Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Calcul certifié des fonctions élémentaires

Vincent Lefèvre

vlefevre@ens-lyon.fr

Responsable de stage : Jean-Michel Muller

4 mars 1996 – 20 juin 1996

Résumé

Etre toujours capable d'arrondir exactement les fonctions élémentaires, comme l'assure la norme IEEE-754 pour les opérations arithmétiques, était un problème ouvert. Des résultats récents montrent qu'il peut être résolu, au prix éventuel de calculs sur des millions de chiffres. Nous étudions deux approches : le lancement de calculs exhaustifs en double précision pour montrer empiriquement que le problème est soluble à coût raisonnable parce que le nombre de bits nécessaire pour assurer l'arrondi exact est en fait faible, et pour calculer ce nombre ; la mise au point d'algorithmes performants pour effectuer les calculs sur de très grands nombres si jamais cela est nécessaire.

Mots-clés : arithmétique virgule flottante, fonctions élémentaires, arrondis, multiprécision.

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens–lyon.fr

Je tiens particulièrement à remercier l'équipe SAAO pour la bonne ambiance qui a régné tout au long de ce stage et pour l'aide que chacun m'a apportée, ainsi que tous les utilisateurs qui nous donnent de leur temps CPU pour pouvoir effectuer les tests exhaustifs.

Table des matières

1 Introduction **2**

2 Le problème de l'arrondi des fonctions élémentaires **4**

2.1 Le dilemme du fabricant de tables 4

2.2 La stratégie multi-niveaux de Ziv 5

2.3 Une approche probabiliste 5

2.4 Une borne théorique pour l'exponentielle 6

3 Tests exhaustifs **7**

3.1 Introduction 7

3.2 Formulation du problème 7

3.3 Algorithme (idée générale) 8

3.4 Première étape 8

3.4.1 Description générale 8

3.4.2 Calcul et test des e^{t+x}, connaissant e^t 9

3.4.3 Calcul des e^t 12

3.4.4 Arrondi de d_0, d_1 et d_2 (avec $r = 16$) 15

3.4.5 Implémentation (sur Sparc) 16

3.5 Parallélisation sur réseau de stations 17

3.5.1 Objectifs 17

3.5.2 Méthode choisie 18

3.5.3 Problèmes rencontrés 20

3.6 Seconde étape 20

3.7 Quelques résultats 21

4 Calcul des fonctions élémentaires en multiprécision **22**

4.1 Introduction 22

4.2 Type de données utilisé et choix de la base 22

4.3 Multiplication 23

4.3.1 Introduction 23

4.3.2 Algorithme en n^2 23

4.3.3 Algorithme de Karatsuba (méthode 2-way) 23

4.3.4 Généralisation (méthode k-way) 24

4.3.5 FFT 24

4.3.6 Résultats sur une SparcStation 5 à 85 MHz 25

4.4 Fonctions algébriques 25

4.5 Fonctions transcendantes 25

5 Conclusion **27**

Chapitre 1

Introduction

Les systèmes à virgule flottante sont presque toujours utilisés pour représenter les nombres réels dans les ordinateurs. Ce choix vient du fait qu'ils sont un bon compromis entre l'amplitude des valeurs représentables, la précision et la simplicité des algorithmes. Une étude sur l'arithmétique à virgule flottante est présentée dans [Gol91].

Dans un système à virgule flottante en base 2, de longueur de mantisse n et d'exposants allant de $e_{\min}$ à $e_{\max}$, un nombre x est représenté par une *mantisse* $m_x = x_0.x_1x_2\ldots x_{n-1}$, qui est un nombre en base 2 à n chiffres vérifiant $0 \leqslant m_x < 2$, un *signe* $s_x = \pm 1$ et un *exposant* e_x, entier compris entre $e_{\min}$ et $e_{\max}$:

$$x = s_x \times m_x \times 2^{e_x}.$$

Pour garder le maximum de précision, la représentation doit être *normalisée* : la mantisse doit toujours être supérieure ou égale à 1, et le premier bit de la mantisse (toujours 1) n'a pas besoin d'être représenté en mémoire. Par conséquent, une représentation spéciale doit être choisie pour zéro.

On appelle *nombre machine* un nombre qui peut être exactement représenté dans le système à virgule flottante utilisé. En général, la somme, le produit ou le quotient de deux nombres machine n'est pas un nombre machine : le résultat de l'opération doit être *arrondi*.

La norme IEEE-754 est de plus en plus souvent implémentée. Elle définit la représentation ($n = 24$ pour la simple précision, $n = 53$ pour la double précision), mais aussi l'arrondi. L'utilisateur a le choix entre les quatre *modes d'arrondi* suivants :

- arrondi vers $-\infty$: $\nabla(x)$ est le plus grand nombre machine inférieur ou égal à x ;
- arrondi vers $+\infty$: $\Delta(x)$ est le plus petit nombre machine supérieur ou égal à x ;
- arrondi vers 0 : $\mathcal{Z}(x)$ est, en valeur absolue, le plus grand nombre machine inférieur ou égal à x ;
- arrondi au plus proche : $\mathcal{N}(x)$ est le nombre machine le plus proche de x (avec une convention spéciale si x est exactement entre deux nombres machine).

Supposons maintenant que le mode d'arrondi actif soit $\diamond$. 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 $\diamond(\sqrt{x})$, i.e. l'*arrondi du résultat exact*. Les opérations vérifiant cette propriété sont dites "exactement arrondies". Une telle propriété a des conséquences très avantageuses :

- elle assure une compatibilité totale : le même programme donnera les mêmes résultats sur des ordinateurs différents ;
- de nombreux algorithmes peuvent utiliser cette propriété, par exemple pour avoir une précision arbitraire [Pri91] ou pour obtenir l'arrondi d'une somme exacte de plusieurs nombres à virgule flottante [Pic72, Kah89] ;
- on peut facilement implémenter une arithmétique d'intervalles [Kul77, KM81], ou plus généralement on peut obtenir une borne inférieure ou supérieure d'un résultat exact d'une séquence d'opérations arithmétiques.

Malheureusement, il n'existe pas encore de norme semblable pour les fonctions élémentaires (exp, log, sin, cos, tan, arctan), car le problème est bien plus complexe. Pour ces fonctions, une étude est effectuée dans [MT96], résumée dans le chapitre 2. L'arrondi exact (suivant l'un des quatre modes d'arrondi) est toujours possible, mais cela peut demander d'effectuer des calculs intermédiaires à très grande précision (cela ne se produit jamais avec les opérations $+$, $-$, $\times$, $\div$ ou $\sqrt{\ }$). Un résultat récent de Yu. Nesterenko et M. Waldschmidt [NW95] fournit une borne supérieure théorique de la précision requise lors des calculs intermédiaires pour le calcul de l'exponentielle, qui est de l'ordre du million de bits pour les nombres en double précision, mais le maximum est certainement beaucoup plus petit (de l'ordre de la centaine de bits). On va donc considérer les deux approches suivantes :

- Tests exhaustifs (chapitre 3). Ces tests permettront de connaître la vraie borne supérieure (au moins, provisoirement, dans un intervalle donné), et de diminuer la borne sur la durée de l'opération et la mémoire utilisée. Cette approche demande beaucoup de temps de calcul pour tester tous les éléments, mais les algorithmes seront très rapides. Avec la puissance de calcul actuelle, l'ensemble des nombres représentables en simple précision est traité rapidement, et il faudra beaucoup de temps (sur de nombreuses machines) pour les nombres en double précision. En revanche, cette méthode est impossible si l'on considère les nombres en quadruple précision, et le restera certainement dans le futur. C'est pourquoi une seconde approche est nécessaire.
- Calculs en multiprécision (chapitre 4). L'inconvénient de cette méthode est que le temps pris pour le calcul demandé est bien plus grand que celui d'un calcul à la précision du nombre. Il y a une très grande probabilité pour que la vraie borne supérieure soit suffisamment petite pour qu'un calcul à très haute précision ne puisse en fait jamais être demandé ; mais il faut quand-même écrire des routines suffisamment optimisées pour que le temps de calcul soit acceptable si jamais ce cas se produit. On fera aussi attention à ne pas utiliser trop de mémoire. Les routines devront être portables et la précision des résultats devra être garantie. Il existe des bibliothèques de calcul en multiprécision, mais elles ne vérifient pas toutes les conditions requises.

Pour éviter les problèmes décrits ci-dessus, on pourrait choisir une norme un peu plus faible que celle demandant l'arrondi exact. Mais il serait alors difficile d'en choisir une, car il n'y aurait plus de norme "naturelle" et on aurait toujours le risque que quelqu'un fasse "mieux" que la norme et ainsi tue la norme.

Chapitre 2

Le problème de l'arrondi des fonctions élémentaires

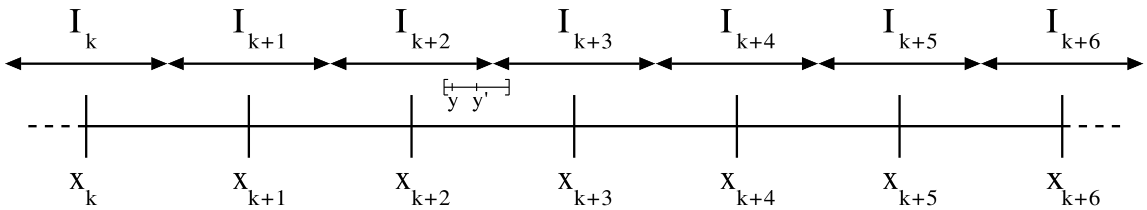
2.1 Le dilemme du fabricant de tables

Pour bien comprendre que le calcul à grande précision est inévitable, on va d'abord considérer le problème géométriquement, en faisant abstraction de la représentation en base 2.

Soit un nombre machine x, et $y = f(x)$, où f est une fonction élémentaire (exp, log, sin, etc...). On cherche à arrondir exactement y en un nombre machine, suivant l'un des quatre modes d'arrondi, fixé ici.

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 sache calculer y à une précision égale à ε'. Cette valeur calculée doit être à une distance de la frontière d'au plus ε' pour être sûr de pouvoir arrondir correctement (en effet, on ne sait pas où se situe la vraie valeur). Pour 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 on ne saura pas toujours 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. 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}.



Supposons que l'on veuille $f(x)$ arrondi à n bits, sachant qu'on peut le calculer avec une erreur de 2^{-m}. Ce ne sera pas possible dans les cas suivants :

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

$$\overbrace{\underbrace{1.xxxx\ldots xx}_{n \text{ bits}}0000\ldots00}^{m \text{ bits}}xxx\ldots$$

ou

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

– avec arrondi au plus proche :

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

ou

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

Ce problème est connu sous le nom du *Table Maker's Dilemma* (TMD), ou *dilemme du fabricant de tables*. Des exemples sont donnés dans la section 3.7.

