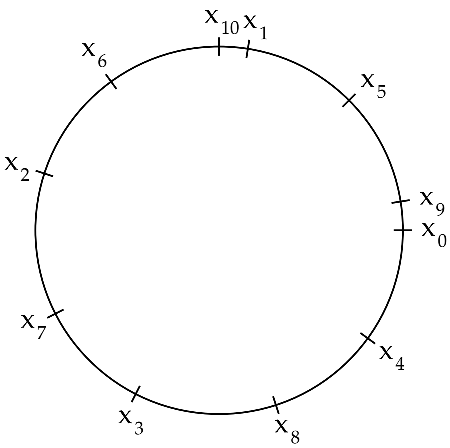

FIG. 2.4: *Théorème des 3 distances. Construction des points suivants. Il y a au plus 3 distances possibles entre deux points voisins.*

Un autre exemple est donné sur la figure 2.5 ci-dessous. Pour avoir des notations assez simples, nous choisissons un nombre rationnel $(17/45)$ pour a et multiplions par 45 tous les rationnels apparaissant sur l'exemple pour obtenir des entiers, et au lieu de travailler avec $\mathbb{R}/\mathbb{Z}$, nous travaillons ici avec $\mathbb{Z}/45\mathbb{Z}$.

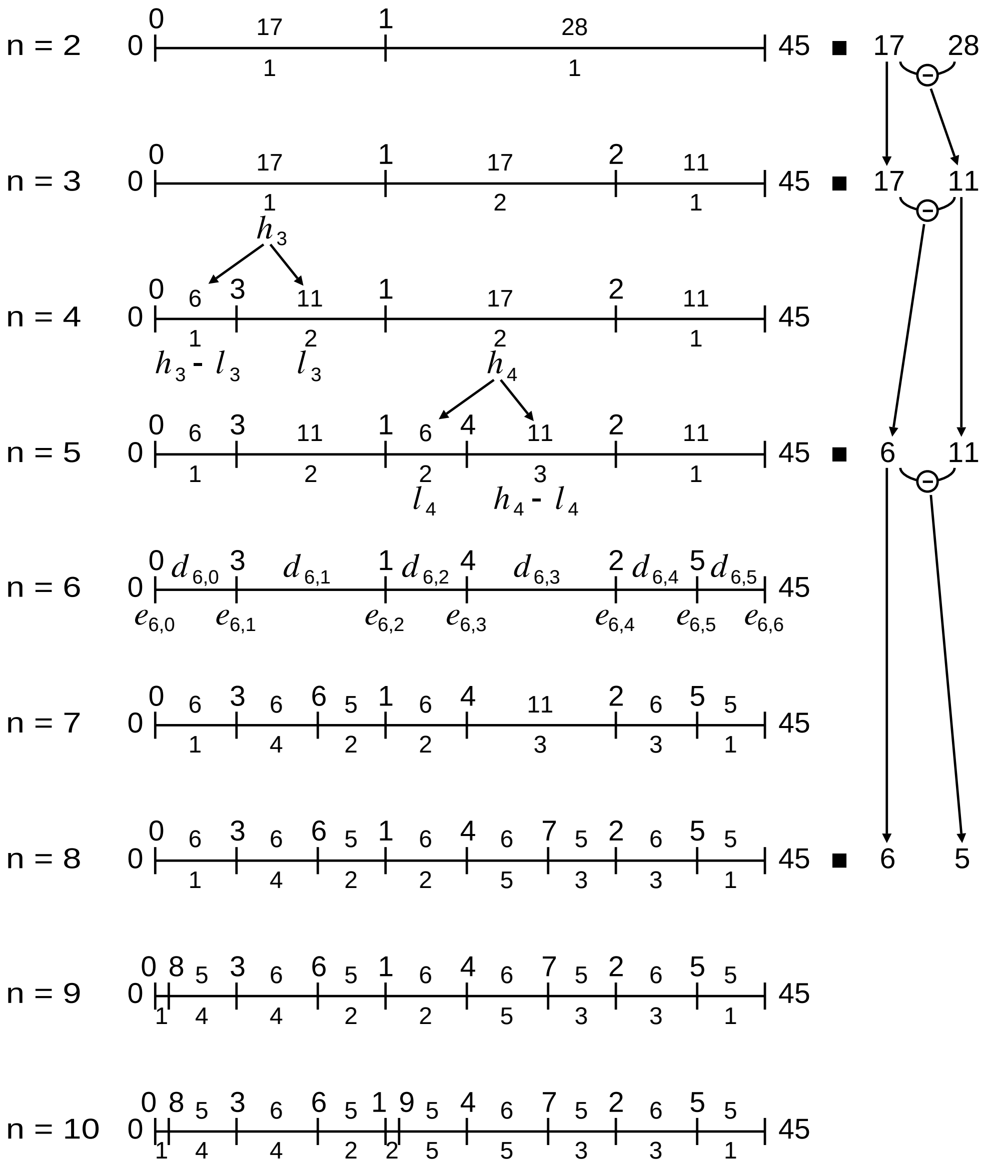

FIG. 2.5: *Construction de E_n et S_n.*

Dans cet exemple, nous avons choisi n suffisamment petit pour éviter d'obtenir des égalités accidentelles et des points multiples. En tenant compte du fait que l'ensemble des irrationnels $\mathbb{R}\backslash\mathbb{Q}$ est dense dans $\mathbb{R}$ et grâce à des propriétés topologiques, nous pouvons supposer que $a \notin \mathbb{Q}$ pour l'étude *mathématique* afin d'éviter de tels problèmes. Ainsi, l'ensemble E_n a exactement n éléments (il ne contient pas de point multiple).

Pour $0 \leqslant i < n$, soient $e_{n,i}$ les valeurs des points de E_n dans $[0, 1[$, dans l'ordre croissant. Nous définissons $e_{n,n} = 1$, qui représente le même point que $e_{n,0} = 0$. Les distances entre deux points consécutifs sur le segment $[0, 1]$ (ou le cercle $\mathbb{R}/\mathbb{Z}$) sont les valeurs $e_{n,i+1} - e_{n,i}$ pour $0 \leqslant i < n$.

Nous donnons maintenant une nouvelle construction de E_n (l'équivalence sera prouvée plus tard), basée sur les distances ; cela permettra de trouver notre algorithme. Pour $n \geqslant 2$, nous définissons un signe $s_n \in \{-1, +1\}$ et une séquence S_n de n quadruplets

$$S_n = (d_{n,i}\,,\ r_{n,i}\,,\ j_{n,i}\,,\ k_{n,i})_{0\leqslant i<n}$$

où $d_{n,i}$ est un réel strictement positif représentant une *distance* (ou une longueur), $r_{n,i}$ est un entier strictement positif représentant un *rang* (induisant un ordre sur les segments $[e_{n,i}, e_{n,i+1}]$ de même longueur), et $j_{n,i}$ et $k_{n,i}$ sont des éléments de $\mathbb{N}$ représentant des *indices de groupe*. Les valeurs initiales sont :

$$d_{2,0} = a, \quad d_{2,1} = 1 - a, \quad r_{2,0} = r_{2,1} = 1, \quad s_2 = \operatorname{sign}(1 - 2a),$$

$$j_{2,0} = 0, \quad k_{2,0} = 1, \quad j_{2,1} = 1, \quad k_{2,1} = 0.$$

Soit $D_n = \{d_{n,i} : 0 \leqslant i < n\}$ l'ensemble des distances de S_n, $h_n = \max D_n$ et $\ell_n = \min D_n$; nous montrerons que D_n a seulement soit deux, soit trois éléments (c'est le théorème des trois distances). La séquence S_n et le signe s_n sont définis par les valeurs initiales et la transformation suivante. Soit i l'indice tel que $d_{n,i} = h_n$ et tel que le rang $r_{n,i}$ soit minimal ; l'unicité de cet indice i est vérifiée pour les conditions initiales et prouvé par récurrence (cela sera expliqué plus bas). Le quadruplet $(d_{n,i}\,, r_{n,i}\,, j_{n,i}\,, k_{n,i})$ est remplacé par deux quadruplets consécutifs définis ci-dessous, et les autres termes de la séquence restent inchangés et dans le même ordre. Ce remplacement correspond à l'ajout d'un point dans E_n. Les distances des deux nouveaux quadruplets sont ℓ_n et $h_n - \ell_n$; leur ordre est déterminé par s_n : ℓ_n puis $h_n - \ell_n$, si $s_n = +1$; $h_n - \ell_n$ puis ℓ_n, si $s_n = -1$. Les nouveaux rangs sont les plus petits entiers strictement positifs tels que tous les rangs associés à une même distance soient différents, c'est-à-dire tous les couples (d, r) doivent être différents ; cette propriété permettra notamment d'assurer l'unicité des futures valeurs de l'indice i considéré plus haut. Notons que, puisque a est irrationnel, on a $h_n - \ell_n \neq \ell_n$, donc il n'y a pas d'ambiguïté. Les indices de groupe $(j_{n,i}\,, k_{n,i})$ sont remplacés par $(j_{n,i}\,, n)$ et $(n\,, k_{n,i})$. Enfin, nous prenons $s_{n+1} = s_n.\operatorname{sign}(h_n - 2\ell_n)$, i.e. le signe s_n change si et seulement si $\ell_{n+1} < \ell_n$; ce choix garantit que les intervalles de même longueur sont découpés de la même façon (voir la figure).

Nous pouvons associer une fonction $f_n : [[0, n - 1]] \to \mathbb{R}/\mathbb{Z}$ à chaque séquence S_n, telle que chaque fonction f_n est une restriction de la fonction $f : \mathbb{N} \to \mathbb{R}/\mathbb{Z}$ vérifiant $f(0) = 0$ et $f(k) - f(j) = d \pmod 1$ pour chaque quadruplet (d, r, j, k) de la séquence S_n.

Reprenons le dernier exemple. Pour $n = 2$, nous avons deux points sur le cercle $\mathbb{Z}/45\mathbb{Z}$, dont les coordonnées respectives sont $f(0) = 0$ et $f(1) = 17$

(modulo 45). Ces deux points forment deux intervalles. Le premier intervalle a pour longueur 17, l'extrémité gauche est le point 0, l'extrémité droite est le point 1, et le rang est 1 (intervalle initial) ; donc $(d, r, j, k) = (17, 1, 0, 1)$. Le second intervalle a pour longueur $45 - 17 = 28$, l'extrémité gauche est le point 1, l'extrémité droite est le point 0, et le rang est aussi 1 ; donc $(d, r, j, k) = (28, 1, 1, 0)$. Maintenant, considérons l'itération passant de $n = 3$ à 4. Pour $n = 3$, nous avons $f(0) = 0$, $f(1) = 17$, et $f(2) = 34$. L'intervalle de longueur $h_3 = 17$ et de rang minimal est $I = (17, 1, 0, 1)$. Ce quadruplet est remplacé par $I' = (6, 1, 0, 3)$ et $I'' = (11, 2, 3, 1)$, dans cet ordre. Puisque $d' + d'' = d$, $j' = j$, $k'' = k$, et $k' = j'' = n$, alors cette transformation définit un nouveau point $f(3) = 6$.

Nous donnons maintenant le théorème montrant que les deux constructions (E_n et S_n) sont équivalentes. Il sera prouvé plus tard.

Théorème 1 *Pour tous $n \geqslant 2$ et $0 \leqslant i < n$, on a : $d_{n,i} = e_{n,i+1} - e_{n,i}$, $e_{n,i} = j_{n,i}.a$ et $e_{n,i+1} = k_{n,i}.a$, i.e., $\forall\, k \geq 0,\ f(k) = k.a$.*

Soient $C_n = \#\{i : d_{n,i} = h_n\}$ et la suite (γ_i, δ_i) définie par la récurrence suivante :

$$(\gamma_0, \delta_0) = (a, 1 - a), \quad (\gamma_{i+1}, \delta_{i+1}) = (\min\{\gamma_i, \delta_i\}, |\gamma_i - \delta_i|),$$

i.e., à chaque itération, on garde le plus petit élément et on remplace le plus grand par la différence des deux. Quand elle est appliquée aux entiers, il s'agit de l'itération de l'algorithme d'Euclide « soustractif » pour calculer le PGCD de deux entiers.

Le théorème suivant dit que certaines séquences contiennent seulement deux distances différentes, et la prochaine paire de distances est obtenue en remplaçant la plus grande distance par la différence des deux distances. Entre deux telles séquences, il y a une période transitoire, où trois distances sont présentes : les deux distances de la séquence initiale et la différence des deux.

Théorème 2 *Il existe une fonction $\varphi : \mathbb{N} \to \mathbb{N}$ strictement croissante telle que $\varphi(0) = 2$, et pour tout $i \geqslant 0$:*

$$D_{\varphi(i)} = \{\gamma_i\,,\, \delta_i\}, \quad \textit{et pour } \varphi(i) < n < \varphi(i+1),\ D_n = \{\gamma_i\,,\, \delta_i\,,\, \delta_{i+1}\}.$$

Pour tous i et n tels que $\varphi(i) \leqslant n < \varphi(i+1)$, on a $\varphi(i+1) = n + C_n$. En particulier, $\varphi(i+1) - \varphi(i) = C_{\varphi(i)}$.

◇ *Preuve.* Le théorème 2 est une conséquence directe de la construction des séquences S_n : nous utilisons seulement le fait qu'à chaque itération, un intervalle de longueur h_n est remplacé par deux intervalles de longueurs ℓ_n et $h_n - \ell_n$. □

Nous déduirons le théorème 1 du lemme suivant.

Lemme 1 *Pour tout n tel que $\#D_n = 2$, i.e., $n \in \varphi(\mathbb{N})$:*

1. $r_{n,0} = r_{n,n-1} = 1$;
2. *si* $s_n = +1$, *alors* $d_{n,0} = \ell_n$ *et* $d_{n,n-1} = h_n$;
   si $s_n = -1$, *alors* $d_{n,0} = h_n$ *et* $d_{n,n-1} = \ell_n$;
3. $j_{n,0} = k_{n,n-1} = 0$,
   $k_{n,0} = \#\{i : d_{n,i} = d_{n,n-1}\}$,
   $j_{n,n-1} = \#\{i : d_{n,i} = d_{n,0}\}$;
4. *pour tout* (d, r, j, k), *les valeurs* $j - r$ *et* $k - r$ *dépendent seulement de la valeur de* d.

◇ *Preuve.* Ce lemme peut être démontré par récurrence sur n.

Pour la preuve, nous notons (i, m) le point i du lemme pour $n = m$ ($1 \leqslant i \leqslant 4$, $m \geqslant 2$). Par exemple, $(2, 6)$ correspond à :

| si $s_6 = +1$, alors $d_{6,0} = \ell_6$ et $d_{6,5} = h_6$;
| si $s_6 = -1$, alors $d_{6,0} = h_6$ et $d_{6,5} = \ell_6$;

Nous pouvons facilement vérifier que, d'après la définition de S_2, le lemme est vrai pour $n = 2$.

Supposons que le lemme soit vrai pour une valeur de n donnée de $\varphi(\mathbb{N})$. Démontrons qu'il est encore vrai pour la prochaine valeur $n' \in \varphi(\mathbb{N})$. Nous rappelons que, entre la valeur initiale n et la prochaine valeur n', chaque intervalle de longueur h_n est découpé en deux.

D'abord, voyons ce qui se passe pour $n + 1$. L'intervalle de longueur h_n et de rang 1 est découpé en deux intervalles de longueurs ℓ_n et $h_n - \ell_n$. D'après le point $(1, n)$ du lemme, cet intervalle est soit le premier, soit le dernier intervalle du segment $[0, 1]$. D'après la construction de S_{n+1}, les deux nouveaux intervalles sont positionnés de telle manière que l'intervalle de longueur $h_n - \ell_n$ se trouve à une extrémité de $[0, 1]$. Cela prouve le point $(1, n + 1)$, donc le point $(1, n')$. En utilisant le point $(2, n)$ et le fait que s change si et seulement si ℓ change, cela prouve aussi le point $(2, n')$.

Maintenant, considérons le point 3. D'après la construction des séquences, les deux indices $j_{m,0}$ et $k_{m,m-1}$ seront nuls pour $m = n'$, et un des indices $k_{m,0}$ et $j_{m,m-1}$, suivant la valeur de s_n, ne changera pas en passant de $m = n$ à $m = n'$. Notons cet indice $\iota(m)$ et l'autre indice $\iota'(m)$. D'après les points $(2, n)$ et $(3, n)$, cet indice $\iota(m)$ est égal à $\#\{i : d_{n,i} = h_n\}$, qui est égal à $\#\{i : d_{n',i} = h_n - \ell_n\}$; ceci prouve la partie du point $(3, n')$ concernant $\iota(n')$. L'autre indice $\iota'(n')$ sera ajouté lorsque $m = n + 1$; il sera donc égal à n. Et nous avons

$$\#\{i : d_{n',i} = \ell_n\} = \#\{i : d_{n,i} = \ell_n\} + \#\{i : d_{n,i} = h_n\} = \#S_n = n,$$

ce qui prouve la dernière partie du point $(3, n')$.

Concernant le point 4, commençons par $d = \ell_n$. Par symétrie, nous pouvons prendre $s_n = +1$ (le cas opposé est similaire). Pour tout (ℓ_n, r, j, k), où

$1 \leqslant r \leqslant j_{n,n-1}$, nous avons $j = r-1$ et $k = k_{n,0}+r-1$. Le nouveau quadruplet $(\ell_n, j_{n,n-1}+1, j, k)$ de S_{n+1} vérifiera $j = j_{n,n-1}$ et $k = n$. Donc nous aurons toujours $j = r-1$ et $k = \#S_n = k_{n,0} + j_{n,n-1} = k_{n,0} + r - 1$. La distance $d = h_n - \ell_n$ (l'autre distance de $S_{n'}$) est nouvelle dans S_{n+1}, donc il n'y a rien d'autre à vérifier pour le moment. Nous venons de prouver le point $(4, n+1)$. Les points $(4, n+2)$ à $(4, n')$ peuvent être déduits du point $(4, n+1)$ et de la construction des séquences. □

Nous pouvons maintenant démontrer le théorème 1.

◇ *Preuve du théorème 1.* Nous démontrons le théorème 1 par récurrence, de manière similaire à la démonstration du lemme. Pour $n = 2$, le théorème 1 est vrai.

Supposons que le théorème soit vrai pour une valeur de n donnée de $\varphi(\mathbb{N})$. Démontrons qu'il est vrai pour $n+1$, puis pour les autres valeurs jusqu'à la prochaine valeur de $\varphi(\mathbb{N})$.

Par symétrie, nous supposons que $s_n = +1$. D'après le lemme, nous avons $j_{n,n-1} + k_{n,0} = n$. Par conséquent,

$$f(n) = j_{n,n-1}.a + \ell_n = j_{n,n-1}.a + d_{n,0} = j_{n,n-1}.a + k_{n,0}.a = n.a,$$

et le théorème est vrai pour $n+1$. En considérant l'intervalle $(h_n, r+1, j, k)$, nous avons $j = j_{n,n-1} + r$ puisque $j - r$ est constant (d'après le lemme). Donc

$$f(n+r) = j.a + \ell_n = (j_{n,n-1} + r).a + k_{n,0}.a = (n+r).a$$

et le théorème est vrai pour $n + r + 1$. □

2.4 Algorithme

Nous rappelons que le segment a une équation de la forme $y = ax - b$, où x vérifie $0 \leqslant x < N$.

Nous allons considérer les $D_{\varphi(i)}$ successifs, et mémoriser la position du point b dans l'intervalle contenant ce point (la distance de b à la borne inférieure de cet intervalle) et la façon dont les intervalles sont découpés, c'est-à-dire, les valeurs h_n, ℓ_n et s_n, où $n = \varphi(i)$. Nous rappelons qu'à chaque itération, les intervalles de longueur h_n sont découpés en deux intervalles de longueurs ℓ_n et $h_n - \ell_n$ (dans l'ordre donné par s_n), et les intervalles de longueur ℓ_n restent inchangés. Nous nous arrêtons lorsque $n \geqslant N$, où N est le nombre initial de valeurs à tester. Ensuite, nous pouvons calculer la distance de b aux deux extrémités de l'intervalle.

En fait, nous voulons savoir si la distance entre le segment et $\mathbb{Z}^2$ est plus grande que ε ou non. Pour éviter d'avoir à calculer la distance de b à la borne supérieure de l'intervalle, nous appliquons l'algorithme à $b + \varepsilon$ au lieu de b, i.e. le segment est décalé de ε vers le bas, et nous avons seulement besoin de connaître la distance de b à la borne inférieure de l'intervalle, qui est directement donnée par l'algorithme.

Notons qu'avec cet algorithme, nous considérons plus de points que voulu, car nous nous arrêtons non pas à N mais à la plus petite valeur de $\varphi(\mathbb{N})$ supérieure ou égale à N. Mais le nombre de points considérés est majoré par le double du nombre de points à tester, i.e. $2N$, ce qui est tout à fait acceptable pour notre problème, puisque la valeur de N peut être choisie de manière à ce que la probabilité pour que le test (dans l'intervalle considéré) échoue soit toujours très petite.

Afin d'éviter d'avoir à recopier ou à échanger des valeurs et tester des « variables d'état » (comme s), nous allons remplacer les variables ℓ et h par les variables $x = d_{n,0}$ et $y = d_{n,n-1}$ (ainsi, nous évitons d'avoir à échanger ℓ et h chaque fois que h devient plus petit que ℓ) et enlever les variables d'état, comme s, en dupliquant le code : une partie pour $s = +1$ et une autre pour $s = -1$. Ainsi, nous connaîtrons la position de ℓ et h sans aucun test : $(x, y) = (\ell, h)$ dans la partie où $s = +1$, et $(x, y) = (h, \ell)$ dans la partie où $s = -1$. Au lieu de comparer ℓ et h et de mettre à jour la variable s, nous pourrons comparer x et y et effectuer un branchement conditionnel. Un autre avantage est que cela utilise moins de variable (pas de variable s et pas de variable temporaire pour l'échange éventuel de ℓ et h).

Nous définissons deux nouvelles variables u et v, qui contiendront le nombre d'intervalles de longueurs respectives x et y ; elles sont seulement utilisées pour calculer n. La variable d sera modifiée de façon à ce qu'elle contienne toujours la distance du point considéré à la borne inférieure de l'intervalle (en particulier, le résultat final) ; sa valeur initiale est $\{b\}$.

Évidemment, l'algorithme sera appliqué à des valeurs rationnelles, alors que l'étude mathématique considérait des valeurs irrationnelles pour des raisons pratiques. L'algorithme reste globalement le même, mais il faut faire attention concernant les cas particuliers ($h = \ell$, puis $\ell = 0$) et s'assurer de ne pas créer de boucle infinie.

Nous avons quatre états possibles :

- $h+$: l'intervalle contenant le point est de longueur h et $s = +1$.
- $h-$: l'intervalle contenant le point est de longueur h et $s = -1$.
- $\ell-$: l'intervalle contenant le point est de longueur ℓ et $s = -1$.
- $\ell+$: l'intervalle contenant le point est de longueur ℓ et $s = +1$.

Nous pouvons regrouper les états $h-$ et $\ell+$ (le point est dans l'intervalle de longueur x), ainsi que les états $h+$ et $\ell-$ (le point est dans l'intervalle de longueur y). L'algorithme 2.1 page suivante peut être implémenté de différentes façons ; une optimisation peut consister à déplacer, enlever ou ajouter des instructions.

Si le résultat d est supérieur à 2ε, alors la distance entre le segment et $\mathbb{Z}^2$

Algorithme 2.1 Minoration de la distance d'un segment de droite à $\mathbb{Z}^2$.

Initialisation : $x = \{a\}$; $y = 1 - \{a\}$; $d = \{b\}$; $u = v = 1$;
Boucle :
```
    si (d < x)
        tant que (x < y)
            si (u + v ⩾ N) fin
            y = y − x; u = u + v;
        si (u + v ⩾ N) fin
        x = x − y; v = v + u;
    sinon
        d = d − x;
        tant que (y < x)
            si (u + v ⩾ N) fin
            x = x − y; v = v + u;
        si (u + v ⩾ N) fin
        y = y − x; u = u + v;
```
Résultat : d

est supérieure à ε. Dans le cas contraire, le test échoue, et un test plus précis est nécessaire (par exemple, en découpant le domaine et le segment, ou en utilisant un autre algorithme, plus lent).

2.5 Conclusion

Cet algorithme a d'abord été implémenté sur des SparcStations Sun pour tester l'exponentielle en double précision dans l'intervalle $[\frac{1}{2}, 1[$ dans des conditions non optimales. Nous avons noté une accélération d'un facteur 90 par rapport à l'algorithme naïf. Des résultats plus précis concernant les temps de calcul seront donnés dans la section 3.3, le facteur d'accélération dépendant beaucoup des différents paramètres : fonction, intervalle, précision des calculs, qualité de l'implémentation, machine, etc.

Remarquons que cet algorithme « contient » l'algorithme d'Euclide (considérer les variables x et y), qui est utilisé pour développer un réel a en fraction continue. Quand les quotients partiels de ce développement en fraction continue sont bornés, le nombre de points générés $(u + v)$ est une fonction exponentielle du nombre d'itérations ; par conséquent, le nombre d'itérations a pour complexité $O(\log N)$. Mais en pratique, N est borné (par exemple, $N < 100\,000$), et seuls les premiers quotients partiels interviennent. Donc c'est la taille des premiers quotients partiels qui est importante. Quand un des tous premiers quotients partiels est très grand, l'algorithme devient très lent ; par exemple, x est beaucoup plus petit que y (ce qui signifie qu'un quotient partiel est grand) et u est petit (ce qui signifie qu'on est dans les toutes premières itérations, donc le quotient partiel est un des premiers), donc le nombre de points ajoutés à chaque itération reste petit, et il faudra beaucoup d'itérations pour atteindre N. Il est possible d'améliorer l'algorithme dans ces cas, par exemple en utilisant des divisions (adaptation de l'algorithme d'Euclide classique), mais le cas général, beaucoup plus courant (sauf cas particulier, comme $\sin x$ avec x petit, car $\sin x \approx x$), sera ralenti. Une solution serait de sélectionner l'algorithme à utiliser suivant la valeur de a : si a est proche d'un rationnel « simple », alors c'est algorithme avec les divisions qui sera choisi, sinon c'est l'algorithme soustractif. D'autre part, la valeur de a ne change pas beaucoup d'un intervalle au suivant ; nous pouvons en tenir compte pour sélectionner l'algorithme de manière un peu plus globale, et nous pouvons peut-être utiliser des formules données par Valérie Berthé dans [7]. Ceci étant, la prise en compte de ces cas particuliers pour l'optimisation n'est pas fondamentale, et l'algorithme reste très rapide en moyenne.

Le travail présenté dans ce chapitre a fait l'objet de publications : [30, 31].

Chapitre 3

Implémentation des tests exhaustifs

3.1 Implémentation choisie

3.1.1 Introduction

Nous allons maintenant présenter comment les algorithmes décrits dans les chapitres précédents ont été implémentés. Nous ne nous sommes intéressés ici qu'à la double précision, d'une part parce que la simple précision peut être résolue avec des programmes existants (n'ayant pas besoin d'être très rapides), d'autre part parce que les précisions plus grandes ne sont pas pour le moment envisageables, car elles demanderaient trop de temps et sont de toute façon moins utilisées que la double précision. Cependant, les programmes qui vont être présentés ont été écrits de façon générique et peuvent être utilisés tels quels avec d'autres précisions (mais certains doivent être adaptés pour les très grandes précisions, à cause de la taille limitée des variables).

Les tests vont se dérouler en 3 étapes :

– La première étape consiste à éliminer la plupart des arguments : en général, on testera si 32 bits consécutifs sont tous nuls, une grande partie des calculs internes se faisant sur 64 bits (en virgule fixe, à l'aide de l'arithmétique entière du processeur) ; donc selon les hypothèses probabilistes, on ne gardera en moyenne qu'un argument sur 2^{32} (environ 4 milliards). Avec une mantisse de 54 bits (53 bits pour la double précision + 1 bit pour les résultats sur la réciproque), on obtient environ $2^{53-32} = 2^{21}$ pires cas par exposant, ce qui fait au total plusieurs dizaines ou centaines de millions de pires cas par fonction testée. Les nombres trouvés ne sont pas forcément de réels pires cas, par exemple si dans le couple (x, y), x et y sont impairs (cf section 1.6), ou quand il y a un changement d'exposant dans le domaine. La première étape est la plus lente ; elle tourne donc en parallèle sur plusieurs dizaines de machines. En utilisant des SparcStations Sun du réseau de l'ENS, nous avons mesuré qu'elle prend entre quelques heures et quelques semaines par exposant, suivant les fonctions et les exposants testés ; par exemple, le test de l'exponentielle est rapide pour l'exposant 0, mais beaucoup plus lent pour l'exposant 8, car la fonction s'approche moins bien.

– La deuxième étape consiste à réduire le nombre de pires cas en utilisant les résultats de la première étape. Elle est lancée juste après la première étape ; elle tourne en fait en parallèle (pipeline) avec la première étape, le domaine étant découpé en intervalles. Tous les « faux » pires cas sont détectés et éliminés. Les pires cas restants sont stockés dans un fichier unique, destiné à être définitivement gardé sur disque. Ce fichier pourra être lu par d'autres programmes (celui de la troisième étape, ou des programmes de statistique) ou par un utilisateur. La deuxième étape est suffisamment rapide pour être lancée sur une seule machine rapide (UltraSparc) ; elle prend alors plusieurs heures de temps CPU, mais peut passer en fait beaucoup de temps à attendre les résultats partiels de la première étape.
– La troisième étape consiste à réduire encore le nombre de pires cas et à calculer les pires cas de la fonction réciproque, en utilisant le fichier produit par la deuxième étape. Le programme est lancé par l'utilisateur quand celui-ci le souhaite. Cette étape est très rapide : elle prend en gros de quelques secondes à quelques dizaines de secondes par exposant.

La plupart des programmes sont écrits en Perl, ce langage étant très adapté à la manipulation de fichiers texte (les données étant transmises et modifiées sous forme de texte lisible par l'homme), de processus, et de signaux (malgré les problèmes connus des versions actuelles de l'interpréteur Perl, 5.004 et 5.005, concernant la gestion des signaux). Seuls les tests eux-mêmes (calcul des valeurs successives d'un polynôme et minoration de la distance d'un segment de droite à $\mathbb{Z}^2$) sont écrits en assembleur Sparc, avec une petite interface en C pour appeler la fonction écrite en assembleur et écrire les résultats dans un fichier.

Certains calculs sont effectués avec Maple : l'approximation de la fonction par des polynômes (pour la première étape), et les deuxième et troisième étapes. Le principal problème consistait à tenir compte des différentes versions de Maple, dépendant de la machine utilisée. Avec Maple V Release 4, il faut utiliser l'option `-f` pour que Maple se termine automatiquement lorsque l'entrée standard a été fermée (comportement voulu). Mais avec Maple V Release 5, cette option n'existe plus (par défaut, Maple se termine automatiquement), et si elle est utilisée, Maple se termine immédiatement avec une erreur ! Malheureusement, il n'y a pas de moyen propre pour lancer Maple comme il le faut. La solution choisie est d'essayer avec l'option `-f` et de faire faire un calcul (très simple) à Maple : on lui donne l'expression « 1 ». Si le résultat n'est pas 1 ou si cela provoque un signal SIGPIPE, Maple est relancé sans l'option `-f`.

Le second problème vient de l'opérateur *ditto*[1], qui n'est pas représenté par le même caractère avec les deux versions : le *double-quote* (`"`) avec l'ancienne version, et le symbole « pourcent » (`%`) avec la nouvelle version, puisque le *double-quote* est dorénavant utilisé comme délimiteur de chaîne de caractères. Bien que l'on puisse savoir quel caractère il faut (puisque la version de Maple est maintenant connue), nous avons choisi de ne pas utiliser cet opérateur pour éviter tout problème à l'avenir.

Avant d'exposer en détail les 3 étapes, expliquons le choix qui a été fait pour le nombre de bits testés lors la première étape, car ce choix est étroitement lié aux deux premières étapes. Un nombre de 32 bits correspond en fait à la taille des registres des machines utilisées ; c'était la raison choisie au départ, mais finalement ce n'est pas une raison valable. Ce choix est discutable, mais difficile à effectuer. En effet, si on diminue le nombre de bits testés, cela permet de calculer avec une précision plus faible, mais les pires cas seront plus nombreux et la deuxième étape sera plus longue ; d'autre part, l'algorithme rapide consistant à minorer la distance entre un segment et $\mathbb{Z}^2$ échouera plus souvent. En revanche, si on augmente le nombre de bits testés, c'est l'inverse. Le meilleur choix correspond à un temps total des deux premières étapes le plus petit possible, mais ce temps est très difficile à évaluer (estimer ou mesurer), car il dépend de nombreux paramètres : fonction, intervalle, degré de parallélisation, charge des machines (difficile à prévoir), plantage de processus ou de machines, etc. Il est difficile d'estimer les variations de temps de calcul suivant le nombre de bits testés, mais une bonne méthode est de voir si une des deux étapes est prédominante. Il semble qu'en pratique, si la fonction testée est régulière (numériquement), le temps de calcul pour chacune des deux premières étapes est du même ordre de grandeur avec le choix de tester 32 bits (en tenant compte évidemment de la parallélisation). Mais par exemple, dans le cas de l'exponentielle avec de gros exposants (disons, supérieurs à 4), la première étape (même parallélisée) devient beaucoup plus lente que la deuxième étape ; en effet, la fonction s'approche plus difficilement par des polynômes, donc la taille des petits segments déterminant les pires cas (cf figures 1.2 et 2.1) est plus grande, et l'algorithme rapide de minoration de la distance entre un segment et $\mathbb{Z}^2$ échoue plus souvent. Dans ce cas, changer le nombre de bits testés n'apporterait pas grand chose.