2.2 La stratégie multi-niveaux de Ziv

La stratégie de Ziv [Ziv91] consiste à choisir une valeur de m petite, mais un peu plus grand que n ($m = n+20$ par exemple), et à calculer la fonction avec une erreur de 2^{-m}. Dans la plupart des cas, cette approximation est suffisante pour pouvoir arrondir correctement. Si cela ne suffit pas, on recommence avec une valeur de m plus grande. Si cela n'est toujours pas suffisant, on prend une valeur de m encore plus grande, et ainsi de suite.

Le problème est de savoir si le processus se termine. En 1882, Lindemann a démontré que l'exponentielle d'un nombre algébrique (éventuellement complexe) non nul n'est pas algébrique [Bak75]. 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, et le logarithme d'un nombre machine différent de 1 n'est pas un nombre machine. Dans ces cas (on ne considère pas ici les cas triviaux 0 ou 1), 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 est en fait bornée. Le problème est donc de trouver un majorant de m.

2.3 Une approche probabiliste

Soit f une fonction élémentaire. On suppose que pour x nombre machine à n bits de mantisse, les bits de $f(x)$ suivant les n premiers bits du résultat peuvent être considérés comme une suite aléatoire de 0 et de 1, avec une probabilité de $\frac{1}{2}$ pour chacun de ces deux chiffres ; ceci peut être vu comme une application des résultats de Feldstein et Goodman [FG76]. On suppose de plus que les suites associées à deux nombres différents x_1 et x_2 peuvent être considérées indépendantes. Ces suppositions seront aussi utilisées pour les tests exhaustifs (chapitre 3) afin de prévoir certains résultats et d'adapter les programmes en conséquence. Notons qu'elles ne sont plus vérifiées dans certains domaines : voisinage de 0 pour l'exponentielle, le sinus et le cosinus, x suffisamment grand pour l'exponentielle (qui donne un overflow)... mais dans ces domaines, l'arrondi ne pose pas de problème.

On pose $k = m - n$. La probabilité que le TMD se produise est alors de 2^{1-k}, pour un mode d'arrondi fixé. Soit n_e le nombre de couples (signe,exposant) considérés (nombre total, ou seulement dans un certain domaine), ce qui donne $N = n_e \times 2^{n-1}$ nombres flottants possibles. Le TMD se produira alors "en moyenne" $2^{n-k+\log_2(n_e)}$ fois sur l'ensemble de ces nombres.

Si on prend m un peu supérieur à $2n + \log_2(n_e)$, le TMD aura donc très peu de chances de se produire.

Cette approche probabiliste est vérifiée en pratique sur des tests exhaustifs en simple précision, et des tests aléatoires en double et quadruple précision.

2.4 Une borne théorique pour l'exponentielle

Grâce à un théorème de Yu. Nesterenko et M. Waldschmidt [NW95], nous avons les bornes suivantes pour l'exponentielle :

n	arguments	m
24	$0 \to \ln 2$	494 416
	$0 \to 10$	3 074 888
53	$0 \to \ln 2$	1 038 560
	$0 \to 10$	5 234 891
112	$0 \to \ln 2$	2 527 507
	$0 \to 10$	10 409 113

On peut en déduire des bornes similaires pour le logarithme. Les résultats sur l'exponentielle se généralisent aux sinus et cosinus.

Il s'agit d'un théorème assez récent et nous espérons qu'il soit amélioré afin d'avoir des bornes théoriques plus petites. Mais d'un autre côté, ces bornes resteront toujours assez grandes, et les meilleures bornes que nous puissions obtenir seront trouvées par tests exhaustifs. De tels tests seront hélas impossibles en précision étendue (80 bits de mantisse) ou en quadruple précision (112 bits de mantisse).

Chapitre 3

Tests exhaustifs

3.1 Introduction

Une des approches pour garantir à coût raisonnable l'arrondi exact d'une fonction élémentaire f est la recherche par tests exhaustifs de la valeur minimale de m pour laquelle le TMD ne se produit jamais. En fait, on se donnera un nombre m_0 de l'ordre de $2n+\log_2(n_e)$ (cf 2.3), et on recherchera les arguments x pour lesquels le TMD se produit si l'on calcule $f(x)$ avec une erreur de 2^{-m_0}. Pour ces éléments, qui devraient être peu nombreux, on pourra calculer $f(x)$ très rapidement à l'aide d'une table ; pour tous les autres, on pourra utiliser la stratégie de Ziv (cf 2.2), limitée à m_0. Le calcul de f avec arrondi exact sera ainsi très rapide dans tous les cas.

Il s'agira en fait de tests exhaustifs restreints à un intervalle donné, en dehors duquel on peut trouver d'autres méthodes. En effet, si x est suffisamment petit (inférieur à 2^{-53}), on peut utiliser la formule de Taylor à l'ordre 1 ; si x est suffisamment grand, l'exponentielle donne un overflow, et les fonctions trigonométriques n'ont plus beaucoup de sens.

Le cas des nombres en simple précision a déjà été traité par J.-M. Muller et A. Tisserand [MT96]. Nous nous intéresserons ici à l'exponentielle avec les arguments en double précision situés dans l'intervalle $[\frac{1}{2}, 1[$, ce qui correspond aux nombres positifs de mantisse quelconque et d'exposant -1. Dans le futur, d'autres intervalles seront testés.

Pour les flottants en double précision ($n = 53$), il y a 2^{52} mantisses possibles au total (le premier bit étant toujours 1), ce qui est énorme. Une des plus grandes préoccupations sera donc la rapidité du test, aux dépens de la portabilité. En effet, un cycle par test correspond à deux ans de calculs sur une machine à 71 MHz. Le moindre cycle est important !

Les calculs se feront sur des Sun SparcStation ainsi que sur une machine Matra Capitan à base de processeurs Sparc. Nous en tiendrons compte pour faire certains choix (mais la plupart resteraient valables pour d'autres types de machines).

3.2 Formulation du problème

Considérons les nombres positifs en double précision d'exposant -1. Ce sont les nombres x de la forme $0.1x_2x_3\ldots x_{53}$ où $x_i \in \{0,1\}$. La mantisse de $y = \exp x$ se compose de 53 bits (dont le premier est 1) suivis d'une suite de bits $b = b_1b_2b_3\ldots$, qui sera calculée à une certaine précision. Le problème revient à trouver cette précision de manière à ce que le TMD ne se produise pas. Ceci revient à trouver l'entier $k \geqslant 0$ tel que

- $b = 00^k1\ldots$ ou $b = 11^k0\ldots$ pour un arrondi dirigé,
- $b = 01^k0\ldots$ ou $b = 10^k1\ldots$ pour un arrondi au plus proche.

Nous ne stockerons que les arguments pour lesquels $k \geqslant k_0$, où k_0 est un entier fixé. En appliquant la formule (probabiliste) donnée en 2.3 ($n \leftarrow 53$, $k \leftarrow k+1$, $n_e \leftarrow 1$), nous stockerons environ 2^{53-k_0} arguments (2^{52-k_0} pour chacun des deux types de modes d'arrondi considérés). Nous prendrons donc $k_0 \approx 50$.

3.3 Algorithme (idée générale)

Pour être rapide, nous allons appliquer une stratégie du style de celle de Ziv. Nous ferons le test en deux étapes : la première doit être très rapide et éliminer un très grand nombre de cas, ceux pour lesquels k est suffisamment petit ; la seconde, qui peut être bien plus lente, consiste à retester les cas qui ont échoué à la première étape (que nous appellerons *exceptions*), en calculant l'exponentielle à une plus grande précision.

Le résultat de l'exponentielle est compris entre $e^{1/2}$ et e. Le poids du premier bit de b est donc 2^{-53} ou 2^{-52}, suivant que $x < \log 2$ ou que $x \geqslant \log 2$. Pour simplifier, nous n'allons pas couper le test en deux, et nous ne testerons que des bits de même poids. Nous ne testerons donc pas le bit de poids 2^{-52} ; il ne sera testé que lors de la seconde étape, si le premier test échoue. D'après les hypothèses probabilistes, cela n'aura aucune influence sur le temps de calcul de la première étape et le nombre d'arguments qui devront passer la seconde étape.

D'autre part, puisque les deux types d'arrondi sont considérés simultanément, Nous ne testerons pas le premier bit de b. Il pourra être pris en compte plus tard (après la seconde étape), si on veut connaître la valeur de k pour chaque type d'arrondi. Le premier bit testé sera alors celui de poids 2^{-54}, et si l'exponentielle est calculée à une précision 2^{-n}, le dernier bit testé sera celui de poids $2^{-(n-1)}$. Le test échoue si et seulement si ces bits sont tous égaux ($000\ldots0$ ou $111\ldots1$), et l'argument devra passer la seconde étape. Dans le cas contraire, on montre facilement que les bits 54 à $n+1$ du résultat exact ne peuvent pas être tous égaux.

La durée moyenne d'un test augmente avec le nombre de bits à calculer et à tester. Il vaut donc mieux tester peu de bits, mais suffisamment de façon à ne pas passer trop de temps sur la seconde étape. Si nous testons p bits, il y a une chance sur 2^{p-1} que le test échoue. Comme nous devons tester 2^{52} arguments, environ 2^{53-p} arguments, à un ordre de grandeur près, devront passer la seconde étape.

3.4 Première étape

3.4.1 Description générale

Différentes méthodes ont été étudiées pour "calculer des exponentielles" (dans le sens "calculer les 2^p bits à tester" d'une approximation à 2^{-n} près). Celle exposée ici semble être la plus rapide.

Nous allons calculer l'exponentielle par approximation polynomiale. Le polynôme choisi devra être de faible degré pour les raisons suivantes : d'une part pour réduire le temps de calcul, d'autre part à cause des erreurs d'arrondi. En contre partie, l'approximation ne sera valable que sur un petit intervalle.

Considérons d'abord un intervalle du type $[-2^{r-53}, 2^{r-53}[$, où r est un entier positif (il sera de l'ordre de 15), pour lequel on connaît un polynôme d'approximation. On pourra utiliser la formule

$$e^{t+x} = e^t.e^x$$

où x appartient à l'intervalle, pour tester tous les arguments compris entre $1/2$ et 1.

Une première idée consiste à calculer tous les e^x et e^t, puis à effectuer les 2^{52} produits possibles à l'aide de plusieurs multiplications en double précision, en fait seulement celles intervenant dans la valeur des bits à tester, ce qui revient à faire une ou deux convolutions. Le principal problème avec la multiplication est que la taille du résultat est double de la taille des opérandes, ce qui complique les calculs. Mais même si une multiplication se fait en un seul cycle, on peut trouver une méthode plus rapide.

La seconde idée est de calculer une approximation polynomiale de l'exponentielle dans les intervalles $[t-2^{r-53}, t+2^{r-53}]$, puis d'évaluer le polynôme sur les valeurs consécutives par table des différences. Cette méthode est avantageuse, car elle ne contient que des additions (deux par argument dans le cas d'un polynôme de degré deux) et les calculs peuvent être faits modulo 2^{-53} (le premier bit testé étant celui de poids 2^{-54}).