Suivant le nombre de pires cas voulus par l'utilisateur, il peut être nécessaire de tester moins de bits que la valeur par défaut 32. C'est par exemple le cas de $\log(x)$, dans des intervalles où x est un nombre machine très proche de 1. Les pires cas pour le logarithme auront par exemple seulement une vingtaine de bits identiques après la mantisse, mais ces pires cas sont surtout intéressants pour la réciproque $\exp$. C'est valable aussi pour le logarithme en base 2. Par exemple,

$$\begin{aligned}&\log_2(1 + 15561019 \times 2^{-53})\\&\quad = 1.0101011010001110100011001110110001001111011001101011\ 1\,1^{27}\ 01\ldots\ 2^{-29}\end{aligned}$$

[1]Cet opérateur permet de rappeler le dernier résultat calculé.

a seulement 27 bits identiques après sa mantisse de 54 bits, alors que le pire cas correspondant pour la réciproque 2^x est très intéressant, car il a 57 bits identiques après sa mantisse de 54 bits :

$$2^{1.0101011010001110100011001110110001001111011001101100 \times 2^{-29}}$$
$$= 1.00000000000000000000000000000111011010111000100111011\,1\,0^{57}\,1110\ldots$$

3.1.2 Première étape

La première étape est implémentée à l'aide de deux scripts Perl, `t3-client` et `test32f`, ainsi qu'un ensemble de fichiers texte `def-exp`, `def-log`, `def-cos`, `def-sin`, etc. définissant les fonctions testées ; ces fichiers contiennent les données exposées dans la section 1.3, les expressions mathématiques étant écrites sous forme d'expressions Maple. Le script `test32f` a pour but de générer les sources C et assembleur du programme de tests lui-même. Le script `t3-client` s'occupe de calculer les paramètres pour `test32f`, de lancer les tests, et de stocker les résultats dans un fichier ; il s'occupe aussi des communications avec le serveur pour la distribution des calculs (voir la section 3.2 sur ce point).

Exposons d'abord les points principaux de l'algorithme implémenté par les sources C et assembleur générés par `test32f`. Puis nous décrirons le script `test32f`, ainsi que la partie du script `t3-client` concernant les calculs (pas celle concernant les communications avec le serveur).

3.1.2.1 Algorithme

Nous avons une fonction f et un intervalle I (contenant un certain nombre de nombres machine), et nous voulons tester t bits d'une approximation de la fonction f à une erreur fixée, pour chaque argument de l'intervalle I. En général, I correspondra à tout un exposant : $[2^k, 2^{k+1}[$. Mais il peut être restreint dans des cas particuliers : par exemple, c'est le cas de la fonction exponentielle au voisinage de l'*overflow*.

L'intervalle I est d'abord découpé en sous-intervalles J_i de I, ayant tous exactement la même longueur. La taille de ces intervalles J_i sera choisie de façon à ce que le test de tous les nombres machine de J_i prenne suffisamment de temps (notamment pour éviter trop de communications), mais pas trop, car un processus sera lancé pour chaque intervalle J_i, et si ce processus est tué, les calculs effectués seront perdus. Disons que nous viserons un temps de test de J_i compris entre 5 minutes et 2 heures. Cela ne peut pas être très précis, car le temps de test peut dépendre beaucoup de l'intervalle J_i en question (il peut y avoir un facteur 100, suivant que la pente de la fonction est proche d'un rationnel simple ou non, cf 2.5) ainsi que de la machine (facteur 10), en supposant la machine utilisée à 100%. En général, I sera découpé en $2^{13} = 8192$ intervalles J_i de $2^{53-13} = 2^{40}$ nombres machine.

Dans J_i, la fonction f est d'abord approchée par un polynôme P_i dont le degré d_i est déterminé par essais successifs (comme indiqué dans la section 1.3 et expliqué plus précisément dans la section 3.1.2.2). Normalement, nous devons tester une égalité du type

$$|P_i(x) - y| < 2^{-m},$$

où x et y sont des nombres machine, c'est-à-dire tester si un certain nombre t de bits de $P_i(x)$ sont tous égaux (soit à 0, soit à 1). Mais nous pouvons ajouter 1 au dernier bit testé (le bit de poids faible) en modifiant le terme constant de P_i, et tester si les $n_t = t - 1$ premiers bits sont tous nuls. Sur les processeurs Sparc (qui sont les processeurs des machines que nous utiliserons, mais cela reste en général valable pour d'autres processeurs), cela donne une implémentation plus rapide.

L'intervalle J_i est découpé en n_k sous-intervalles $K_{i,j}$ de même longueur[2], dans lesquels P_i est approché par des polynômes $Q_{i,j}$ de degré 2 (en utilisant l'algorithme décrit en section 1.5). Maintenant, plaçons-nous dans un intervalle $K_{i,j}$ (nous le noterons aussi K pour simplifier, de même que le polynôme $Q_{i,j}$ sera aussi noté Q). En négligeant le terme de degré 2, Q est approché par un polynôme Q' de degré 1, et l'algorithme de minoration de la distance entre le segment correspondant à Q' et $\mathbb{Z}^2$ peut être alors appliqué (sauf si Q s'approche trop mal par Q', auquel cas on fait comme si l'algorithme de minoration avait échoué). En cas de succès, on déduit qu'il n'y a aucun pire cas, et c'est terminé. En cas d'échec, l'intervalle K est découpé en n_ℓ sous-intervalles L_k (si $n_\ell \neq 1$, 1 étant une valeur acceptée), dans lesquels Q est approché par des polynômes de degré 1, et on réapplique l'algorithme de minoration. En cas de succès pour un intervalle L_k, on peut passer à l'intervalle suivant L_{k+1} (s'il y en a un), à l'aide de quelques additions et décalages. En cas d'échec, la méthode des différences finies est appliquée au polynôme Q sur L_k, puis on passe à l'intervalle suivant. Note : si $n_\ell = 1$, alors en cas d'échec de l'algorithme de minoration sur l'intervalle K, on passe directement à la méthode des différences finies sur K.

La méthode des différences finies (appliquée à un polynôme Q de degré 2) consiste ici à effectuer deux additions modulo 1, puis à tester si un certain nombre de bits sont tous nuls. L'implémentation tient compte du fait qu'en général, ces bits ne sont pas tous nuls. Les branchements sont donc optimisés de façon à favoriser ce cas-là.

3.1.2.2 Script `test32f`

Le script `test32f` permet de tester une fonction dans un intervalle donné. Il est lancé avec 6 arguments, dont les 4 premiers définissent complètement les arguments à tester :

[2] Le nombre n_k est indépendant de i. La lettre k est ici un symbole faisant référence aux $K_{i,j}$ et n'est pas un nombre.

1. Le *nom de la fonction*, qui indique quel fichier `def-*` devra être lu.
2. La *mantisse* en base 2 de « l'argument initial » de l'intervalle x_0 (plus petit nombre machine de l'intervalle), sous forme d'une suite de chiffres 0 et 1. La mantisse doit être normalisée, i.e. le premier bit doit être un 1. Le nombre de bits définit la longueur de la mantisse (les 0 situés tout à droite sont donc importants). À cause des deux types de mode d'arrondi (et pour avoir les résultats concernant la réciproque), il faut ajouter un bit à la mantisse. Par exemple, pour la double précision, la mantisse devra faire 54 bits.
3. L'*exposant* (écrit en base 10) de l'argument initial x_0 : logarithme binaire du poids du bit le plus significatif de la mantisse (premier bit).
4. Le *nombre d'arguments* n_x à tester (écrit en base 10), entre 2 et $2^{64} - 1$, avec la contrainte suivante : l'exposant des arguments doit être constant dans l'intervalle. En fait, `test32f` pourra tester un peu plus d'arguments que demandé, de façon à avoir des tests d'arrêt simples et rapides pour les différentes boucles. Notons que pour les arguments testés supplémentaires, les résultats peuvent être faux (en particulier parce que l'approximation n'est plus valide), mais ce sera détecté à la deuxième étape.
5. Le *préfixe* : il s'agit d'une chaîne de caractères déterminant les noms des fichiers qui seront créés :
   - *`prefix`*`.log` : fichier de *log* ;
   - *`prefix`*`.c` : source C ;
   - *`prefix`*`.s` : source assembleur.

   Le fichier de *log* sert à plusieurs choses : des informations importantes y sont stockées pour savoir quels sont les bits testés exactement (l'exposant de $f(x)$ n'étant pas forcément constant et pas forcément connu à l'avance) ; des informations comme le degré du polynôme P_i ou la taille des nombres manipulés y sont aussi stockées et peuvent toujours être intéressantes pour connaître l'efficacité de l'algorithme, et voir éventuellement s'il est préférable de diminuer la taille des intervalles J_i pour être plus rapide (les approximations sont d'autant meilleures que les intervalles sont petits) ; avec certains paramètres, il se peut que les tests ne soient pas possibles, et le fichier de *log* peut permettre de savoir pourquoi.
6. Le *nombre de chiffres* `Digits` pour les calculs intermédiaires avec Maple. Ce nombre de chiffres serait trop compliqué à calculer automatiquement. S'il est trop petit, les résultats n'auront pas assez de précision, et cela sera détecté (notamment à l'aide du *package* d'arithmétique d'intervalles intpak) ; `test32` se terminera alors avec une erreur. Mais il ne faut pas le choisir très grand, sinon les calculs prendront beaucoup de temps. Par défaut, `t3-client` utilise 300 chiffres, mais ce choix peut facilement être changé par option.

En plus des arguments, on peut donner autant d'options que nécessaire. Voici les plus importantes :

– Option `-n` suivie d'un entier n_t compris entre 1 et 32 : il s'agit du nombre de bits testés. Sa valeur par défaut est 32.

– Option `-k` suivie d'un entier strictement positif r : cet entier permet de fixer une valeur maximale de la longueur (mesurée en nombres machine) des sous-intervalles $K_{i,j}$: 2^r nombres machine. En pratique, `test32f` pourra imposer une longueur inférieure à 2^r pour des questions de précision. Par défaut, aucune longueur maximale n'est fixée.

– Option `-l` suivie d'un entier naturel s : cet entier permet de fixer le nombre de sous-intervalles L_k d'un intervalle K : 2^s. L'entier s vaut par défaut 0.

– Option `-p` suivie d'un entier naturel p : cet entier permet d'imposer une taille minimale de la mémoire tampon qui contiendra les pires cas d'un intervalle J_i : 2^p. Mais cette option n'est normalement pas nécessaire, car `test32f` est capable de prévoir une taille suffisante et pas trop grosse en utilisant les hypothèses probabilistes. Si jamais la taille de la mémoire tampon est trop petite, le programme de tests se terminera avec une erreur (sans renvoyer les pires cas trouvés) ; le programme devra être relancé avec une mémoire tampon plus grande. En pratique, cela ne se produit jamais, grâce aux hypothèses probabilistes.

– Option `-t` : cette option permet de sélectionner le mode *test* : des informations sur le succès de l'algorithme de minoration sur $K_{i,j}$ sont affichées à l'écran. Cela permet de voir si cet algorithme est très utile dans les intervalles considérés, ou bien s'il faut augmenter la valeur de s. Cette option ne doit pas être utilisée pour les vrais tests.

Les options `-k` et `-l` permettent d'optimiser le découpage de J_i et de $K_{i,j}$ en sous-intervalles. Le passage d'un intervalle K au suivant prend beaucoup plus de temps que le passage d'un intervalle L au suivant. L'option `-k` ne sera donc pas utilisée en pratique. Pour savoir quelle valeur s donner à l'option `-l`, on peut soit tester avec l'option `-t` (mais les informations affichées ne seront pas forcément très significatives), soit mesurer les temps en utilisant différentes valeurs et prendre la meilleure. Ces tests pour connaître la meilleure valeur de s ne se feront que sur un intervalle J_i particulier, la valeur optimale de s changeant très peu fréquemment (voir les exemples en section 3.3). On fera tout de même attention à ne pas choisir un intervalle J_i contenant des pentes de f proches d'un rationnel simple, car la valeur obtenue ne serait pas du tout significative. Il n'est probablement pas possible de connaître la valeur optimale pour s sans tester différentes valeurs, car trop de paramètres entrent en jeu : fonction testée, intervalle testé, plate-forme utilisée, etc.

Maintenant, expliquons ce que fait `test32f`.

1. Certaines variables sont initialisées et les options sont lues et traitées.

2. Les arguments sont traités et les principaux paramètres sont mémorisés dans le fichier de *log* : en cas de problème, c'est le seul moyen de les connaître si `test32f` est lancé par `t3-client`.

3. Le fichier définissant la fonction sous forme de formule de Taylor est lu. La première ligne contient la définition de la fonction, utilisant l'arithmétique d'intervalles, avec la notation d'intpak. Cela peut être simplement `&exp(x)` pour l'exponentielle, ou des expressions plus compliquées, comme `&exp(x &* &ln(2))` pour 2^x. La fonction $f(x)$ sera approchée par une fonction polynomiale $P(t)$, où $t = x - x_0$ (rappelons que x_0 est l'argument initial) est compris dans un intervalle $[0, T[$. Les lignes suivantes du fichier sont formées d'un motif, suivi d'une tabulation et d'une expression Maple. Le motif peut être :
   - Une constante : la définition de constantes permet d'avoir des calculs un peu plus rapides. Par exemple, pour 2^x, on peut faire intervenir la constante `&exp(x0 &* &ln(2))`, qui intervient dans les coefficients de la formule de Taylor.
   - Le caractère `<` : l'expression Maple suivante donne un majorant de T pour lequel la définition de la fonction est valable ; `test32f` vérifiera que la valeur de T est bien inférieure à ce majorant, et dans le cas contraire, renverra une erreur.
   - Le caractère `E` : l'expression Maple suivante donne un majorant de l'erreur due à l'approximation de Taylor, en fonction du degré d du polynôme P et de la borne T. Ce motif doit apparaître au moins une fois (en fait, une fois seulement).
   - Une condition en fonction de i : l'expression Maple suivante donne le coefficient associé à t^i si la condition est vérifiée. La condition n'est pas une expression Maple, mais une expression Perl dans laquelle le caractère `i` est remplacé par sa valeur (entier naturel). Par exemple, la condition `1` est toujours vérifiée, la condition `i&1` est vérifiée si et seulement si i est impair (utile pour les fonctions sinus et cosinus), la condition `i>=1` est vérifiée si et seulement si i est supérieur ou égal à 1... Pour une approximation à l'ordre d, au moins une condition doit être vérifiée pour chaque i compris entre 0 et d.

   Notons que ce format ne permet de définir que certaines fonctions. Si besoin est, il pourra être étendu ultérieurement. Mais le principal problème peut être d'ordre mathématique : il faut savoir exprimer formellement les coefficients de Taylor (éventuellement jusqu'à un degré D fixé)... ou alors il faut utiliser une autre méthode.
4. Le programme Maple est lancé avec un double *pipe* : écriture pour lancer des calculs, lecture pour récupérer les résultats.
5. La taille de la mémoire où seront stockés les pires cas est calculée : 2^q avec $q = \max(\lceil \log_2(n_x + 1/2) \rceil - n_t + 3, p)$, où n_x est le nombre d'arguments à tester, n_t est le nombre de bits testés (comparaison à 0) et p l'entier donné éventuellement par l'option `-p` (sa valeur par défaut est 6). En général, n_x est une puissance de deux : $n_x = 2^h$. Dans ce cas, la formule devient : $q = \max(h - n_t + 4, p)$. On prévoit donc un buffer pouvant contenir 16 fois le nombre moyen de pires cas, ce qui devrait être largement suffisant.

6. Le domaine de l'approximation polynomiale est vérifié. D'abord, on vérifie que l'exposant des arguments est constant dans l'intervalle. Ensuite, on vérifie que la valeur de T est inférieure à la borne donnée dans le fichier de définition de la fonction (si une telle borne est donnée).
7. La plage des exposants des valeurs de la fonction dans l'intervalle considéré est déterminée grâce au *package* d'arithmétique d'intervalles. Notons qu'ici, le *package* d'arithmétique d'intervalles n'est pas uniquement utilisé pour contrôler l'erreur. On en déduit le poids des bits qui seront testés, ainsi que l'erreur maximale ε avec laquelle les valeurs de la fonction devront être calculées. Si 2^u est le poids du bit testé le plus significatif, alors $\varepsilon = 2^{u-t} = 2^{u-n_t-1}$ (cf section 3.1.2.1).
8. La fonction f sera approchée par un polynôme P_0 de $\mathbb{R}[X]$. Le degré d de P_0 est déterminé de la façon suivante : on part de $d = 2$, on calcule un majorant ε_0 de l'erreur donnée dans le fichier de définition de la fonction, et si cette erreur est inférieure à $\varepsilon/8$, alors on garde le degré d ; sinon, on recommence avec le degré $d + 1$. L'erreur maximale pour la suite des opérations devient donc maintenant : $\varepsilon_P = \varepsilon - \varepsilon_0$.
9. La fonction f est approchée par un polynôme P_0 de la façon suivante. Nous voulons approcher $f(x)$ par $P_0(t)$, où $t = x - x_0$. Soit le polynôme P vérifiant $P(m) = P_0(t)$, où $t = m.\mathrm{d}x$, $\mathrm{d}x$ étant la différence entre deux arguments consécutifs. On calcule, en arithmétique d'intervalles, les coefficients de P dans la base $\{1, X, X^2, X^3, ...\}$, à l'aide des expressions symboliques données dans le fichier de définition de la fonction. Si plusieurs conditions sur le degré du monôme sont vérifiées, alors l'une des expressions correspondantes est choisie. Si aucune condition n'est vérifiée (cela peut arriver avec certaines fonctions, dont les approximations sont définies jusqu'à un certain degré), alors `test32f` se termine avec une erreur.
10. L'intervalle J va être décomposé en plusieurs sous-intervalles K_j de même longueur (éventuellement excepté le dernier, qui peut être plus court, suivant la valuation dyadique de n_x), dans lesquels P sera approché par des polynômes Q_j de degré 2. Le nombre d'arguments $k = 2^q$ que contiennent ces sous-intervalles est calculé de la façon suivante. Il est intéressant que k soit le plus grand possible, donc nous cherchons d'abord des majorants simples :
    – Nous utilisons d'abord le fait que nous faisons des approximations par des polynômes de degré 2, sur 64 bits. En nous ramenant modulo 1 (le premier bit testé étant celui de poids 2^{-1}), nous avons : $\varepsilon = 2^{-t-1} = 2^{-n_t-2}$. Nous appliquons la formule de la section 1.5.3.3 : l'erreur due à l'approximation sur 64 bits est majorée par :
    $$2^{-64}\left[\binom{k}{1} + \binom{k}{2}\right] = 2^{-65}k(k+1).$$
    Nous voulons $2^{-65}k(k+1) \leqslant 2^{-n_t-2}$, et avec $k = 2^q$, nous voulons donc : $2^{2q-64} \leqslant 2^{-n_t-2}$, i.e. $q \leqslant (62-n_t)/2$. Notons que pour $n_t = 32$

(valeur généralement utilisée), $q \leqslant 15$, i.e. $k \leqslant 32768$.

- L'implémentation de l'algorithme de minoration dans K_j demande $q \leqslant 16$. Cette limite pourrait être retirée en modifiant très légèrement l'implémentation, mais cela n'apporterait pas grand chose car on a généralement $q \leqslant 15$ (cf point précédent).
- Il y a évidemment le majorant éventuel donné par l'utilisateur en option.
- La longueur de l'intervalle J (nombre d'arguments n_x) donne aussi un majorant de k.

On calcule le minimum de tous ces majorants : c'est la valeur provisoire de k. Sa valeur définitive sera déterminée plus tard.

11. On calcule, en arithmétique d'intervalles, les coefficients initiaux $a_i(0)$ de P dans la base $\{1, X, \frac{X(X-1)}{2}, ...\}$ (cf section 1.5.2.2), à l'aide des coefficients de P dans la base $\{1, X, X^2, ...\}$.
12. La valeur définitive de k est déterminée de la façon suivante. Tant que le majorant de l'erreur donné en section 1.5 (en tenant compte de ε_1 parce que $\deg Q = 2$, ainsi que de ε_2 parce que les 3 coefficients de Q seront sur 64 bits) est supérieur ou égal à $\frac{4}{7}\varepsilon_P$, la valeur de k est divisée par 2. Si on arrive à $k = 1$, `test32f` s'arrête avec une erreur ; mais cela ne devrait jamais arriver (sauf pour des fonctions très irrégulières). Note : l'erreur maximale autorisée prise en compte pour le calcul de k est supérieure à $\frac{4}{7} \times \frac{7}{8}\varepsilon = \varepsilon/2$, i.e. au moins à la moitié de l'erreur finale. Une telle part dans l'erreur finale a été choisie, car la valeur de k est importante concernant le temps de calcul.
13. Du nombre de sous-intervalles L_k (option `-l`) et du nombre d'arguments que contiennent les intervalles K, on en déduit le nombre d'arguments que contiennent les intervalles L_k.
14. Comme indiqué dans la section 1.5.2, pour $0 \leqslant i \leqslant 2$, les coefficients $a_i(n)$ des polynômes P_n de degré 2 sont des polynômes en n de degré $d-i$. Les valeurs de ces polynômes seront calculées (par le programme généré) à l'aide de la méthode des différences finies, où tous les calculs seront effectués modulo 1 et les coefficients $a_{i,j}$ seront exprimés sur n_j bits en virgule fixe par un tableau d'entiers sur 32 bits (n_j est donc un multiple de 32) : le bit de poids faible de $a_{i,j}$ a pour poids 2^{-n_j}. Les valeurs des n_j sont calculées dès maintenant (algorithme 3.1 page suivante), de façon à ce que les coefficients des polynômes Q puissent être calculés à 2^{-64} près.
15. On calcule les $a_i(n)$, comme expliqué en section 1.5.2.2.
16. On calcule les $a_{i,j}(0)$, comme expliqué en section 1.5.2.2, et on ajoute 2^{u-n_t} à $a_{0,0}(0)$. On décompose ces valeurs en mots de 32 bits en vue de les inclure dans le source assembleur.
17. On génère les sources C et assembleur.

Algorithme 3.1 Calcul des n_j.

$\varepsilon_{\max} = 2^{-64}$;
$n = 64$;
pour j **allant de** 0 **à** $d - 1$
    $b = \binom{n_x}{j+1}$;
    tant que $(\varepsilon_{\max} - \frac{3}{2}.2^{-n}.b < 0)$
        $n = n + 32$;
    $\varepsilon_{\max} = \varepsilon_{\max} - 2^{-n_j}.b$;
    $n_j = n$;
$n_d = n$;

Il est prévu de réécrire toute la partie assembleur en C ISO, de manière à pouvoir faire tourner les programmes sur n'importe quelle machine. Il y a de nombreux calculs en multiprécision, faisant intervenir le bit *carry* du processeur, et allant au delà du plus grand type entier des compilateurs C classiques (64 bits si on accepte le type `long long`). Malheureusement, ce bit n'est pas directement accessible en C (ainsi que dans d'autres langages de haut niveau). La solution est d'effectuer les opérations en arithmétique non signée, la norme C exigeant que les calculs se fassent alors en arithmétique modulaire. Une expression du type `a + b < a` (`a` et `b` étant d'un type `unsigned`) permet de détecter un dépassement de capacité et faire de la multiprécision, mais en tenant compte des retenues, il est peu probable que les compilateurs soient capables de détecter ce genre d'astuces et de générer du code assembleur optimal. Une autre solution, certainement meilleure, est d'utiliser une bibliothèque de calculs en multiprécision, style GMP, où les routines de calculs en multiprécision sont déjà écrites en assembleur. Mais l'appel des fonctions correspondantes fera quand même perdre du temps.

3.1.2.3 Script `t3-client`

Le script `t3-client` permet de lancer les tests pour une fonction donnée, une taille de mantisse donnée, et un exposant donné (i.e. un intervalle du type $[2^k, 2^{k+1}[$ contenant les arguments à tester) ; ce script connaît aussi la taille des sous-intervalles de $[2^k, 2^{k+1}[$ ($\log_2$ du nombre d'arguments). Ces paramètres sont donnés soit par option en ligne de commandes, soit par le serveur. Les sous-intervalles sont numérotés à partir de 0, dans l'ordre, le premier commençant au point 2^k (même s'il n'est pas dans le domaine considéré). Les sous-intervalles appartenant au domaine considéré seront testés un par un, dans un ordre fixé par l'utilisateur ou par le serveur.

Reprenons les 6 arguments de `test32f` : le nom de la fonction et l'exposant sont directement connus. La mantisse de l'argument initial s'obtient par la taille de la mantisse, et le numéro et la taille du sous-intervalle. Pour calculer le nombre d'arguments à tester dans le sous-intervalle, il suffit de calculer la puissance de 2 correspondante, mais ce nombre étant généralement supérieur à 2^{32}, il doit être calculé en multiprécision (module bigint). Le préfixe est déterminé dès le début ; plusieurs clients pouvant tourner en même temps (sur une ou plusieurs machines) et le préfixe devant être unique, il est formé du nom de la machine, de l'heure (à la seconde près) à laquelle est lancée le client et de l'identificateur du processus (*pid*) : `tst-`*`host`*`-`*`time`*`-`*`pid`*.

En ce qui concerne les options, les plus importantes peuvent être données en ligne de commandes lors du lancement de `t3-client`, ou bien par le serveur. L'option indiquant le nombre de bits testés est très importante : en effet, si trop de bits sont testés, certains pires cas pouvant être intéressants seront filtrés, comme expliqué à la fin de la section 3.1.1. Mais le nombre maximum de chiffres égaux après la mantisse des valeurs filtrées est stocké dans le fichier de résultats (pour chaque intervalle J_i), et une erreur sera renvoyée lors de la deuxième étape si trop de bits sont testés.

Après le lancement de `test32f`, si l'exécution s'est terminée sans erreur, `gcc` est lancé pour compiler et assembler les sources C et assembleur *`prefix`*`.c` et *`prefix`*`.s`, et générer un exécutable *`prefix`*. On lit ensuite dans le fichier de *log* de `test32f` les valeurs minimales et maximales des exposants des $f(x_m)$ (ainsi que le nombre de bits testés, par mesure de précaution), pour en déduire le poids des bits testés et savoir à quel niveau les arguments ont pu être filtrés. Notons qu'à cause des diverses approximations, les valeurs minimales et maximales des exposants des $f(x_m)$ ne correspondent pas forcément à ceux qui seraient obtenus par raisonnement mathématique ; leur lecture dans le fichier de *log* est donc indispensable. Le programme *`prefix`* généré par `gcc` est ensuite lancé ; les résultats sont récupérés par *pipe* et écrits sur la sortie standard (généralement redirigée vers un fichier). En pratique, cette méthode de récupération des résultats peut poser des problèmes : le processus fils *`prefix`* peut mourir avant que le père `t3-client` n'ait pu récupérer tous les résultats par *pipe*, et une partie des résultats peuvent ainsi être perdus. Cependant, les pertes de résultats que nous avons remarquées étaient toutes liées à un *bug* de l'interpréteur Perl. Cette méthode (utilisation d'un *pipe*) devra être changée en une méthode plus sûre : le processus de calculs *`prefix`* devra écrire directement tous les résultats dans le fichier de résultats de `t3-client`.

Note : certaines machines n'ont pas le logiciel Maple ou gcc d'installé, ou alors ces machines n'ont pas beaucoup de mémoire, ou bien il y a des problèmes avec NFS, ce qui provoque parfois des erreurs lors du lancement de Maple ou de gcc. Pour ces raisons (ou pour d'autres raisons), il est possible de lancer `test32f` et `gcc` sur une autre machine, faisant tourner un serveur (script Perl `t3-tserv`, différent du serveur dont nous parlerons dans la section 3.2) : `t3-client` se connecte à ce serveur, donné par l'option `-t=`*`serveur`*, lui envoie tous les paramètres nécessaires à `test32f`, et récupère l'exécutable *`prefix`* et les trois valeurs qui auraient été lues dans le fichier de *log*. L'exécutable contient des données binaires, il est donc transféré en *base64* (format permettant de transférer des données binaires avec un transfert du type ASCII), et les données sont vérifiées par MD5.

Il est important de donner la signification des résultats renvoyés. Tous les calculs se faisant en virgule fixe, les bits testés ont le même poids. Voici un exemple où il y a un écart de 5 entre l'exposant minimal et l'exposant maximal (même si en pratique, cet écart est seulement de 0 ou 1) :

Exposant minimum : `     mmmmmm...mmmmmmmtttt...ttttb`
Exposant maximum : `mmmmmm...mmmmmmmxxxxxtttt...ttttb`

Les bits `mmmmmm...mmmmmmm` sont les 54 bits de mantisse des approximations calculées[3], et les bits `tttt...tttt` sont les n_t bits testés (après avoir ajouté 1 au bit `b`). À cause de l'écart entre les exposants, ils ne suivent pas forcément la mantisse : il peut y avoir un certain nombre de bits (notés `x`) entre les deux, qui seront pris en compte seulement lors de la deuxième étape. Les arguments x_i éliminés sont ceux dont les bits `tttt...ttttb` de l'approximation de $f(x_i)$ ne sont pas tous égaux. On peut donc en déduire que pour tout argument x_i éliminé, $f(x_i)$ a une mantisse de la forme :

$$\underbrace{1.xxx...xx}_{n\text{ bits}}\underbrace{r0000...00}_{k\text{ bits}}1...$$

ou

$$\underbrace{1.xxx...xx}_{n\text{ bits}}\underbrace{r1111...11}_{k\text{ bits}}0...$$

avec $k \leqslant n_t + 2 + d$, où d est l'écart maximal entre les exposants.

Nous allons maintenant expliquer le format des résultats générés par `t3-client`. Il a été choisi de façon à ce qu'une perte de données puisse être généralement détectée (on ne s'occupera pas des pertes de données du genre un seul caractère au milieu d'une ligne, certainement aussi improbable que d'autres erreurs impossibles à détecter). C'est d'autant plus important que des pertes de données ont parfois réellement lieu (si le fils meurt trop tôt ou si le disque dur est plein).

La première ligne est : `*** t3-client ***` (elle indique qu'il s'agit d'un fichier créé par `t3-client`. On a ensuite une ligne blanche, suivie de quatre lignes donnant des informations sur les tests en cours (pour pouvoir interpréter les résultats de manière non ambiguë), d'une autre ligne blanche, et d'un ensemble de blocs, chaque bloc représentant les résultats d'un intervalle J_i. Les quatre lignes d'information sont par exemple :

```
function:       exp
exponent:       3
msize:          54
ibits:          40
```

Cela signifie que la fonction est celle définie par `def-exp` (c'est la fonction exponentielle), que l'exposant testé est 3, que la mantisse fait 54 bits, et que les intervalles J_i ont 2^{40} bits.

[3]Une valeur $f(x_m)$ et son approximation calculée n'ont pas forcément le même exposant si la valeur est très proche d'une puissance de deux, mais cela ne change pas du tout le raisonnement.

La première ligne d'un bloc est de la forme : `[`i`]\t(`d`:`$k_{\max}$`)`, où i, d et $k_{\max}$ sont des entiers écrits en base 10 ; i est le numéro de l'intervalle testé, d la différence entre l'exposant maximal et l'exposant minimal des approximations de $f(x)$ sur l'intervalle (cette valeur ne sera pas prise en compte lors de la seconde étape, mais peut servir à l'utilisateur), et $k_{\max} = n_t + 2 + d$. La deuxième ligne est de la forme : `<`p`>`, où p est le nombre de pires cas trouvés dans J_i. Ensuite, pour chaque pire cas, on a une ligne de la forme suivante, désignant la position m du pire cas $x_m = x_0 + m.\mathrm{d}x$ dans l'intervalle : `[`h_{lo}`,`h_{hi}`]\t[`d_{lo}`,`d_{hi}`]`. Les deux premiers entiers sont les 32 bits de poids faible puis les 32 bits de poids fort de m écrits en hexadécimal (sur 8 chiffres), et les deux entiers suivants représentent les mêmes valeurs écrites en décimal. Enfin, la dernière ligne est `{`i`}` ; elle sert à vérifier s'il ne manque pas de données.