L'algorithme se décompose en deux parties :

- le calcul des e^t à une précision 2^{-s}, où t sera de la forme $(2\ell+1)2^{r-53}$ (et compris entre $1/2$ et 1),
- le calcul des e^{t+x} pour $x \in [-2^{r-53}, 2^{r-53}]$ à la précision 2^{-n}, connaissant e^t à la précision 2^{-s}.

(n et s seront déterminés plus loin.)

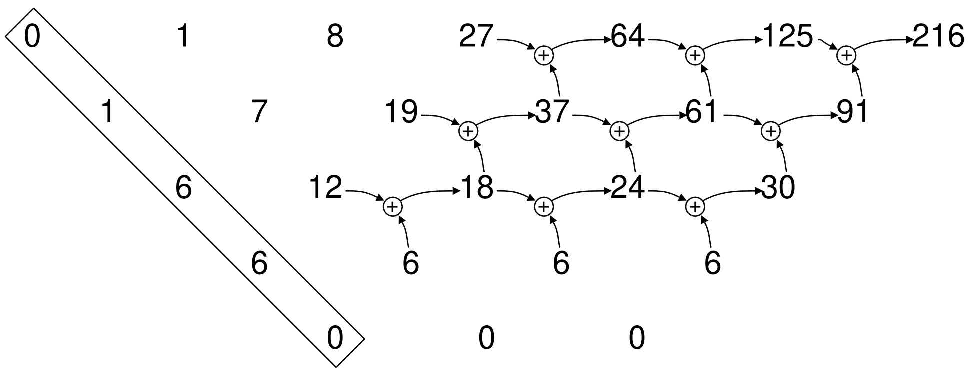
Commençons d'abord par étudier le calcul et le test des e^{t+x}, qui devrait être la partie la plus lente.

3.4.2 Calcul et test des e^{t+x}, connaissant e^t

Table des différences

Le calcul des e^{t+x} se fera par la méthode des différences, qui 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 avec un exemple simple : $P(x) = x^3$.

On calcule d'abord $P(0)$, $P(1)$, ..., $P(d)$ (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 $y - x$. 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.



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

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

(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 : c'est le plus pratique, donc c'est le choix que nous ferons pour le calcul des e^{t+x}.

Application à l'exponentielle

Soit $x = k.2^{-53}$, avec $k \in \mathbb{Z}$ tel que $|k| \leqslant k_0 = 2^r$. Considérons l'approximation

$$e^x = 1 + x + \frac{1}{2}x^2 + \varepsilon(x) \quad \text{où} \quad \varepsilon(x) = \frac{1}{6}x^3 + \frac{1}{24}x^4 + \ldots$$

(nous pourrions choisir un polynôme de degré 2 un peu meilleur, mais nous verrons que celui-ci suffit, i.e. que la précision est limitée par une autre raison). En faisant intervenir k et la base de polynômes $\{1, k, \frac{k(k-1)}{2}\}$ (utilisée pour la table des différences), on obtient :

$$e^x = 1 + (2^{-53} + 2^{-107})k + 2^{-106}\frac{k(k-1)}{2} + \varepsilon(x).$$

On pose $u = e^t$, dont on sait calculer une approximation $\hat{u}$ telle que $|u - \hat{u}| \leqslant 2^{-s}$ (cf section 3.4.3). On a alors :

$$e^{t+x} = u.e^x = \hat{u} + (2^{-53} + 2^{-107})\hat{u}k + 2^{-106}\hat{u}\frac{k(k-1)}{2} + \hat{u}.\varepsilon(x) + (u - \hat{u}).e^x\,.$$

Le calcul des exponentielles (modulo 2^{-53}) dans l'intervalle choisi fait intervenir les trois nombres $d_0 = \hat{u}$, $d_1 = (2^{-53} + 2^{-107})\hat{u}$ et $d_2 = 2^{-106}\hat{u}$. Une étape de calcul dans le sens positif ($k = 1, 2, 3$, etc...) consiste à effectuer $d_1 \leftarrow d_1 + d_2$ puis $d_0 \leftarrow d_0 + d_1$; dans le sens négatif ($k = -1, -2, -3$, etc...), ce sont des soustractions.

Nous allons arrondir ces nombres de façon à diminuer les calculs. Ceci dépend du type de données utilisé ici, qui va être choisi maintenant. Sur Sparc, on a le choix entre flottants en simple, double ou quadruple précision, et entiers (sur 32 bits). Les flottants en simple précision ne conviennent pas, car ils n'ont pas assez de précision et ne sont pas plus rapides que les flottants en double précision. Les flottants en quadruple précision sont émulés donc lents ; par conséquent, ils ne sont pas intéressants. Les flottants en double précision pourraient sembler intéressants si chaque nombre est stocké sur un seul flottant à des précisions absolues différentes, mais pour certaines raisons (modulo, test des bits, pas de multiplication) les entiers sont préférables.

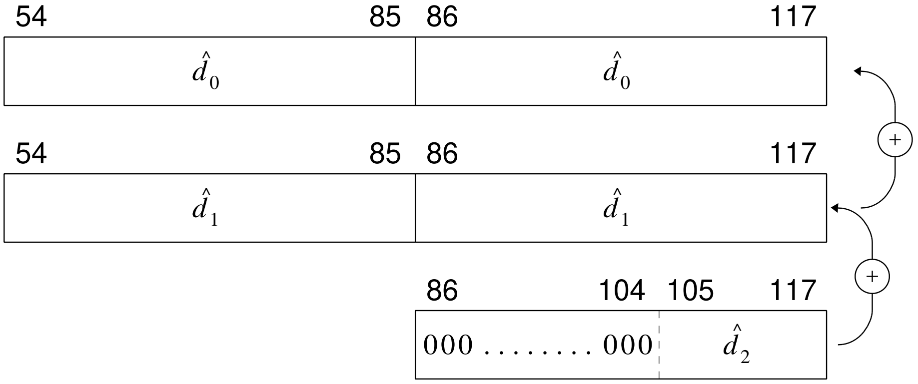
Les 3 coefficients seront arrondis et stockés sur 64 bits (bits 54 à 117) pour d_0 et d_1, sur 32 bits (bits 86 à 117) pour d_2 ; les nombres arrondis sont notés $\hat{d}_0$, $\hat{d}_1$ et $\hat{d}_2$.

On a alors :

$$e^{t+x} = \hat{d}_0 + \hat{d}_1 k + \hat{d}_2\frac{k(k-1)}{2} + \varepsilon'(t,x)$$

avec

$$\varepsilon'(t,x) = (d_0 - \hat{d}_0) + (d_1 - \hat{d}_1)k + (d_2 - \hat{d}_2)\frac{k(k-1)}{2} + \hat{u}.\varepsilon(x) + (u - \hat{u}).e^x\,.$$



$\varepsilon'(t, x)$ est l'erreur que l'on fait. Nous allons en chercher une majoration. Notons tout de suite que l'erreur maximale est de l'ordre de celle due au terme de degré 2. Par conséquent, si on veut un résultat plus précis, il faut que l'addition $d_1 \leftarrow d_1 + d_2$ se fasse à la précision 2^{-149} au lieu de 2^{-117} (mais la précision considérée ici se révélera suffisante).

Majorons d'abord $|\varepsilon(x)|$, sachant que $x \leqslant \frac{1}{3}$:

$$|\varepsilon(x)| \leqslant \frac{1}{6}(x^3 + x^4 + x^5 + \ldots) = \frac{x^3}{6(1-x)} \leqslant \frac{x^3}{4}.$$

Ensuite, puisque $e^x \leqslant \frac{3}{2}$, alors $|u - \hat{u}|.e^x \leqslant \frac{3}{2}2^{-s}$. D'autre part, puisque nous faisons les calculs à la précision 2^{-117}, nous prendrons $s < 117$ et nous pouvons arrondir d_0 de telle sorte que $|\hat{d}_0 - \hat{u}| \leqslant 2^{-s-1}$.

Nous allons choisir les arrondis de manière à ce qu'ils se compensent partiellement. Pour d_2, nous prenons évidemment un arrondi au plus proche, si bien que $|d_2 - \hat{d}_2| \leqslant 2^{-118}$. Pour d_1, l'arrondi est choisi de telle sorte que $\operatorname{sign}(d_1 - \hat{d}_1) \neq \operatorname{sign}((d_2 - \hat{d}_2)k)$, à une précision 2^{r-120}. Dans ces conditions, on a :

$$\begin{aligned} |\varepsilon'| &\leqslant \max(2^{2r-119} - 2^{r-119}, 2^{2r-120}) + 2^{3(r-53)} + 2^{1-s} \\ &\leqslant 2^{2r-119} + 2^{1-s} - (2^{r-119} - 2^{3r-159}). \end{aligned}$$

Nous choisirons r et s tels que $r \leqslant 20$ et $s \geqslant 120 - 2r$, si bien que $|\varepsilon| \leqslant 2^{2r-118}$. D'autre part, nous prendrons le plus petit s vérifiant les contraintes, d'où $s = 120 - 2r$. Puisque $r \simeq 15$, nous aurons $s \simeq 90$.

Nous prendrons $-k_0 \leqslant k < k_0$, de manière à tester tous les $1/2 \leqslant x + t < 1$.

Test des bits (sur Sparc)

Nous cherchons à tester très rapidement si les p bits de poids fort de d_0 sont tous égaux. Si $p > 32$, nous pouvons d'abord tester les 32 premiers bits (registre de poids fort de d_0), et nous ne nous occuperons pas ici du test des bits suivants, qui peut être relativement lent car cela n'arrive en moyenne qu'une fois sur 2^{31}. Supposons donc que $3 \leqslant p \leqslant 32$.

Le jeu d'instructions du Sparc ne permet pas d'effectuer le test en une seule instruction (même dans le cas $p = 32$). Deux instructions sont nécessaires et suffisantes. Mais l'astuce suivante permet de n'utiliser qu'une seule instruction (donc de gagner un cycle) : le registre à tester contiendra en permanence le vrai résultat + 2^{32-p} (les additions restant valides). Le test sera alors une comparaison non signée à 2^{33-p}.

Après l'instruction de test se trouve une instruction de branchement sur une routine qui doit stocker la valeur de k dans un tableau. Le branchement sera pris uniquement si le test échoue, donc rarement (de l'ordre de un million de fois sur les 2^{52} tests) ; par conséquent, il n'est pas nécessaire que cette routine soit optimisée au maximum.

3.4.3 Calcul des e^t

Algorithme

Nous cherchons à calculer les $u_\ell = e^{y+\ell z}$ à 2^{-s} près, où $y = 1/2 + 2^{r-53}$, $z = 2^{r-52}$ et ℓ est un entier tel que $0 \leqslant \ell < 2^{51-r}$.

Le calcul de u_0 et e^z, puis l'application de la formule $u_{\ell+1} = e^z.u_\ell$ pour calculer les termes suivants est une mauvaise méthode, car on perd de la précision à chaque multiplication. L'utilisation d'une approximation polynomiale n'est pas non plus une bonne idée, car le degré serait trop élevé.

Nous allons considérer une méthode permettant de calculer un nouveau terme en faisant une seule multiplication (grâce à la formule $e^{a+b} = e^a.e^b$), en ayant un arbre de calcul assez équilibré pour ne pas trop perdre de précision et sans avoir besoin de stocker trop de valeurs u_ℓ (la taille du disque étant limitée).

On décompose ℓ en base 2 : $\ell = \ell_{50-r}\ell_{49-r}\ldots\ell_0$. On a $e^{y+\ell z} = e^y.e_0^{\ell_0}.e_1^{\ell_1}\ldots e_{50-r}^{\ell_{50-r}}$ où $e_i = (e^z)^{2^i}$; pour simplifier, e^y et les e_i seront précalculés.

Le problème revient maintenant à calculer pour tout sous-ensemble $I \subseteq \{0, 1, \ldots, n\}$ contenant n :

$$P_I = \prod_{i \in I} e_i$$

les e_i étant les valeurs précalculées ci-dessus. Pour cela, on partitionne $\{0, 1, \ldots, n\}$ en deux sous-ensembles à peu près de même taille (pour équilibrer), on applique cette méthode récursivement sur chacun des deux sous-ensembles, en s'arrêtant quand on a un singleton, puis on calcule tous les produits xy, où x est un élément du premier sous-ensemble et y un élément du second sous-ensemble (*diviser-pour-régner*).

Nous effectuerons tout d'abord les multiplications sauf celles de la dernière étape, qui seront effectuées juste avant le test de l'intervalle correspondant (donc en parallèle). Il y aura environ un million de multiplications. Ce sera très rapide ; elles pourront donc être effectuées en séquentiel sur une seule machine. Nous verrons que le fichier qui contiendra les résultats occupera environ 17 Mo sur disque.

Représentation des résultats

D'après des tests faits sur SparcStation 5, la multiplication d'entiers est très lente (22 cycles pour une multiplication de deux entiers 32 bits, résultat sur 32 bits), et la multiplication en double précision peut se faire en 1 ou 2 cycles. Nous allons donc faire toutes les opérations sur des flottants, un nombre étant représenté par plusieurs flottants. Le résultat final (i.e. les e^t) sera ensuite converti dans le format voulu.

Un nombre x sera stocké sous la forme de deux flottants double précision, qui représenteront deux entiers x_1 et x_2 tels que $0 \leqslant x_1 < 2^{53}$ et $|x_2| < 2^q$:

$$x = x_1 2^{-q} + x_2 2^{-2q}$$

où q est pair, car il faut faire une décomposition suivant $2^{-q/2}$ pour la multiplication (puisque la taille du résultat d'une multiplication exacte est double de celle des opérandes) :

$$x = x'_1 2^{-q/2} + x'_2 2^{-2q/2} + x'_3 2^{-3q/2} + x'_4 2^{-4q/2}$$

avec $|x_i| < 2^{q/2}$ pour $i \geqslant 2$. On doit avoir $q \leqslant 51$ pour pouvoir représenter tous les éléments de l'ensemble

$$\left\{x \in 2^{-2q}\mathbb{Z} : 1 \leqslant x < e + 2^{-s}\right\}$$

où $2 < e + 2^{-s} < 4$. Nous prendrons donc $q = 50$, et nous ferons des opérations dans $2^{-100}\mathbb{Z}$. Il faudra éviter de perdre plus d'une dizaine de bits en précision puisque nous voulons $s \simeq 90$.

Les opérations seront valables dans n'importe quel mode d'arrondi. En fait, elles seront effectuées avec l'arrondi au plus proche, qui est le mode d'arrondi par défaut. Nous autoriserons x_2, et par conséquent x'_4, à n'être pas forcément entier ; cela évitera quelques calculs supplémentaires. Pour simplifier, nous supposerons par la suite x_2 entier, mais ce qui sera dit pourra s'étendre au cas x_2 non entier, considéré comme un représentant de l'intervalle $[\lfloor x_2 \rfloor, \lceil x_2 \rceil]$. Lors de l'implémentation, les nombres pourront être manipulés à une puissance de 2 près : par exemple $x'_3 2^{q/2}$ au lieu de x'_3.

Calcul d'un produit et majoration de l'erreur

Considérons deux nombres x et y, représentés par des valeurs approchées $\hat{x}$ et $\hat{y}$ à ε_x et ε_z près, dans $\varepsilon\mathbb{Z}$ (ici, $\varepsilon = 2^{-100}$). Nous cherchons à calculer rapidement une valeur approchée $\hat{p}$ de $p = xy$ dans $\varepsilon\mathbb{Z}$. Posons :

$$\hat{x} = x'_1 2^{-q/2} + x'_2 2^{-2q/2} + x'_3 2^{-3q/2} + x'_4 2^{-4q/2}$$

et

$$\hat{y} = y'_1 2^{-q/2} + y'_2 2^{-2q/2} + y'_3 2^{-3q/2} + y'_4 2^{-4q/2}$$

où les x'_i, y'_i sont des entiers, et $|x'_i|, |y'_i| < 2^{q/2}$ pour $i \geqslant 2$. Nous allons calculer :

$$\begin{cases} z_2 = x'_1 y'_1 \\ z_3 = x'_1 y'_2 + x'_2 y'_1 \\ z_4 = x'_1 y'_3 + x'_2 y'_2 + x'_3 y'_1 \\ z_5 = x'_1 y'_4 + x'_2 y'_3 + x'_3 y'_2 + x'_4 y'_1 \end{cases}$$

Montrons qu'ici ces quatre nombres sont bien représentables, et que chaque résultat intermédiaire est aussi représentable. Comme il n'y a que des entiers, il suffit de montrer que $\tilde{z}_i < 2^{53}$, où $\tilde{z}_i$ correspond au calcul de z_i en remplaçant chaque nombre par sa valeur absolue. La plus grande valeur calculée est e^{1-z} plus l'erreur effectuée ; elle peut être majorée par e. On a $|x'_1|, |y'_1| < e \cdot 2^{q/2} + 1$. On a donc pour $i \geqslant 3$: $\tilde{z}_i < 2(e \cdot 2^q + 2^{q/2}) + 2 \cdot 2^q < 8 \cdot 2^q = 2^{53}$, et $\tilde{z}_2 < e \cdot 2^q + 2^{q/2} < 2^{53}$.

Ensuite le résultat est converti dans le format voulu. On décompose $z_3 = z'_3 2^{q/2} + z''_3$ avec $|z_3| < 2^{q/2}$. L'entier z'_3 est ajouté à z_2, qui reste représentable (pour la même raison). On ajoute $z''_3 2^{q/2}$ et $z_5 2^{-q/2}$ à z_4 ; on a encore $\tilde{z}_4 < 2^{53}$. On décompose $z_4 = z'_4 2^q + z''_4$ avec $|z''_4| < 2^q$, et on ajoute z'_4 à z_2. Le résultat est alors (z_2, z''_4).

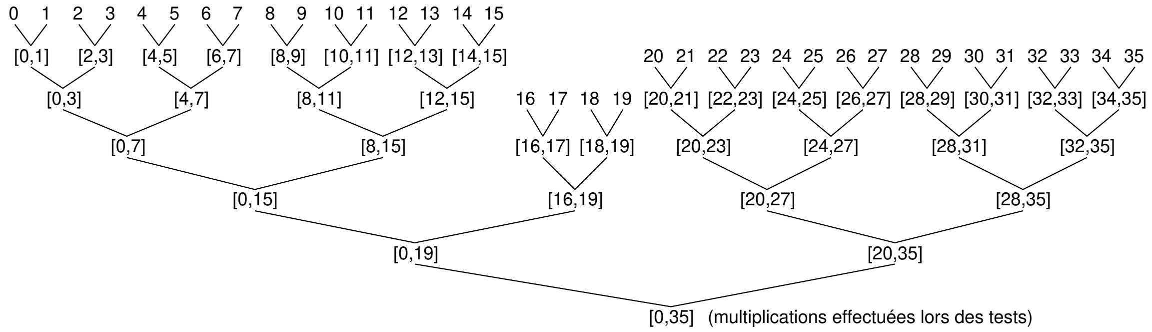
On a $|\hat{p} - \hat{x}\hat{y}| < 7\varepsilon$: $1\,\varepsilon$ provient de l'arrondi de $z_4 + z_5 2^{-q/2}$, $5\,\varepsilon$ proviennent de z_6 (non calculé) et $1\,\varepsilon$ provient de z_7 et z_8 (non calculés). On a alors :

$$|\hat{p} - xy| < x\varepsilon_y + y\varepsilon_x + \varepsilon_x\varepsilon_y + 7\varepsilon.$$

Arbre des produits pour $r = 16$ et majoration de l'erreur

Nous prenons $r = 16$. Alors $s = 120 - 2r = 88$. Nous allons donc chercher à calculer les e^t à $2^{-88} = 2^{12}\varepsilon$ près (au maximum).

Posons $e_{35} = e^y$. Les constantes $e_0, e_1, \ldots, e_{35}$ sont précalculées à $\varepsilon = 2^{-100}$ près. Nous allons calculer tous les produits P_I où $I \subseteq \{0, 1, \ldots, 35\}$, en partitionnant les ensembles de la façon suivante :



Calculons maintenant une majoration de l'erreur maximale effectuée.