Ainsi, les quatre lignes d'information de l'en-tête ainsi que chaque bloc identifient parfaitement ce qui a été testé et la signification des résultats.

3.1.3 Deuxième étape (script `t3-secstep`)

La deuxième étape est implémentée à l'aide d'un script Perl `t3-secstep`, qui lit les résultats renvoyés lors de la première étape par `t3-client` (dans des fichiers), filtre plus précisément les pires cas, et stocke ses résultats dans un autre fichier.

De manière à pouvoir arrêter `t3-secstep` et à le relancer plus tard, et pour que les résultats puissent être lus automatiquement lors d'une troisième étape, il faut que le fichier donne toutes les informations nécessaires pour connaître parfaitement la signification des résultats. Comme les fichiers générés par `t3-client`, celui généré par `t3-secstep` contient un en-tête suivi d'une ligne blanche, qui donne toutes ces informations, par exemple :

```
*** results from t3-secstep ***

function:       exp
exponent:       3
msize:          54
ibits:          40
nbits:          44
interval:       0..8191
```

En plus des données déjà présentes dans le fichier de résultats de `t3-client`, `nbits` indique le nombre minimal de bits identiques situés après la mantisse de 54 bits des pires cas, et `interval` indique quels sont les sous-intervalles J_i de I qui doivent être testés[4] : ici, les J_i pour i allant de 0 à 8191, i.e. I doit être testé en entier.

Les fichiers générés par `t3-client` sont lus un par un. Il est possible de faire tourner `t3-secstep` en boucle, et dans ce cas, un fichier n'est relu que

[4] Rappelons que I, défini en section 3.1.2.1, est l'intervalle regroupant *tous* les sous-intervalles J_i.

si sa taille n'a pas changé : cela permet d'aller plus rapidement (les fichiers pouvant être gros et nombreux) et d'éviter de faire trop d'accès aux fichiers, qui surchargeraient inutilement le serveur NFS.

Pour chacun de ces fichiers, on commence par vérifier que les paramètres `function`, `exponent`, `msize` et `ibits` sont les mêmes ; sinon, on passe au fichier suivant. Pour chaque sous-intervalle J_i testé, on vérifie que la valeur $k_{\max}$ est inférieure ou égale à la valeur de `nbits`. Chaque pire cas de J_i est ensuite testé (si l'intervalle J_i n'a pas déjà été testé). La valeur de l'argument x à tester est donnée par la formule suivante :

$$x = 2^{\texttt{exponent}-(\texttt{msize}-1)}(2^{\texttt{msize}-1} + i.2^{\texttt{ibits}} + 2^{32}d_{\mathrm{hi}} + d_{\mathrm{lo}})$$

et la valeur $f(x)$ est évaluée en arithmétique d'intervalles, son expression étant définie dans le fichier `def-`*`function`*. On récupère la partie entière des deux bornes de l'intervalle englobant $|f(x)|$ multiplié par une certaine puissance de deux, de manière à obtenir les premiers bits de la mantisse de $f(x)$; plus précisément, cette puissance de deux est : $2^{2.\texttt{msize}+30-e}$, où e est un entier proche de $\log_2|f(x)|$. On a : $a \leqslant |f(x)|.2^w \leqslant b$, avec a et b entiers. On vérifie que $b-a \leqslant 1$, sinon il faudra recalculer $f(x)$ avec plus de précision ; les premiers bits de la mantisse de $f(x)$ sont soit les bits de a, soit ceux de $a+1$. Puis on convertit a en binaire. C'est un nombre de la forme :

$$\underbrace{1xxx...xx}_{n\text{ bits}}\underbrace{r000...00}_{k\text{ bits}}\underbrace{1bbb...bb}_{s\text{ bits}}$$

ou

$$\underbrace{1xxx...xx}_{n\text{ bits}}\underbrace{r111...11}_{k\text{ bits}}\underbrace{0bbb...bb}_{s\text{ bits}}$$

avec $n = \texttt{msize} - 1$, $k \geqslant 2$ et $s \geqslant 0$ (s pouvant éventuellement être nul). La valeur de k correspondant à la mantisse de $f(x)$ peut donc être fausse si et seulement si $s \leqslant 1$ ou si tous les bits b sont égaux à 1 ; c'est notamment le cas si $f(x)$ est un nombre machine sur $n+1$ bits (cela peut arriver avec certaines fonctions, comme 2^x, il ne faut donc pas considérer cette possibilité comme une erreur). En fait, on teste une condition un peu plus faible : $s \leqslant 2$ ou les bits b sont tous égaux ; si cette condition est vérifiée, alors on le signale dans le fichier de résultats, en incluant les paramètres i, d_{hi} et d_{lo}. Sinon, la valeur de k obtenue est forcément correcte. Si le bit r n'est pas le même que le bit qui suit (i.e. si $f(x)$ est « proche » du milieu de deux nombres machine sur n bits consécutifs) et si d_{lo} est impair (i.e. si x est « proche » du milieu de deux nombres machine sur n bits consécutifs), alors le candidat pire cas est rejeté. Si le nombre de bits identiques situés juste après la mantisse de $n+1$ bits est strictement inférieur à `nbits` (i.e. si $k \leqslant \texttt{nbits}$), alors le pire cas est également rejeté. Sinon, le pire cas est gardé : on écrit dans le fichier de résultats la mantisse de x (sur $n+1$ bits), le nombre de bits identiques situés juste après la mantisse de $n+1$ bits ($k-1$), et le type du pire cas :

- $(0,0)$: x est un nombre machine sur n bits et $f(x)$ est très proche d'un nombre machine sur n bits ; il y aura donc un pire cas pour f et sa réciproque, en mode d'arrondi dirigé ;

– $(0,1)$: x est un nombre machine sur n bits et $f(x)$ est très proche du milieu de deux nombres machine consécutifs ; il y aura donc seulement un pire cas pour f en mode d'arrondi au plus près ;

– $(1,0)$: x est le milieu de deux nombres machine consécutifs et $f(x)$ est très proche d'un nombre machine sur n bits ; il y aura donc seulement un pire cas pour la réciproque f^{-1} en mode d'arrondi au plus près ;

– $(1,1)$: ce cas n'est pas possible, car il a été rejeté.

Notons que jusque là, la signification des pires cas (valeur de k) est toujours relative à f, quel que soit le type du pire cas, jamais à sa réciproque. Des résultats rigoureux sur la réciproque ne seront obtenus qu'à la troisième étape.

3.1.4 Troisième étape (script `t3-lastep`)

La troisième étape est implémentée à l'aide d'un script Perl `t3-lastep`, qui lit les résultats renvoyés par `t3-secstep`, et sort les pires cas de f et de f^{-1} en fonction de la valeur de k minimale fixée par l'utilisateur ; cette valeur de k peut correspondre à la fonction f uniquement, ou aux deux fonctions f et f^{-1}, comme expliqué plus loin.

L'utilisateur choisit un entier k. Le script `t3-lastep` testera les pires cas x tels que $k_{f(x)} \geqslant k$, où $k_{f(x)}$ est le nombre de bits identiques (à part le bit d'arrondi r, qui peut avoir n'importe quelle valeur) situés juste après la mantisse de n bits de $f(x)$. Maintenant, supposons que l'utilisateur choisisse un entier k' et qu'il soit intéressé par *tous* les pires cas y de la réciproque vérifiant $k_{f^{-1}(y)} \geqslant k'$. Les tests devront se faire à partir des arguments x (stockés dans le fichier de résultats de `t3-secstep`) : pour chaque x sélectionné (en fonction de k), on calcule $f(x)$, on arrondit convenablement en un nombre y sur $n+1$ bits, on calcule $f^{-1}(y)$, et enfin on compte le nombre de bits identiques (tout cela se fait bien sûr en arithmétique d'intervalles). Mais comment choisir k de manière à ce que les tests sur les arguments x tels que $k_{f(x)} \geqslant k$ suffisent pour obtenir *tous* les pires cas de f^{-1} voulus par l'utilisateur ?

D'après la section 1.6.3, il suffit de choisir $k = k' + d$, avec

$$d \leqslant \log_2(2^{e'-e}/M) = e' - e - \log_2(M),$$

où M est un majorant de $|f'|$ sur l'intervalle $[f^{-1}(y), x]$ ou $[x, f^{-1}(y)]$, e est l'exposant de x et e' l'exposant de y. On peut chercher un minorant de

$$\left\lfloor \log_2 |f(x)| - \log_2 |x| - 1 - \log_2 \left|f'(x)\right| \right\rfloor,$$

bien que l'inégalité ne soit pas tout à fait rigoureuse (il faut le vérifier en pratique, pour chaque fonction f considérée). On prendra une valeur inférieure à

$d(x, e, x_{\min})$, où :

$f(x)$	$d(x, e, x_{\min})$
$\exp x$	$-e-2$
$\log x$	$\log_2 \lvert\log x\rvert - 1$
2^x	$-e-3$
$\log_2 x$	$\log_2 \lvert\log x\rvert - 1$
$\sin x$	$\log_2(\tan x_{\min}) - (e+2)$

Le script `t3-lastep` doit être appelé avec une valeur $k_{\min}$, un ensemble de fichiers de résultats de `t3-secstep`, et un ensemble d'options. Contrairement aux autres étapes, où les pires cas n'étaient pas donnés dans l'ordre (car ils n'étaient pas directement destinés à l'utilisateur final), ici ils peuvent être triés dans l'ordre croissant des arguments si l'option `-s` (`--sort`) est donnée. On doit aussi donner obligatoirement l'une des deux options `-x` ou `-y` pour indiquer le mode de fonctionnement (signification de $k_{\min}$) :

– option `-x` : la valeur $k = k_{\min}$ est utilisée pour sélectionner les pires cas de la fonction f, i.e. on teste les pires cas pour lesquels $k_{f(x)} \geqslant k_{\min}$, et à partir de ces pires cas, les résultats concernant f et f^{-1} sont renvoyés ;

– option `-y` : la valeur $k = \min(k_{\min} + d, k_{\min})$ est utilisée pour sélectionner les pires cas de la fonction f, on teste donc les pires cas pour lesquels $k_{f(x)} \geqslant k$, mais on ne renvoie que les pires cas pour lesquels $k_{f(x)} \geqslant k_{\min}$ (pour f) ou $k_{f^{-1}(y)} \geqslant k_{\min}$ (pour f^{-1}). La valeur de k choisie permet d'obtenir *tous* les pires cas vérifiant ces inégalités.

Note : les données concernant les différentes fonctions, en particulier $d(x)$, sont stockées en dur dans le script. Il aurait évidemment été préférable de les stocker dans les fichiers `def-*`, qui sont faits pour cela.

3.1.5 Cas particulier de `nbits` nul

Pour certaines fonctions dans certains intervalles (par exemple, $\log x$ avec x très proche de 1), les deux premières étapes sont inutiles et peuvent même compliquer les calculs, car le nombre d'arguments est relativement petit et il peut y avoir de nombreux changements d'exposant pour $f(x)$. Il est donc possible de demander à `t3-secstep`, avec l'option `--nbits=0` (valeur `nbits` nulle), de générer toutes les mantisses possibles des arguments, en vue d'effectuer uniquement une troisième étape.

3.2 Parallélisation

Comme nous l'avons vu, la première étape demande beaucoup de temps de calculs. Il est donc nécessaire de lancer les calculs en parallèle. Du point de vue algorithmique, cela ne pose pas de problème, car il s'agit de tester des intervalles indépendants les uns des autres ; il n'y aura donc aucune communication entre deux processus tournant en parallèle. Le principal problème consiste à répartir les tâches ; c'est ce dont nous parlerons en premier (section 3.2.1). Du point de vue matériel, nous utiliserons un réseau de stations de travail que nous avons à notre disposition, ces stations de travail n'étant pas utilisées en permanence, soit parce que leurs utilisateurs sont absents (par exemple, la nuit ou le week-end), soit parce qu'ils ne demandent pas beaucoup de temps CPU (travail interactif, comme du traitement de textes ou du *mail*). Mais nous ne devons pas pour autant les gêner ; nous avons choisi diverses solutions à ce propos (section 3.2.2). Nous indiquerons aussi comment sont stockés les résultats et comment ils seront récupérés (section 3.2.3). Enfin, nous expliquerons comment les scripts seront utilisés en pratique (section 3.2.4).

3.2.1 Répartition des tâches

Pour la répartition des tâches, la solution choisie est de faire tourner un serveur (script Perl `t3-server`) en permanence sur une machine peu souvent *rebootée*, et des clients (script Perl `t3-client`) sur différentes machines, qui se connectent au serveur et lui demandent des intervalles à tester.

Dans la toute première implémentation, avant la réécriture complète de tous les scripts, les clients étaient lancés par le serveur sur les différentes machines, suivant certaines règles. Mais cela a posé de gros problèmes avec des machines qui ne répondaient pas ou qui répondaient trop lentement. Il est aussi arrivé que des processus ne pouvaient plus être tués (processus bloqués dans des entrées-sorties) : même un `kill -KILL` n'avait aucun effet ; et la machine pouvait se retrouver avec plusieurs processus de tests et une charge très élevée. Nous avons donc décidé que les clients seraient dorénavant lancés à la main, mais avec une possibilité de les faire tourner en permanence.

En général, nous ne lancerons qu'un client par machine. De manière à pouvoir nous assurer de l'unicité du client (sur une machine donnée), mais aussi à pouvoir identifier ce client, nous lancerons généralement les clients avec l'option `-u` (`--unique`), qui a pour effet de créer un verrou et de stocker le *pid* du client dans un fichier. Le seul inconvénient est qu'il faut parfois retirer le verrou à la main en cas de problèmes. Sur les machines multiprocesseur, l'un des clients pourra être lancé avec l'option `-u`, et les autres sans cette option.

Pour simplifier, le serveur ne peut s'occuper que d'un intervalle et une fonction à la fois (fixés au lancement). Si plusieurs intervalles ou fonctions doivent être testés à la fois, il faut lancer plusieurs serveurs (sur des ports différents s'ils tournent sur la même machine). Cela peut être intéressant si un intervalle I a presque entièrement été testé et que seules quelques machines travaillent sur les derniers sous-intervalles J_i : pour faire travailler les autres machines, on peut commencer à tester un autre intervalle I' (éventuellement d'une autre fonction). En revanche, l'avantage qu'un serveur s'occupe d'un seul intervalle est qu'il peut donner tous les paramètres nécessaires aux clients (fonction, exposant, etc.), ce qui simplifie le lancement des clients.

Lorsque le serveur est lancé, il commence par lire le fichier de résultats de `t3-secstep`, où se trouvent tous les paramètres (fonction, exposant, etc.) ; c'est facultatif, mais si ce fichier n'est pas lu, alors les paramètres doivent être donnés en arguments lors du lancement. Ensuite, le serveur lit tous les fichiers de résultats de `t3-client`. Cela lui permet de connaître tous les intervalles qui ont été testés, ainsi que ceux qui sont en cours de test. Le serveur fournit en priorité aux clients les intervalles qui ne sont ni testés, ni en cours de test ; lorsqu'il n'y en a plus, le serveur fournit les intervalles qui sont en cours de test : ainsi, lorsque presque tout a été testé, certains intervalles peuvent être en cours de test sur plusieurs machines, au cas où les calculs seraient interrompus sur certaines machines.

Nous avons choisi un protocole similaire à de nombreux protocoles existants (NNTP, POP, etc.) : il s'agit d'un protocole texte sur un port TCP fixé (par défaut, 4913), de manière à ce que l'utilisateur puisse voir comment ça se passe ou puisse même agir avec une connexion *telnet*. Le client envoie des commandes, et le serveur renvoie la réponse éventuelle. Les commandes sont les suivantes, où *num* indique une valeur numérique :

– `QUIT` : le serveur ferme la connexion.
– `INFO` : le serveur renvoie la liste des intervalles qui ont été testés, leur nombre, ainsi que la liste des intervalles en cours de test.
– `RESET` : cette commande force le serveur à relire tous les fichiers de résultats (comme lorsqu'il a été lancé) et à mettre à jour sa liste d'intervalles testés et en cours de test (quand un processus de calcul est définitivement arrêté, l'intervalle qui était en cours de test ne l'est plus).
– `GET` *num* : cette commande est suivie d'un entier positif n. Le client l'utilise pour demander n numéros d'intervalles à tester.
– `TESTED` *num* : cette commande est suivie du numéro d'un intervalle. Elle indique au serveur que cet intervalle a été testé.
– `LKMAX` *num* : cette commande est suivie d'un entier positif ; elle est normalement donnée par l'utilisateur avec une connexion *telnet*. Elle permet de fixer la valeur de l'option `-k` (`--lkmax`) de `t3-client` (qui correspond aussi à l'option `-k` de `test32f`) la prochaine fois que le client se connectera au serveur.
– `LNSSI` *num* : cette commande est suivie d'un entier positif ; elle est normalement donnée par l'utilisateur avec une connexion *telnet*. Elle permet de fixer la valeur de l'option `-l` (`--lnssi`) de `t3-client` (qui correspond aussi à l'option `-l` de `test32f`) la prochaine fois que le client se connectera au serveur.

Pour simplifier, et parce qu'il reçoit relativement peu de connexions, le serveur ne crée pas de nouveau *thread* à chaque connexion. S'il n'a pas reçu de commande après 8 secondes, il ferme la connexion, pour éviter que toutes les autres connexions soient bloquées.

3.2.2 Contrôle des processus

Pour ne pas trop gêner les utilisateurs des machines, les processus de calculs sont lancés avec une faible priorité à l'aide de la commande `nice`. Cela permet de n'utiliser le processeur pour ces calculs que lorsqu'il n'a rien d'autre à faire : un processeur passe généralement la plus grande partie de son temps à attendre que l'utilisateur fasse une action (lorsque le travail de celui-ci est interactif). Malheureusement, ceci semble assez théorique, et en pratique, même lorsque le processus a une faible priorité, il semble que cela puisse ralentir considérablement les machines (même des machines puissantes comme des UltraSparc). Nous avons donc dû trouver des solutions pour arrêter temporairement ou définitivement les processus de calculs. Pour cela, nous avons implémenté trois méthodes, qui peuvent être utilisées ensemble (i.e. l'utilisation d'une méthode n'exclut pas les autres) :

– n'importe quel utilisateur peut arrêter temporairement ou tuer un processus de calculs (sous certaines conditions) ;

– le processus de calculs peut s'arrêter après un certain temps, fixé au lancement ;

– l'utilisation de la machine (clavier ou souris) peut être détectée automatiquement.

Ces trois méthodes sont détaillées ci-dessous. Note : les arrêts temporaires sont préférables, car rien n'est perdu, alors qu'avec un arrêt définitif, les résultats sur l'intervalle en cours sont perdus.

3.2.2.1 Script `t3-kill`

Le script Perl *setuid* `t3-kill` permet à n'importe quel utilisateur d'arrêter le processus de calculs, si le client a été lancé avec l'option `--unique` (pour que le client puisse être identifié). L'utilisateur peut donner en argument le temps, en minutes, pendant lequel le processus doit être arrêté (avec un temps maximal de 360 minutes, i.e. 6 heures), le temps par défaut étant de 180 minutes, i.e. 3 heures : l'heure de reprise est stockée dans un fichier et un signal SIGALRM est envoyé au client, qui, à la réception de ce signal, envoie alors un signal SIGSTOP au processus de calculs pour l'arrêter temporairement. Après le temps fixé, le client enverra un signal SIGCONT pour faire reprendre les calculs. À tout moment, l'utilisateur peut relancer `t3-kill` avec une autre valeur, pour changer l'heure de reprise ; en particulier, lancer `t3-kill` avec la valeur 0 permet de faire immédiatement reprendre les calculs.

Si l'utilisateur donne l'option `-TERM` en lançant `t3-kill`, c'est le signal SIGTERM qui est envoyé au client, au lieu de SIGALRM ; cela a pour effet d'arrêter définitivement le processus de calculs et le client.

3.2.2.2 Option `-q` (`--quit`)

Le client peut arrêter le processus de calculs au bout d'un certain temps fixé au lancement, à l'aide de l'option `-q=`*`num`* (nombre de minutes). Cela permet par exemple de lancer un processus pour la nuit, pour le week-end, ou pendant une période de vacances.

3.2.2.3 Détection de l'utilisation d'une machine

La méthode certainement la meilleure est la détection automatique de l'utilisation de la machine : utilisation du clavier, de la souris, et de certains *pseudo-tty*. Cela se fait en regardant périodiquement l'heure de dernier accès aux fichiers `/dev/kbd`, `/dev/mouse` et `/dev/pts/`*`num`*, mais aussi le propriétaire actuel des fichiers `/dev/kbd` et `/dev/mouse` : par exemple, la souris peut être bougée alors que personne n'est *loggé* sur la machine (le propriétaire est *root*) ; nous considérerons quand même que la machine n'est pas utilisée. Par défaut, cette période est de 30 secondes, ce qui signifie que l'utilisateur peut voir sa machine ralentie pendant *au plus* 30 secondes après un temps de non utilisation choisi au lancement du client (cf ci-dessous) ; la valeur de cette période peut être changée par option, au lancement du client.

Pour que cette méthode soit utilisée, le script `t3-client` doit être lancé avec l'option `--idletime=`*`num`*, qui donne la durée en minutes pendant laquelle le processus de calculs doit être arrêté après la dernière utilisation de la machine.

3.2.3 Fichiers de résultats

Dans la toute première implémentation, les résultats de la première étape étaient stockés dans un fichier commun. Comme il y avait des écritures concurrentes par NFS, il fallait utiliser un verrou résistant à NFS pour éviter des pertes de données. Mais cette méthode a posé des problèmes ; le serveur devait parfois forcer le retrait du verrou...

Nous avons alors choisi une méthode sans écritures concurrentes et sans verrou, donc plus sûre : chaque client écrit ses résultats dans un fichier qui lui est propre. Les résultats ne sont regroupés dans un fichier unique que lors de la deuxième étape, laquelle n'est pas parallélisée, donc il n'y a pas non plus d'écritures concurrentes.

3.2.4 Utilisation des scripts

Nous allons indiquer comment lancer les différents scripts, et décrire quelques autres fonctionnalités non données précédemment.

3.2.4.1 Liste des fichiers

Donnons d'abord la liste des fichiers utilisés, autres que ceux créés automatiquement. Les plus importants ont déjà été décrits. Les autres permettent surtout de faciliter le lancement et le déroulement des tests et seront décrits plus loin.

– `def-`*`function`* : il s'agit de fichiers texte définissant les fonctions à tester ; ils sont utilisés par les scripts `test32f` et `t3-secstep`, et définis en section 3.1.2.2.

– `test32f` : script Perl générant les sources C et assembleur pour le test d'une fonction dans un intervalle. Il est utilisé par `t3-client` et `t3-tserv` et n'a normalement pas à être lancé directement par l'utilisateur.

– `t3-client` : script Perl testant une fonction dans un ou plusieurs intervalles. Il utilise `test32f` pour générer les sources C et assembleur correspondant à l'intervalle, et `gcc` pour compiler. Les paramètres de test peuvent être donnés soit en arguments au lancement, soit par un serveur (script `t3-server`). Les résultats sont écrits dans un fichier dont le nom est fixé de manière unique (il y a un fichier par client).

– `t3-exec` : script Perl permettant de lancer un client `t3-client` sur une ou plusieurs machines distantes, par `rsh`.

– `t3-server` : script Perl jouant le rôle de serveur pour `t3-client` (répartition des tâches).

– `t3-tserv` : script Perl permettant de lancer `test32f` et la compilation à distance ; il s'agit aussi d'un serveur.

– `t3-kill` : script Perl *setuid* permettant d'arrêter temporairement ou de tuer un processus de calculs, si le client correspondant a été lancé avec l'option `-u` (`--unique`), comme cela est fait par défaut avec `t3-exec`.

– `t3-secstep` : script Perl effectuant la deuxième étape.

– `t3-stat` : script Perl donnant quelques statistiques (intervalles testés, machines utilisées) concernant les résultats de la deuxième étape.

– `t3-lastep` : script Perl effectuant la troisième (et dernière) étape.

– `t3-start` : script Perl permettant de lancer les scripts `t3-server` et `t3-secstep` dans `screen` (utilisation de terminaux virtuels), et permettant donc de faciliter le contrôle des tests à partir de différentes machines.

3.2.4.2 Lancement des scripts

L'aide sur l'utilisation des scripts `t3-*` peut être obtenue grâce à l'option `-h` (`--help`). Mais nous allons expliquer comment ils sont généralement utilisés.

Il faut d'abord choisir une fonction à tester, un exposant, une taille de mantisse `msize`, la taille des sous-intervalles $2^{\texttt{ibits}}$, le nombre minimum de bits identiques situés après la mantisse pour les pires cas `nbits`, et si besoin est, les sous-intervalles à tester. On doit d'abord appeler `t3-secstep` avec ces paramètres pour créer le fichier de résultats, qui sera notamment utilisé par le serveur. Par exemple, pour tester l'exponentielle avec l'exposant 3, une taille de mantisse de 54 bits (double précision), des sous-intervalles J_i de taille 2^{40} (i.e. 2^{40} arguments), et 44 bits identiques (au minimum) pour les pires cas :

```
t3-secstep -f=exp -e=3 -m=54 -i=40 -n=44
```

Ceci crée un fichier de résultats `results.exp.3.54` contenant les différents paramètres, et `t3-secstep` pourra dorénavant être relancé avec :

```
t3-secstep -r=results.exp.3.54
```

(et éventuellement d'autres options). Si on veut tester le logarithme en base 2 entre $1 + 2^{-13}$ et $1 + 2^{-9}$, on peut utiliser :

```
t3-secstep -f=lg2 -e=0 -m=54 -i=36 -n=36 --imin=16 --imax=255
```

Cela définit $2^{54-1-36} = 2^{17}$ sous-intervalles de longueur 2^{-17} entre 1 et 2. L'intervalle 16 (premier intervalle à tester) commence bien à $1+16\times 2^{-17} = 1+2^{-13}$, et l'intervalle 255 (dernier intervalle à tester) finit bien à $1+256\times 2^{-17} = 1+2^{-9}$.

L'option `--loop=`*`num`* permet de faire tourner `t3-secstep` en boucle. La valeur est le nombre de secondes de la pause entre deux itérations. Le choix de cette valeur dépend des paramètres des tests, des machines utilisées pour la première étape, etc. On la choisira d'autant plus grande que le temps moyen de test d'un intervalle J_i est grand. Chaque itération consommant des ressources (accès NFS surtout), on ne prendra pas inutilement une petite valeur.

L'option `--remove` permet d'effacer les fichiers de résultats de `t3-client` lorsque le client correspondant s'est terminé proprement (par l'option `--quit` ou par `kill -TERM`). En effet, on n'a généralement plus besoin de ces fichiers.

Une fois le fichier de résultats de `t3-secstep` créé, un serveur `t3-server` peut être lancé en donnant comme argument le nom de ce fichier (les paramètres y sont lus, on n'a donc pas besoin de les rappeler). En général, on lance aussi la deuxième étape en boucle, par exemple :

```
t3-secstep -r=results.exp.3.54 --remove --loop=30
```

Les scripts `t3-server` et `t3-secstep` envoient des informations sur les fichiers de sortie et d'erreur standard. On choisira donc soit de les lancer dans des fenêtres séparées (éventuellement des terminaux virtuels, ce qui est la meilleure solution, cf section 3.2.4.3), soit de rediriger la sortie et l'erreur standard dans des fichiers.

On peut lancer ensuite un client sur une ou plusieurs machines distantes (l'une d'elles pouvant être la machine locale) avec `t3-exec`. On donne en arguments : le nom du serveur, la priorité pour `nice`, le nom des machines en question, et éventuellement des arguments pour le client (comme `--idletime=`*`num`* ou `-l=`*`num`*, si nécessaire). Les clients sont lancés par `rsh` en parallèle, puis chaque `rsh` est tué dès que le client correspondant a créé son fichier de résultats. Une limite maximale de 3 `rsh` en cours est fixée pour éviter de surcharger la machine locale et le réseau. Il est possible de tuer le `rsh` situé en tête de la file (typiquement, lorsque la machine correspondante ne répond pas) par `Ctrl-C`, qui est le moyen classique d'envoyer un signal SIGINT sous Unix[5]. Les clients sont lancés par défaut avec l'option `--unique`, pour s'assurer de leur unicité sur chaque machine et pour qu'ils puissent être arrêtés par n'importe quel utilisateur. Si l'on ne veut pas de cette option pour une raison ou pour une autre, on peut toujours ajouter l'option `--nounique` pour l'ignorer ; cependant, dans ce cas, le `rsh` peut être tué prématurément, ce qui a pour effet d'interrompre le lancement du client. S'il n'est pas possible de faire de `rsh` sur une machine, ce qui peut arriver pour certaines machines configurées comme cela, alors `t3-client` peut être lancé directement avec les options `--unique` et `-s=`*`serveur`* (pour indiquer quel est le serveur).

Sur une machine multiprocesseur, il faut lancer le client autant de fois qu'il y a de processeurs à utiliser. L'un peut être lancé avec `t3-exec` ou avec `t3-client` et l'option `--unique`, mais les autres doivent être lancés avec soit `t3-client` sans l'option `--unique`, soit `t3-exec` avec l'option `--nounique`. Seul le client lancé avec l'option `--unique` pourra être arrêté par n'importe quel utilisateur.

On peut avoir besoin de tester des intervalles particuliers, en donnant les intervalles en arguments. Si l'option `-s` est présente, i.e. si le client peut demander des intervalles au serveur, les intervalles donnés en arguments sont d'abord testés, et une fois ces tests terminés, le client demandera des intervalles au serveur. Ceci évite d'avoir à relancer le client.

L'option `--log` permet de *logger* les messages d'erreur des clients dans un fichier commun `client.out`. L'utilisation d'un fichier commun facilite la lecture des messages d'erreur, surtout s'il y a beaucoup de clients, mais pose des problèmes avec NFS : en cas d'écritures simultanées, des messages d'erreur peuvent être tronqués ou perdus. Heureusement, les messages d'erreur sont assez peu nombreux, et les pertes (pas trop importantes) sont peu courantes. Mais même en tenant compte de cela, il restait un problème lié aux buffers, qui provoquait de très nombreuses pertes de messages d'erreur. La solution a été d'ouvrir le fichier `client.out` avec l'appel système `open` (`sysopen` en Perl) et le *flag* `O_SYNC` (en plus de `O_CREAT`, `O_WRONLY` et `O_APPEND`) pour synchroniser.

Enfin, pour faciliter la détection de problèmes avec les clients, on peut les lancer avec l'option `--fork` : le client effectue un *fork* au début, le processus fils continue normalement, et le père attend que le fils termine pour récupérer

[5]Les `rsh` sont exécutés dans des processus fils, et il a fallu détacher ces processus du terminal à l'aide de l'appel système `setsid` pour éviter que le `Ctrl-C` ne tue tous les `rsh`. La redéfinition du signal SIGINT avant d'exécuter le `rsh` n'est pas suffisante, car le `rsh` écrase cette redéfinition ; et redéfinir le signal après le `rsh` n'est pas non plus acceptable, car cela se produirait un peu trop tard : en pratique, il a été remarqué que la commande pouvait être tuée entre le temps où le `rsh` redéfinit les signaux et le temps où la commande lancée par `rsh` redéfinit SIGINT pour l'ignorer.

le code d'erreur et le numéro du signal éventuel qui a tué le fils et écrire ces données dans le fichier d'erreur standard. Cette option avait en fait été ajoutée parce que des clients mourraient parfois de façon inexpliquée ; ces problèmes semblent venir de *bugs* (connus) de l'interpréteur Perl.

3.2.4.3 Script `t3-start`

Pour faciliter le lancement de `t3-server` et de `t3-secstep` (une fois le fichier de résultats de `t3-secstep` créé) et le suivi des tests, un nouveau script a été écrit : `t3-start`. Ce script permet de lancer `t3-server` et `t3-secstep` chacun dans un terminal virtuel de `screen`, les informations de ces deux scripts étant affichées dans ces terminaux virtuels ; il définit aussi quelques macros pouvant être utiles, par exemple pour se connecter au serveur. Toujours pour faciliter son utilisation, `t3-start` requiert un nombre minimal d'arguments : uniquement la valeur de l'option `--loop` de `t3-secstep`.

Les 3 fichiers suivants sont créés : `.screenrc`, `server.out` et `secstep.out` ; si l'un d'eux existe déjà, `t3-start` se termine avec une erreur, et ceci par mesure de précaution, pour éviter que `t3-start` tourne deux fois. Le fichier `.screenrc` est le fichier d'initialisation de `screen`, et les deux autres fichiers `server.out` et `secstep.out` contiendront les messages d'erreur de `t3-server` et de `t3-secstep`.

3.2.4.4 Mode *batch*

Sur certaines machines, les processus doivent être lancés via le système LSF. Pour pouvoir faire tourner les tests sur ces machines, le client doit être lancé sur une machine « normale » et assez puissante pour créer tous les exécutables, avec l'option `-b=`*`num`*, qui indique le nombre d'intervalles à considérer. Des fichiers de la forme `tst-`*`host`*`-`*`time`*`-`*`pid`*`-batch*` sont créés et doivent être transférés sur les machines en question (par exemple, avec `tar`, `gzip`, puis `rcp` ou `ftp`). Ensuite, le *job* peut être lancé via LSF. Les résultats des tests seront stockés dans des fichiers de la forme `tst-*-batch.results`, et ces fichiers devront être récupérés à l'aide de l'option `-g` de `t3-secstep` (meilleure solution, car automatique), ou par `rcp` ou `ftp`.

La récupération automatique des résultats se fait de la façon suivante. Le script `t3-secstep` doit être lancé avec `-g=`*`host`*`:/`*`path`*, et `t3-secstep` fera périodiquement un `rcp` utilisant les paramètres de l'option `-g` pour aller récupérer les résultats. Il arrive qu'un `rcp` ne puisse pas fonctionner entre les deux machines. La solution est de passer par une troisième machine avec un `rsh` : on utilise alors `-g=`*`host1`*`,`*`host2`*`:/`*`path`*.

3.3 Temps de calcul

Nous allons maintenant présenter quelques mesures de temps de calcul. Nous montrerons en particulier les effets des options `--lnssi` et `--lkmax` de `t3-client`. Rappelons que ces options sont passées au script `test32f` (section 3.1.2.2) et influent sur la manière dont les intervalles sont découpés en sous-intervalles. La machine sur laquelle ont tourné ces tests est une station Sun Ultra-5 à 333 MHz sous Solaris 2.6.

Le tableau 3.1 ci-dessous donne les temps de calcul avec diverses valeurs pour `--lnssi` (0 étant la valeur par défaut), et diverses fonctions, avec l'exposant 0 pour les fonctions $\exp$, $\sin$ et $\cos$, et l'exposant 1 pour la fonction $\log$, des intervalles de 2^{40} arguments (`ibits` $= 40$). Dans les quatre cas, c'est l'intervalle numéro 16 qui est testé.

lnssi	exp	log	sin	cos
0	152 s	277 s	182 s	140 s
1	101 s	203 s	118 s	100 s
2	95 s	196 s	111 s	95 s
3	95 s	198 s	110 s	94 s
4	95 s	204 s	111 s	95 s
5	96 s	217 s	113 s	96 s
6	99 s	241 s	117 s	98 s
7	103 s	289 s	123 s	101 s
8	110 s	293 s	130 s	108 s
9	124 s	298 s	143 s	118 s
10	143 s	309 s	172 s	136 s

TAB. 3.1: *Temps de calcul mesurés pour le test d'un intervalle particulier de* 2^{40} *arguments avec les fonctions* $\exp$, $\log$, $\sin$, $\cos$. *Il s'agit d'un intervalle au voisinage de* 1 *pour les fonctions* $\exp$, $\sin$ *et* $\cos$, *et d'un intervalle au voisinage de* 2 *pour la fonction* $\log$.

Le nombre moyen d'arguments testés par cycle d'horloge avec la meilleure valeur pour `--lnssi` est donc d'environ 35 pour l'exponentielle et le cosinus, de 30 pour le sinus, et de 17 pour le logarithme.

Avec l'exemple précédent, l'option `--lnssi` permet de gagner un peu de temps. Mais avec d'autres exemples, cette option est beaucoup plus importante. C'est le cas de la fonction $\log_2$ avec l'exposant 0, des intervalles de 2^{38} bits et l'intervalle numéro 1024. Lorsqu'il n'y a pas d'option `--lnssi`, le programme de tests met 480 secondes, et avec l'option `--lnssi=3`, le programme met 36 secondes, soit plus de 13 fois moins de temps ! Ces différences de temps s'expliquent par le fait que si la valeur de l'option `--lnssi` est trop petite, alors l'algorithme de minoration échoue très souvent, et l'algorithme naïf (beaucoup plus lent) doit alors être utilisé ; augmenter la valeur de l'option `--lnssi` permet de réduire la taille du domaine de test pour l'algorithme de minoration, qui réussit alors plus souvent.

D'autres exemples sont donnés sur les tableaux 3.2 et 3.3.

$\#K_{i,j}$	lnssi	$\#L_k$	temps
32768	0	32768	9530 s
4096	0	4096	930 s
32768	2	8192	430 s
32768	3	4096	360 s
32768	4	2048	500 s

TAB. 3.2: *Temps de calcul pour le test de l'exponentielle sur l'intervalle de longueur 2^{40} commençant à la valeur $2^4 = 16$ ($\#E$ dénote le cardinal d'un ensemble E).*

$\#K_{i,j}$	lnssi	$\#L_k$	temps
32768	2	8192	3247 s
16384	2	4096	716 s
32768	3	4096	683 s
16384	3	2048	313 s
32768	4	2048	282 s
16384	4	1024	378 s
32768	5	1024	352 s
32768	6	512	568 s

TAB. 3.3: *Temps de calcul pour le test de l'exponentielle sur l'intervalle numéro 8 de longueur 2^{38} et d'exposant 7.*

Notons que ces tests ont été effectués dans des intervalles dont l'image par la fonction testée ne contient pas de valeur rationnelle « simple » (2, $\frac{1}{2}$, 3, $\frac{1}{3}$, $\frac{2}{3}$, etc.). Sinon, le problème décrit dans la conclusion du chapitre 2 se produirait, et les calculs pourraient prendre plusieurs heures au lieu de quelques minutes.

3.4 Résultats

Ces programmes de test ont tourné pendant de nombreux mois sur des stations Sun du réseau de l'ENS-Lyon (utilisation des machines du laboratoire LIP, du PSMN[6], et du LHPC[7]). Nous avons ainsi obtenu les résultats complets pour les fonctions 2^x et $\log_2(x)$ (nous avons d'ailleurs implémenté la fonction 2^x avec arrondi exact, cf section 4.5.2), ainsi que pour l'exponentielle et le logarithme, et des résultats partiels pour les fonctions sinus et cosinus, ainsi que leur réciproque : le sinus a été testé entre 2^{-5} et 2 (6 exposants) et le cosinus a été testé entre 2^{-6} et 1 (6 exposants). Les tests continuent afin d'augmenter la taille de ces domaines.

[6]Pôle Scientifique de Modélisation Numérique

[7]Laboratoire des Hautes Performances en Calcul

Nous allons d'abord expliquer comment les fonctions 2^x et $\log_2(x)$ ont été testées et nous donnerons les principaux pires cas pour ces deux fonctions (section 3.4.1). Nous montrerons ensuite, dans la section 3.4.2, que les résultats concernant les autres fonctions élémentaires permettent de vérifier en pratique les hypothèses probabilistes (sauf dans des domaines particuliers). Quelques pires cas seront donnés dans l'annexe B.

3.4.1 Cas des fonctions 2^x et $\log_2(x)$

Comme nous l'avons dit dans la section 1.1, pour ces fonctions, il n'est pas nécessaire de tester tous les intervalles. En effet, pour le test de 2^x, nous pouvons écrire $x = n + t$, avec $n \in \mathbb{Z}$ et $|t| \leqslant \frac{1}{2}$; nous avons alors la relation $2^x = 2^n \times 2^t$, que nous pouvons utiliser puisque t est représentable en machine. D'autre part, 2^n est une puissance de 2, donc les nombres 2^x et 2^t ont la même mantisse ; par conséquent, il y aura correspondance des pires cas. Il est donc suffisant de connaître les pires cas pour $|x| \leqslant \frac{1}{2}$. Mais dans ce domaine, il est plus intéressant de tester la fonction réciproque $\log_2$ (cf section 1.6).

La mantisse de t est d'autant plus petite (en nombre de bits, si on ne renormalise pas) que $|n|$ est grand. Par exemple, dans un système avec 13 bits de mantisse, $x = 10.01100110001$ donne la mantisse 1100110001 pour t, ou 1100110001000 en ajoutant des 0 à droite pour se ramener sur 13 bits. Donc il y aura en moyenne deux fois moins de pires cas réduits[8] entre 2^k et $2^{k+1} - 1$ qu'entre 2^{k-1} et $2^k - 1$ (bien qu'il y ait le même nombre de nombres machine), mais un pire cas réduit apparaîtra deux fois plus souvent (une fois pour chaque partie entière possible de x). Une conséquence est que la valeur maximale de m pour laquelle le TMD se produira est en général plus petite quand $|n|$ est grand (de manière probabiliste, elle diminue de 1 à chaque fois que n, ou x, est multiplié par 2).

Les résultats obtenus sont utiles pour déduire les pires cas du logarithme. Mais pour les grands arguments, il faut connaître aussi les pires cas dont la valeur de m est plus petite que celle considérée dans le voisinage de 1 (rappelons qu'ici, l'entier m associé à un nombre machine x est le plus grand entier tel que le TMD puisse se produire pour le calcul de $\log_2(x)$ à 2^{-m} près sur la mantisse du résultat[9]). En effet, en posant $x = 2^\alpha t$ (avec $\alpha \in \mathbb{Z}$ et t proche de 1), on a la relation $\log_2(2^\alpha t) = \alpha + \log_2(t)$. L'ajout de l'entier α introduit un décalage dans la mantisse de $\log_2(t)$. Par exemple, considérons un pire cas t pour lequel $k = m - n = 30$ (30 bits identiques après la mantisse représentée en machine) ; l'ajout de α peut introduire 10 bits identiques à gauche des k bits identiques, donnant un pire cas avec 40 bits identiques (cette suite de 40 bits identiques était présente dans la représentation binaire de $\log_2(t)$, mais ses 10 premiers bits faisaient partie de la mantisse représentée en machine et n'étaient donc

---

[8] Le terme « réduit » fait référence à la réduction d'argument : il s'agit du pire cas t correspondant au pire cas x.

[9] Il s'agit aussi de la longueur de la séquence allant du premier bit de la mantisse au dernier des bits identiques considérés pour le TMD.

pas pris en compte pour la valeur de k). Pour obtenir ces pires cas supplémentaires, il faut donc tester 2^x dans d'autres intervalles, avec x assez gros.

Plus précisément, à partir des résultats de 2^x sur un intervalle du type $[2^n, 2^n + 1[$, nous pouvons générer *rapidement* les résultats de 2^x sur un intervalle du type $[2^{n+p}, 2^{n+p} + 1[$, où p est un entier strictement positif : il suffit d'effectuer un décalage sur la partie fractionnaire de l'argument. Considérons par exemple le pire cas suivant de l'intervalle $[1, 2[$:

$$x = 1.11010010111001111101001011101000111010001101111011100$$

(il y a 54 bits). Dans l'intervalle $[2, 3[$, cela donnera le pire cas suivant :

$$x = 10.1101001011100111110100101110100011101000110111101110,$$

qui a la même partie fractionnaire, mais avec un bit de moins (parce que la partie entière a un bit de plus). Dans l'intervalle $[4, 5[$, on aura :

$$x = 100.110100101110011111010010111010001110100011011110111,$$

qui a encore la même partie fractionnaire avec un bit de moins. Mais dans l'intervalle suivant $[8, 9[$, le prochain bit perdu de la partie fractionnaire est un « 1 » ; cela donnerait un nombre avec une partie fractionnaire différente. Donc dans ce cas, le pire cas n'est pas pris en compte.

Nous générons ainsi des pires dans d'autres intervalles du type $[2^n, 2^n + 1[$.

Pour trouver tous les pires cas de 2^x et de $\log_2(x)$, nous avons choisi de tester :

$\log_2(x)$	pour x dans $[1/2, 2[$;
2^x	pour x dans $[1, 2[$ et $[32, 33[$.

Avec les paramètres que nous avons choisis lors du test de 2^x dans $[1, 2[$, nous pouvons obtenir tous les pires cas de la réciproque $\log_2$ dans $[2^1, 2^2[$, tels que $k \geqslant 48$. Nous en déduisons alors les pires cas de $\log_2$ dans $[2^2, 2^4[$ tels que $k \geqslant 49$, puis les pires cas de $\log_2$ dans $[2^4, 2^8[$ tels que $k \geqslant 50$, et ainsi de suite. Mais arrivé à l'intervalle $[2^{32}, 2^{64}[$, nous n'avons que les pires cas tels que $k \geqslant 53$, ce qui nous a semblé insuffisant. Nous avons donc retesté 2^x dans l'intervalle $[32, 33[$. Avec les paramètres que nous avons choisis, nous obtenons les pires cas de $\log_2$ dans $[2^{32}, 2^{64}[$ tels que $k \geqslant 47$, et ainsi de suite. Bien sûr, tous ces choix sont arbitraires ; ce n'est qu'un exemple. Nous avons d'ailleurs dû retester ces deux fonctions pour l'implémentation de 2^x (section 4.5.2), à cause du choix de notre algorithme (qui n'est pas le meilleur), au prix de très grosses tables.

Les tableaux 3.4 et 3.5 présentent les pires cas de $\log_2(x)$ et de 2^x. Concernant les pires cas de $\log_2(x)$, seuls ceux ayant un exposant égal à -1, 0 ou une puissance de 2 sont donnés. Les autres peuvent être obtenus en utilisant la relation :

$$\log_2(2^\alpha x) = \alpha + \log_2(x).$$

– Pour trouver tous les pires cas d'exposant E positif, nous prenons la puissance de 2 immédiatement inférieure à E, que nous notons E'. Nous prenons alors les mantisses de tous les pires cas d'exposant E'. D'après la relation ci-dessus, seuls les premiers chiffres de la mantisse du résultat changent, donc les bits situés après la mantisse du résultat (i.e. ceux qui déterminent si on a un pire cas ou non) restent les mêmes. Par exemple, si $E = 289$, nous considérons les pires cas d'exposant $E' = 256$ (ici, il y en a deux) ; nous gardons les mantisses de ces pires cas, et nous remplaçons l'exposant 256 par l'exposant 289.
– Pour trouver tous les pires cas d'exposant $E \leqslant -2$, nous prenons α égal au plus petit entier tel que $\log_2(2^\alpha x)$ soit positif et ait une partie entière de même taille qu'avec les pires cas d'exposant E. Les bits de la partie fractionnaire du résultat sont inversés. Plus précisément, si nous considérons un exposant compris entre -2^{p+1} et $-(2^p + 1)$, alors les pires cas se déduisent de ceux de l'exposant 2^p.

E	mantisse	R	m
0	1.1011010011101011111001000000110010010101101000000001	N	**107**
1	1.0001101111010001110011111111100101000111000111101010	D	**106**
2	1.0001101111010001110011111111100101000111000111101010	D	**107**
2	1.1000100111011001010010001010100101001111111000010111	N	**106**
4	1.0001101111010001110011111111100101000111000111101010	N	**108**
16	1.1001010101101011011011110011010000011111010111010000	N	**106**
64	1.0110000101010101010111110111010110001000010110110100	D	**106**
128	1.0110000101010101010111110111010110001000010110110100	D	**107**
128	1.1101001100001010010000110111011100111101110100011011	D	**106**
256	1.0110000101010101010111110111010110001000010110110100	D	**108**
256	1.1101001100001010010000110111011100111101110100011011	N	**107**
512	1.0110000101010101010111110111010110001000010110110100	D	**109**

TAB. 3.4: *Pires cas de* $\log_2(x)$ *pour lesquels* $m \geqslant 106$. *La première colonne indique l'exposant de l'argument ; seules les valeurs* -1, 0 *et les puissances de* 2 *sont données, car les autres exposants s'en déduisent ; en particulier, notons qu'il n'y a aucun pire cas pour* $E = -1$. *La deuxième colonne donne la mantisse de l'argument. La troisième colonne donne le mode d'arrondi pour lequel il s'agit d'un pire cas : D pour arrondi dirigé, N pour arrondi au plus près. La quatrième colonne donne la valeur de* m.

3.4.2 Vérification des hypothèses probabilistes

Nous allons donner le nombre de pires cas obtenus, dans différentes conditions. Cela permet de vérifier en pratique que les hypothèses probabilistes sont satisfaites. Les notations sont celles de la section 3.1.3.

Le tableau 3.6 page suivante donne le nombre moyen de pires cas x par exposant pour diverses fonctions et diverses valeurs de k, ainsi que l'estimation donnée par les hypothèses probabilistes. Rappelons que k est la longueur

E	mantisse	R	m
-15	$-1.0010100001100011101010111010111010101111011110110010$	D	**111**
-20	$-1.0100000101101111011011000110010001000101101011001111$	D	**111**
-32	$-1.0000010101010110000000001110010001010101100111110001$	D	**111**
-33	$-1.0001100000101101110001101101101101101010110000011101$	D	**111**
-29	$1.0101011010001110100011001110110001001111011001101100$	N	**111**
-27	$1.0001001010110001010010100011000110001111100100000100$	D	**112**
-25	$1.1011111110111011110111100100010011101101111111000101$	D	**113**
-10	$1.1110010001011001011001010010011010111111100101001101$	N	**113**
-10	$1.1110011101100000001001001000001110011000001100111111$	N	**111**
-8	$1.1111100110011010111111101111101000110000110101100101$	D	**111**
-6	$1.1000111111101010100000001111110110111010100010010110$	N	**111**

TAB. 3.5: *Pires cas de 2^x pour lesquels $m \geqslant 111$.*

de la séquence formée par le bit d'arrondi et les bits identiques qui suivent dans la mantisse de $f(x)$:

$$\underbrace{1.xxx...xx}_{n \text{ bits}}\underbrace{r0000...00}_{k \text{ bits}}1...$$

ou

$$\underbrace{1.xxx...xx}_{n \text{ bits}}\underbrace{r1111...11}_{k \text{ bits}}0...$$

Nous avons choisi de donner le nombre moyen de pires cas, et non leur nombre total, afin de faciliter la comparaison avec l'estimation. Le nombre total peut être retrouvé en multipliant la moyenne par le nombre d'exposants et en arrondissant à l'entier le plus proche.

k_0	estimation	$\exp(x)$	$\exp(-x)$	$\sin(x)$	$\cos(x)$
45	768	788.7	761.0	774.8	762.5
46	384	393.4	377.9	398.0	396.5
47	192	200.3	190.1	198.7	190.0
48	96	98.2	95.9	104.5	94.5
49	48	52.9	46.8	52.8	41.5
50	24	27.4	23.5	26.2	18.0
51	12	14.0	11.4	12.2	6.5
52	6	7.0	5.2	6.3	3.0
53	3	2.6	2.4	3.3	2.0

TAB. 3.6: *Nombre moyen de pires cas par exposant dont la valeur k vérifie $k \geqslant k_0$, pour différentes fonctions élémentaires et différentes valeurs de k_0. Les exposants considérés sont : -1 à 8 (10 exposants) pour $\exp(x)$, 0 à 7 (8 exposants) pour $\exp(-x)$, -5 à 0 (6 exposants) pour $\sin(x)$, et -2 à -1 (2 exposants) pour $\cos(x)$.*

L'estimation s'obtient de la manière suivante. Pour tout réel x aléatoire, nous avons $k \geqslant k_0$ avec une probabilité égale à 2^{2-k_0}. Comme il y a 2^{53} arguments par exposant, nous devrions obtenir environ, selon les hypothèses probabilistes, 2^{55-k_0} pires cas vérifiant $k \geqslant k_0$. Puisque nous éliminons immédiatement les pires cas du type $(1,1)$, et que les quatre types doivent être équiprobables, l'estimation du nombre de pires cas passe à $3 \times 2^{53-k_0}$. Pour $k_0 = 45$, nous avons une estimation de 768 pires cas par exposant. Ensuite, ce nombre est divisé par 2 à chaque fois que k_0 augmente de 1. Nous remarquons que dans l'ensemble, les hypothèses probabilistes semblent vérifiées. Ceci est confirmé par le tableau 3.7 ci-dessous, où nous distinguons les trois types de pires cas.

type	f	f^{-1}	estimation	$\exp(x)$	$\exp(-x)$	$\sin(x)$	$\cos(x)$
$(0,0)$	D	D	256	263.3	246.8	254.0	232.5
$(0,1)$	N	-	256	259.7	255.8	259.8	262.0
$(1,0)$	-	N	256	265.7	258.5	261.0	268.0
total			768	788.7	761.0	774.8	762.5

TAB. 3.7: *Nombre moyen de pires cas de chaque type, par exposant, dont la valeur k vérifie $k \geqslant 45$. Dans la colonne f, un « D » indique que les pires cas sont valables pour les modes d'arrondi dirigés uniquement, et un « N » indique que les pires cas sont valables pour le mode d'arrondi au plus près uniquement. En ce qui concerne la colonne f^{-1}, c'est la même signification, mais pour la fonction réciproque.*

Le fait que les hypothèses probabilistes soient confirmées est pour nous très important. En effet, si on peut espérer que de légères améliorations des algorithmes et l'augmentation de la rapidité des machines permettent d'ici quelques années de résoudre le dilemme du fabricant de tables pour la précision étendue (64 bits de mantisse), il semble illusoire d'espérer autre chose que des estimations probabilistes pour la quadruple précision (113 bits de mantisse).

Dans certains domaines particuliers, les hypothèses probabilistes ne sont plus vérifiées ; par exemple, c'est le cas de la fonction $\exp(x)$ pour x très petit en valeur absolue : environ $|x| < 2^{-32}$. Ceci est notamment dû au fait que les premiers termes de la formule de Taylor s'expriment très simplement en base 2 : $\exp x \approx 1 + x + 2^{-1}x^2$. On obtient ainsi de très grandes valeurs de k. Par exemple, pour

$$\begin{aligned} x &= 1.1111111111111111111111111111111111111111111110000000 \times 2^{-46} \\ &= 2^{-45} - 2^{-91}, \end{aligned}$$

$\exp(x)$ est égal à :

$$1.0000000000000000000000000000000000000000000001111111\ 1\,1^{83}\ 0101\ldots$$

Cependant, ce sont des pires cas particuliers dans un domaine particulier ; ce domaine devra être traité séparément lors de l'implémentation de la fonction exponentielle, et ne devrait pas poser de problèmes.

Chapitre 4

Implémentation des fonctions élémentaires

4.1 Introduction

Nous cherchons maintenant à implémenter de manière efficace des fonctions élémentaires f avec arrondi exact, et en particulier à montrer que le calcul avec arrondi exact est possible *en pratique* et que le temps moyen de calcul n'est pas beaucoup plus grand que celui des algorithmes actuels, pour lesquels l'arrondi exact n'est pas garanti.

En gros, tout ce que nous savons faire, c'est pour tout argument x (nombre machine, ou ayant éventuellement plus de précision) et tout entier m, calculer $f(x)$ avec une erreur sur la mantisse inférieure à 2^{-m}, sous réserve que les ressources nécessaires (temps CPU et mémoire) soient disponibles. Le problème des ressources est très important, puisqu'il peut être *a priori* nécessaire de prendre une très grande valeur de m (dans le cas où on n'aurait pas tous les résultats des tests exhaustifs) : s'il n'y a pas assez de mémoire pour un argument donné, l'arrondi exact ne pourra pas être garanti, mais cela peut aussi provoquer un dysfonctionnement ou le « plantage » d'autres programmes tournant sur la même machine ; et si le temps est trop long, cela peut revenir à considérer que le programme est bloqué, du moins du point de vue de l'utilisateur, et ce n'est pas non plus acceptable. Ceci dit, d'après les hypothèses probabilistes exposées dans l'introduction, il est extrêmement improbable que ces problèmes de ressources puissent se produire, mais puisque nous voulons prévoir le cas, nous devons le faire rigoureusement.

Nous allons commencer par montrer comment les calculs peuvent être faits pour que le temps moyen de calcul ne soit pas beaucoup plus grand qu'avec un algorithme classique (ne garantissant pas l'arrondi exact), mais tout de même assez précis (section 4.2). Nous parlerons ensuite des majorants de m_0, valeur maximale de m pour laquelle le dilemme du fabricant de tables (TMD) peut se produire, ainsi que du calcul à très grande précision (disons, plusieurs millions de bits), ce qui pourra donner une idée du temps maximal garanti au cas très improbable où un tel calcul soit nécessaire... (sections 4.3 et 4.4). Puis nous expliquerons comment les résultats des tests exhaustifs peuvent être utilisés pour minimiser les calculs sur les pires cas (section 4.5) ; nous présenterons une implémentation de la fonction 2^x, et nous donnerons des mesures des temps de calcul.

4.2 Stratégie de Ziv

La méthode de Ziv [53] consiste à évaluer $f(x)$, à l'aide d'une méthode classique [37, 33], avec tout d'abord une valeur de m (précision des calculs intermédiaires) un peu plus grande que n (taille de la mantisse des nombres machine), et en cas d'échec, prendre une valeur de m plus grande, et ainsi de suite. Par exemple, on peut commencer avec $m = n + 20$. Supposons que les hypothèses probabilistes restent valables pour la valeur calculée, i.e. que les mantisses des valeurs calculées soient équiréparties modulo 2^{1-n} (ce n'est évidemment pas tout à fait vrai, mais cela permet d'obtenir un ordre de grandeur des probabilités d'échec). Alors l'arrondi exact de $f(x)$ peut être déduit avec une probabilité égale à $1 - 2^{1-n}$, i.e. la probabilité d'échec 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. Et ainsi de suite...

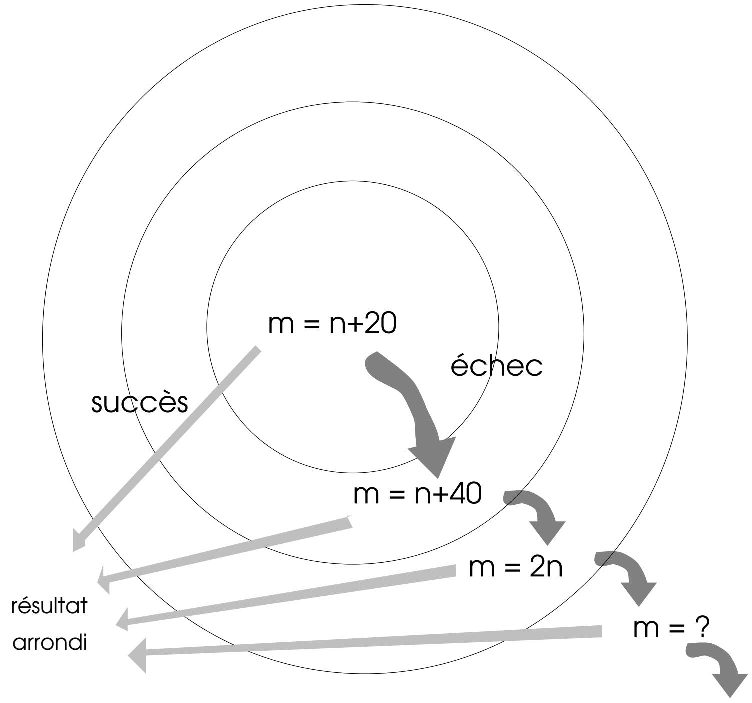

FIG. 4.1: *La stratégie « peau d'oignon » de Ziv.*

Pour être sûr de pouvoir garantir l'arrondi exact (à condition que les ressources soient suffisantes), il faut connaître une borne sur m, qui peut être donnée soit par tests exhaustifs, qui donneront en fait la valeur maximale de m, soit à l'aide de théorèmes de théorie des nombres, comme celui de Nesterenko et Waldschmidt (cf section 4.3).

Ziv a implémenté sa méthode avec Olshansky pour la bibliothèque ml4j (Mathematical Library for Java) d'IBM, en utilisant notamment l'arrondi exact des quatre opérations arithmétiques en double précision (suivant la norme IEEE-754) pour la portabilité. Seul l'arrondi au plus près est supporté. D'après la documentation fournie avec leur bibliothèque :

– La première étape utilise la double précision, soit $m \approx n$.
– La deuxième étape utilise une arithmétique où chaque nombre est représenté par une somme de deux nombres machine en double précision, le second servant en quelque sorte de correction au premier [16] ; donc $m \approx 2n$.
– La troisième étape utilise de la multiprécision où les nombres sont des flottants avec une mantisse de 32 chiffres en base 2^{24}, donc nous avons $m \approx 32 \times 24 = 768$; les chiffres sont représentés par des nombres machine en double précision.

Puisque cette implémentation ne peut pas faire de calculs à une plus grande précision, l'arrondi exact *n'est pas garanti*, même s'il est très très improbable que m_0 soit suffisamment grand pour qu'un arrondi inexact puisse se produire.

4.3 Théorème de Nesterenko et Waldschmidt

4.3.1 Théorème principal

Nous allons présenter le théorème de Nesterenko et Waldschmidt [39] et en donner des applications pour trouver des majorants de m_0 pour différentes fonctions et différents intervalles.

Définition 1 (Hauteur logarithmique absolue de Weil) *Soit α un nombre algébrique de polynôme minimal $P(X) = a_0 X^d + \ldots + a_d \in \mathbb{Z}[X]$, et $\alpha_1, \ldots, \alpha_d$ les racines (complexes) de P. La hauteur logarithmique absolue de Weil de α est définie par :*

$$h(\alpha) = \frac{1}{d}\left(\log|a_0| + \sum_{i=1}^{d} \log\max\{1, |\alpha_i|\}\right).$$

Théorème 3 (Yu. Nesterenko and M. Waldschmidt (1995)) *Soient $\theta \in \mathbb{C}^*$, et α, β des nombres algébriques. Soient $\mathbb{K} = \mathbb{Q}(\alpha, \beta)$ et $d = [\mathbb{K} : \mathbb{Q}]$. Soient A, B et E des réels strictement positifs, avec $E \geqslant e$, satisfaisant*

$$\log A \geqslant \max\{h(\alpha), d^{-1}\} \quad \textit{et} \quad \log B \geqslant h(\beta).$$

Alors

$$\begin{aligned}
&\left|e^{\theta}-\alpha\right|+|\theta-\beta| \\
&\quad \geqslant \exp\Big(-211d \times (\log B+\log\log A+4\log d+2\log(E\,|\theta|_+)+10) \\
&\qquad \times (d\log A+2E\,|\theta|+6\log E)\times(3.3d\log(d+2)+\log E)\times(\log E)^{-2}\Big)
\end{aligned}$$

où $|\theta|_+ = \max\{1, |\theta|\}$.

4.3.2 Application à l'exponentielle et au logarithme

Nous voulons appliquer ce théorème aux nombres machine, qui sont des rationnels (nombres algébriques de degré 1). Soit un rationnel non nul $\frac{p}{q}$, avec $p \wedge q = 1$. Alors $h(\frac{p}{q}) = \log\max\{|p|, |q|\}$. Posons $H(\frac{p}{q}) = \max\{|p|, |q|\}$. Du théorème 3, nous en déduisons :

Corollaire 1 *Soient $\theta \in \mathbb{C}^*$, et α, β des nombres rationnels. Soient A, B et E des réels strictement positifs, avec $E \geqslant e$, satisfaisant*

$$A \geqslant \max\{H(\alpha), e\} \quad \textit{et} \quad B \geqslant H(\beta).$$

Alors

$$\begin{aligned}
&\left|e^{\theta}-\alpha\right|+|\theta-\beta| \\
&\quad \geqslant \exp\Big(-211 \times (\log B+\log\log A+2\log(E\,|\theta|_+)+10) \\
&\qquad \times (\log A+2E\,|\theta|+6\log E)\times(3.3\log(3)+\log E)\times(\log E)^{-2}\Big)
\end{aligned}$$

où $|\theta|_+ = \max\{1, |\theta|\}$.

En posant $\theta = \beta$, nous obtenons une minoration de la distance entre l'exponentielle d'un rationnel (en particulier, d'un nombre machine) et un rationnel (en particulier, un nombre machine) :

Corollaire 2 *Soient α et β des nombres rationnels, avec $\beta \neq 0$. Soient A, B et E des réels strictement positifs, avec $E \geqslant e$, satisfaisant*

$$A \geqslant \max\{H(\alpha), e\} \quad \textit{et} \quad B \geqslant H(\beta).$$

Alors

$$\begin{aligned}
&\left|e^{\beta}-\alpha\right| \\
&\quad \geqslant \exp\Big(-211 \times (\log B+\log\log A+2\log(E\,|\beta|_+)+10) \\
&\qquad \times (\log A+2E\,|\beta|+6\log E)\times(3.3\log(3)+\log E)\times(\log E)^{-2}\Big)
\end{aligned}$$

où $|\beta|_+ = \max\{1, |\beta|\}$.

Et en posant $\theta = \log \alpha$, nous obtenons une minoration de la distance entre le logarithme d'un rationnel (en particulier, d'un nombre machine) et un rationnel (en particulier, un nombre machine) :

Corollaire 3 *Soient α et β des nombres rationnels, avec $\alpha > 0$ et $\alpha \neq 1$. Soient A, B et E des réels strictement positifs, avec $E \geqslant e$, satisfaisant*

$$A \geqslant \max\{H(\alpha), e\} \quad \textit{et} \quad B \geqslant H(\beta).$$

Alors

$$\begin{aligned}&|\log \alpha - \beta| \\ &\quad \geqslant \exp\Big(-211 \times (\log B + \log\log A + 2\log(E\,|\log \alpha|_+) + 10) \\ &\qquad \times (\log A + 2E\,|\log \alpha| + 6\log E) \times (3.3\log(3) + \log E) \times (\log E)^{-2}\Big)\end{aligned}$$

où $|\log \alpha|_+ = \max\{1, |\log \alpha|\}$.