- Intervalle $[0, 19]$. Pour tout $I \subseteq [0, 19]$, $P_I \leqslant e_{20} = e^{1/65536}$. On a les majorations suivantes au bout de chaque étape (un produit x est obtenu au bout de la n^{e} étape si le sous-arbre de racine x a une profondeur égale à n) :

$$\begin{cases} \text{1 étape :} & \varepsilon_1 \leqslant \varepsilon(2e_{20} + \varepsilon) + 7\varepsilon \leqslant 10\varepsilon, \\ \text{2 étapes :} & \varepsilon_2 \leqslant 10\varepsilon(2e_{20} + 10\varepsilon) + 7\varepsilon \leqslant 28\varepsilon, \\ \text{3 étapes :} & \varepsilon_3 \leqslant 28\varepsilon(2e_{20} + 28\varepsilon) + 7\varepsilon \leqslant 64\varepsilon, \\ \text{4 étapes :} & \varepsilon_4 \leqslant 64\varepsilon(2e_{20} + 64\varepsilon) + 7\varepsilon \leqslant 136\varepsilon, \\ \text{5 étapes :} & \varepsilon_5 \leqslant 136\varepsilon(2e_{20} + 136\varepsilon) + 7\varepsilon \leqslant 280\varepsilon. \end{cases}$$

- Intervalle $[20, 35]$. On a les majorations suivantes :

$$\begin{cases} \text{1 étape :} & \varepsilon'_1 \leqslant \varepsilon(e^y + e^{1/4} + \varepsilon) + 7\varepsilon \leqslant 10\varepsilon, \\ \text{2 étapes :} & \varepsilon'_2 \leqslant 10\varepsilon(e^{y+1/4} + e^{3/16} + 10\varepsilon) + 7\varepsilon \leqslant 41\varepsilon, \\ \text{3 étapes :} & \varepsilon'_3 \leqslant 41\varepsilon(e + e^{1/16} + 41\varepsilon) + 7\varepsilon \leqslant 163\varepsilon, \\ \text{4 étapes :} & \varepsilon'_4 \leqslant 163\varepsilon(e + e^{1/256} + 163\varepsilon) + 7\varepsilon \leqslant 614\varepsilon. \end{cases}$$

  L'erreur maximale peut donc être majorée par :

$$e \cdot 280\varepsilon + e_{20} \cdot 614\varepsilon + 280\varepsilon \cdot 614\varepsilon + 7\varepsilon \leqslant 1383\varepsilon < 2^{12}\varepsilon.$$

Décomposition d'un entier en deux

Dans le calcul d'un produit, il faut décomposer un flottant x en une somme $x'\,2^k + x''$, où x' est un entier et x'' vérifie $|x''| < 2^k$. On suppose que les opérations d'addition $\oplus$, de soustraction $\ominus$ et de multiplication $\otimes$ sont effectuées en double précision (53 bits) avec l'un des quatre modes d'arrondi. Ici, on a $k = q/2 = 25$.

Soit $C = 3 \cdot 2^{51+k}$. On calcule $\tilde{x} = (x \oplus C) \ominus C$, puis $x'' = x \ominus \tilde{x}$. Ensuite, on peut éventuellement calculer $x' = \tilde{x} \otimes 2^{-k}$ ou utiliser $\tilde{x}$ dans les calculs.

Note : cette décomposition reste valable pour tout x tel que $|x| \leqslant 2^{51+k} - 2^k$ et pour tout entier k. Elle est aussi utilisée dans la multiplication en multiprécision (cf section 4.3).

3.4.4 Arrondi de d_0, d_1 et d_2 (avec $r = 16$)

Dans la section 3.4.3, nous avons décrit comment calculer les nombres $\hat{u}$, représentés par deux flottants double précision. Nous voulons maintenant calculer les arrondis $\hat{d_0}$, $\hat{d_1}$, $\hat{d_2}$ de d_0, d_1 et d_2 (comme indiqué dans la section 3.4.2).

Représentation intermédiaire de $\hat{u}$

Nous devons faire une conversion de format de $\hat{u}$: nous devons passer d'un couple de flottants (u_1, u_2) en double précision (type `double`) à plusieurs variables entières sur 32 bits (type `int`). Cela se fera en passant par la représentation en mémoire d'un flottant double précision, déterminée par la norme IEEE. Il faut d'abord fixer l'exposant des deux flottants u_1 et u_2 de telle sorte qu'un bit de poids donné soit situé à un endroit fixe.

- u_2 : nous avons $u_2 < 2^{50}$. Il est plus pratique de rendre u_2 positif ; nous ajoutons donc 2^{50}. En même temps, nous pouvons fixer l'exposant en ajoutant 2^{52}. D'une part nous avons maintenant $2^{52} \leqslant u_2 \leqslant 2^{52} + 2^{51}$ (l'exposant est bien fixé à 52). D'autre part, u_2 n'a pu être arrondi qu'à $\lfloor u_2 \rfloor$ ou $\lceil u_2 \rceil$: le bit de poids faible de la mantisse a pour exposant 0. Le nombre formé par les 52 bits représentés de la mantisse avec leurs poids respectifs (2^{-49} à 2^{-100}) est égal à $(u_2 + 2^{50})\, 2^{-100}$. Remarque : il faudra tenir compte du bit de poids 2^{-49}, car il n'est pas forcément nul (si u_2 a été arrondi à 2^{50}).
- u_1 : nous avons $u_1 \in \mathbb{Z}$ et $1 < u_1\, 2^{-50} < 3$. Nous ajoutons alors $2^{52}-1$ (addition exacte), le "-1" servant à compenser le 2^{50} ajouté à u_2. Nous avons maintenant $2^{52} \leqslant u_1 < 2^{53}$; l'exposant est donc bien fixé à 52. Le nombre formé par les 52 bits représentés de la mantisse avec leur poids respectif (2^1 à 2^{-50}) est égal à $(u_1 - 1)\, 2^{-50}$.

Arrondi de d_0

Pour d_0, nous ne faisons aucun arrondi : nous récupérons les bits de poids 2^{-54} à 2^{-100} de $\hat{u}$; les bits de poids 2^{-101} à 2^{-117} sont annulés.

Arrondi de d_1

L'arrondi de d_1 est le plus compliqué. Nous avons besoin d'un arrondi inférieur et d'un arrondi supérieur de $(2^{-53} + 2^{-107})\hat{u}$ à 2^{-104} près (l'un pour le sens positif, l'autre pour le sens négatif). Nous calculons d'abord d'_1, arrondi inférieur de $2^{-53}\hat{u}$ dans $2^{-117}\mathbb{Z}$. Nous avons évidemment $0 < d_1 - d'_1 < 3 \cdot 2^{-107}$. Nous calculons ensuite $\hat{d_1}^-$, arrondi inférieur de d'_1 dans $2^{-105}\mathbb{Z}$. Nous avons $0 \leqslant d'_1 - \hat{d_1}^- < 2^{-105}$. Par conséquent,

$$0 < d_1 - \hat{d_1}^- < 7 \cdot 2^{-107} < 2^{-104}.$$

Enfin, nous calculons $\hat{d_1}^+ = \hat{d_1}^- + 2^{-104}$. Alors

$$0 < 2^{-104} - 7 \cdot 2^{-107} < \hat{d_1}^+ - d_1 < 2^{-104}.$$

Les deux nombres $\hat{d_1}^-$ et $\hat{d_1}^+$ conviennent comme arrondis de d_1.

Arrondi de d_2

Le bit de poids 2^{-117} de d_2 correspond au bit de poids 2^{-11} de $\hat{u}$ (car $\hat{u}$ est multiplié par 2^{-106}). Nous considérons donc l'entier formé par les bits de poids 2^1 à 2^{-12} de $\hat{u}$; l'arrondi au plus proche se fait en ajoutant 1 et en gardant les bits de poids 2^1 à 2^{-11}.

3.4.5 Implémentation (sur Sparc)

Calcul des e^t

Le calcul des e^t nécessite certaines valeurs précalculées, notées e_i. Nous avons choisi la méthode suivante : un script Perl fait calculer ces valeurs par Maple, récupère les résultats (un couple d'entiers par e_i) et génère un source C contenant ces résultats. Ainsi il n'y a pas de risque d'erreur en recopiant des données.

Calcul des e^{t+x}

La partie qui prend le plus de temps est le calcul et le test des e^{t+x}, connaissant $\hat{d}_0$, $\hat{d}_1$ et $\hat{d}_2$. Le compilateur C n'est pas capable de générer le code complètement optimisé. En particulier, les blocs de base ne sont pas placés comme on le voudrait, et il y a des grosses pertes de temps au niveau des branchements. Ceci était à prévoir, car le compilateur ne sait pas que certains branchements ne sont quasiment jamais pris (nous le savons notamment grâce à l'approche probabiliste). Malheureusement, le langage C ne possède pas de directive permettant de donner des indications sur les branchements ; le seul moyen, qui semble marcher avec le compilateur `gcc`, est de placer soi-même les blocs de base dans le source C et d'utiliser des `gotos`. Mais il est plus simple et plus sûr de programmer cette partie directement en assembleur.

Ce calcul est constitué de deux boucles séquentielles (une dans le sens positif, l'autre dans le sens négatif). Chaque boucle est constituée de $2^r = 2^{16} = 65536$ itérations. On peut gagner environ un cycle en déroulant partiellement la boucle (2^k tests par itération) : le compteur de boucle sera décrémenté 2^k fois moins souvent. Mais il ne faut pas la dérouler trop, pour qu'elle puisse tenir dans le cache de niveau 1. L'optimal (approché) est obtenu en essayant différentes valeurs de k. Sur Sparc 5, la valeur optimale de k est 6 environ.

Estimation du temps de calcul

Pour chaque test, nous faisons deux additions sur 64 bits : $d_1 \leftarrow d_1 + d_2$ et $d_0 \leftarrow d_0 + d_1$, ce qui prend au total 4 cycles (processeur 32 bits). Le test se compose d'une comparaison et d'un branchement, qui n'est quasiment jamais pris (une fois sur 2^{31} en moyenne) ; cela prend un cycle sur Sparc. De plus, les instructions peuvent être placées de manière à ne pas avoir besoin d'ajouter d'instruction `nop` après les branchements. En négligeant la décrémentation du compteur de boucle, nous obtenons un temps de 5 cycles par test.

En fait, en faisant tourner le programme, i.e. en tenant compte de tout (sauf de la seconde étape[1]), le nombre de cycles moyen par test se situe entre 6 et 7 sur une Sparc 5, ce qui représente une dizaine d'années de calculs. Il faudra donc paralléliser les calculs, ce qui est facile puisque tous les intervalles peuvent être testés de manière indépendante.

Regroupement des intervalles

Pour que la parallélisation des calculs soit efficace, nous allons considérer des groupes d'intervalles ; cela diminuera les accès disque et réseau. Les résultats seront validés uniquement lorsque le groupe aura été entièrement testé ; il ne faut donc pas que le test d'un

1. La seconde étape devrait prendre un temps de l'ordre du jour si on écrit un programme spécialement pour elle.

groupe prenne trop de temps, car le processus pourra être arrêté à tout moment, et il faudra reprendre à zéro le travail correspondant au groupe. Nous avons choisi de faire des groupes de 2^{15} intervalles consécutifs, ce qui correspond à environ 5 minutes sur une Sparc 5 (20 à 30 minutes sur les machines les plus lentes). Tous les arguments nécessaires à la dernière étape du calcul des e^t ($2^{15} + 1$ valeurs) sont lus tout au début pour être plus efficace.

Maintenant nous ne nous préoccuperons plus des intervalles considérés jusqu'à présent (contenant 2^{17} valeurs). Nous ne considérerons que des groupes d'intervalles, qui forment des intervalles à 2^{32} valeurs, et que nous appellerons tout simplement intervalles. Ils sont numérotés de 0 à $2^{20} - 1$. D'après l'approche probabiliste, il y aura en moyenne 2 *exceptions* par intervalle.