4.3.3 Application aux fonctions trigonométriques

Ici, nous ne chercherons pas à être tout à fait rigoureux, mais juste à montrer comment on peut appliquer le théorème de Nesterenko et Waldschmidt pour obtenir les bornes cherchées.

4.3.3.1 Sinus et cosinus

Nous pouvons aussi utiliser le théorème de Nesterenko et Waldschmidt pour en déduire une minoration de la distance entre le sinus ou le cosinus d'un nombre machine et un nombre machine. Pour cela, posons $\theta = \beta = ix$ avec x rationnel (nombre machine, en fait) et $\alpha = a + ib$ vérifiant $|\alpha| = 1$ (i.e. $a^2 + b^2 = 1$) ainsi que a ou b rationnel (nombre machine, en fait). Supposons que a soit rationnel (le cas b rationnel est similaire). Nous cherchons alors une minoration de $|\cos x - a|$.

Soient $c = |\cos x - a|$ et $s = |\sin x - b|$. Puisque a est rationnel et $a^2 + b^2 = 1$, alors b est algébrique de degré inférieur ou égal à 2, et α aussi. Donc nous pouvons appliquer le théorème de Nesterenko et Waldschmidt, qui nous donne une minoration du style : $|c + is| \geqslant \varepsilon$, i.e. $c^2 + s^2 \geqslant \varepsilon^2$. D'autre part,

$$\cos^2 x + \sin^2 x = 1 = a^2 + b^2,$$

donc

$$|\cos x + a| \times |\cos x - a| = |\sin x + b| \times |\sin x - b|$$

et $s = k.c$ avec

$$k = \frac{|\cos x + a|}{|\sin x + b|}.$$

Par conséquent, $c^2 + s^2 = (1 + k^2)c^2$, et

$$c \geqslant K_c\,\varepsilon \quad \text{avec} \quad K_c = \frac{1}{\sqrt{1+k^2}}.$$

Essayons d'évaluer la valeur de K_c approximativement. Puisque nous nous intéressons à la minoration de $c = |\cos x - a|$, nous supposons que le nombre machine a est choisi de telle façon à ce qu'il soit très proche de $\cos x$ (par exemple, c'est l'arrondi de $\cos x$). Alors b est aussi très proche de $\sin x$. Donc $k \approx |\cot x|$, et $K_c \approx |\sin x|$.

De même, pour le cas b rationnel, $s \geqslant K_s\,\varepsilon$ avec $K_s \approx |\cos x|$.

4.3.3.2 Tangente

Pour la tangente, nous utilisons le fait que les fonctions $\tan$ et $\cos$ sont algébriquement liées :

$$1 + \tan^2 x = \frac{1}{\cos^2 x}.$$

Nous voulons minorer $|\tan x - t|$, où x et t sont des rationnels (nombres machine, en fait). Si $\tan x$ est proche de t, alors $\cos x$ sera proche d'un nombre algébrique a tel que :

$$1 + t^2 = \frac{1}{a^2}.$$

Le raisonnement pour minorer la distance entre le cosinus d'un nombre machine et un nombre machine s'applique aussi aux nombres algébriques. Donc nous avons une inégalité du type :

$$|\cos x - a| \geqslant \varepsilon.$$

Donc

$$\left|\cos^2 x - a^2\right| \geqslant \varepsilon\left|\cos x + a\right|,$$

puis

$$\left|\frac{1}{a^2} - \frac{1}{\cos^2 x}\right| \geqslant \frac{\varepsilon\left|\cos x + a\right|}{a^2\cos^2 x}.$$

Or

$$\left|\frac{1}{a^2} - \frac{1}{\cos^2 x}\right| = \left|(1+t^2) - (1+\tan^2 x)\right|$$

Donc

$$\left|t^2 - \tan^2 x\right| \geqslant \frac{\varepsilon\left|\cos x + a\right|}{a^2\cos^2 x}$$

et

$$|\tan x - t| \geqslant \frac{\varepsilon\left|\cos x + a\right|}{a^2\cos^2 x\left|\tan x + t\right|}.$$

4.3.4 Application aux fonctions hyperboliques

Comme pour les fonctions trigonométriques, nous ne chercherons pas à être tout à fait rigoureux, mais juste à montrer comment on peut appliquer le théorème de Nesterenko et Waldschmidt pour obtenir les bornes cherchées.

4.3.4.1 Sinus et cosinus hyperboliques

La méthode est similaire à celle utilisée pour les fonctions sinus et cosinus. Posons $\theta = \beta = x$ avec x rationnel (nombre machine) et $\alpha = a + b$ vérifiant $a^2 - b^2 = 1$ ainsi que a ou b rationnel (nombre machine). Supposons que a soit rationnel (le cas b rationnel est similaire). Nous cherchons alors une minoration de $|\operatorname{ch} x - a|$.

Soient $c = |\operatorname{ch} x - a|$ et $s = |\operatorname{sh} x - b|$. Puisque a est rationnel et $a^2 - b^2 = 1$, alors b est algébrique de degré inférieur ou égal à 2, et α aussi. Donc nous pouvons appliquer le théorème de Nesterenko et Waldschmidt, qui nous donne une minoration du style : $|e^x - \alpha| \geqslant \varepsilon$, i.e. $|(\operatorname{ch} x - a) + (\operatorname{sh} x - b)| \geqslant \varepsilon$, donc $c + s \geqslant \varepsilon$. D'autre part,

$$\operatorname{ch}^2 x - \operatorname{sh}^2 x = 1 = a^2 - b^2,$$

donc

$$|\operatorname{ch} x + a| \times |\operatorname{ch} x - a| = |\operatorname{sh} x + b| \times |\operatorname{sh} x - b|$$

et $s = k.c$ avec

$$k = \frac{|\operatorname{ch} x + a|}{|\operatorname{sh} x + b|}.$$

Par conséquent, $c + s = (1 + k)c$, et

$$c \geqslant K_c\, \varepsilon \quad \text{avec} \quad K_c = \frac{1}{1+k}.$$

Essayons d'évaluer la valeur de K_c approximativement. On a par hypothèse $\operatorname{ch} x \approx a$, donc $k \approx \coth x$, et

$$K_c = \frac{1}{1+k} \approx \frac{\operatorname{sh} x}{e^x} = \frac{1 - e^{-2x}}{2}.$$

De même, pour le cas b rationnel, $s \geqslant K_s\, \varepsilon$ avec

$$K_s \approx \frac{\operatorname{ch} x}{e^x} = \frac{1 + e^{-2x}}{2}.$$

4.3.4.2 Tangente hyperbolique

De manière similaire à la fonction $\tan$, nous utilisons le fait que les fonctions th et ch sont algébriquement liées :

$$1 - \operatorname{th}^2 x = \frac{1}{\operatorname{ch}^2 x}.$$

Nous voulons minorer $|\mathrm{th}\, x - t|$, où x et t sont des rationnels (nombres machine, en fait). Si $\mathrm{th}\, x$ est proche de t, alors $\mathrm{ch}\, x$ sera proche d'un nombre algébrique a tel que :

$$1 - t^2 = \frac{1}{a^2}.$$

Le raisonnement pour minorer la distance entre le cosinus hyperbolique d'un nombre machine et un nombre machine s'applique aussi aux nombres algébriques. Donc nous avons une inégalité du type :

$$|\mathrm{ch}\, x - a| \geqslant \varepsilon.$$

Après calculs similaires à ceux de la section 4.3.3.2, nous obtenons :

$$|\mathrm{th}\, x - t| \geqslant \frac{\varepsilon\,|\mathrm{ch}\, x + a|}{a^2 \,\mathrm{ch}^2\, x\,|\mathrm{th}\, x + t|}.$$

4.3.5 Bornes sur m_0 obtenues

Nous cherchons maintenant à appliquer tous ces résultats pour trouver des bornes sur m_0 (nous nous contenterons des ordres de grandeur) en double précision, que nous noterons M_0.

Dans le théorème de Nesterenko et Waldschmidt, nous avons trois paramètres qu'il faut fixer : A, B et E. Nous pouvons les choisir de manière à ce qu'ils donnent les meilleures bornes possibles. Pour A et B, il suffit de prendre les valeurs minimales données par les conditions du théorème. La valeur optimale de E se trouve en maximisant la fonction correspondante.

Soit x un nombre machine. Cherchons un ordre de grandeur de $H(x)$, où $H(\frac{p}{q}) = \max\{|p|, |q|\}$, qui nous servira pour fixer A et B. Nous pouvons écrire $x = M.2^k$, où M et k sont des entiers, et M est impair. Nous voulons en fait un minorant de $H(x)$ sur tout un intervalle, donc nous considérons les nombres machine x les plus « compliqués », c'est-à-dire pour lesquels $H(x)$ sera *a priori* très grand par rapport aux autres : ce sont les nombres machine dont la mantisse se termine par un 1, i.e. l'entier M s'écrit sur 53 bits. Nous pouvons considérer ces nombres machine dans n'importe quel domaine du fait de leur densité. Maintenant, nous avons en gros trois cas, dont les frontières ne sont pas tout à fait strictes puisque nous travaillons toujours en ordre de grandeur :

– $|x| \leqslant 1$: lorsque l'on écrit $x = p/q$, c'est le dénominateur qui est le plus grand, donc $H(x) = |q|$, qui a pour ordre de grandeur $2^{53}/x$.

– $1 \leqslant |x| \leqslant 2^{53}$: le numérateur de $x = p/q$ est plus grand que le dénominateur ; il a pour ordre de grandeur 2^{53}.

– $2^{53} \leqslant |x|$: x est un entier, donc $H(x) = |x|$.

4.3.5.1 Exponentielle et logarithme

Nous allons considérer trois domaines pour $\exp(x)$:

1. $2^{-53} \leqslant |x| \leqslant 1 : A = 2^{53}, B = 2^{100}$.
2. $1 \leqslant |x| \leqslant 36 : A = B = 2^{53}$.
3. $|x| \geqslant 36 : A = 2^{1000}, B = 2^{53}$.

domaine	$\|x\|$	M_0
1	2^{-50}	3.5×10^5
2	1	1.2×10^6
2	2	1.7×10^6
2	36	1.5×10^7
3	700	3.3×10^8

Note : si x est suffisamment petit (par exemple $|x| < 2^{-53}$), nous pouvons utiliser la formule de Taylor à l'ordre 1 pour arrondir exactement. C'est pour cela que nous n'avons pas considéré ce cas ci-dessus.

En ce qui concerne le logarithme, nous obtenons les mêmes résultats que pour l'exponentielle (dans les domaines correspondants, en tant que fonction réciproque). Nous devons donc prévoir de faire des calculs sur plusieurs millions de bits, et jusqu'à plusieurs centaines de millions de bits.

Concernant la quadruple précision, les valeurs de M_0 pour $x = 1$, 2 et 36 sont environ deux fois plus grandes : 2.9×10^6, 4.0×10^6, 2.8×10^7. Pour x proche de la valeur maximale ($x = 11000$, $A = 2^{16000}$, $B = 2^{113}$), nous obtenons $M_0 = 8.8 \times 10^9$.

4.3.5.2 Sinus et cosinus

Par symétrie du problème, considérons le cas du cosinus. Nous reprenons les notations de la section 4.3.3.1.

Le théorème de Nesterenko et Waldschmidt doit être appliqué à $\alpha = a + ib$, qui est un nombre algébrique de module 1 et de degré 2, à β, qui est un rationnel imaginaire, et nous prendrons $\theta = \beta$. Nous avons alors

$$d = [\mathbb{Q}(\alpha, i) : \mathbb{Q}] = 4.$$

Si $a = \frac{p}{q}$, alors le polynôme minimal de α est de la forme :

$$P(X) = qX^2 - 2pX + q,$$

donc $h(\alpha) = \frac{1}{2} \log |q|$. Nous avons toujours, comme pour l'exponentielle et le logarithme, $h(\beta) = \log \max\{|p|, |q|\}$. Les contraintes sur A et B sont donc :

$$A \geqslant \max\{\sqrt{|q_a|}, e^{1/4}\} \quad \text{et} \quad B \geqslant \max\{|p_x|, |q_x|\}.$$

En considérant $x \approx 1$, nous prenons $A = 2^{27}$ et $B = 2^{53}$. Nous obtenons alors $M_0 \approx 2.7 \times 10^7$, c'est-à-dire encore plusieurs millions.

Le cas général est beaucoup plus difficile, car il faut prendre en compte le fait que $\sin x$ ou $\cos x$ peut être très proche de 0. Mais de telles valeurs de x peuvent être trouvées par développement en fraction continue de $\frac{\pi}{2}$, et il est possible de faire les minorations nécessaires.

4.4 Multiprécision

4.4.1 Introduction

Puisque le TMD peut *a priori* se produire avec de grandes valeurs de m, nous devons prévoir des routines de calcul en multiprécision, avec contrôle de l'erreur. Il existe déjà des bibliothèques de calcul en multiprécision des fonctions élémentaires, mais actuellement aucune ne garantit la précision du résultat renvoyé. P. Zimmermann a commencé l'écriture d'une telle bibliothèque portable et avec arrondi exact (extension de la norme IEEE-754) : MPFR, basée sur la bibliothèque GMP [22].

Il existe différents types de représentations des nombres en multiprécision. Il y a les expansions sur les flottants [42, 45, 46, 13, 14, 15], où chaque nombre est représenté par une somme de flottants qui ne se « recouvrent » pas, c'est-à-dire le poids du bit le moins significatif d'un flottant est toujours supérieur au poids du bit le plus significatif du flottant suivant. Cette représentation peut être utile à précision assez faible (la bibliothèque ml4j d'IBM en utilise un cas particulier, avec deux flottants seulement pour la deuxième étape de la stratégie de Ziv, cf section 4.2). Mais au moins pour les plus grandes précisions, ce choix ne conduit pas à des implémentations très efficaces, ou ne peut même plus du tout être utilisé à cause de la plage très limitée des exposants des flottants. Nous choisirons une représentation « classique » dans une grande base, et nous ne parlerons que de ces représentations (qui n'interviendront en fait qu'essentiellement pour l'addition et la multiplication, les autres opérations s'exprimant en fonction de celles-là). Nous considérons des précisions allant jusqu'à l'ordre du million de bits, et les algorithmes choisis doivent être les plus rapides pour ces précisions.

Les entiers positifs sont représentés par un tableau de chiffres dans une grande base B. Comme nous nous intéressons à des résultats en base 2, B sera de la forme 2^N ; c'est de toute façon plus efficace, tant en rapidité de traitement qu'en occupation de mémoire (le choix de Maple de faire tous les calculs en base 10 a été une grande erreur). Un entier positif s'écrit alors :

$$x = \sum_{i=0}^{r} x_i B^i = \sum_{i=0}^{r} x_i 2^{Ni},$$

où l'entier r et les chiffres x_i, entiers compris entre 0 et $B - 1 = 2^N - 1$, définissent complètement le nombre x. Concernant l'implémentation, les chiffres peuvent être stockés soit poids faible en tête, soit poids fort en tête, mais cela n'a pas beaucoup d'importance. La manipulation d'entiers négatifs, et plus généralement, de réels (approchés), peut se faire avec l'ajout d'un bit de signe et d'un exposant (entier qui correspond généralement à un type de données de base du langage, mais qui peut aussi être un entier en multiprécision, suivant l'implémentation).

Un choix important à faire est le type de données utilisé pour stocker les chiffres des nombres en multiprécision. On peut prendre :

– soit des entiers, dont la taille correspond de préférence à celle des registres du processeur, par exemple de 32 bits (et la valeur de N correspond aussi de préférence à cette taille) ;

– soit des flottants contenant en fait des entiers compris entre 0 et $B - 1$ (on peut aussi introduire le Ni dans la valeur de l'exposant, notamment dans des représentations internes, mais il faut tenir compte du fait que la plage des exposants est limitée pour ne pas avoir d'exceptions du genre $\pm\infty$ ou NaN).

Le choix devrait se faire de manière à avoir l'implémentation la plus rapide. Or de plus en plus, les processeurs sont plus rapides en calcul flottant qu'en calcul entier dès que la multiplication intervient. Cela peut paraître étrange car un calcul sur des entiers est plus simple qu'un calcul sur des flottants, mais cela vient du fait que les calculs sur des flottants sont aujourd'hui beaucoup plus utilisés que les calculs sur des entiers, et que les processeurs sont donc optimisés pour les calculs flottants. C'est par exemple le cas du MicroSPARC II (processeur des stations Sun Sparc 4 et 5), où une multiplication entière sur 32 bits prend toujours 22 cycles et où la multiplication flottante en double précision prend un ou deux cycles. Nous avons comparé deux implémentations différentes, sur processeur Sparc, d'un même algorithme de multiplication de nombres de grande taille : une effectuant les calculs élémentaires sur les entiers de 32 bits, l'autre sur les flottants en double précision ; le calcul sur les flottants était environ 7 fois plus rapide que celui sur les entiers. C'est aussi le cas du Cray-2 (ordinateur utilisé par Bailey pour calculer 29 360 000 décimales de π [3]), qui doit faire des conversions avec les flottants pour effectuer une multiplication entière.

Une méthode mixte peut aussi être employée : choisir les entiers comme type de base, et faire des conversions pour utiliser la multiplication flottante ; c'est ce que fait par exemple la nouvelle version de la bibliothèque GMP avec les processeurs Sparc.

Nous devons aussi tenir compte du fait que le bit *carry* servant entre autres à la propagation des retenues dans les additions et soustractions entières en multiprécision (et existant sur la plupart des processeurs) n'est pas directement accessible en C et dans les autres langages de haut niveau. Même si en

C, en s'appuyant sur la norme, on peut écrire des routines entièrement équivalentes à l'utilisation du bit *carry* comme expliqué à la fin de la section 3.1.2.2, les compilateurs ne sont pas capables de le détecter (ou très partiellement), et le code n'est donc pas du tout optimisé. Si un type entier est choisi pour stocker les chiffres en base B, il vaut donc mieux utiliser une bibliothèque de fonctions où les routines de bas niveau sont déjà écrites en assembleur, ou écrire les routines de bas niveau directement en assembleur ; pour les processeurs non supportés, des routines génériques (portables) peuvent être écrites en C, mais elles seront évidemment plus lentes. Comme bibliothèques de multiprécision pouvant servir de base, nous pouvons citer par exemple GMP de Torbjörn Granlund [22] et Apfloat de Mikko Tommila [50]. Les calculs avec des flottants n'ont pas ces problèmes de portabilité et peuvent être écrits entièrement en C à l'aide de la norme IEEE-754 (en supposant que le compilateur, avec les bonnes options, respecte cette norme, ce qui est généralement le cas).

Nous allons présenter successivement l'addition et la soustraction d'entiers, puis la multiplication d'entiers, qui sont les opérations sur lesquelles reposent tous les autres algorithmes (division, racine carrée, fonctions élémentaires), dont nous parlerons ensuite. Il est donc important de bien optimiser ces opérations. Les algorithmes d'addition et de soustraction sont classiques ; le principal problème concerne leur implémentation. Le cas de la multiplication est plus compliqué, d'une part en ce qui concerne l'implémentation, mais aussi pour le choix de l'algorithme, car le meilleur choix dépend de la taille des nombres à multiplier, du processeur et de la qualité de l'implémentation. Nous allons donc en parler plus en détail, en présenter une implémentation, et la comparer avec d'autres bibliothèques.

4.4.2 Addition et soustraction

L'algorithme est classique : l'opération se fait chiffre par chiffre, des chiffres de poids faible vers les chiffres de poids fort, sans oublier les retenues. Cependant, si l'on doit soustraire deux entiers positifs sans savoir lequel est le plus grand, nous devons commencer par soustraire les chiffres de poids fort, jusqu'à obtenir deux chiffres de poids fort différents ; à ce moment, la soustraction peut être faite normalement, sachant lequel des deux opérandes il faut réellement soustraire à l'autre.

Note : avec certaines représentations, notamment concernant les flottants en multiprécision, mixant base 2 et base 2^N, un décalage peut être nécessaire pour additionner des chiffres de même poids.

Parlons maintenant du problème des retenues, suivant que le type de données de base est entier ou flottant.

– Type de base entier : la plupart des processeurs permettent de faire très simplement des additions et soustractions en multiprécision, en utilisant le bit *carry* (et en déroulant suffisamment les boucles dans les cas où il doit être sauvé et récupéré plus tard, au niveau des tests de fin de boucle).

– Type de base flottant : il n'y a pas de bit *carry*. Nous choisirons donc une valeur de N pas trop grande de manière à pouvoir stocker la retenue dans le résultat de l'addition de deux chiffres. Parfois, nous devons faire plusieurs additions ou soustractions à la suite, ce qui arrive notamment avec certains algorithmes de multiplication (cf section 4.4.3) ; pour éviter de propager les retenues trop souvent, nous pouvons choisir N de manière à pouvoir accumuler un certain nombre de retenues. Cela permet aussi d'effectuer facilement des multiplications par de très petits entiers (2, 3, 4...). Dans notre implémentation de la multiplication, nous choisirons $N = 50$: les chiffres vont donc de 0 à $2^{50} - 1$, mais ils pourront temporairement contenir les entiers de -2^{53} à 2^{53} (en double précision). Cela peut être vu comme une représentation redondante. La valeur $N = 50$ permet d'accumuler jusqu'à 7 retenues. Lorsque l'on a un flottant `x` contenant un chiffre et des retenues, pour séparer la retenue `r` du « vrai » chiffre `y`, il suffit d'effectuer les deux opérations suivantes avec le mode d'arrondi vers $-\infty$: `r = (x + C) - C`, puis `y = x - r`, où `C` est une constante convenablement choisie. Nous prendrons $3 \times 2^{51+N}$. Si nous lui ajoutons une valeur comprise entre -2^{53} et 2^{53}, alors nous obtenons un nombre compris entre 2^{52+N} et $2^{53+N} - 1$, donc le bit de poids fort est 2^{52+N}, et le bit de poids faible est 2^N. De plus, l'arrondi vers $-\infty$ permet d'avoir `y` positif. Nous obtenons donc bien la retenue, multipliée par 2^N, après la première opération, et le chiffre compris entre 0 et $2^N - 1$ après la seconde opération. Si on autorise que le chiffre `y` puisse être négatif, alors n'importe quel mode d'arrondi convient. Notons que certains processeurs (par exemple, les processeurs compatibles Intel i386) font par défaut leurs calculs internes en précision étendue (avec 64 bits de mantisse) ; pour éviter un double arrondi (le premier lors de l'opération, le second lors du stockage en mémoire) et donc des résultats incorrects, il *faut* configurer le processeur pour que les calculs soient effectués en double précision (53 bits de mantisse) *exactement*.

4.4.3 Multiplication

Nous considérons ici la multiplication sur des entiers de même taille. En fait, ce seront des flottants en multiprécision que nous devrons multiplier pour effectuer les autres opérations (division, racine carrée) ou calculer les fonctions élémentaires ; mais la seule différence est la gestion des exposants qu'il faut faire en plus pour les flottants et l'arrondi éventuel du résultat pour obtenir un flottant à la même précision que les entrées.

On connaît actuellement trois types d'algorithmes pour calculer le carré ou le produit d'entiers :

– l'algorithme « classique » en $O(n^2)$: n^2 multiplications élémentaires (avec résultat de taille double) pour un produit, et $n(n-1)/2$ multiplications et n carrés pour un carré ;

- les algorithmes du type Toom-Cook [51, 12], qui généralisent une méthode inventée par Karatsuba et Ofman [26] : un entier est vu comme un polynôme (par décomposition en blocs de k chiffres), et on doit calculer le carré ou le produit de polynômes en faisant le moins possible de carrés ou multiplications ;
- les algorithmes utilisant la transformée de Fourier rapide (FFT).

Zuras en a étudié certains dans [54], plus particulièrement les algorithmes du type Toom-Cook.

Les algorithmes de type Toom-Cook et certains algorithmes utilisant la FFT sont basés sur le principe de programmation *diviser-pour-régner* : on transforme le problème pour se ramener à des multiplications de nombres plus petits. Tant que les nombres à multiplier sont assez grands, on applique récursivement cette transformation ; dans le cas contraire, on utilise un autre algorithme (basé sur une autre transformation, ou alors l'algorithme en n^2), de complexité en temps plus grande, mais en pratique plus rapide sur de petits entiers.

4.4.3.1 Algorithme en $O(n^2)$

Il s'agit d'un algorithme classique, mais il y a de nombreuses façons de l'implémenter. Il est important de bien l'optimiser, car il est utilisé par les algorithmes du type *diviser-pour-régner*, en fin de récursion.

Une des opérations élémentaires est la multiplication de nombres formés d'un seul chiffre, qui donne un nombre de deux chiffres. Or, suivant la base B choisie, le processeur ne peut pas forcément faire une telle multiplication. Par exemple, la multiplication entière peut donner le résultat sur un registre seulement, et la multiplication flottante de deux nombres en double précision donne un nombre en double précision (arrondi). Évidemment, le langage peut permettre des opérations supplémentaires, mais à la compilation, de telles opérations sont décomposées en plusieurs sous-opérations, ce qui ne nous intéresse pas forcément. La solution est de considérer une base $B = 2^N$, où B est un carré b^2 (i.e. N est pair), et avant chaque multiplication, de découper chaque chiffre relatif à la base B en deux chiffres relatifs à la base b, et de faire des regroupements convenables (additions, générant éventuellement des retenues).

Note : nous aurions pu considérer que les nombres étaient représentés directement en base b, i.e. que les chiffres étaient déjà découpés en deux. Cela peut être intéressant avec certains processeurs, par exemple capables de manipuler efficacement des demi-mots, ou de faire efficacement des multiplications de deux flottants en simple précision avec résultat en double précision. Mais cela peut être pénalisant avec d'autres processeurs. De toute façon, le résultat d'une multiplication élémentaire (nombre de deux chiffres en base b) doit être redécomposé pour obtenir deux chiffres séparés : nous nous retrouverions avec le même genre de problèmes.

Si le type élémentaire utilisé est un type entier, alors la décomposition en deux chiffres peut se faire avec des décalages logiques. Nous allons donner plus de détails concernant le choix d'un type flottant (double précision), puisque c'est ce que nous avons implémenté. Rappelons que $N = 50$, donc $B = 2^{50}$ et $b = 2^{25}$.

La décomposition d'un chiffre en deux est similaire à la décomposition d'un chiffre temporaire en retenue et chiffre compris entre 0 et $B - 1$ (cf section 4.4.2). Au lieu de $3 \times 2^{51+N}$, on utilise $3 \times 2^{51+N/2}$ pour la constante `C` ; dans l'algorithme, arrondi(z, C) correspond à l'opération $(z+C)-C$. L'implémentation que nous allons présenter correspond en fait naturellement à une multiplication avec accumulation : nous allons multiplier deux entiers x et y de taille n (i.e. à n chiffres) et ajouter en même temps le résultat à un entier z de taille potentiellement infinie ; c'est-à-dire z est au moins de taille $2n$, et tant qu'il y a une retenue après ces $2n$ chiffres, on propage cette retenue en supposant que cela a un sens. Non seulement cela simplifie l'implémentation de la routine de multiplication en $O(n^2)$, mais c'est en fait bien le comportement voulu quand cette routine sera appelée avec les méthodes du type Toom-Cook. L'algorithme détaillé est donné sur la figure 4.1 page suivante, avec des indications sur le contenu des variables utilisées permettant de prouver l'algorithme.

Un certain nombre de constantes (exactement représentables en double précision) sont définies. On devra décomposer les chiffres de x et de y en deux, et on aura deux boucles imbriquées : une parcourant les chiffres de x, et l'autre parcourant les chiffres de y. Nous choisissons que celle parcourant les chiffres de x est la boucle interne (l'autre possibilité était de toute façon équivalente). Pour éviter d'avoir à décomposer les chiffres de x à chaque itération de la boucle externe, nous le faisons une fois pour toutes dès le début et mémorisons les résultats dans un tableau x' de taille $2n$. La décomposition des chiffres de y se fait au début de la boucle externe : les deux chiffres en base b sont y_0 (chiffre de poids faible) et y_1 (chiffre de poids fort). Lors de la décomposition des chiffres en deux (que ce soit pour x ou pour y), chaque chiffre de poids fort obtenu correspond au vrai chiffre multiplié par $b = 2^{N/2}$. Pour obtenir le vrai chiffre, il faudrait donc le multiplier par $2^{-N/2}$; mais une telle opération se révélerait en fin de compte inutile et ferait perdre du temps. Donc les chiffres de rang pair de x et de y écrits en base b appartiennent à l'intervalle $\left[-2^{N/2}, 2^{N/2}\right]$, et ceux de rang impair appartiennent à l'intervalle $2^{N/2}\left[-2^{N/2}, 2^{N/2}\right]$.

La valeur de la variable z_1 est réutilisée pour l'itération suivante sur i : elle joue le rôle de retenue. Une itération calcule en gros le produit de deux nombres à un chiffre en base B (ou deux chiffres en base b) et ajoute la retenue z_1 et le chiffre courant (car on fait une accumulation) ; le résultat tient sur deux chiffres en base B : $[z_1' z_0']_B = \mathrm{retenue} + [z_0]_B + [x_1 x_0]_b \times [y_1 y_0]_b$. Le poids faible z_0' est stocké en mémoire (à la place de z_0) et le poids fort est multiplié par 2^{-N} et gardé comme retenue.

Les intervalles correspondant à chaque variable sont indiqués à droite de chaque instruction dans l'algorithme. Par exemple, le $[0 : N + 3]$ (deuxième

Algorithme 4.1 Algorithme de multiplication en $O(n^2)$. À droite sont indiqués des intervalles contenant les valeurs des variables correspondantes ; ils servent à vérifier qu'il n'y aura jamais de dépassement de capacité (concernant la précision). La notation $[E : M_1 \mid M_2 \mid \ldots \mid M_k]$ représente l'intervalle $2^{EN/2} \sum \left[-2^{M_i}, 2^{M_i}\right]$.

$C_0 = 3 \times 2^{51}$;
$C_1 = C_0 \times 2^{N/2}$;
$C_2 = C_1 \times 2^{N/2}$;
$C_S = 2^{-N}$;
pour i **allant de** 0 **à** $n-1$
    $x_0 = x[i];\ x_1 = $ arrondi(x_0, C_1);
    $x'[2i] = x_0;\ x'[2i+1] = x_1$;
pour j **allant de** 0 **à** $n-1$
    $z_1 = 0$;
    $y_0 = y[j];\ y_1 = $ arrondi$(y_0, C_1);\ y_0 = y_0 - y_1$;
    pour i **allant de** 0 **à** $n-1$

$x_0 = x'[2i]$;	x_0, y_0 :	$[0 : N/2]$
$x_1 = x'[2i+1]$;	x_1, y_1 :	$[1 : N/2]$
$z_0 = x_0 y_1 + x_1 y_0$;	z_0 :	$[1 : N+1]$
$z_r = $ arrondi(z_0, C_2);	z_r :	$[2 : N/2+1]$
$z_0 = z_1 + (z_0 - z_r) + x_0 y_0 + z[i+j]$;	z_0 :	$[0 : N+3]$
$z_1 = x_1 y_1 + z_r$;	z_1 :	$[2 : N \mid N/2+1]$
$z_r = $ arrondi(z_0, C_2);	z_r :	$[2 : 3]$
$z[i+j] = z_0 - z_r$;	$z[i+j]$:	$[0 : N]$
$z_1 = (z_1 + z_r)\, C_S$;	z_1 :	$[0 : N \mid N/2+2]$

    $[i = n]$
    tant que $z_1 \neq 0$
        $z_0 = z[i+j] + z_1$;
        $z_1 = $ arrondi(z_0, C_2);
        $z[i+j] = z_0 - z_1$;
        incrémenter i;

valeur de z_0) provient de :

$$\begin{cases} z_1: & [-2^N - 2^{N/2+2}, 2^N + 2^{N/2+2}] \\ z_0 - z_r: & 2^{N/2} \times [-2^{N/2}, 2^{N/2}] \subset [-2^N, 2^N] \\ x_0 y_0: & [-2^N, 2^N] \\ z[i+j]: & [-2^N, 2^N] \end{cases}$$

dont la somme appartient à un intervalle de la forme $[-X, X]$ avec

$$X = (2^N + 2^{N/2+2}) + 2^N + 2^N + 2^N = 4 \times (2^N + 2^{N/2}) \leqslant 8 \times 2^N = 2^{N+3}.$$

4.4.3.2 Algorithme de Karatsuba (méthode *2-way*)

En 1963, Karatsuba et Ofman [26] ont remarqué qu'il suffit, grâce à l'égalité suivante appliquée à $x = 2^{Nw}$, de 3 produits de nombres de longueur w et quelques additions pour effectuer un produit de nombres A et B de longueur $2w$:

$$(A_1x+A_0)(B_1x+B_0) = A_1B_1x^2+(A_1B_1+A_0B_0-(A_1-A_0)(B_1-B_0))x+A_0B_0.$$

On obtient ainsi un coût asymptotique en $O(n^{\log 3/\log 2}) \approx O(n^{1.585})$ en appliquant récursivement la méthode ci-dessus, et on accélère l'implantation en utilisant dans la récursion la méthode en $O(n^2)$ quand la taille des nombres est suffisamment petite. Le seuil en-dessous duquel on utilise la méthode en $O(n^2)$ se trouve par essais successifs ; il dépend de la machine et de l'implémentation elle-même (en particulier, du compilateur utilisé). Les différentes valeurs trouvées pour notre implémentation sont données en section 4.4.3.5.

4.4.3.3 Généralisation (méthode *k-way*)

L'algorithme de Karatsuba peut être généralisé en découpant les entiers en p blocs au lieu de 2. Pour alléger les notations, nous considérerons le cas du carré ; pour la multiplication, il suffit de remplacer dans les expressions $(\sum k_{ij}A_j)^2$ par $(\sum k_{ij}A_j)(\sum k_{ij}B_j)$, où les k_{ij} sont des constantes (i.e. ne dépendant ni de A, ni de B).

Ainsi on considère l'équation

$$(A_{p-1}x^{p-1} + ... + A_1x + A_0)^2 = C_{2p-2}x^{2p-2} + ... + C_1x + C_0$$

et on détermine les C_i en choisissant $2p-1$ valeurs de x (rationnels ou/et ∞). Zuras suggère de prendre dans l'ordre : 1, ∞, 0, 2, $\frac{1}{2}$, -2, $-\frac{1}{2}$, 3, $\frac{1}{3}$, -3, $-\frac{1}{3}$, $\frac{3}{2}$, $\frac{2}{3}$... On privilégie la symétrie multiplicative par rapport la symétrie additive. La résolution du système linéaire (effectuée une fois pour toutes lors de l'implémentation de l'algorithme) mène à un produit d'une matrice constante à coefficients rationnels par le vecteur des $(\sum k_{ij}A_j)^2$. Un exemple pour $p = 3$, correspondant à notre implémentation, sera donné en section 4.4.3.5.

On montre que la complexité est $O(n^{\log(2p-1)/\log p})$.

4.4.3.4 FFT

Il existe plusieurs algorithmes utilisant la FFT. Certains sont décrits dans [54] ; ils ont des complexités de la forme $O(n^{k/(k-2)})$, mais ils semblent peu efficaces en pratique. Le meilleur (asymptotiquement) algorithme connu est l'algorithme de Schönhage–Strassen [8, 2], qui consiste à décomposer un nombre de n chiffres en $\sqrt{n}$ blocs de $\sqrt{n}$ chiffres ; sa complexité est $O(n \log n \log\log n)$. Mais il est peu souvent implémenté. Les algorithmes les plus rapides pour les très grands nombres restent ceux utilisant la FFT, mais sont limités à des nombres d'une certaine taille, dépendant de l'implémentation. Une des solutions pour augmenter cette taille est de faire plusieurs FFT et d'utiliser le théorème des restes chinois ; c'est le choix de D. Bailey pour le calcul de π [3], ainsi que celui de M. Tommila pour sa bibliothèque Apfloat.

4.4.3.5 Implémentation de la méthode *3-way*

Appliquons le principe décrit en section 4.4.3.3 pour la multiplication avec $p = 3$. On choisit $x \in \{\infty, 2, 1, 1/2, 0\}$. On a alors :

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} P_4 \\ P_3 \\ P_2 \\ P_1 \\ P_0 \end{bmatrix}$$

avec

$$\begin{bmatrix} P_4 \\ P_3 \\ P_2 \\ P_1 \\ P_0 \end{bmatrix} = \begin{bmatrix} A_2B_2 \\ (4A_2 + 2A_1 + A_0)(4B_2 + 2B_1 + B_0) \\ (A_2 + A_1 + A_0)(B_2 + B_1 + B_0) \\ (A_2 + 2A_1 + 4A_0)(B_2 + 2B_1 + 4B_0) \\ A_0B_0 \end{bmatrix}$$

et la résolution de ce système donne :

$$\begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ -21 & 2 & -12 & 1 & -6 \\ 7 & -1 & 10 & -1 & 7 \\ -6 & 1 & -12 & 2 & -21 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_4 \\ P_3 \\ P_2 \\ P_1 \\ P_0 \end{bmatrix}$$

ce qui donne un produit par une matrice constante entière et trois divisions. Dans l'implémentation choisie, le produit de matrice s'effectue en calculant simultanément (donc sans accès mémoire) les valeurs intermédiaires suivantes :

$$S_0 = 7P_0 \quad \text{et} \quad S_4 = 7P_4,$$

$$T_0 = S_0 - P_1 + 3P_2 \quad \text{et} \quad T_4 = S_4 - P_3 + 3P_2,$$

$$X_2 = T_0 + T_4 + 4P_2, \quad X_1 = P_4 - T_4 - 3T_0 - P_1 \quad \text{et} \quad X_3 = P_0 - T_0 - 3T_4 - P_3$$

en tenant compte du fait que nous avons seulement 3 bits de retenues. Les trois divisions (par des constantes), peuvent, toujours en s'appuyant sur la norme IEEE-754, s'effectuer sans utiliser de division flottante pour être plus rapides.

Les tableaux 4.1 et 4.2 donnent les plates-formes et compilateurs[1] utilisés, puis les seuils « optimaux » `MAXN` et `MAXK` obtenus en pratique par essais successifs. `MAXN` est le nombre maximal de mots pour lequel l'algorithme en $O(n^2)$ est utilisé, et `MAXK` est le nombre maximal de mots pour lequel l'algorithme de Karatsuba (2-way) est utilisé.

id	machine	processeur	fréquence	système	compilateur
S4	Sun Sparc-4	MicroSPARC II	110 MHz	Solaris 2.6	gcc 2.8.0
U1	Sun Ultra-1	UltraSPARC	143 MHz	Solaris 2.5.1	gcc 2.7.2.2
U5	Sun Ultra-5	UltraSPARC	333 MHz	Solaris 2.6	gcc 2.8.0
PP	Intel PC	Pentium Pro	150 MHz	Solaris 2.6	gcc 2.8.1
P2	PC (DELL)	Pentium II	300 MHz	Linux 2.0.33	gcc 2.7.2.3

TAB. 4.1: *Plates-formes et compilateurs utilisés ; gcc est lancé avec l'option* `-O`.

id	base	`MAXN`	`MAXK`
S4	2^{50}	15	95
U1	2^{50}	19	62
U5	2^{50}	19	70
PP	2^{50}	15	86
P2	2^{50}	15	89

TAB. 4.2: *Seuils* `MAXN` *et* `MAXK` *obtenus par essais successifs.*

Pour faciliter la recherche des seuils « optimaux », on peut tester la routine avec des nombres d'une taille de la forme $n \times k^u$, où $k = 2$ pour la recherche de `MAXN`, $k = 3$ pour la recherche de `MAXK`, u est un entier choisi de manière à avoir des nombres assez grands et n est le seuil à tester. On peut aussi prendre une valeur plus petite, en particulier $u = 0$. Une telle valeur de u est même préférable pour avoir des résultats plus précis. La multiplication peut alors être effectuée plusieurs fois, en boucle, de manière à obtenir une mesure du temps assez précise. Par exemple, pour $k = 2$ (recherche du `MAXN` optimal) et $n = 17$, on teste la routine avec `MAXN` égal à 16, puis `MAXN` égal à 17, et on en déduit si le seuil optimal est inférieur ou égal à 16 ou bien supérieur ou égal à 17. Mais il faut au moins tester tout un intervalle de seuils, car d'une part les mesures se font sur des nombres particuliers et ne sont pas très précises (certaines inégalités peuvent changer de sens en pratique), d'autre part il n'y a pas forcément de seuil optimal strict, en particulier pour `MAXK` ; en effet, les valeurs de n pour lesquelles il vaut mieux découper les nombres de taille n en 3 ne forment pas toujours un intervalle $[n_0, \infty[$. Ce problème est certainement dû à la discrétisation : n n'est pas toujours divisible par 2 ou 3. Cela peut se vérifier

[1]Le compilateur est lancé avec l'option `-O` (optimisations simples).

en analysant les différences de temps dans le tableau 4.3 : pour le découpage en 3, il y a des sauts importants quand on passe de $3p$ à $3p+1$, de même pour le découpage en 2, quand on passe de $2p$ à $2p+1$ (et encore plus quand on passe de $4p$ à $4p+1$). Ceci dit les différences de temps de calcul entre le découpage en 2 et le découpage en 3 à cause de ce problème sont relativement faibles (au plus quelques pour cent). Par exemple, lors de la recherche de `MAXK` sur une station Sun Ultra-5, il a été remarqué que pour $n = 69$, il valait mieux découper en 3, mais que pour $n = 70$, il valait mieux découper en 2, comme l'indique le tableau 4.3. Nous n'avons pas tenu compte de ce problème dans notre algorithme, c'est-à-dire nous considérons que le choix du découpage se fait suivant que la taille du nombre est inférieure ou supérieure à un seuil fixé ; la raison de ce choix est que la perte de temps est relativement négligeable[2].

n	3-way	2-way	meilleur découpage
67	200.68 s	196.49 s	2-way
68	200.52 s	198.57 s	2-way
69	203.67 s	207.95 s	3-way
70	215.24 s	212.07 s	2-way
71	215.48 s	216.82 s	3-way
72	217.35 s	219.08 s	3-way
73	227.12 s	228.83 s	3-way

TAB. 4.3: *Recherche du seuil* `MAXK` *sur une station Sun Ultra-5. Des nombres aléatoires de $n = 67$ à 73 chiffres (en base 2^{50}) sont testés un par un dans deux cas : un découpage en 3 est effectué (le seuil est un peu inférieur à n), et un découpage en 2 est effectué (le seuil est supérieur ou égal à n). Dans chaque cas, 5 mesures sont effectuées (sur des nombres différents). Il s'agit des temps de calcul sur une boucle de* 500 000 *tests.*

Les seuils trouvés sont relativement petits. Il peut être intéressant d'implémenter l'algorithme 4-way. Mais il y a d'autres solutions, comme les algorithmes à base de FFT.

Nous avons comparé les implémentations des algorithmes suivants sur la multiplication de nombres de 20 000 mots (i.e. un million de bits) :

– algorithme classique en $O(n^2)$;
– algorithme 2-way (qui utilise l'algorithme classique) ;
– algorithme 3-way (qui utilise l'algorithme 2-way).

Les temps d'exécution en secondes sont donnés dans le tableau 4.4.

Nous avons également testé l'implémentation de l'algorithme 3-way sur des nombres de différentes tailles. Les temps d'exécution en secondes sont donnés dans le tableau 4.5.

[2]Notons que pour des nombres de grande taille, le rapport des temps de calcul entre les deux choix possibles est encore beaucoup plus proche de 1.

id	$O(n^2)$	2-way	3-way
S4	419	25.1	14.5
U1	150	8.6	4.6
U5	62	3.6	2.0
PP	221	10.8	6.2
P2	91	4.0	2.4

TAB. 4.4: *Comparaison des algorithmes en $O(n^2)$, 2-way et 3-way (temps en secondes).*

#Kmots	1	2	4	10	20	40	100	200
#Kbits	50	100	200	500	1000	2000	5000	10000
U1	0.056	0.16	0.44	1.70	4.6	13.0	50	136
U5	0.024	0.07	0.19	0.74	2.0	5.6	22	59
PP	0.072	0.21	0.57	2.16	6.2	17.1	64	178
P2	0.028	0.08	0.22	0.85	2.4	6.6	25	70

TAB. 4.5: *Temps de calcul (en secondes) pour l'algorithme 3-way sur des nombres de différentes tailles.*

Nous avons comparé notre implémentation avec GMP (qui utilise, dans les versions testées[3], l'algorithme de Karatsuba) et Apfloat (qui implémente la FFT). Les temps de calcul pour multiplier deux entiers aléatoires d'un million de bits sont donnés dans le tableau 4.6 ci-dessous.

Plate-forme	PC/Linux	UltraSparc/Solaris
Fréquence	300 MHz	143 MHz
GMP 2.0.2	3.7 s	12.8 s
GMP-980312	2.7 s	11.0 s
Apfloat 1.40	1.4 s	10.0 s
Threeway	2.4 s	4.6 s

TAB. 4.6: *Temps de calcul pour multiplier deux entiers aléatoires d'un million de bits. Pour Apfloat, plusieurs types élémentaires sont possibles, et le plus rapide (entiers de 32 bits) a été choisi.*

Nous remarquons, que bien qu'asymptotiquement plus rapide, Apfloat est plus lent que notre implémentation Threeway sur UltraSparc, pour des entiers d'un million de bits. Sur cette machine, Apfloat devient plus rapide pour un seuil compris entre trois et quatre millions de bits. Cela est dû en partie au fait qu'Apfloat n'est pas optimisé pour les processeurs qui sont bien meilleurs en

[3] Il s'agit de la dernière version publique 2.0.2 (sortie en juin 1996) et d'une pré-version datant de mars 1998.

calcul flottant qu'en calcul entier. Mais cela montre aussi que l'implémentation et les tailles de nombres considérées sont très importantes.

4.4.4 Fonctions algébriques

On suppose l'algorithme de multiplication donné. Les fonctions algébriques peuvent se calculer à l'aide de l'itération de Newton, dont nous rappelons brièvement le principe. Sous certaines conditions (vérifiées quand les itérations plus loin seront appliquées), on peut calculer une solution ζ de l'équation $f(x) = 0$ à l'aide de l'itération suivante :

$$x_{i+1} = x_i - \frac{h_i f(x_i)}{f(x_i + h_i) - f(x_i)} \quad \text{(méthode des cordes)}$$

ou

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad \text{(méthode de Newton)}.$$

En supposant que la racine ζ de f est simple et que x_0 est dans un voisinage de ζ suffisamment petit, on peut montrer que si $h_i = 2^{-n/2}$, $x_i - \zeta = O(2^{-n/2})$ et le calcul est effectué à la précision n, alors $x_{i+1} - \zeta = O(2^{-n})$, i.e. la convergence est quadratique (en supposant que $\zeta \neq 0$ car nous parlons de précision relative).

On utilise le fait que cette itération est « autocorrective » (i.e. x_i n'a besoin d'être connu qu'à $O(2^{-n/2})$ près) et que la convergence est quadratique : si x_i est connu à la précision $n/2$ (ainsi que ζ), alors le calcul de x_{i+1} est effectué à la précision n, i.e. la précision est doublée à chaque itération. Ainsi, la complexité en temps est $O(M(n))$, où $M(n)$ est la complexité de la multiplication[4] [9].

Le calcul des fonctions transcendantes utilise des divisions et des racines carrées. L'inverse d'un nombre c peut être calculé en appliquant l'itération $x_{i+1} = x_i(2 - cx_i)$. À partir de l'inverse et de la multiplication, nous obtenons la division. Pour la racine carrée d'un nombre c positif, le plus rapide est de calculer $1/\sqrt{c}$ en résolvant $cx^2 = 1$ par l'itération $x_{i+1} = x_i(3 - cx_i^2)/2$, puis de multiplier le résultat par c.

Il existe d'autres algorithmes de division [11] et de racine carrée [52], non forcément asymptotiquement optimaux mais rapides en pratique.

4.4.5 Fonctions transcendantes

4.4.5.1 Considérations générales

Les fonctions transcendantes sont calculées à l'aide de résultats de la théorie des intégrales elliptiques [9], encore à l'aide d'algorithmes itératifs avec

[4] $O(n \log n \log \log n)$ si on utilise l'algorithme de Schönhage–Strassen.

convergence quadratique, mais ceux-ci ne sont plus autocorrectifs. Par conséquent, on ne peut plus faire les calculs intermédiaires avec une précision réduite. Il faut même calculer avec une précision légèrement supérieure à cause des erreurs d'arrondi. La complexité en temps est alors $O(M(n) \log n)$.

Par exemple, l'algorithme 4.2 ci-dessous permet de calculer π (la valeur de π est utilisée dans les algorithmes de calcul des fonctions élémentaires).

Algorithme 4.2 (Brent-Salamin). Calcul de π.

$A = 1;\ B = 2^{-1/2};\ T = 1/4;\ X = 1;$
tant que $(A - B > 2^{-n})$
  $Y = A;\ A = (A + B)/2;\ B = (YB)^{1/2};$
  $T = T - X(A - Y)^2;\ X = 2X;$
Résultat : $(A + B)^2/(4T)$

Les algorithmes pour évaluer les fonctions transcendantes sont décrits dans [10, 9, 4]. Les temps de calcul sont, en pratique, jusqu'à environ 30 fois plus grands que celui d'une multiplication, soit au plus quelques minutes pour des nombres d'un million de bits.

4.4.5.2 Exemple pour le logarithme

Appelons $A(x)$ la limite de la suite arithmético-géométrique

$$\begin{array}{lcl} a_0 & = & 1 \\ b_0 & = & x \\ a_{n+1} & = & \frac{1}{2}(a_n + b_n) \\ b_{n+1} & = & \sqrt{a_n b_n} \end{array}$$

et F la fonction elliptique définie par

$$F(x) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - (1 - x^2)\sin^2\theta}}.$$

On a la relation suivante :

$$A(x) = \frac{\pi}{2F(x)}.$$

L'algorithme de Salamin pour le calcul du logarithme utilise le premier terme du développement asymptotique $(s \to \infty)$:

$$\begin{aligned} F(4/s) &= \log(s) + \frac{4\log(s) - 4}{s^2} + \frac{36\log(s) - 42}{s^4} \\ &\quad + \frac{1200\log(s) - 1480}{3s^6} + O\left(\frac{\log(s)}{s^8}\right) \end{aligned}$$

Pour calculer $\log(t)$, on calcule le logarithme d'un terme de la forme $s = 2^m t$, tel que s est suffisamment grand pour que l'erreur de méthode soit suffisamment petite. On obtient alors la relation utilisée par l'algorithme de Salamin :

$$\log(t) \approx \frac{\pi}{2A(4/s)} - m \times \log(2).$$

On est donc ramené au calcul de $A(4/s)$.

Notons que plus x est petit, moins $A(x)$ converge rapidement. Il peut donc être intéressant d'utiliser les deux ou trois premiers termes du développement asymptotique de $F(x)$, et choisir une valeur de x plus grande.

4.5 Utilisation des résultats des tests exhaustifs

4.5.1 Méthodes générales

Les tests exhaustifs peuvent non seulement servir à prouver l'algorithme de calcul d'une fonction élémentaire avec arrondi exact, en donnant une borne explicite sur la précision maximale nécessaire pour les calculs intermédiaires, mais ils peuvent aussi permettre de réduire cette précision maximale en tabulant les pires cas. Les calculs seront ainsi plus rapides pour les cas où une grande précision intermédiaire est demandée, même si cela n'accélère pas le cas général (où une faible précision est suffisante), déjà très rapide grâce à la stratégie de Ziv (cf section 4.2).

Le calcul pourra s'effectuer de la façon suivante :

– la fonction est calculée à une certaine précision par un algorithme classique (à l'aide de la stratégie de Ziv) ;

– si la précision n'est pas suffisante, une lecture dans une table est effectuée pour savoir comment il faut arrondir.

Si la fonction est implémentée entièrement en matériel, ces deux étapes peuvent s'effectuer en parallèle (la lecture dans la table est alors effectuée même si la précision est finalement suffisante, mais cela n'a aucune importance).

Notons que les tables dépendent de l'implémentation, à moins d'utiliser des tables plus grandes que nécessaires. En effet, tout dépend des distances relatives entre la valeur calculée, la valeur exacte et la frontière entre deux arrondis. Par exemple, considérons le nombre $2549/150 = 16.99333...$, que nous cherchons à arrondir à l'entier supérieur (arrondi sur deux chiffres vers $+\infty$), et supposons qu'il soit calculé avec une erreur maximale de 10^{-2}. Si la valeur calculée par l'implémentation est 16.991, alors, du point de vue du programme, la valeur exacte est dans l'intervalle $[16.981, 17.001]$ et on ne sait pas s'il faut arrondir à 17 ou à 18. Mais si la valeur calculée est 16.987, alors la valeur exacte est dans l'intervalle $[16.977, 16.997]$, et on peut en déduire qu'il faut arrondir à 17.

Maintenant, expliquons comment tabuler efficacement. Les nombres machine étant très nombreux, il faut utiliser un code de hachage pour se restreindre à un nombre raisonnable d'entrées dans la table. Par exemple, les premiers ou les derniers bits de l'argument peuvent être utilisés comme clé. Puisque les arguments tabulés sont connus lors de l'implémentation, différentes méthodes de hachage peuvent être comparées et celle donnant le meilleur compromis entre temps d'accès et taille de la table peut être prise.

Nous venons d'expliquer comment effectuer la lecture dans la table à partir de l'argument x, donnons maintenant deux méthodes permettant d'obtenir le résultat $f(x)$ arrondi exactement. Nous pouvons nous permettre de ne pas stocker le résultat en entier, qui prendrait beaucoup de place. En effet, grâce au calcul approché déjà effectué, nous avons des informations que nous pouvons utiliser. Par exemple, l'une des deux méthodes suivantes peut être utilisée :

- Nous connaissons une approximation très précise de $f(x)$, et nous avons le choix entre deux arrondis possibles. La valeur dans la table peut alors être représentée par un seul bit, indiquant lequel des deux arrondis il faut renvoyer.
- Pour tout argument x, la valeur $f(x)$ est évaluée avec une certaine précision (plus grande que la précision finale), ce qui donne un nombre y. Au moment de l'implémentation, l'algorithme permet de déterminer un majorant global ε de l'erreur finale sur $f(x)$: $|f(x) - y| < \varepsilon$. À l'exécution, les valeurs de ε et y indiquent si le TMD se produit ou non. Mais nous pouvons tenir compte du fait qu'en général l'erreur réelle $|f(x) - y|$ est beaucoup plus petite que le majorant ε : en effet, d'une part les majorations d'erreur sont souvent assez pessimistes, et d'autre part les majorations sont globales, pour x appartenant à tout un intervalle. Par conséquent, même si le TMD se produit, l'arrondi de y est généralement exact.

  Mais suivant les implémentations, cela ne sera pas forcément utilisé. En effet, la valeur de y peut par exemple être représentée par la somme $y_h + y_\ell$ de deux nombres machine avec y_ℓ petit devant y_h, et nous pouvons supposer que y_h sera l'arrondi de $f(x)$ choisi *a priori* (car obtenu immédiatement). Mais y_h n'est pas forcément l'arrondi exact de y, et par conséquent la remarque sur l'ordre de grandeur de $|f(x) - y|$ par rapport à ε ne s'applique pas forcément à y_h ; ce problème se produira dans notre implémentation de 2^x pour l'arrondi au plus près (section 4.5.2). On peut toujours corriger les valeurs de y_h et y_ℓ pour que y_h soit l'arrondi exact de y, mais cela n'est pas forcément avantageux. En pratique, il faudrait comparer les deux solutions, et choisir le meilleur compromis.

  Supposons maintenant que l'on ait calculé y_h tel que y_h soit en général l'arrondi exact de $f(x)$. Nous pouvons alors construire une table contenant *uniquement* les valeurs de x telles que la valeur de y_h ne soit pas l'arrondi exact. À l'exécution, la valeur de y_h sera alors corrigée si et seulement si x est dans la table, et l'arrondi exact de $f(x)$ sera ainsi obtenu.

Il n'est pas possible de dire *a priori* laquelle de ces deux méthodes est la meilleure : cela dépend beaucoup de l'implémentation, et la comparaison doit s'effectuer en pratique, pour chaque fonction.

4.5.2 Exemple : implémentation de la fonction 2^x

Pour montrer que les méthodes décrites dans les sections précédentes peuvent être utilisées en pratique, nous avons implémenté une fonction élémentaire en double précision. Nous avons choisi la fonction 2^x pour les deux raisons suivantes (qui sont liées) :

– la réduction d'argument est très simple : nous nous ramenons au calcul avec x dans l'intervalle $[-0.5, +0.5]$ par simple soustraction en double précision, sans erreur d'arrondi, et la reconstruction du résultat est aussi très simple et s'effectue de manière exacte (multiplication par une puissance de 2) ;

– la recherche des pires cas pour cette fonction demande moins de calculs que pour les autres fonctions élémentaires (grâce à la simplicité de la réduction d'argument).

La routine de calcul de 2^x est écrite en C ISO, et utilise la bibliothèque standard. Mais le C ISO ne supporte pas en standard la norme IEEE-754, en particulier les modes d'arrondi et la précision des calculs (double précision ou précision étendue). Puisque la routine sera testée sous le système Solaris, nous utilisons donc aussi la bibliothèque sunmath pour changer le mode d'arrondi et fixer la précision des calculs à la double précision. Nous supposons que le compilateur génère du code respectant la norme IEEE-754, par exemple, en considérant que les opérations mathématiquement associatives (addition, multiplication) ne le sont plus et en respectant strictement les changements de mode d'arrondi. Nous supposons aussi que les constantes sont calculées à la compilation en arrondi au plus près.

Nous n'avons pas cherché à faire quelque chose de très rapide, mais à avoir rapidement une implémentation correcte, afin de montrer la faisabilité de notre approche. Nous n'avons donc pas utilisé la stratégie de Ziv, qui aurait demandé du code supplémentaire. Il ne faut donc pas s'attendre à des temps de calcul très rapides par rapport aux fonctions existantes dans les bibliothèques de calcul en double précision. Les tables de pires cas doivent avoir une taille raisonnable ; la précision des calculs intermédiaires ne doit donc pas être trop petite. Nous avons visé une précision de l'ordre de 2^{-100} (permettant notamment de représenter les nombres dans cette précision sous forme d'une somme de deux flottants en double précision).

Nous avons choisi d'implémenter la fonction 2^x à l'aide d'un algorithme à base de tables pour nous ramener dans un très petit intervalle, et de l'évaluation d'un polynôme de petit degré. Après avoir décrit la réduction d'argument (section 4.5.2.2), nous présenterons dans la section 4.5.2.3 les formules utilisées ainsi que la construction des tables en pratique. L'algorithme sera précisément décrit dans la section 4.5.2.4, puis prouvé (nous donnerons en particulier une majoration de l'erreur finale) dans la section 4.5.2.5. Les petits arguments ($|x| < 2^{-40}$, par exemple) posent des problèmes pour l'arrondi exact[5]. Ils sont donc traités à l'aide d'un autre algorithme à base de tables. Cet algorithme et la construction des tables sont décrits dans la section 4.5.2.6. Enfin, nous montrerons comment arrondir exactement pour les autres arguments (section 4.5.2.7). Mais d'abord, donnons quelques notations.

4.5.2.1 Notations

Soit $x \in \mathbb{R}$ et $k \in \mathbb{Z}$. Nous notons respectivement $\lfloor x \rfloor_k$, $\lceil x \rceil_k$, $[x]_k$, $\lfloor x \rceil_k$, les arrondis de x dans $2^k\,\mathbb{Z}$ vers $-\infty$, vers $+\infty$, vers 0, et au plus près. Si $k = 0$ (arrondi dans $\mathbb{Z}$), nous ne mettrons pas le 0 en indice.

Soit x un *nombre machine*. Nous définissons les fonctions suivantes :

– $\mathrm{MSB}(x)$ dénote l'exposant du bit de poids fort de x (qui est aussi l'exposant de la représentation flottante de x), ou $-\infty$ si $x = 0$.

– $\mathrm{LSB}(x)$ dénote l'exposant du bit de poids faible non nul de x, ou $+\infty$ si $x = 0$.

– $\mathrm{ULP}(x)$ dénote l'exposant du bit de poids faible de x (dernier bit représenté), ou $-\infty$ si $x = 0$. Puisque la mantisse a une taille de 53 bits, on a alors : $\mathrm{ULP}(x) = \mathrm{MSB}(x) - 52$.

Par exemple, si $x = 1.001101_2 \times 2^{-28}$, alors $\mathrm{MSB}(x) = -28$, $\mathrm{LSB}(x) = -34$ et $\mathrm{ULP}(x) = -80$.

Les valeurs de $\mathrm{MSB}(x)$ et de $\mathrm{LSB}(x)$ pour $x = 0$ sont les extensions à 0 les plus naturelles, permettant de conserver les diverses propriétés utilisées plus loin. Cela peut aussi s'exprimer plus formellement, comme suit. Soit x un nombre machine quelconque (éventuellement nul). Un tel nombre s'écrit, en valeur absolue, de manière unique :

$$|x| = \sum_{-\infty}^{+\infty} a_i 2^i$$

où $(a_i)_{i \in \mathbb{Z}}$ est une suite bi-infinie presque nulle sur l'ensemble $\{0, 1\}$. Soit $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$. On a alors :

$$\mathrm{MSB}(x) = \max\nolimits_{\overline{\mathbb{Z}}} \{i \in \mathbb{Z} :\ a_i \neq 0\}$$

et

$$\mathrm{LSB}(x) = \min\nolimits_{\overline{\mathbb{Z}}} \{i \in \mathbb{Z} :\ a_i \neq 0\}.$$

[5] Il faudrait faire des modifications des programmes de tests exhaustifs pour traiter les pires cas consécutifs, ou il faudrait un algorithme calculant plus précisément.

Pour faire des calculs exacts, nous aurons besoin de découper des nombres machine en deux, en utilisant la même méthode que celle donnée dans les sections 4.4.2 et 4.4.3.1. Notons $\mathrm{ROUND}(x, n)$ l'arrondi de x dans $2^{-n}\,\mathbb{Z}$ suivant le mode d'arrondi actif. Le découpage en deux s'effectue de la façon suivante : $x_h = \mathrm{ROUND}(x, n)$ contient la partie de poids fort, $x_\ell = x - x_h$ contient la partie de poids faible, et on a : $\mathrm{LSB}(x_h) > \mathrm{MSB}(x_\ell)$.

4.5.2.2 Réduction d'argument

Nous commençons par tester si $x \geqslant 1024$ ou si $x < -1075$; si c'est le cas, nous renvoyons respectivement un *overflow* (et la variable `errno` est positionnée à `ERANGE`) ou 0.

Sinon, nous positionnons la précision des calculs internes au processeur à la double précision[6] et nous changeons le mode d'arrondi actif pour avoir un arrondi au plus proche (qui sera utilisé ci-dessous pour la fonction `rint`).

La réduction d'argument est basée sur la formule

$$2^x = 2^n \times 2^{x-n},$$

où nous choisissons n entier tel que $|x - n| \leqslant \frac{1}{2}$. Le calcul de n se fait simplement en arrondissant x à l'entier le plus proche, à l'aide de la fonction `rint`. La soustraction $x - n$ se fait en double précision, et le résultat mathématique est représentable, donc le résultat renvoyé est exact. Enfin, la multiplication par 2^n se fait aussi simplement à l'aide de la fonction `ldexp`, et le résultat de cette multiplication est exact.

Nous sommes donc ramenés au calcul de 2^x avec $|x| \leqslant \frac{1}{2}$. Dans les sections suivantes, nous supposerons alors que x vérifie cette inégalité.

4.5.2.3 Algorithme à base de tables

Pour calculer 2^x avec $|x| \leqslant \frac{1}{2}$, nous utiliserons l'approximation suivante :

$$2^x \approx 2^u \times 2^v \times \left(1 + z + \frac{z^2}{2} + \frac{z^3}{6}\right)$$

où u est une approximation de x (qui sera obtenue à partir d'une table), v est une approximation de $x - u$ (qui sera également obtenue à partir d'une table), et $z = (x - u - v)\log(2)$. Les tables seront construites de manière à ce que $x - u$ soit de l'ordre de 2^{-13} au maximum, et à ce que $x - u - v$ et z soient de l'ordre de 2^{-25} au maximum. Puisque nous visons une précision de l'ordre de 2^{-100}, le développement de Taylor de $\exp(z)$ à l'ordre 3 est suffisant.

[6] Cela est nécessaire si le programme tourne sur un processeur compatible Intel i386, par exemple.

Nous voulons que les multiplications par 2^u et par 2^v puissent se faire rapidement (et à la précision voulue, évidemment). Nous choisirons donc de tabuler des valeurs de u et v (ainsi que les valeurs correspondantes 2^u et 2^v) de telle sorte que 2^u et 2^v s'écrivent sur peu de bits ; u et v auront donc des valeurs réelles et seront arrondies pour pouvoir être représentées. Cependant, nous avons une autre contrainte : la valeur maximale de $|x - u|$ doit être proche de la valeur optimale, correspondant au cas où les valeurs tabulées de u sont régulièrement espacées ; idem pour $|x - u - v|$. Pour respecter cette contrainte, le nombre choisi de bits de 2^u et celui de 2^v ne doivent pas être trop petits. Ces choix seront fixés en essayant différentes valeurs.