Le passage des données se fera de la manière suivante :

- Les numéros des intervalles à tester sont passés en arguments au programme.
- Les résultats sont envoyés dans la sortie standard. Leur format doit permettre de détecter certaines erreurs (perte, entrelacement de données sur une machine parallèle). Pour chaque intervalle, le résultat est formé d'un ensemble de lignes, chaque ligne contenant le numéro de l'intervalle (pour identification). Nous avons d'abord une ligne `BEGIN`, indiquant le début du résultat, puis des lignes identifiant les exceptions, puis une ligne `END` contenant le nombre d'exceptions. Note : après la ligne `END`, il est utile de "flusher" la sortie standard, au cas où la machine serait "rebootée".

Exécution sur une machine parallèle

Le programme est écrit de manière à ce qu'il puisse tourner sur une machine parallèle MIMD, avec les conditions suivantes (c'est le cas de la machine Capitan qui sera utilisée) : la même liste d'arguments est passée à tous les nœuds, il y a une variable qui donne le nombre de nœuds n, et une autre qui donne le numéro i du nœud entre 0 et $n - 1$. Alors le nœud i ne testera que les intervalles de numéro congru à i modulo n.

3.5 Parallélisation sur réseau de stations

3.5.1 Objectifs

Nous voulons les résultats des tests assez rapidement (quelques mois). Nous devons donc paralléliser les calculs. Nous allons utiliser le réseau de stations de travail (une centaine de SparcStations ainsi qu'une machine Capitan) que nous avons à notre disposition au LIP et à l'ENS. Les stations de travail ont souvent une charge nulle : par exemple la nuit, quand l'utilisateur n'est pas là, ou même le jour, quand il fait un travail fortement interactif (traitement de textes, mail...). On cherche à utiliser chaque machine au maximum sans gêner l'utilisateur ; on donnera donc une très faible priorité au processus de test. D'autre part, le processus occupe très peu de mémoire : dans chaque intervalle, les calculs se font uniquement sur des registres (sauf quand un test échoue, mais c'est très rare), et le code est prévu pour tenir dans le cache ; par conséquent, le processus ne provoque pas de swap (ou très peu), ce qui ne ralentit pas la machine. Il y a aussi très peu de communications (accès NFS...), car la plus grande partie du temps est prise par les tests eux-mêmes, les exceptions étant très rares ; la machine n'est donc pas non plus ralentie par les communications.

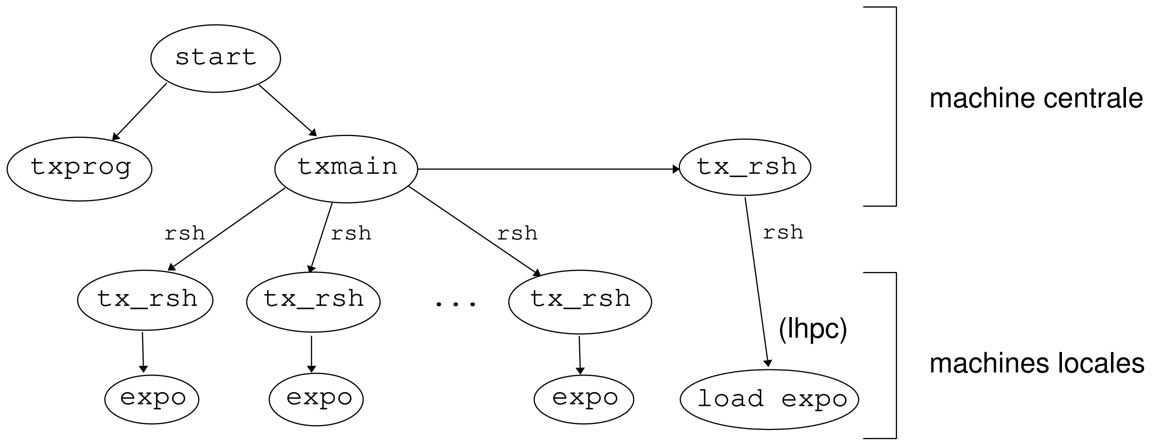
On doit aussi s'adapter à l'utilisateur. Il se peut que celui-ci ne veuille pas qu'on utilise sa machine le jour, quand il est là. Il faut donc un système permettant que le processus soit lancé uniquement à certains moments donnés (la nuit, pendant le week-end, les vacances entre telle et telle date).

Il faudra aussi tenir compte que des machines peuvent être "rebootées" à tout moment, ou peuvent ne pas répondre. On stockera toutes les données qui peuvent servir à détecter les éventuels problèmes.

3.5.2 Méthode choisie

Nous avons un processus de contrôle central `txmain` qui doit tourner sur une machine peu souvent rebootée, car s'il est tué il faut le relancer à la main. Ce processus se charge de lancer par `rsh` des processus de contrôle local `tx_rsh` sur chaque machine suivant les modalités choisies par les utilisateurs de ces machines ; il y a une petite exception : pour la machine parallèle Capitan (*lhpc*), le processus `tx_rsh` est lancé sur la machine centrale car le système de fichiers de *lhpc* est différent. Les processus de contrôle local devront lancer le processus de tests `expo`.

Le lancement des tests se fait en exécutant un processus `start`, qui crée d'abord un *fichier de log*, dont le nom contient la date du lancement, pour être sûr de l'unicité de ce fichier. La sortie (`stdout` et `stderr`) est redirigée vers ce fichier, ce permettra de détecter tout ce qui se passe, et s'il y a des problèmes avec certaines machines. Ensuite est lancé un processus `txprog` chargé de stocker le nombre total d'intervalles testés à chaque heure, ainsi que le nombre moyen d'intervalles testés par minute pendant la dernière heure ; ceci permettra de détecter d'éventuels problèmes (en remarquant une perte de performance) et d'estimer la date à laquelle les tests seront terminés. Le processus de contrôle central est alors lancé, et on attend que ce processus se termine. On récupère le code d'erreur de ce processus, que l'on stocke dans le fichier de log ; cela permettra de savoir comment le processus s'est terminé. Enfin, le processus `txprog` est tué, et on stocke la date de fin.



Note : nous avons choisi d'écrire tous ces processus (sauf `expo`) en Perl, ce langage étant très adapté à la manipulation de fichiers texte, processus, signaux ; les résultats sont manipulés de telle manière que tout bug (en dehors de `expo`) ou autre problème courant (perte de données, processus tués brusquement, etc...) n'ait aucune influence sur la validité des résultats.

Modalités

L'utilisateur de la machine a le choix entre différentes modalités. Ces modalités sont stockées dans un fichier (un par machine) et peuvent être modifiées à tout moment, sans avoir besoin de tout relancer. D'autre part elles peuvent changer suivant la date (pour prendre en compte une période de vacances). Un fichier de modalités se compose de plusieurs blocs : un bloc par période. Pour chaque période, on a le choix entre machine inutilisée, machine utilisée en permanence, machine utilisée tel jour de la semaine entre telle et telle heure (en général la nuit et les week-ends, avec horaires choisis par l'utilisateur), et machine utilisée seulement quand personne n'est physiquement connecté dessus (cas des machines en libre service).

L'utilisateur a toujours la possibilité de tuer le processus à n'importe quel moment. Le nom d'un script setuid lui a été donné (`killexpo`). Ceci tue le processus et empêche son redémarrage pendant trois heures (valeur par défaut). Ensuite il pourra être relancé par le processus central suivant les modalités.

Processus de contrôle local `tx_rsh`

Le processus de contrôle local est lancé par le processus central, qui lui donne les numéros des intervalles à tester en arguments ; il doit en avoir assez pour une dizaine d'heures au moins. Ensuite, ce processus est complètement autonome ; si le processus central meurt, le processus local peut continuer à travailler et à stocker les résultats tant qu'il n'a pas terminé.

Le processus `tx_rsh` lance `expo` au tout début et stocke le `pid` dans un fichier (fichier de sortie) de manière à ce que le processus central puisse tuer `expo` en cas de problème. Les résultats sont récupérés par *pipe* et stockés dans un fichier texte commun ; un système de verrouillage qui résiste à NFS est utilisé pour éviter des pertes de données (des essais ont été effectués sans verrouillage et les pertes de données étaient relativement courantes). Le processus vérifie régulièrement s'il doit s'arrêter (cf les deux dernières modalités, dans la section précédente).

Lorsque le processus doit se terminer ou est tué proprement, il écrit le mot `exit` dans le fichier de sortie pour indiquer au processus central qu'il a terminé. Evidemment cela ne fonctionne pas si la machine est rebootée ; c'est pourquoi il simule périodiquement une écriture dans le fichier de sortie (`touch`) pour indiquer qu'il est toujours vivant et qu'il n'a pas besoin d'être relancé.

Processus de contrôle central `txmain`

La fonction principale du processus central est de lancer les processus locaux suivant les modalités. Il doit aussi détecter si un processus local a terminé (mot `exit` dans le fichier de sortie du processus local) ou si s'il a été tué brusquement ; on considère qu'il a été tué brusquement s'il n'y a eu aucune écriture dans le fichier de sortie pendant deux heures. Dans le premier cas, le processus peut être relancé rapidement. Si le second cas se produit trop de fois à la suite, la machine ne sera plus réutilisée. Lorsqu'un processus est ajouté ou enlevé, `txmain` l'écrit dans le fichier de log.

Il est aussi capable de détecter d'éventuels interblocages (dus au système de verrouillage) et d'enlever le verrou.

Lorsque le fichier commun de résultats devient trop gros ou à la demande (signal `USR1`), les résultats sont transférés dans un autre fichier après conversion en binaire et retrait de certaines informations pour prendre moins de place.

Arrêt

Le processus central crée au début un script permettant de tout arrêter (il s'agit de l'envoi d'un signal `TERM` au processus). Mais il se peut que cela se passe mal et que le processus central meure sans rien arrêter. Dans ce cas, les processus locaux continuent à tourner et à stocker des résultats. Pour tout arrêter, il faut alors effacer ce script, ce qui empêchera les processus locaux d'écrire leurs résultats et les tuera à ce moment.

3.5.3 Problèmes rencontrés

Parfois certaines machines ne répondent pas ou répondent trop lentement. Le processus local est automatiquement relancé (au bout de deux minutes environ). Si le problème persiste, il vaut mieux désactiver la machine par `killexpo`.

Il arrive aussi (c'est assez rare) que les processus `expo` ne font plus rien et ne peuvent plus être tués : ils résistent au `kill -KILL`. La machine doit alors être rebootée. Ce problème n'a pas été résolu ; mais dans tous les cas qui se sont produits, il y avait déjà un processus extérieur qui n'a pas été tué correctement et prenait tout le temps cpu.

La gestion des signaux `CHLD` est pour l'instant assez mauvaise. Des processus "zombies" peuvent parfois apparaître. Mais une gestion correcte ne fonctionne pas. Ceci est dû à deux bugs de la version de perl/Solaris installée ici. Pour l'instant, seulement l'un des deux a pu être corrigé.