Nous allons maintenant décrire le programme de construction des tables donnant les valeurs exactes de u et v, et les valeurs approchées de 2^u et 2^v. Il s'agit d'un programme assez générique, de manière à pouvoir essayer différents choix possibles : précision de 2^u, de 2^v, précision finale, taille des tables. Ce programme est écrit en Perl, et utilise Maple avec de l'arithmétique d'intervalles (*package* intpak).

Le programme prend en entrée 7 arguments :

- 2 nombres a et b écrits en base 2 et formant un intervalle $[a, b]$,
- 2 nombres c et d écrits en base 2 et formant un intervalle $[c, d]$ inclus dans $[a, b]$,
- 3 entiers positifs m, n et q.

La signification de ces arguments est donnée ci-dessous.

Nous cherchons à tabuler 2^x, où $x \in [a, b]$. Le calcul de 2^x se fera à l'aide de la formule

$$2^x = 2^{\lambda_i} \times 2^{x-\lambda_i},$$

où λ_i est une valeur tabulée (proche de x).

Les nombres c et d sont des nombres « réguliers » respectivement très proches de a et b. Leur choix est arbitraire et doit permettre de simplifier l'implémentation. Nous donnerons des explications plus loin, une fois que nous aurons décrit la tabulation.

L'intervalle $[c, d]$ est découpé en n sous-intervalles de même longueur

$$s = \frac{d - c}{n}.$$

Les extrémités des premier et dernier sous-intervalles sont modifiées pour couvrir l'intervalle $[a, b]$ en entier. Plus précisément, pour $0 \leqslant i < n$, les extrémités des intervalles $[a_i, b_i]$ sont définies par : $a_i = c + is$ et $b_i = a_i + s$, sauf $a_0 = \min(a, b_0)$ au lieu de c et $b_{n-1} = \max(b, a_{n-1})$ au lieu de d. Nous définissons x_i comme étant le milieu du sous-intervalle $[a_i, b_i]$:

$$x_i = \frac{a_i + b_i}{2}.$$

Pour chaque sous-intervalle i, 2^{x_i} est calculé en arithmétique d'intervalles, puis arrondi dans $2^{-q}\,\mathbb{Z}$; cela donne la valeur 2^{λ_i}, qui sera tabulée. On en prend le logarithme en base 2 (qui est le nombre que l'on note λ_i), et on arrondit le résultat au plus près dans $2^{-m}\,\mathbb{Z}$: $\hat{\lambda}_i = \lfloor \lambda_i \rceil_{-m}$, qui sera également tabulé.

En plus des valeurs à tabuler, les deux nombres suivants sont calculés et renvoyés sur la sortie standard (pour être stockés dans un fichier) :

$$t_{\min} = \left\lfloor \min_{0 \leqslant i < n} \{a_i - \hat{\lambda}_i\} \right\rfloor_{-m} \quad \text{et} \quad t_{\max} = \left\lceil \max_{0 \leqslant i < n} \{b_i - \hat{\lambda}_i\} \right\rceil_{-m}.$$

Dans l'implémentation du calcul de 2^x avec $x \in [a, b]$, on choisira λ_i tel que $x \in [a_i, b_i]$, et on appliquera la formule

$$2^x = 2^{\lambda_i} \times 2^{x - \lambda_i},$$

ce qui ramène le problème au calcul de 2^t, où t se trouve dans un intervalle plus petit, contenu dans $[t_{\min}, t_{\max}]$. Ces nombres $t_{\min}$ et $t_{\max}$ peuvent être repris comme valeurs de a et b si cette méthode à base de tables est appliquée une seconde fois (pour 2^t) ; c'est ce qui se fera dans notre implémentation, comme nous l'avons dit plus haut.

Parlons maintenant de la sélection du λ_i dans l'implémentation. L'index i peut se calculer par

$$i = \left\lfloor \frac{x - c}{s} \right\rfloor \quad \text{ou} \quad i = \left\lceil \frac{x - c}{s} \right\rceil$$

où i doit être compris entre 0 et $n-1$. Si i sort de ces bornes (parce que $a \leqslant x < c$ ou $d \leqslant x \leqslant b$), alors on ramène i à 0 ou $n - 1$ suivant que i est trop petit ou trop grand[7]. La soustraction peut se faire avec une erreur d'arrondi, mais dans ce cas, cette erreur devra être prise en compte pour l'implémentation ($t_{\min}$ ou $t_{\max}$ peut être modifié). Le nombre s ne dépend pas de x, donc on fera en pratique une multiplication par la constante $1/s$, plus rapide qu'une division par s. Suivant l'implémentation, il vaudra mieux choisir c, d et n de façon à ce que $1/s$ soit une puissance de 2 (cela revient à sélectionner les premiers bits de $x - c$). Sinon, il faudra prendre en compte l'erreur d'arrondi éventuelle. Concernant les erreurs d'arrondi, il suffit d'étudier ce qui se passe aux bornes des intervalles $[a_i, b_i]$.

Dans notre implémentation, nous construisons la première table (pour u) avec les paramètres suivants : $a = c = -1/2$, $b = d = 1/2$, $m = 104$, $n = 2^{12} = 4096$, $q = 16$. Ainsi, $1/s = 2^{12}$. La valeur de $t_{\min}$ est minorée par

$$-1.0001111111111000001100110011010110001111000012 \times 2^{-13}$$

[7]En pratique, cela peut se faire en gardant la valeur de i telle qu'elle a été calculée, et en agrandissant les tables de telle manière que les premières valeurs (correspondant à $i = 0$) soient égales, ainsi que les dernières valeurs (correspondant à $i = n - 1$).

et celle de $t_{\max}$ est majorée par

$$+1.000111111110001011001100000110001100000100000010_2 \times 2^{-13}.$$

La seconde table (pour v) est construite avec les paramètres suivants : les $t_{\min}$ et $t_{\max}$ de la première table pour a et b, $c = -9 \times 2^{-16}$, $d = 9 \times 2^{-16}$, $m = 104$, $n = 9 \times 2^9 = 4608$, $q = 32$. Ainsi, $1/s = 2^{24}$. La valeur de $t_{\min}$ pour cette table est minorée par

$$-1.0000000101110000011110010001100100100011000101_2 \times 2^{-25}$$

et celle de $t_{\max}$ est majorée par

$$+1.0000000101110000011110010001100100100011000101_2 \times 2^{-25}.$$

Expliquons maintenant nos choix des paramètres m, n et q.

– La valeur de m a été choisie en vue d'obtenir une précision finale de l'ordre de 2^{-100} : l'arrondi de λ_i introduit une erreur à peu près égale à l'erreur sur λ_i multipliée par la dérivée de 2^x, et cette dérivée est de l'ordre de 1. D'autre part, les nombres manipulés doivent être représentables par une somme de deux nombres machine en double précision. Nous avons donc choisi $m \approx -100$.
– La valeur de n donne la taille des tables. Celles-ci ne doivent pas être trop grosses, mais doivent permettre de suffisamment diminuer l'intervalle contenant x après l'utilisation successive des deux tables pour pouvoir calculer 2^x rapidement. Nous choisissons ainsi l'ordre de grandeur de n. Ensuite, la valeur de n est ajustée pour éviter des erreurs d'arrondi supplémentaires, comme indiqué plus haut.
– La valeur de q est choisie en fonction de celle de n, de façon à ce que les λ_i soient assez régulièrement espacés ; q doit donc être assez grand. Mais d'autre part, q ne doit pas être trop grand pour ne pas avoir à effectuer de multiplications trop compliquées.

Les nombres seront manipulés sous forme de sommes de deux ou trois nombres machine. Les tables ci-dessus doivent être aussi représentées sous cette forme. C'est ce que fait un second programme (aussi écrit en Perl), qui génère des tables de `doubles` pour la routine de calcul de 2^x, écrite en C.

Les tables pour u seront indexées par i allant de 0 à 4096 ; le cas $i = 4096$ est obtenu pour $x = \frac{1}{2}$ (valeur limite) et doit donc être pris en compte. Nous construisons trois tables `EU`, `MUH` et `MUL` de telle façon que

$$\texttt{EU}[i] = 2^u \quad \text{et} \quad \texttt{MUH}[i] + \texttt{MUL}[i] = -\lfloor u \rceil_{-104}$$

avec

$$\mathrm{LSB}(\texttt{MUH}[i]) \geqslant -51 \quad \text{et} \quad \mathrm{MSB}(\texttt{MUL}[i]) \leqslant -52.$$

Les tables pour v seront indexées par i allant de 0 à 4607. Nous construisons trois tables `EV`, `MVH` et `MVL` de telle façon que

$$\texttt{EV}[i] = 2^v \quad \text{et} \quad \texttt{MVH}[i] + \texttt{MVL}[i] = -\lfloor v \rceil_{-104}$$

avec également

$$\mathrm{LSB}(\texttt{MVH}[i]) \geqslant -51 \quad \text{et} \quad \mathrm{MSB}(\texttt{MVL}[i]) \leqslant -52.$$

4.5.2.4 Description de l'algorithme

Rappelons que nous avons un nombre machine x tel que $|x| \leqslant \frac{1}{2}$ et que nous voulons calculer 2^x avec arrondi exact. L'algorithme de calcul approché (algorithme 4.3 page ci-contre) peut se décomposer en 10 étapes, qui peuvent s'effectuer avec n'importe quel mode d'arrondi, sauf la dernière, qui devra être effectuée dans le mode d'arrondi voulu, afin d'obtenir un peu plus facilement l'arrondi exact. Tous les calculs se font en double précision.

Les étapes sont numérotées de 1 à 10, et les sous-étapes par des lettres minuscules. Il y a une réutilisation naturelle des variables, mais pour plus de clarté, nous avons noté en indice le numéro de l'étape dans laquelle l'affectation de la variable a eu lieu.

C_1, C_2 et C_3 sont trois constantes dont la somme donne une approximation de $\log(2)$ avec une erreur inférieure à 2^{-75}. Elles ont été en partie choisies en minimisant leur taille de mantisse : $\mathrm{MSB}(C_k) - \mathrm{LSB}(C_k) + 1$ doit être « assez petit » et $\mathrm{LSB}(C_k) - \mathrm{MSB}(C_{k+1})$ doit être « assez grand ». Cela s'obtient en considérant les longues séquences de 0 consécutifs et de 1 consécutifs dans l'écriture en base 2 de $\log(2)$. Voici les choix effectués :

$$\begin{array}{lll} C_1 = +1.01100010111001000011_2 \times 2^{-1} & (\mathrm{MSB} = -1 & \text{-} \ \mathrm{LSB} = -21) \\ C_2 = -1.00000101110001100001_2 \times 2^{-29} & (\mathrm{MSB} = -29 & \text{-} \ \mathrm{LSB} = -49) \\ C_3 = -1.10010101000011011_2 \quad \times 2^{-54} & (\mathrm{MSB} = -54 & \text{-} \ \mathrm{LSB} = -71) \end{array}$$

4.5.2.5 Preuve de l'algorithme

Nous allons maintenant prouver que l'algorithme ci-dessus calcule 2^x avec une certaine erreur et majorer cette erreur, étape par étape (sans donner tous les détails). Pour chaque étape, une explication est donnée sur ce que fait l'algorithme.

Pour alléger les notations, les indices correspondant à l'étape courante ne sont pas écrits.

1. On décompose l'argument x en deux : x_h et x_ℓ.
   On a $x_h + x_\ell = x$ avec les inégalités suivantes : $\mathrm{LSB}(x_h) \geqslant -51$ et $\mathrm{MSB}(x_\ell) \leqslant -52$.
2. On sélectionne une première approximation u de x, on calcule $x - u$ et on lit 2^u.
   Le résultat de $x - u$ est représenté par $x_h + x_\ell$. Par construction des tables, on a toujours $\mathrm{LSB}(x_h) \geqslant -51$. Et d'après les valeurs minimale

Algorithme 4.3 Calcul approché de 2^x.

1a.	$x_{h,1}$	$=$	$\mathrm{ROUND}(x, 51)$
1b.	$x_{\ell,1}$	$=$	$x - x_{h,1}$
2a.	i_2	$=$	$\left[(x + 0.5) \times 2^{12}\right]$
2b.	$x_{h,2}$	$=$	$x_{h,1} + \mathtt{MUH}[i_2]$
2c.	$x_{\ell,2}$	$=$	$x_{\ell,1} + \mathtt{MUL}[i_2]$
2d.	$y_{0,2}$	$=$	$\mathtt{EU}[i_2]$
3a.	i_3	$=$	$\left[(x_{h,2} + 9/2^{16}) \times 2^{24}\right]$
3b.	$x_{h,3}$	$=$	$x_{h,2} + \mathtt{MVH}[i_3]$
3c.	$x_{\ell,3}$	$=$	$x_{\ell,2} + \mathtt{MVL}[i_3]$
3d.	$y_{0,3}$	$=$	$y_{0,2} \times \mathtt{EV}[i_3]$
4a.	$x_{h,4}$	$=$	$x_{h,3}$
4b.	$x_{m,4}$	$=$	$\mathrm{ROUND}(x_{\ell,3}, 78)$
4c.	$x_{\ell,4}$	$=$	$x_{\ell,3} - x_{m,4}$
5a.	$x_{\ell,5}$	$=$	$C_1\, x_{\ell,4} + C_2\, x_{m,4} + C_3\, x_{h,4}$
5b.	$x_{m,5}$	$=$	$C_1\, x_{m,4} + C_2\, x_{h,4}$
5c.	$x_{h,5}$	$=$	$C_1\, x_{h,4}$
6a.	$x_{h,6}$	$=$	$x_{h,5}$
6b.	$x_{r,6}$	$=$	$x_{\ell,5} + x_{m,5} + x_{h,5}\,(x_{m,5} + x_{h,5}\,(1 + x_{h,5} \times (1/3)) \times (1/2))$
7a.	$x_{s,7}$	$=$	$\mathrm{ROUND}(x_{h,6}, 50)$
7b.	$x_{r,7}$	$=$	$x_{r,6} + (x_{h,6} - x_{s,7})$
8a.	$y_{h,8}$	$=$	$\mathrm{ROUND}(y_{0,3}, 25)$
8b.	$y_{\ell,8}$	$=$	$y_{0,3} - y_{h,8}$
9a.	$y_{0,9}$	$=$	$y_{0,3}$
9b.	$y_{1,9}$	$=$	$x_{s,7}\, y_{h,8}$
9c.	$y_{2,9}$	$=$	$x_{s,7}\, y_{\ell,8} + x_{r,7}\, y_{0,3}$
10a.	Sélection du mode d'arrondi voulu		
10b.	$y_{h,10}$	$=$	$(y_{2,9} + y_{1,9}) + y_{0,9}$
10c.	$y_{\ell,10}$	$=$	$((y_{0,9} - y_{h,10}) + y_{1,9}) + y_{2,9}$

et maximale de $x - u$, on a $\mathrm{MSB}(x_h) \leqslant -13$. On a $\mathrm{MSB}(x_\ell) \leqslant -51$, donc $\mathrm{ULP}(x_\ell) \leqslant -51 - 52 = -103$. L'erreur sur x est donc majorée par $2^{-104} + 2^{-103} = 3 \times 2^{-104}$.

3. On sélectionne une seconde approximation v de $x - u$, puis on calcule $x - u - v$ et 2^{u+v}.

   Le résultat de $x - u - v$ est représenté par $x_h + x_\ell$. Par construction des tables, on a toujours $\mathrm{LSB}(x_h) \geqslant -51$. Et d'après les valeurs minimale et maximale de $x-u-v$ (arrondi), on a $\mathrm{MSB}(x_h) \leqslant -25$. On a $|x_\ell| < 3 \times 2^{-51}$, donc $\mathrm{MSB}(x_\ell) \leqslant -50$, et $\mathrm{ULP}(x_\ell) \leqslant -102$. L'erreur sur x est donc majorée par 7×2^{-104}.

   On calcule ensuite $y_0 = \mathtt{EU}[i_2] \times \mathtt{EV}[i_3]$. Puisque $\mathrm{MSB}(\mathtt{EU}[i_2]) \leqslant 0$ et $\mathrm{LSB}(\mathtt{EU}[i_2]) \geqslant -16$, $\mathtt{EU}[i_2]$ tient au maximum sur 17 bits. Puisque $\mathrm{MSB}(\mathtt{EV}[i_3]) \leqslant 0$ et $\mathrm{LSB}(\mathtt{EV}[i_3]) \geqslant -32$, $\mathtt{EV}[i_3]$ tient au maximum sur 33 bits, donc la multiplication se fait exactement. D'autre part, le produit est inférieur à 2, donc $\mathrm{MSB}(y_0) \leqslant 0$ et $\mathrm{LSB}(y_0) \geqslant -48$.

4. On décompose $x' = x - u - v$ en trois (x_h ne change pas, et x_ℓ est décomposé en deux).

   On obtient :
   $\mathrm{MSB}(x_h) \leqslant -25$ et $\mathrm{LSB}(x_h) \geqslant -51$,
   $\mathrm{MSB}(x_m) \leqslant -50$ et $\mathrm{LSB}(x_m) \geqslant -78$,
   $\mathrm{MSB}(x_\ell) \leqslant -79$.

   Concernant x_m, on a même : $|x_m| \leqslant 3 \times 2^{-51}$.

   Rappelons que l'erreur sur ce nombre est majorée par 7×2^{-104}. Nous voulons évaluer $2^{x'}$ et majorer l'erreur.

5. On calcule $t = x' \log(2)$.

   Le résultat t sera représenté par $x_h + x_m + x_\ell$; les opérations intervenant dans le calcul de x_h et de x_m se font de manière exacte.

   Cherchons un majorant de $|x_m|$. On a $x_m = C_1\, x_{m,4} + C_2\, x_{h,4}$ avec $|C_1| < 3/4$, $|x_{m,4}| \leqslant 3 \times 2^{-51}$, $|C_2| < 2^{-29} + 2^{-34}$, $|x_{h,4}| \leqslant 2^{-25} + 2^{-32}$. Donc $|x_m| < 9 \times 2^{-53} + 2^{-53} = 5 \times 2^{-52}$.

   On obtient :
   $\mathrm{MSB}(x_h) \leqslant -25$ et $\mathrm{LSB}(x_h) \geqslant -72$,
   $\mathrm{MSB}(x_m) \leqslant -50$ et $\mathrm{LSB}(x_m) \geqslant -100$,
   $\mathrm{MSB}(x_\ell) \leqslant -76$, et l'erreur est majorée par 2^{-127}.

   La somme des termes non calculés dans la multiplication de x' par $C_1 + C_2 + C_3$ est majorée par 21×2^{-107}.

   On a par construction : $|x'| \leqslant 2^{-25} + 2^{-32}$. L'erreur globale sur la valeur de $t = x_h + x_m + x_\ell$ calculée ici est donc majorée par :

   $$2^{-75}\,(2^{-25} + 2^{-32}) + 2^{-127} + 21 \times 2^{-107} + 7 \times 2^{-104} \log(2),$$

   donc par : $2^{-100} + 2^{-101}$.

6. On calcule $P(t) = t + \frac{1}{2}t^2 + \frac{1}{6}t^3$, dont la valeur est représentée par $x_h + x_r$. L'erreur sur $1/3$ est majorée par 2^{-54}. L'erreur sur $x_h \times (1/3)$ est donc majorée par 2^{-77}.

On a $\mathrm{MSB}(1+x_h\times(1/3)) \leqslant 0$, donc $\mathrm{ULP}(1+x_h\times(1/3)) \leqslant -52$, et l'erreur effectuée dans le calcul de $1 + x_h \times (1/3)$ est majorée par $2^{-52} + 2^{-77}$. Ensuite, $|x_h| \leqslant 2^{-25} + 2^{-32}$, donc l'erreur sur $x_h\,(1 + x_h \times (1/3))$ est majorée par :

$$(2^{-52} + 2^{-77})(2^{-25} + 2^{-32}) + 2^{-77} \leqslant 2^{-76} + 2^{-83}.$$

L'erreur sur $x_h\,(1 + x_h \times (1/3)) \times (1/2)$ est donc majorée par $2^{-77} + 2^{-84}$. Et on a $\mathrm{MSB}(x_h\,(1 + x_h \times (1/3)) \times (1/2)) \leqslant -26$. Donc après ajout de x_m, l'erreur est majorée par $2^{-77} + 2^{-78} + 2^{-84}$.

On a $\mathrm{MSB}(x_h\,(x_m + x_h\,(1 + x_h \times (1/3)) \times (1/2))) \leqslant -51$. L'erreur sur $x_h\,(x_m + x_h\,(1 + x_h \times (1/3)) \times (1/2))$ est majorée par :

$$(2^{-77} + 2^{-78} + 2^{-84})(2^{-25} + 2^{-32}) + 2^{-103} \leqslant 2^{-101} + 2^{-107}.$$

On a $|x_\ell| < 2^{-75}$, $|x_m| < 5 \times 2^{-52}$, et le 3e terme (ci-dessus) est majoré par $2^{-51} + 2^{-56}$. Donc $|x_r| < 15 \times 2^{-53}$, et $\mathrm{MSB}(x_r) \leqslant -50$. Donc $\mathrm{ULP}(x_r) \leqslant -102$. À cause des deux additions et de l'erreur sur le 3e terme, l'erreur sur x_r est majorée par $2^{-100} + 2^{-107}$.

En tenant compte de l'erreur sur t ($2^{-100}+2^{-101}$) multipliée par $\exp(t_{\max})$ ainsi que de l'approximation par un polynôme (erreur majorée par 2^{-104}), on obtient une erreur globale majorée par : $2^{-99} + 2^{-101} + 2^{-103}$.

7. On normalise $x_h + x_r$.

   Par construction de x_s, on a $\mathrm{MSB}(x_s) \leqslant -25$ et $\mathrm{LSB}(x_s) \geqslant -50$. De plus, $\mathrm{MSB}(x_h - x_s) \leqslant -51$ et $\mathrm{LSB}(x_h - x_s) \geqslant -72$.

   On a maintenant :

   $$|x_r| < 15 \times 2^{-53} + 2^{-50} = 23 \times 2^{-53}.$$

   Donc $\mathrm{MSB}(x_r) \leqslant -49$. La dernière addition génère alors une erreur inférieure à 2^{-101}. L'erreur globale sur $x_s + x_r$ est donc majorée par : $2^{-99} + 2^{-100} + 2^{-103}$.

   D'autre part, d'après la valeur maximale de t, on a :

   $$|x_s + x_r| < 2^{-25} + 2^{-32}.$$

8. On décompose y_0 (calculé à l'étape 3) en deux : $y_0 = y_h + y_\ell$, ce qui donne :

   $\mathrm{MSB}(y_h) \leqslant 0$ et $\mathrm{LSB}(y_h) \geqslant -25$.
   $\mathrm{MSB}(y_\ell) \leqslant -26$ et $\mathrm{LSB}(y_\ell) \geqslant -48$.

9. On calcule $y_{0,3} \times (1 + P(t))$.

   On a $|x_r\,y_0| < 33 \times 2^{-53}$, donc $|y_2| < 41 \times 2^{-53}$. Par conséquent :

   $\mathrm{MSB}(y_0) \leqslant 0$ et $\mathrm{LSB}(y_0) \geqslant -48$.
   $\mathrm{MSB}(y_1) \leqslant -25$ et $\mathrm{LSB}(y_1) \geqslant -75$.
   $\mathrm{MSB}(y_2) \leqslant -48$.

   Les erreurs d'arrondi interviennent seulement dans y_2. La multiplication $x_s\,y_\ell$ est effectuée exactement. On a : $|x_r\,y_0| < 33 \times 2^{-53}$, donc $\mathrm{ULP}(x_r\,y_0) \leqslant -100$, donc l'erreur sur y_2 est majorée par 2^{-99}, à laquelle on ajoute l'erreur due à x_r (multipliée par un majorant de y_0).

   Donc l'erreur globale est majorée par $2^{-98} + 2^{-99} + 2^{-101}$.

10. On renormalise avec le mode d'arrondi voulu.

   Le calcul de y_h se fait à l'aide de deux additions. Soit y_t le résultat arrondi de la première addition. On peut écrire les égalités suivantes :

$$\begin{cases} y_1 + y_2 = y_t + \varepsilon_1 \\ y_t + y_0 = y_h + \varepsilon_2 \end{cases}$$

   où y_t et y_h sont des nombres machine, ε_1 et ε_2 sont les erreurs d'arrondi.

   On a : $\mathrm{MSB}(y_h) = 0$ ou 1, et $\mathrm{MSB}(y_t) \leqslant -25$. Donc $1/2 < y_0/y_h < 2$, et la soustraction $y_0 - y_h$ s'effectue exactement.

   On a : $y_0 - y_h = \varepsilon_2 - y_t = -y_1 + \varepsilon$, avec $\varepsilon = \varepsilon_1 + \varepsilon_2 - y_2$; donc $|\varepsilon| \leqslant 2^{-77} + 2^{-52} + 41 \times 2^{-53}$. Donc $\mathrm{MSB}((y_0 - y_h) + y_1) \leqslant -48$, et $\mathrm{ULP}((y_0 - y_h) + y_1) \leqslant -100$. Donc l'erreur due à cette addition est majorée par 2^{-100}.

   On a aussi $\mathrm{ULP}(y_2) \leqslant -100$, donc l'erreur due à la dernière addition est majorée par 2^{-100}.

   2^x est donc représenté par $y_h + y_\ell$ avec une erreur finale majorée par $2^{-98} + 2^{-99} + 2^{-101} + 2 \times 2^{-100}$. En résumé :

$$|2^x - (y_h + y_\ell)| < \varepsilon_{\max}, \quad \text{avec} \quad \varepsilon_{\max} = 2^{-97} + 2^{-101}.$$

4.5.2.6 Cas particulier : petits arguments

Pour des raisons pratiques, nous avons choisi d'utiliser une autre méthode pour les très petites valeurs de l'argument x : si $|x| < 2^{-40}$, alors 2^x est approché par $1 + C\,x$, où C est une approximation de $\log(2)$ sur un seul nombre machine, et nous utilisons une table pour arrondir exactement.

En fait, on ne calculera pas la valeur $1 + C\,x$. Nous nous intéressons aux valeurs frontières entre deux arrondis, de la forme $1 + i\,U$, avec i entier naturel, et $U = -2^{-54}$ pour $x < 0$, $U = +2^{-53}$ pour $x \geqslant 0$. À l'aide de Maple avec arithmétique d'intervalles, nous construisons deux tables indexées par i :

- $\mathtt{TN}[i] = \lceil \log_2(1 - i.2^{-54}) \rfloor$ pour $x < 0$ ($0 \leqslant i \leqslant 11356$) ;
- $\mathtt{TP}[i] = \lceil \log_2(1 + i.2^{-53}) \rfloor$ pour $x \geqslant 0$ ($0 \leqslant i \leqslant 5678$).

Nous utilisons alors l'algorithme 4.4 page suivante.

4.5.2.7 Arrondi exact pour le cas général

Nous voulons savoir dans quelles conditions y_h est l'arrondi exact de 2^x (dans le mode d'arrondi voulu). Rappelons d'abord que x vérifie :

$$x_0 \leqslant |x| \leqslant \frac{1}{2}, \quad \text{avec} \quad x_0 = 2^{-40}.$$

Étudions d'abord ce qui se passe en général. La valeur de y_h est obtenue par l'addition $y_t + y_0$ effectuée dans le mode d'arrondi voulu. L'erreur sur le

Algorithme 4.4 Calcul de 2^x avec arrondi exact pour $|x|$ petit.

- **si** $(x < 0)$
  - $i = \left[-C.2^{54}.x\right]$;
  - **si** $(x < \mathtt{TN}[i])$
    - décrémenter i;
  - **si** $(x \geqslant \mathtt{TN}[i+1])$
    - incrémenter i;
  - **si** (mode d'arrondi au plus près)
    - incrémenter i;
  - $i = i/2$;
  - **si** (mode d'arrondi vers $-\infty$ ou vers 0)
    - incrémenter i;
  - **renvoyer** $1 - i.2^{-53}$;
- **sinon**
  - $i = \left[C.2^{53}.x\right]$;
  - **si** $(x < \mathtt{TP}[i])$
    - décrémenter i;
  - **si** $(x \geqslant \mathtt{TP}[i+1])$
    - incrémenter i;
  - **si** (mode d'arrondi au plus près)
    - incrémenter i;
  - $i = i/2$;
  - **si** (mode d'arrondi vers $+\infty$)
    - incrémenter i;
  - **renvoyer** $1 + i.2^{-52}$;

résultat exact de $y_t + y_0$ est égale à l'erreur sur y_t, qui est la somme de l'erreur sur y_2 et de l'erreur due à l'addition flottante $y_1 + y_2$. Cette erreur est majorée par un nombre de l'ordre de 2^{-77}, qui est très petit devant $2^{\mathrm{ULP}(y_h)}$. Par conséquent, la valeur de y_h sera en général l'arrondi exact de 2^x dans le mode d'arrondi voulu. Le temps passé à corriger l'arrondi dans les autres cas ne sera pas très grand, et il sera donc négligeable en moyenne.

Cherchons maintenant à détecter un arrondi incorrect. On a :

$$2^x = y_h + y_\ell + \varepsilon, \quad \text{avec} \quad \varepsilon < \varepsilon_{\max} = 2^{-97} + 2^{-101}.$$

La valeur de y_h est l'arrondi exact de 2^x si et seulement si la différence vérifie une inégalité du type : $a \leqslant 2^x - y_h \leqslant b$, i.e. $a \leqslant y_\ell + \varepsilon \leqslant b$, où a et b ont les valeurs suivantes :

Arrondi	a	b
vers $-\infty$	0	$+2^{E-52}$
vers $+\infty$	-2^{E-52}	0
au plus près	-2^{E-53}	$+2^{E-53}$

où E est l'exposant de 2^x (valeur exacte). Dans notre cas, E, donc a et b également, ne dépendent que du signe de x : $E = -1$ pour $x < 0$, et $E = 0$ pour $x \geqslant 0$. Nous n'avons pas mentionné la mode d'arrondi vers 0, car il est ici identique au mode d'arrondi vers $-\infty$, 2^x étant toujours positif.

Si y_ℓ vérifie $a + \varepsilon_{\max} \leqslant y_\ell \leqslant b - \varepsilon_{\max}$, alors l'arrondi exact est garanti. Sinon, il est possible que ce soit dû au double arrondi lors du calcul de y_h (deux additions) : la première addition a introduit une erreur de l'ordre de 2^{-77}. On essaye donc de corriger y_h et y_ℓ, en ajoutant à l'un et en retranchant à l'autre la valeur $2^{E-52} = 2^{\mathrm{ULP}(2^x)}$. Si on est toujours en dehors des bornes, cela est dû à une erreur trop grande sur $y_h + y_\ell$ (dilemme du fabricant de tables). Dans ce cas, on fait une lecture dans la table des pires cas.

4.5.2.8 Recherche des arguments à tabuler

Indiquons maintenant comment trouver les arguments à tabuler. Pour tout argument qui n'a pas pu être arrondi exactement, y_ℓ est à une distance inférieure à $\varepsilon_{\max}$ d'une frontière entre les deux arrondis possibles ; et puisque 2^x a été calculé à $\varepsilon_{\max}$ près, alors la valeur exacte de 2^x est à une distance inférieure à $2\,\varepsilon_{\max}$ de la frontière. Pour construire la table des pires cas, nous allons considérer tous les nombres machines x (dans l'intervalle $[x_0, 1/2]$ en valeur absolue) pour lesquels 2^x est à une distance inférieure à $2\,\varepsilon_{\max}$ d'une frontière. Pour chacun des trois modes d'arrondi, ces pires cas seront testés avec l'algorithme présenté en section 4.5.2.4. L'argument sera considéré comme un pire cas final si et seulement si l'algorithme a détecté un arrondi non garanti (i.e. l'inégalité $a + \varepsilon_{\max} \leqslant y_\ell \leqslant b - \varepsilon_{\max}$ n'est pas vérifiée).

$2\,\varepsilon_{\max} < 2^{-95}$, donc il suffit de trouver les arguments pour lesquels le résultat exact 2^x a :

– 41 bits identiques après le bit d'arrondi pour $x < 0$,
– 42 bits identiques après le bit d'arrondi pour $x > 0$.

Pour $x < 0$, `t3-lastep` sera lancé avec $k_{\min} = 42$, et pour $x > 0$, `t3-lastep` sera lancé avec $k_{\min} = 43$ (cf section 3.1.4).

Voici les résultats :

Mode d'arrondi	Pires cas	Arrondi exact	Arrondi à corriger
vers $-\infty$	52 231	51 148	1 083
vers $+\infty$	52 231	51 028	1 203
au plus près	52 224	26 174	26 050

4.5.2.9 Tabulation des pires cas

Nous allons expliquer comment la tabulation a été implémentée. Les trois modes d'arrondi sont considérés séparément. Donc ce qui suit doit être répété pour chaque mode d'arrondi.