Il est arrivé plusieurs fois que le processus central soit tué par un "segmentation fault" ou un "bus error" (numéro du signal récupéré grâce au processus `start`). Ce genre d'erreur ne devrait pas arriver. Il y a peut-être une relation avec les bugs de `CHLD`.

Il est arrivé une fois que des intervalles soient calculés en double ; ceci est apparemment dû à un problème de communication par NFS. Quand une telle erreur se produit, cela est détecté par le processus central, qui s'arrête.

3.6 Seconde étape

La seconde étape consiste à retester les arguments qui ont échoué à la première étape. Cela se fait simplement en calculant l'exponentielle à une assez grande précision, 2^{-150} par exemple. La probabilité pour qu'un argument échoue de nouveau est extrêmement faible. Nous pouvons aussi prendre une précision plus faible, les éventuels cas restants pouvant être testés avec Maple.

Pour l'instant, les programmes nécessaires à la seconde étape ne sont pas encore écrits ; nous pourrons éventuellement utiliser ceux des calculs en multiprécision (cf chapitre 4). Mais il faut commencer la seconde étape dès maintenant, d'une part pour détecter d'éventuels bugs, d'autre part pour s'assurer que les hypothèses probabilistes sont vérifiées. Nous utilisons Maple, ce qui est simple, mais assez lent (bien qu'il soit possible d'exécuter entièrement la seconde étape comme cela en quelques jours avec plusieurs machines) ; en particulier, on est obligé avec Maple de calculer en base 10, puis de convertir en base 2, ce qui n'est pas du tout naturel en ce qui nous concerne.

3.7 Quelques résultats

Le première étape a été lancée il y a un mois, vers le 20 mai, d'abord sur quelques machines, puis sur toutes les machines du LIP. Nous avons ajouté récemment deux machines du laboratoire de chimie. Des machines de l'ENS seront bientôt ajoutées.

227 385 intervalles (sur 1 048 576) ont pour l'instant été testés, soit environ 21,7%. La nuit et les week-ends, entre 8 et 11 intervalles sont testés par minute ; ce nombre tombe à 5 pendant la journée en semaine. Pour l'instant (avant l'ajout de nouvelles machines), cela donne environ 80 000 intervalles testés par semaine. La première étape devrait être terminée vers la mi-août.

Sur les 227 385 intervalles testés, 454 688 exceptions ont été trouvées. L'approche probabiliste en prévoyait 454 770. L'estimation est donc excellente.

Nous avons commencé la seconde étape. 134 886 exceptions ont pour l'instant été testées (avec Maple). Dans le tableau suivant, nous donnons en fonction des nombres k_0 (avec $k_0 \geqslant 40$) et m_0 intervenant dans le dilemme du fabricant de tables (cf chapitre 2) :

- le nombre d'exceptions pour lesquelles $k = k_0$;
- le nombre d'exceptions pour lesquelles $k \geqslant k_0$;
- le nombre estimé (par les arguments probabilistes) d'exceptions pour lesquelles $k \geqslant k_0$.

k_0	m_0	$k = k_0$	$k \geqslant k_0$	estimation
40	93	535	1070	1053,8
41	94	267	535	526,9
42	95	135	268	263,4
43	96	56	133	131,7
44	97	41	77	65,9
45	98	16	36	32,9
46	99	9	20	16,5
47	100	4	11	8,2
48	101	4	7	4,1
49	102	1	3	2,1
50	103	0	2	1,0
51	104	1	2	0,5
52	105	0	1	0,3
53	106	1	1	0,1

Là encore, l'estimation est très bonne.

Les exceptions trouvées pour $k \geqslant 49$ sont :

$x_1 = 0.10000000001011010111000000100001001010110101110101011$,
avec $\exp(x_1) = 1.1010011001011101100010011010101111110011110100011110 10^{52}10\ldots$

$x_2 = 0.10000011100100111101011110101111111101110100011110110$,
avec $\exp(x_2) = 1.101011000000001100101010100011010010111011000010001100^{50}10\ldots$

$x_3 = 0.10000000110101110001001011100011000001001101000111000$,
avec $\exp(x_3) = 1.1010011101111010111000110110000011101100011010011110111^{48}00\ldots$

Chapitre 4

Calcul des fonctions élémentaires en multiprécision

4.1 Introduction

L'arrondi exact des fonctions élémentaires nécessite d'avoir des routines de calculs à très grande précision. Elles pourront d'abord servir à résoudre le problème du fabricant de tables pour des implantations en double précision, tant que les tests exhaustifs n'auront pas été complètement terminés, ce qui prendra beaucoup de temps. Elles seront de toutes façons obligatoires pour les implantations en quadruple précision. Notons qu'en employant la méthode multi-niveaux de Ziv (cf section 2.2), l'utilisation de ces routines influe très peu sur le temps de calcul moyen.

Nous cherchons à écrire des routines optimisées pour des nombres de l'ordre du million de bits (borne supérieure pour les flottants en double précision donnée par le théorème de Yu. Nesterenko et M. Waldschmidt). Les algorithmes d'addition et de soustraction sont classiques et ne posent pas de problème particulier ; nous n'en parlerons donc pas. Ensuite, il y a la multiplication, sur laquelle reposent tous les autres algorithmes (division, racine carrée, fonctions élémentaires) ; il est donc important de bien l'optimiser, et nous en parlerons plus en détail. D. Zuras a décrit un certain nombre d'algorithmes de multiplication dans [Zur94]. Les algorithmes des autres opérations et fonctions élémentaires sont donnés par R.P. Brent dans [Bre76] ; des algorithmes semblables mais plus rapides sont donnés par D.H. Bailey dans [Bai93].

4.2 Type de données utilisé et choix de la base

De plus en plus, les processeurs sont plus rapides en calcul flottant qu'en calcul entier, dès que la multiplication intervient. Nous avons comparé deux implémentations différentes d'un algorithme de multiplication de nombres de grande taille : une effectuant les calculs élémentaires sur les entiers, l'autre sur les flottants en double précision. Le calcul sur les flottants était environ 7 fois plus rapide que celui sur les entiers. Nous avons donc choisi d'utiliser les flottants en double précision comme type de données élémentaire.

Nous devons calculer dans une base du type 2^N. N sera fixé plus loin par des problèmes d'implémentation, mais il vaut mieux le choisir plutôt grand afin de ne pas utiliser trop de mémoire.

4.3 Multiplication

4.3.1 Introduction

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 ou fonctions élémentaires ; mais la seule différence est la gestion des exposants qu'il faut faire en plus pour les flottants et l'arrondi du résultat pour avoir 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 :

1. L'algorithme "classique" en n^2 (n^2 multiplications pour un produit, et $n(n-1)/2$ multiplications et n carrés pour un carré).
2. Les algorithmes du type Toom–Cook, qui généralisent une méthode inventée par Karatsuba et Ofman. Un entier est vu comme un polynôme (par décomposition en blocs de k chiffres). On doit calculer le carré ou le produit de polynômes en faisant le moins possible de carrés ou multiplications.
3. Les algorithmes utilisant la transformée de Fourier rapide (FFT).

Les algorithmes de type (2) et (3) utilisent 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 l'algorithme en n^2), de complexité en temps plus grande, mais plus rapide sur des petits nombres.

4.3.2 Algorithme en n^2

L'algorithme étant classique, nous ne parlerons que de l'implémentation. Puisque le produit de deux nombres de K bits est un nombre de $2K$ bits, nous avons besoin de savoir convertir un nombre de $2K$ bits en un nombre de K bits pour revenir dans la représentation d'origine ; la méthode a été expliquée dans la section 3.4.3. Nous avons aussi besoin d'extraire les retenues lors d'une addition ou d'une soustraction ; la même méthode s'applique. Nous avons choisi de travailler en base 2^{50}, ce qui laissera notamment 3 bits pour les retenues, utiles pour faire plusieurs additions ou soustractions simultanées (cf section 4.3.4).

4.3.3 Algorithme de Karatsuba (méthode 2-way)

En 1963, Karatsuba et Ofman ont remarqué qu'il suffit, grâce à l'égalité suivante appliquée à $x = 2^{Nw}$, de 3 produits de mots de longueur w pour effectuer un produit de mots 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 $O(w^{\log 3/\log 2}) = O(w^{1.585})$ en appliquant récursivement la méthode ci-dessus.

4.3.4 Généralisation (méthode k-way)

On peut généraliser l'algorithme de Karatsuba en découpant l'entier 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 dépendent ni de A, ni de B).

Ainsi on considère l'équation

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

et on détermine les C_i en choisissant $2p-1$ valeurs de x (rationnels ou/et ∞). On remarque qu'un bon choix est de prendre dans l'ordre 1, ∞, 0, 2, $\frac{1}{2}$, -2, $-\frac{1}{2}$, 3, $\frac{1}{3}$, -3, $-\frac{1}{3}$, $\frac{3}{2}$, $\frac{2}{3}$, $\ldots$; on privilégie la symétrie multiplicative sur la symétrie additive. La résolution du système linéaire (effectuée une fois pour toute lors de l'implémentation de l'algorithme) mène à un produit d'une matrice constante à coefficients rationnels par le vecteur des $(\sum k_{ij}A_j)^2$.

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

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

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 16 & 8 & 4 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} C_4 \\ C_3 \\ C_2 \\ C_1 \\ C_0 \end{bmatrix} = \begin{bmatrix} P_4 \\ P_3 \\ P_2 \\ P_1 \\ P_0 \end{bmatrix} = \begin{bmatrix} A_2B_2 \\ (4A_2 + 2A_1 + A_0)(4B_2 + 2B_1 + B_0) \\ (A_2 + A_1 + A_0)(B_2 + B_1 + B_0) \\ (A_2 + 2A_1 + 4A_0)(B_2 + 2B_1 + 4B_0) \\ A_0B_0 \end{bmatrix}$$

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 peuvent s'effectuer sans utiliser de division flottante pour être plus rapides.

4.3.5 FFT

Il existe plusieurs algorithmes utilisant la FFT. Certains sont décrits dans [Zur94] ; 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 [BMZ85, Bai87], qui consiste à décomposer un nombre de n chiffres en $\sqrt{n}$ blocs de $\sqrt{n}$ chiffres ; sa complexité est $O(n \log n \log\log n)$.

4.3.6 Résultats sur une SparcStation 5 à 85 MHz

Nous avons implémenté 3 algorithmes : l'algorithme en n^2, celui de Karatsuba, et le 3-way. L'algorithme de Karatsuba a été implémenté avec des flottants en double précision (base 2^{50}) et avec des entiers (base 2^{31}). L'implémentation sur les flottants s'est avérée être 7 fois plus rapide sur des nombres de 1 000 bits ; ce n'est pas surprenant, car sur Sparc la multiplication en double précision est environ 50 fois plus rapide que celle sur des entiers (en se ramenant à des nombres de même taille). Les algorithmes suivants ont donc été implémentés avec des flottants de 50 bits.

Des simulations sur des nombres de 1 000 000 de bits ont montré que l'algorithme de Karatsuba devient efficace par rapport à l'algorithme classique pour les nombres excédant 17 mots (soit 850 bits). L'algorithme de Karatsuba donne un temps de 32 secondes sur une Sparc 5 à 85 MHz, pour des nombres de 20 000 mots (soit 1 000 000 de bits). L'algorithme 3-way donne un temps de 18,3 secondes (avec les mêmes données). La frontière entre le 2-way et le 3-way se situe aux alentours de 30–70 mots (de 1 500 à 3 500 bits).

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 nous appliquerons l'itération plus loin), on peut calculer une solution de l'équation $f(x) = 0$ à l'aide de l'itération suivante :

$$x_{i+1} = x_i - \frac{h_i f(x_i)}{f(x_i + h_i) - f(x_i)} \text{ (ou } x_i - \frac{f(x_i)}{f'(x_i)}\text{).}$$

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).

Nous utiliserons 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.

Nous avons besoin d'algorithmes de division et de racine carrée pour le calcul des fonctions transcendantes. Nous pouvons calculer l'inverse d'un nombre c en appliquant l'itération $x_{i+1} = x_i(2 - cx_i)$. A 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 par c.

4.5 Fonctions transcendantes

Les fonctions transcendantes sont calculées à l'aide de résultats de la théorie des intégrales elliptiques. Les fonctions sont encore calculées à l'aide d'algorithmes itératifs avec 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 faiblement supérieure à cause des erreurs d'arrondi.

Par exemple, on peut calculer π de la façon suivante (la valeur de π est utilisée dans les algorithmes de calcul des fonctions élémentaires) :

$A \leftarrow 1;\ B \leftarrow 2^{-1/2};\ T \leftarrow 1/4;\ X \leftarrow 1;$
`While` $A - B > 2^{-n}$ `do`
  $Y \leftarrow A;\ A \leftarrow (A+B)/2;\ B \leftarrow (YB)^{1/2};$
  $T \leftarrow T - X(A-Y)^2;\ X \leftarrow 2X;$
`Return` $(A+B)^2/(4T)$

Les algorithmes pour les fonctions transcendantes sont décrits dans [Bre76] et [Bai93]. On a des temps qui sont jusqu'à environ 30 fois plus grands que celui d'une multiplication, soit quelques minutes pour des nombres de 1 000 000 de bits.

Chapitre 5

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) 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 du résultat exact. Pour résoudre ce problème, nous avons considéré deux approches :

- Pour les flottants en double précision, le problème peut être résolu à l'aide de tests exhaustifs. Pour l'instant, nous effectuons les tests sur l'exponentielle entre $1/2$ et 1 à l'aide d'une centaine de machines ; nous espérons obtenir les résultats complets vers la mi-août. Ces tests permettront aussi d'obtenir des bornes pour la fonction réciproque : le logarithme. Nous étendrons ensuite les calculs à d'autres intervalles et d'autres fonctions, en utilisant plus de machines. Mais pour les flottants en quadruple précision, il est impossible d'effectuer de tels tests, et les machines ne seront certainement jamais assez puissantes.
- Pour résoudre complètement le problème, nous écrivons une bibliothèque de calculs en multiprécision, optimisée pour des nombres ayant une taille de l'ordre du million de bits. Nous avons implémenté trois algorithmes de multiplication, qui est l'opération la plus importante : l'algorithme classique en n^2, l'algorithme de Karatsuba (décomposition en 2) et l'algorithme 3-way (décomposition en 3). Il est encore possible d'accélérer la multiplication :
  - L'implémentation de l'algorithme en n^2 peut être améliorée, en prenant par exemple une base plus petite et en propageant moins souvent les retenues (dues aux additions).
  - Les frontières n^2 / 2-way et 2-way / 3-way sont assez proches et nous gagnons environ un facteur 2 avec l'algorithme 3-way par rapport à l'algorithme 2-way. Ceci nous suggère qu'il serait intéressant d'implémenter un algorithme 4-way et éventuellement d'aller plus loin (k-way généralisé ou algorithme de Schönhage–Strassen).
  - Nous pourrions aussi implémenter le carré. Il est ainsi possible de gagner un facteur d'environ 1,5 lorsque le cas se présente [Zur94].
  - Puisque la multiplication est en fait utilisée pour calculer la division, la racine carrée, etc..., nous avons en fait seulement besoin d'un résultat ayant la même précision que les opérandes : lorsque nous multiplions deux nombres de n mots,

nous sommes seulement intéressés par les n mots les plus significatifs du produit (au lieu des $2n$ mots), i.e. par le résultat du *produit court* [KJ93a, KJ93b]. Nous pouvons ainsi gagner un facteur 2 sur les petits nombres (1000 bits), ce qui est intéressant pour implémenter la stratégie de Ziv (cf section 2.2) ; si les nombres sont très grands, on y gagne très peu.

Il reste aussi à implémenter les algorithmes de la division, la racine carrée, et des fonctions élémentaires. Notons que ces routines pourront servir à des tâches autres que l'arrondi exact des fonctions élémentaires.

Nous avons montré que l'arrondi exact des fonctions élémentaires est possible. Cela peut coûter très cher, mais nous devons considérer que :

– La vraie borne supérieure de m est certainement beaucoup plus faible que la borne théorique. Les tests exhaustifs en double précision permettront d'avoir une petite borne (les arguments probabilistes laissent à penser qu'elle sera comprise entre 106 et 115), en tabulant les arguments dont la borne sera jugée trop grande. Lorsque l'argument n'a pas été testé (quadruple précision par exemple), si jamais on tombe sur une grande borne nécessitant des calculs à grande précision, 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 très peu de chances d'être nécessaire) sera de l'ordre de quelques minutes.

– La stratégie multi-niveaux de Ziv permet d'évaluer la plupart des opérations avec une précision un peu supérieure à n bits. Le temps requis est quasiment le même que celui nécessaire aux bibliothèques existantes pour calculer les fonctions élémentaires sans garantir l'arrondi exact.

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? Si un tel mode d'arrondi est fourni, doit-on prévoir un flag indiquant un arrondi non exact?

– Doit-on prévoir un arrondi exact sur un certain domaine seulement? Comment choisir ce domaine?

– Que doit-il se passer si jamais il n'y a pas assez de mémoire lors d'un calcul à grande précision?

Bibliographie

[Bai87] Bailey (D.H.). – A high-performance FFT algorithm for vector supercomputers. *Journal of Supercomputing*, novembre 1987.

[Bai93] Bailey (D.H.). – Multiprecision translation and execution of fortran programs. *ACM Transactions on Mathematical Software*, vol. 19, n° 3, septembre 1993, pp. 288–319.

[Bak75] Baker (A.). – *Transcendental Number Theory*. – Cambridge University Press, 1975.

[BMZ85] Brassard (G.), Monet (S.) et Zuffellato (D.). – Algorithmes pour l'arithméthique des très grands entiers. *Technique et Science Informatiques*, 1985.

[Bre75] Brent (R.P.). – Multiple precision zero-finding methods and the complexity of elementary function evaluation. *In : Analytic Computational Complexity*, éd. par Traub (J.F.). – Academic Press, New York.

[Bre76] Brent (R.P.). – Fast multiple precision evaluation of elementary functions. *Journal of the ACM*, vol. 23, 1976, pp. 242–251.

[Bre78] Brent (R.P.). – A fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software*, vol. 4, n° 1, mars 1978, pp. 57–70.

[CK93] Collins (G.E.) et Krandick (W.). – A hybrid method for high precision calculation of polynomial real roots. *In : International Symposium on Symbolic and Algebraic Computation (ISSAC)*.

[FG76] Feldstein (A.) et Goodman (R.). – Convergence estimates for the distribution of trailing digits. *Journal of the ACM*, vol. 23, 1976, pp. 287–297.

[Gol91] Goldberg (D.). – What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, vol. 23, n° 1, mars 1991, pp. 5–47.

[Kah89] Kahan (W.). – Paradoxes in concepts of accuracy. *In : Lecture notes from Joint Seminar on Issues and Directions in Scientific Computations, U.C. Berkeley.*

[KJ93a] Krandick (W.) et Johnson (J.R.). – Efficient multiprecision floating point multiplication with optimal directional rounding. *In : 11th Symposium on Computer Artithmetic*, éd. par E.E. Swartzlander (M.J. Irwin) et Jullien (G.). pp. 228–233. – Los Alamitos, CA, juin 1993.

[KJ93b] Krandick (W.) et Johnson (J.R.). – *Efficient Multiprecision Floating Point Multiplication with Exact Rounding*. – Technical Report n° 93–76, Johannes Kepler University, Linz, Austria, RISC-Linz, 1993.

[KM81] Kulisch (U.W.) et Miranker (W.L.). – *Computer Arithmetic in Theory and Practice.* – Academic Press, 1981.

[Kul77] Kulisch (U.W.). – Mathematical foundation of computer arithmetic. *IEEE Transactions on Computers*, vol. C-26, n° 7, juillet 1977, pp. 610–621.

[MT96] Muller (J.M.) et Tisserand (A.). – Towards exact rounding of the elementary functions. *In: Scientific Computing and Validated Numerics (proceedings. of SCAN'95)*, éd. par Alefeld (Frommer) et Lang. – Akademie Verlag.

[Mul89] Muller (J.M.). – *Arithmétique des Ordinateurs.* – Masson, Paris, 1989.

[NW95] Nesterenko (Yu. V.) et Waldschmidt (M.). – On the approximation of the values of exponential function and logarithm by algebraic numbers. *A paraître*, 1995.

[Pic72] Pichat (M.). – Correction d'une somme en arithmétique à virgule flottante. *Numerische Mathematik*, vol. 19, 1972, pp. 400–406.

[Pri91] Priest (D.M.). – Algorithms for arbitrary precision floating point arithmetic. *In: Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, éd. par Kornerup (P.) et Matula (D.). pp. 132–144. – IEEE Computer Society Press.

[SS93] Schulte (M.) et Swartzlander (E.E.). – Exact rounding of certain elementary functions. *In: 11th Symposium on Computer Artithmetic*, éd. par E.E. Swartzlander (M.J. Irwin) et Jullien (G.). pp. 138–145. – Los Alamitos, CA, juin 1993.

[SVH89] Serpette (B.), Vuillemin (J.) et Hervé (J.C.). – *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic.* – Rapport technique n° 2, Digital Paris Research Laboratory, mai 1989.

[Zim92] Zimmermann (P.). – Comparison of three public-domain multiprecision libraries: Bignum, gmp and pari. – février 1992.

[Ziv91] Ziv (A.). – Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, vol. 17, n° 3, septembre 1991, pp. 410–423.

[Zur94] Zuras (D.). – More on squaring and multiplying large integers. *IEEE Transactions on Computers*, vol. 43, n° 8, août 1994, pp. 899–908.