Fixons un mode d'arrondi et définissons les ensembles suivants. Soit E_0 l'ensemble des pires cas trouvés dont l'arrondi par l'algorithme présenté en section 4.5.2.4 est correct. Soit E_1 l'ensemble des pires cas trouvés dont l'arrondi est incorrect (et doit donc être corrigé). Soit $E = E_0 \cup E_1$ l'ensemble des pires cas. Nous définissons deux codes de hachage simples

$$h_1 : E \to H_1 \quad \text{et} \quad h_2 : E \to H_2$$

(où H_1 et H_2 seront en fait les ensembles des entiers positifs pouvant s'écrire sur n_1 et n_2 bits), ainsi que des ensembles $T(h) \subset H_2$ tels que pour tout $h \in h_1(E)$, l'ensemble $T(h)$ vérifie :

$$\forall\, x \in h_1^{-1}(h), \quad \begin{cases} \text{si} \quad x \in E_1, & \text{alors} \quad h_2(x) \in T(h); \\ \text{si} \quad x \in E_0, & \text{alors} \quad h_2(x) \notin T(h). \end{cases}$$

Comme fonctions h_1 et h_2, nous choisirons des champs de bits fixés dans la représentation machine de $x \in E$. La recherche de fonctions h_1 et h_2 se fait à la main, mais la vérification des conditions ci-dessus se fait par programme. Dans notre implémentation, nous avons trouvé les fonctions suivantes[8] : h_1 extrait le champ allant du bit 32 au bit 43 (longueur $n_1 = 12$), et h_2 extrait le champ allant du bit 48 au bit 63 (longueur $n_2 = 16$)[9]. Ces valeurs permettent d'obtenir des ensembles $T(h)$ pas trop gros : pour tout h, on a $\#T(h) \leqslant 4$ pour les modes d'arrondi dirigés, et $\#T(h) \leqslant 17$ pour le mode d'arrondi au plus près, où $\#X$ désigne le nombre d'éléments de l'ensemble X.

[8]Ce sont les mêmes pour chaque mode d'arrondi, car l'implémentation actuelle ne permet pas d'en choisir des différentes.

[9]Ce champ sera représenté sous forme d'un nombre entier d'octets (une puissance de 2, en fait) en mémoire. Un seul octet aurait demandé une valeur de n_1 trop grosse, et deux octets suffisent.

Nous voulons stocker en mémoire les ensembles $T(h)$. Pour cela, nous devons tenir compte du fait que la répartition des valeurs dans les ensembles $T(h)$ est toujours très irrégulière ; par exemple, le choix de stocker les ensembles $T(h)$ dans des structures de longueur fixe (indépendant de h) ferait perdre énormément de place. Nous avons choisi la solution suivante. Nous stockons les éléments des ensembles les uns à la suite des autres dans une table U, en commençant par $T(0)$, puis $T(1)$, et ainsi de suite jusqu'à $T(2^{n_1}-1)$; la longueur de cette table est au plus $\#E_1$. Une autre table, de longueur $2^{n_1}+1$, donne des pointeurs (en fait, des indices) $P(h)$ sur les ensembles $T(h)$, le nombre d'éléments étant donné par $P(h+1)-P(h)$.

La recherche d'un élément x dans la table des pires cas se fait en calculant $h = h_1(x)$, puis en comparant successivement à $h_2(x)$ les éléments $U(P(h))$, $U(P(h)+1)$, $U(P(h)+2)$, et en s'arrêtant soit lorsque les valeurs sont égales, soit juste avant $U(P(h+1))$.

4.5.2.10 Temps de calcul

Nous avons fait des mesures de temps de calcul de 2^x avec notre routine. La machine utilisée est une station Sun Ultra-5 à 333 MHz sous Solaris 2.6.

Le temps varie entre $2.8\,\mu s$ et $3.8\,\mu s$ pour des arguments aléatoires, et il est environ de $2.55\,\mu s$ pour les arguments « très petits » (cf section 4.5.2.6). Si l'argument est un pire cas, il faut ajouter environ $0.3\,\mu s$. Le temps le plus long observé est ainsi de $4.10\,\mu s$ pour $x = 1.42636434598407579$ (arrondi au plus près) dans le mode d'arrondi vers $+\infty$.

Notons qu'un changement de mode d'arrondi semble nécessiter entre $0.15\,\mu s$ et $0.20\,\mu s$. Un tel changement de mode d'arrondi se fait en appelant une fonction d'assez haut niveau (`ieee_flags` de la bibliothèque sunmath). Il y a 8 appels à `ieee_flags` par calcul de 2^x, ainsi que d'autres traitements liés à ces appels. L'utilisation de fonctions de plus bas niveau avec certains compilateurs pourrait donc faire gagner un temps non négligeable. Et surtout, implémenter la stratégie de Ziv pourrait faire gagner encore beaucoup de temps en moyenne.

À titre de comparaison, nous avons aussi fait des mesures sur la bibliothèque ml4j d'IBM, qui implémente la stratégie de Ziv. La fonction 2^x n'est pas implémentée ; nous avons alors testé la fonction exponentielle[10]. De même, seul le mode d'arrondi au plus près est implémenté. Sur la même station Sun, le calcul de $\exp(x)$ prend en général $0.55\,\mu s$, mais $4.4\,ms$ (millisecondes) pour les pires cas, soit plus d'un facteur 1000 par rapport à notre routine (et il n'y a pour l'instant aucune preuve de l'exactitude du résultat).

[10] La réduction d'argument est bien plus complexe, mais elle a très peu d'influence sur le rapport de temps pour les pires cas.

Conclusion

Le but du problème posé, à long terme, est de certifier une bibliothèque de calculs en vue d'instaurer une norme garantissant pour les fonctions élémentaires ($\exp$, $\log$, $\sin$, $\cos$, $\tan$, $\arctan$, etc.) ce que garantit déjà la norme IEEE-754 pour les fonctions arithmétiques ($+$, $-$, $\times$, $\div$ et $\sqrt{}$), c'est-à-dire que le résultat d'une opération soit l'arrondi correct du résultat exact. D'autre part, le coût en temps et en mémoire d'une telle bibliothèque doit être raisonnable. Pour résoudre ce problème, nous avons considéré deux approches :

– Pour les flottants en simple et double précision, le problème peut être résolu à l'aide de tests exhaustifs, afin de trouver à quelle précision maximale la bibliothèque devra effectuer les calculs *intermédiaires*. Cette précision sera de l'ordre du double de la taille de la mantisse, d'après les hypothèses probabilistes. Cela permettra d'avoir des algorithmes plus rapides en tabulant les pires cas, et de connaître le coût maximal (en temps et en mémoire) des calculs.

  Le cas de la simple précision est rapide à traiter. Nous nous intéressons donc uniquement à la double précision.

  Nous avons déjà des résultats complets pour les fonctions 2^x, $\log_2(x)$, l'exponentielle et le logarithme, et des résultats partiels pour d'autres fonctions élémentaires ($\sin$ et $\cos$, et leurs réciproques). Les tests continuent, et nos programmes pourront être portés pour d'autres plates-formes que des stations Sun, et beaucoup plus de machines pourront être utilisées. En tenant compte de l'évolution des machines, qui sont de plus en plus rapides, nous pouvons même espérer résoudre prochainement le cas de la précision étendue (64 bits de mantisse).

  Cependant les fonctions $\sin$, $\cos$ et $\tan$ avec de gros arguments (c'est-à-dire des arguments dont le bit de poids faible n'est pas négligeable devant π) sont très irrégulières du point de vue numérique. Pour ces fonctions, nous ne pourrons pas avoir de résultats complets, et nous devrons nous rabattre sur la seconde approche. Même si le sinus ou le cosinus d'un très grand nombre n'a généralement pas beaucoup de sens (car un tel nombre n'est généralement qu'une approximation très grossière par rapport à la période de ces fonctions), l'arrondi exact est important d'une part pour assurer la portabilité des programmes, d'autre part parce qu'il peut y avoir des corrélations dans le programme qui utilisera ces fonctions : par exemple, la propriété $\sin^2 x + \cos^2 x \approx 1$ peut être requise. Une autre solution serait de considérer les fonctions $\sin(\pi x)$, $\cos(\pi x)$ et $\tan(\pi x)$, pour lesquelles les tests exhaustifs seraient très simples (un peu comme pour 2^x), d'autant plus que ces fonctions ont de meilleures propriétés de périodicité du point de vue numérique.

La quadruple précision demanderait également trop de temps (sauf si des algorithmes de tests beaucoup plus rapides sont trouvés). Il faut alors utiliser la seconde approche.

– Nous devons prévoir des calculs sur un très grand nombre de bits, dont les bornes sont données par des théorèmes de théorie des nombres ; nous avons utilisé le théorème de Nesterenko et Waldschmidt, qui nous donne des bornes de l'ordre de plusieurs millions à plusieurs milliards de bits. La bibliothèque MPFR de P. Zimmermann pourra être utilisée pour cela, par exemple.

  L'arrondi exact peut *a priori* coûter très cher, mais nous devons considérer que :

  – La stratégie de Ziv permet d'évaluer les fonctions pour la plupart des arguments avec une précision interne un peu supérieure à n bits, où n est la taille finale de la mantisse. Le temps requis est donc en moyenne quasiment le même que celui nécessaire aux bibliothèques existantes pour calculer les fonctions élémentaires sans garantir l'arrondi exact. Reste le problème du temps « au pire »...
  – D'après les hypothèses probabilistes, la (vraie) borne supérieure de m_0 est très certainement beaucoup plus faible que la borne théorique. Si jamais quelqu'un tombe sur une grande borne nécessitant des calculs à grande précision (il est très peu probable que cela puisse arriver), il faut prévoir de le signaler à l'utilisateur, de manière à pouvoir tabuler cette valeur dans une future version de la bibliothèque.
  – Les ordinateurs deviennent de plus en plus rapides. Le temps maximal théorique d'une opération à très grande précision (qui a, rappelons-le, très peu de chances d'être nécessaire) sera de l'ordre de quelques minutes.
  – De nouveaux théorèmes donnant de meilleures bornes sur m_0 peuvent toujours être découverts ; le théorème de Nesterenko et Waldschmidt est relativement récent (1995).

L'arrondi exact permet de préserver certaines propriétés mathématiques utiles. Mais dans certains cas (qui sont connus), c'est l'inverse qui se produit. Considérons l'exemple suivant [37] : nous voulons évaluer $\arctan(2^{30})$ en simple précision avec l'arrondi au plus près. Le nombre machine le plus proche de $\arctan(2^{30})$ est :

$$\frac{13\,176\,795}{8\,388\,608} = 1.57079637050628662109375 > \frac{\pi}{2}.$$

Donc si l'arctangente est arrondie exactement, avec arrondi au plus près, nous obtenons une valeur supérieure à $\frac{\pi}{2}$ (cependant égale à la représentation machine de $\frac{\pi}{2}$). Par conséquent, nous aurons dans ces conditions :

$$\tan(\arctan(2^{30})) = -2.2877... \times 10^{7}.$$

La propriété mathématique $|\arctan x| < \frac{\pi}{2}$ n'est plus vérifiée. Mais ce genre de problème ne se produit pas avec toutes les fonctions :

Théorème 4 *Si une fonction f vérifie :*

– *$f(x)$ est défini pour tout $x \in [a, b]$,*

– *$c = \min_{x\in[a,b]} f(x)$ est un nombre machine,*

– *$d = \max_{x\in[a,b]} f(x)$ est un nombre machine,*

alors si $f(x)$ est calculé avec arrondi exact pour un nombre machine $x \in [a, b]$, le résultat obtenu sera dans $[c, d]$.

◇ *Preuve.* Soit $x \in [a, b]$ et $y = f(x)$. Considérons l'ensemble des nombres machine inférieurs ou égaux à y. Par définition de c, cet ensemble contient c, et en particulier, il est non vide. Soit $\lfloor y \rfloor$ son plus grand élément. De même, soit $\lceil y \rceil$ le plus petit élément de l'ensemble des nombres machine supérieurs ou égaux à y. On a alors :

$$c \leqslant \lfloor y \rfloor \leqslant y \leqslant \lceil y \rceil \leqslant d,$$

ce qui prouve que l'arrondi exact de $f(x)$ dans le mode d'arrondi courant ($\lfloor y \rfloor$ ou $\lceil y \rceil$) est dans $[c, d]$. □

Par conséquent, le problème décrit ci-dessus n'apparaîtra pas pour les fonctions $\sin$, $\cos$, th.

L'arrondi exact des fonctions élémentaires devenant plus facile, il est temps de penser à une norme. Les questions que nous nous posons à ce sujet sont les suivantes :

– Doit-on prévoir un arrondi « moins cher », mais bien défini, pour les applications dont la rapidité est essentielle ? Quelles propriétés mathématiques doit-il vérifier (par exemple, monotonie, erreur maximale garantie) ? Si un tel mode d'arrondi est fourni, doit-on prévoir un drapeau indiquant un arrondi non exact ?

– Doit-on prévoir un arrondi exact sur tout le domaine de définition de la fonction, ou sur un certain domaine seulement ? La première solution est préférable, mais nous avons vu qu'elle peut être plus complexe (par exemple, pour les fonctions $\sin$ et $\cos$ avec de grands arguments).

– Que doit-il se passer si jamais il n'y a pas assez de mémoire lors d'un calcul à grande précision ?

D’un point de vue plus théorique, il serait intéressant d’obtenir des résultats concernant la complexité de nos algorithmes de tests exhaustifs. Mais cela semble être un problème très compliqué, car beaucoup de paramètres interviennent. Nous devrons aussi chercher à améliorer l’algorithme de multiplication par une constante et à prouver certains résultats.

Annexe A

Quelques démonstrations

Nous allons prouver ici certaines propriétés données dans l'introduction.

Nous supposons que les calculs décrits ici ne génèrent jamais de dépassement de capacité (*overflow* ou *underflow*).

A.1 $\frac{x}{\sqrt{x^2+y^2}}$ avec les modes d'arrondi dirigés

Considérons une machine conforme à la norme IEEE-754. Kahan a montré que sur une telle machine, le calcul de

$$z = \frac{x}{\sqrt{x^2 + y^2}}$$

en arrondi au plus près donne toujours un résultat compris entre -1 et 1. Nous voulons prouver que cette propriété reste vraie en arrondi vers $+\infty$, mais ne l'est plus toujours en arrondi vers $-\infty$ ou vers 0.

Choisissons n'importe quel mode d'arrondi. Pour alléger les notations, les opérations que nous allons considérer dans ce paragraphe sont celles de l'arithmétique flottante utilisée : elles sont arrondies (exactement). On a : $y^2 \geqslant 0$, donc $x^2 + y^2 \geqslant x^2$, $\sqrt{x^2 + y^2} \geqslant \sqrt{x^2}$, et enfin :

$$0 \leqslant \frac{|x|}{\sqrt{x^2 + y^2}} \leqslant \frac{|x|}{\sqrt{x^2}}.$$

Il suffit donc de considérer uniquement les cas où $y = 0$.

À partir de maintenant, nous travaillons dans $\mathbb{R}$: les opérations sont celles effectuées dans $\mathbb{R}$, et l'arrondi d'un résultat y réel (dans le mode d'arrondi courant) sera noté $\diamond(y)$.

A.1.1 Cas de l'arrondi vers $+\infty$

On a $\diamond(x^2) \geqslant x^2$, puis

$$x' = \diamond(\sqrt{\diamond(x^2)}) \geqslant \sqrt{\diamond(x^2)} \geqslant \sqrt{x^2} = |x|$$

(le première inégalité est due au mode d'arrondi vers $+\infty$, et la seconde utilise le fait que la fonction racine carrée est croissante). Donc $|x|/x' \leqslant 1$, et puisque 1 est un nombre machine, $\diamond(|x|/x') \leqslant 1$. CQFD.

A.1.2 Cas de l'arrondi vers $-\infty$ ou vers 0

Soit le nombre machine immédiatement supérieur à 1 : $x = 1 + u$, avec $u = \mathrm{ulp}(1)$. On a $x^2 = 1 + 2u + u^2$, et $\diamond(x^2) = 1 + 2u$, car $u^2 < u$. Puisque $\diamond(x^2) < x^2$, on a :

$$x' = \diamond(\sqrt{\diamond(x^2)}) \leqslant \sqrt{\diamond(x^2)} < \sqrt{x^2} = 1 + u.$$

Puisque x' est un nombre machine *strictement* inférieur à $1 + u$ et que $1 + u$ est le nombre machine *immédiatement* supérieur à 1, alors $x' \leqslant 1$. On montre facilement que $x' \geqslant 1$. Par conséquent, $x' = 1$, et $\diamond(x/x') = 1 + u > 1$. La propriété voulue n'est donc pas vérifiée dans ce cas particulier.

A.2 Condition pour que 2^x soit un nombre machine

Nous voulons montrer de façon élémentaire que si x est un nombre machine (donc rationnel), alors 2^x est un nombre machine si et seulement si x est entier.

◇ *Preuve.* Nous allons montrer les deux implications séparément.

Si x est entier, alors 2^x est un nombre machine (en supposant qu'il n'y a pas de dépassement de capacité), par définition de la représentation flottante en base 2.

Supposons maintenant que 2^x est un nombre machine. Le nombre machine x est rationnel, donc x peut s'écrire $x = p/q$ avec p et q entiers, et $q \geqslant 1$. Le nombre 2^x est aussi rationnel, donc il peut s'écrire $2^x = a/b$ avec a et b entiers strictement positifs premiers entre eux. On a $2^{p/q} = a/b$, et après multiplication par b et élévation à la puissance q, on obtient : $2^p b^q = a^q$. Supposons p positif. On a $q \geqslant 1$, b^q divise a^q et les entiers a et b sont premiers entre eux, donc $b = 1$. Alors $2^p = a^q$, et par conséquent, a est de la forme : $a = 2^k$. D'où $p = kq$. Donc $x = p/q = k$ est un entier. Le cas $p < 0$ est similaire (il suffit de changer p en $-p$ et d'inverser les rôles de a et b). □

Annexe B

Quelques pires cas

Nous allons donner certains pires cas trouvés pour certaines fonctions dans certains domaines en double précision ($n = 53$ bits de mantisse). Nous rappelons qu'à chaque nombre réel y qui n'est ni un nombre machine, ni le milieu de deux nombres machine consécutifs, nous associons deux entiers m et k (vérifiant $m = n + k$) tels que la mantisse de y soit de la forme suivante :

$$\overbrace{\underbrace{1.xxx...xx}_{n \text{ bits}}\underbrace{r0000...00}_{k \text{ bits}}}^{m \text{ bits}}1...$$

ou

$$\overbrace{\underbrace{1.xxx...xx}_{n \text{ bits}}\underbrace{r1111...11}_{k \text{ bits}}}^{m \text{ bits}}0...$$

Dans les tableaux décrivant les pires cas d'une fonction f, nous avons successivement :

- l'exposant E de l'argument x ;
- la mantisse de l'argument x ;
- le mode d'arrondi pour lequel il s'agit d'un pire cas : D pour arrondi dirigé, N pour arrondi au plus près ;
- la valeur de m correspondant au résultat exact $f(x)$.

Note : il s'agit de résultats partiels, i.e. dans des domaines restreints, mais à l'intérieur de ces domaines, *tous* les pires cas vérifiant la condition sur m sont donnés. D'autre part, tous les résultats connus actuellement ne sont pas donnés ici (des domaines plus grands ont été testés, comme indiqué dans la section 3.4).

E	mantisse	R	m
-1	1.1010110011001111101111100100011010110100111011110000	N	**108**
-1	1.1101000010000110010101000011011010010100110001011010	N	**107**
0	1.1010110010100111101011101000110110100101101001111011	N	**107**
0	1.1101011000110011011010101000100000000111011110101010	D	**107**
2	1.0010011100111100000110001000101010100111101100010100	N	**107**
2	1.1000001111010100101111001101111010111011001111110100	D	**111**
4	1.0000100011110101000101000011010001100101001011000011	D	**107**
4	1.0001110101011100001011011010111010111110001101100111	D	**108**
4	1.1100010001001100111000001101011100010110101000011010	D	**108**

TAB. B.1: *Pires cas de* $\exp(x)$ *pour lesquels* $1/2 \leqslant x < 32$ *(exposants* -1 *à* 4*) et* $m \geqslant 107$ *(i.e.* $k \geqslant 54$*).*

E	mantisse	R	m
0	1.0010101010011100101011011001100110011000001001100010	D	**107**
0	1.1100110000110111111011110111111011110011101010000001	D	**107**
1	1.0000001000111001001111010101100101110110011101101001	N	**107**
3	1.0101010011001001001000011000101110101010010110000001	N	**107**
3	1.1100011110010100110111011100101111010110011000011010	N	**107**
3	1.1110100010111101101111111100110110010001010001001110	D	**108**

TAB. B.2: *Pires cas de* $\exp(-x)$ *pour lesquels* $1/2 \leqslant x < 16$ *(exposants* -1 *à* 3*) et* $m \geqslant 107$ *(i.e.* $k \geqslant 54$*).*

E	mantisse	R	m
2	1.0101101101101110011111100100111010010110111110000110	N	**107**
2	1.0110110111100001000100001100011110011111101011000101	N	**107**
2	1.1001000110101000110111111111010101000000111111110111	D	**108**
6	1.1110011001000010001101010100110000110100101000110100	D	**108**
7	1.1011110111010111100100111111111001111011010000011100	N	**108**
8	1.0001010110010101011010110010011100111100001101100101	N	**108**
8	1.1010110001010000101101000000100111001000101011101110	D	**114**
9	1.1000001111101100001001010011111111111010010001001110	D	**107**
13	1.1100011011101111000001110001000011000110110111101011	D	**107**
14	1.0010000100110010001000111011000001101000001101010110	D	**107**
14	1.1010101011001011100100011010101111010010101011100010	N	**107**
14	1.1111011000111110100010100000001011100011000000101010 1	D	**107**
15	1.0111001011110011001111010000011100110110111000110011	N	**108**
15	1.1010011001011010001101011011101001010100101110001010	N	**107**
16	1.1100011110110001100101001000001000110100111010101000	D	**107**
16	1.1100101100001000011010000101000100011001011110001000	N	**108**
16	1.1101111001111100110101100111010100010000001010011010	D	**109**

TAB. B.3: *Pires cas de* $\log(x)$ *pour lesquels* $1 < x < 2^{17}$ *(exposants* 0 *à* 16*) et* $m \geqslant 107$ *(i.e.* $k \geqslant 54$*).*

E	mantisse	R	m
-5	1.0110100110010100100110110011110101010001111110110001	D	**108**
-5	1.1001001010000011010110000110010100000011111111100000	D	**109**
-5	1.1101011110111101110011010111011110000000001001001111 1	D	**109**
-2	1.0001110001100011110111110110001111010101100111010000	N	**110**
-2	1.1110110100100101110001011110101110001100100100010110	D	**107**
-2	1.1111111001110110011101110011100111010000111101101101	D	**119**
-1	1.1101100110001100010011000110000100100111000110001101	N	**108**
0	1.1001001000011111101101010100010001000010110100011000	D	**108**

TAB. B.4: *Pires cas de* $\sin(x)$ *pour lesquels* $1/32 \leqslant x < 2$ *(exposants* -5 *à* 0*) et* $m \geqslant 107$ *(i.e.* $k \geqslant 54$*). Note : le dernier pire cas est particulier, car il correspond à* $\lfloor \pi/2 \rceil$ *; pour* $\lceil \pi/2 \rceil$*, on obtient* $m = 105$.

E	mantisse	R	m
-5	1.0110100101110110100011011100100010011011101100000000	D	**108**
-5	1.1001001001011001111000110111000010001011110100111010	D	**109**
-5	1.1010010010000000101101011001000000110011001110000011 1	N	**108**
-5	1.1101011101111011000100010111111110010001100001101011 0	D	**109**
-2	1.1101101001001110000011100110110001110001011110100101	D	**107**
-2	1.1110100110010101000001110011000011000100011010010110	D	**118**

TAB. B.5: *Pires cas de* $\arcsin(x)$ *pour lesquels* $1/32 \leqslant x \leqslant 1$ *(exposants* -5 *à* -1*) et* $m \geqslant 107$ *(i.e.* $k \geqslant 54$*).*

E	mantisse	R	m
-5	1.0101010010011110110000001100000011000101101011111010	D	**108**
-4	1.0001011011100101001101001110111000110110010110000000	N	**107**
0	1.0110101110001010011000100111001111010111110000100001	N	**108**

TAB. B.6: *Pires cas de* $\cos(x)$ *pour lesquels* $1/32 \leqslant x \leqslant 12867/8192$ *(un peu inférieur à* $\pi/2$*) et* $m \geqslant 107$ *(i.e.* $k \geqslant 54$*).*

E	mantisse	R	m
-11	1.1111110101110011011110111110100100010100010101111000	D	**116**

TAB. B.7: *Pire cas de* $\arccos(x)$ *pour* $\cos(12867/8192) < x < \cos(1/32)$*. Pour toutes les autres valeurs de* x *de ce domaine, on a* $m < 116$.

Bibliographie

[1] P. Alessandri et V. Berthé. Three distance theorems and combinatorics on words. *L'Enseignement Mathématique - Revue Internationale*, 44:103–132, 1998.

[2] D. H. Bailey. A high-performance FFT algorithm for vector supercomputers. *Journal of Supercomputing*, novembre 1987.
URL : *http://www.nas.nasa.gov/NAS/TechReports/RNRreports/dbailey/fftzp/fftzp.ps*

[3] D. H. Bailey. The computation of pi to 29,360,000 decimal digits using Borweins' quartically convergent algorithm. *Mathematics of Computation*, 50(181):283–296, janvier 1988.
URL : *http://www.nas.nasa.gov/Pubs/TechReports/RNRreports/dbailey/pi/pi.ps*

[4] D. H. Bailey. Multiprecision translation and execution of fortran programs. *ACM Transactions on Mathematical Software*, 19(3):288–319, septembre 1993.

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

[6] R. Bernstein. Multiplication by integer constants. *Software – Practice and Experience*, 16(7):641–652, juillet 1986.

[7] V. Berthé. Étude arithmétique et dynamique de suites algorithmiques. Habilitation à diriger des recherches, Université de la Méditerranée, juin 1999.

[8] G. Brassard, S. Monet, et D. Zuffellato. Algorithmes pour l'arithméthique des très grands entiers. *Technique et Science Informatiques*, 1985.

[9] R. P. Brent. Fast multiple precision evaluation of elementary functions. *Journal of the ACM*, 23:242–251, 1976.

[10] R. P. Brent. Multiple-precision zero-finding methods and the complexity of elementary function evaluation. Dans J.F. Traub, editor, *Analytic Computational Complexity*, pages 151–176. Academic Press, New York, 1976.

[11] C. Burnikel et J. Ziegler. Fast recursive division. Rapport de recherche MPI-I-98-1-022, MPI Saarbrücken, octobre 1998.

[12] S. A. Cook. On the minimum computation time of functions. Thèse, Harvard Univ., mai 1966.

[13] M. Daumas. Multiplications of floating point expansions. Dans *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, 1999.

URL : *ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-39.ps.Z*

[14] M. Daumas et C. Finot. Division of floating point expansions. Dans *IMACS-GAMMM International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, pages 45–46, Budapest, Hungaria, 1998.

[15] M. Daumas et C. Finot. Division of floating point expansions. Rapport de recherche RR1999-03, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

[16] T. J. Dekker. A floating-point technique for extending the available precision. *Numr. Math.*, 18:224–242, 1971.

[17] C. B. Dunham. Feasibility of "perfect" function evaluation. *ACM Sigum Newsletter*, 25(4):25–26, octobre 1990.

[18] A. Feldstein et R. Goodman. Convergence estimates for the distribution of trailing digits. *Journal of the ACM*, 23:287–297, 1976.

[19] S. Gal. Computing elementary functions: a new approach for achieving high accuracy and good performance. Dans Springer-Verlag, editor, *Accurate Scientific Computations. Lecture Notes in Computer Science, No 253*, volume 235, pages 1–16, 1986.

[20] S. Gal et B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, mars 1991.

[21] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, mars 1991.

[22] T. Granlund. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 2.0.2 edition, juin 1996.
URL : *http://www.swox.com/gmp/*

[23] C. Iordache et D. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. Dans Jean-Michel Muller, editor, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, pages 233–240. IEEE Computer Society Press, 1999.

[24] W. Kahan. Paradoxes in concepts of accuracy. Dans *Lecture notes from Joint Seminar on Issues and Directions in Scientific Computations, U.C. Berkeley*, 1989.

[25] W. Kahan. A test for correctly rounded SQRT. Technical report, Elect. Engineering and Computer Science, Univ. of California at Berkeley, 1996.
URL : *http://HTTP.CS.Berkeley.EDU/~wkahan/SQRTest.ps*

[26] A. Karatsuba et Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Phys. Doklady*, 7(7):595–596, janvier 1963.

[27] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1973.

[28] U. W. Kulisch. Mathematical foundation of computer arithmetic. *IEEE Transactions on Computers*, C-26(7):610–621, juillet 1977.

[29] U. W. Kulisch et W. L. Miranker. *Computer Arithmetic in Theory and Practice*. Academic Press, 1981.

[30] V. Lefèvre. 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, France, 1997.

URL : *ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1997/RR1997-18.ps.Z*

[31] V. Lefèvre. An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. Dans *Developments in Reliable Computing*, pages 203–212. Kluwer, Dordrecht, Netherlands, 1999.

[32] V. Lefèvre. Multiplication by an integer constant. Rapport de recherche RR1999-06, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1999.

URL : *ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-06.ps.Z*

[33] V. Lefèvre et J.-M. Muller. Table methods for the elementary functions. Dans F.T. Luk, editor, *Proceedings of the SPIE — The International Society for Optical Engineering*, volume 3807, pages 43–49, Denver, Colorado, 1999.

[34] V. Lefèvre, J.-M. Muller, et A. Tisserand. Towards correctly rounded transcendentals. Dans *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, Asilomar, USA, 1997. IEEE Computer Society Press, Los Alamitos, CA.

[35] V. Lefèvre, J.-M. Muller, et A. Tisserand. The table maker's dilemma. Rapport de recherche RR1998-12, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1998.

URL : *ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/RR/RR1998/RR1998-12.ps.Z*

[36] V. Lefèvre, J.-M. Muller, et A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, novembre 1998.

[37] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[38] J.-M. Muller et A. Tisserand. Towards exact rounding of the elementary functions. Dans Frommer Alefeld et Lang, editors, *Scientific Computing and Validated Numerics (proceedings of SCAN'95)*. Akademie Verlag, 1996.

[39] Yu. V. Nesterenko et M. Waldschmidt. On the approximation of the values of exponential function and logarithm by algebraic numbers (in Russian). *Mat. Zapiski*, 2:23–42, 1996.

[40] M. Parks. Number-theoretic tests for directed rounding. Dans Jean-Michel Muller, editor, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, 1999.

URL : *http://euler.ecs.umass.edu/paper/final/paper-131.ps ;*
http://euler.ecs.umass.edu/paper/final/paper-131.pdf

[41] M. Pichat. Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, 19:400–406, 1972.

[42] D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. Dans P. Kornerup et D. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 132–144. IEEE Computer Society Press, juin 1991.

[43] M. Schulte et E. E. Swartzlander. Exact rounding of certain elementary functions. Dans M.J. Irwin E.E. Swartzlander et G. Jullien, editors, *Proceedings of the 11th IEEE Symposium on Computer Artithmetic*, pages 138–145, Los Alamitos, CA, juin 1993. IEEE Computer Society Press.

[44] M. J. Schulte et E. E. Swartzlander. Hardware designs for exactly rounded elementary functions. *IEEE Transactions on Computers*, 43(8):964–973, août 1994.

[45] J. R. Shewchuk. Robust adaptative floating-point geometric predicates. Dans *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 141–150, Philadelphia, Pensylvania, 1996.
URL : *http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-predicates.ps*

[46] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. Dans *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.
URL : *http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps*

[47] V. T. Sós. On the distribution mod 1 of the sequence $n\alpha$. *Ann. Univ. Sci. Budapest, Eötvös Sect. Math.*, 1:127–134, 1958.

[48] J. Surányi. Über die Anordnung der Vielfachen einer reellen Zahl mod 1. *Ann. Univ. Sci. Budapest, Eötvös Sect. Math.*, 1:107–111, 1958.

[49] S. Swierczkowski. On successive settings of an arc on the circumference of a circle. *Fundamenta Math.*, 46:187–189, 1958.

[50] M. Tommila. Apfloat.
URL : *http://www.jjj.de/mtommila/apfloat/*

[51] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Math.*, 3:714–716, 1963.

[52] P. Zimmermann. Karatsuba square root. 1999.
URL : *http://www.loria.fr/~zimmerma/bignum/sqrtrem.ps.gz*

[53] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, septembre 1991.

[54] D. Zuras. More on squaring and multiplying large integers. *IEEE Transactions on Computers*, 43(8):899–908, août 1994.