École Normale Supérieure de Lyon

Magistère d'Informatique de Modélisation, 1ère année

# Calcul sur les expansions de taille fixe

Brice Goglin

Stage encadré par Claire Moreau

Laboratoire d'Informatique et du Parallélisme — Équipe Arénaire

Mai-Juin 2000

# Introduction

La complexité des problèmes rencontrés en physique conduit les scientifiques à les résoudre numériquement. La précision des calculs prend alors une grande importance. Les astrophysiciens ne sauraient en effet se satisfaire de quelques décimales quand il s'agit d'envoyer une sonde se poser sur Mars.

Les informaticiens ont donc développé différentes méthodes visant à mener des calculs sur des nombres flottants avec une très grande précision.

L'équipe ARÉNAIRE du Laboratoire d'Informatique et du Parallélisme de l'École normale supérieure de Lyon s'intéresse à ces problèmes d'arithmétique des ordinateurs et de multiprécision.

Une bibliothèque de calcul a été développée pour manipuler des expansions. Il s'agit d'une technique de représentation des réels sur plusieurs nombres à virgule flottante. Son principal avantage est de faire de la multiprécision tout en évitant le stockage des zéros intermédiaires.

Cette bibliothèque utilise actuellement des expansions de taille quelconque, c'est-à-dire que le nombre de flottants utilisés pour la représentation d'une expansion est variable.

Un des domaines actuellement étudié est le calcul en ligne qui consiste en un calcul progressif des différentes composantes d'une expansion. Cela impose de traiter tout d'abord les composantes de poids fort. On est alors conduit à s'intéresser à des pseudo-expansions, ce qui simplifie par ailleurs l'implantation des algorithmes.

S'il peut être nécessaire de travailler avec une grande précision, il semble inutile d'utiliser un nombre arbitrairement grand de composantes. Travailler avec des expansions de taille fixe semble donc être une voie intéressante, même si en tronquant ainsi les expansions, on perd l'exactitude des résultats.

J'ai été amené au cours de mon stage à implanter des algorithmes effectuant les opérations de base sur des expansions de taille fixe, puis à tester leur efficacité et la précision des résultats obtenus.

# Table des matières

# Chapitre 1

# Les flottants

Il y a une vingtaine d'années, les constructeurs d'unités de calcul en virgule flottante utilisaient tous des tailles, précisions et techniques d'arrondis différentes, rendant ainsi très difficile la réalisation de programmes ou de bibliothèques mathématiques portables.

Dans les années 80, un groupe de travail sur les flottants crée la norme IEEE 754, visant à définir précisément toute l'arithmétique flottante des ordinateurs et ainsi faciliter la portabilité des applications.

## 1.1 Stockage des nombres flottants

La norme impose l'implantation de deux types de flottants, **single** et **double**. Les **double-extended** et **quadruple** ne sont pas imposés mais sont souvent implantés par les constructeurs.

### 1.1.1 Stockage en mémoire

La méthode de stockage de ces nombres est la suivante : on dispose de $K$ bits pour l'exposant (qui fixe l'ordre de grandeur du nombre), $N$ pour la mantisse (qui précise les décimales du nombre) et 1 pour le signe. Le tableau ci-dessous décrit les valeurs de $N$ et $K$ imposés par la norme IEEE-754 pour les single et double, et celles couramment utilisées pour les double-extended et quadruple.

| Format | Octets | $K$ | $N$ |
|---|---|---|---|
| Single | 4 | 8 | 23 |
| Double | 8 | 11 | 52 |
| Double-Extended | $\geqslant 10$ | $\geqslant 15$ | $\geqslant 63$ |
| Quadruple | 16 | 15 | 112 |

Ainsi, la mantisse prend des valeurs entre 0 et $2^N - 1$. En fait, on considère que les $N$ bits de la mantisse sont les $N$ décimales suivant la virgule dans un nombre commençant par un 1. Ainsi, on n'a pas besoin de stocker ce premier chiffre. La mantisse varie donc entre 1 et $1,11\ldots 11 = 2 - 2^{-N}$.

L'exposant n'est pas stocké avec un signe. Pour gérer les exposants négatifs, on stocke en fait l'exposant augmenté de $2^{K-1} - 1$. Ainsi, en stockant entre 0 et $2^K - 1$, on peut gérer des exposants entre $-2^{K-1} + 1$ et $2^{K-1}$. On dit qu'on stocke un *exposant biaisé.*

Finalement, avec $d_0, \ldots, d_{N-1}$ les bits de mantisse, $e_0, \ldots, e_{K-1}$ les bits d'exposant, et $s$ le bit signe, le nombre représenté est :

$$(-1)^s \times 1, d_0 \ldots d_{N-1} \times 2^{e_0 \ldots e_{K-1} - 2^{K-1} + 1}$$

On peut remarquer que la valeur absolue de ce nombre se situe toujours dans l'intervalle $\left[2^e, 2^{e+1}\right[$ avec $e$ l'exposant $e_0 \ldots e_{K-1} - 2^{K-1} + 1$.

### 1.1.2 Gestion des infinis, zéros, dénormalisés et Nans

Les valeurs extrèmes de l'exposant ne sont pas utilisées pour le stockage des nombres normaux :

- ⋆ Si l'exposant est maximal et la mantisse nulle, on représente en fait un **infini**, dont le signe est précisé par le bit de signe ;
- ⋆ Si l'exposant est maximal et la mantisse non nulle, on réprésente un **NaN** (voir 1.4) ;
- ⋆ Si l'exposant est minimal et la mantisse nulle, on représente 0. A noter que le bit de signe permet de distinguer 0+ et 0- (c'est notamment utile pour la division par zéro. Voir 1.3) ;
- ⋆ Si l'exposant est minimal et la mantisse non-nulle, on représente les nombres **subnormaux** ou **dénormalisés**. Ces nombres permettent de distinguer différents ordres d'Underflow (voir 1.3).

## 1.2 Techniques d'arrondi

La norme IEEE 754 impose la gestion de 4 techniques d'arrondis, afin que l'utilisateur puisse choisir lui-même l'arrondi qu'il va utiliser. Dans tous les cas, le résultat des 4 opérations de bases, l'extraction de racine carrée et diverses conversions entre flottants et entiers, doit toujours être correctement arrondi.

On a le choix entre :

- ⋆ l'arrondi vers $+\infty$,
- ⋆ l'arrondi vers $-\infty$,
- ⋆ l'arrondi vers 0,
- ⋆ l'arrondi au plus près.

Par défaut, c'est l'arrondi au plus près qui est utilisé.

## 1.3 Les exceptions

Au cours du calcul, si des opérations incorrectes apparaissent, le programme va lever certains drapeaux afin de signaler à l'utilisateur que quelquechose de particulier s'est produit. On dit qu'on a **lancé une exception**.

Il existe 5 types d'exceptions :

- ⋆ Invalid : signale qu'une opération a engendré un NaN (voir 1.4) ;
- ⋆ Divide by zero : renvoie un infini du même signe que le 0 utilisé ;
- ⋆ Overflow : signale un dépassement de capacité vers le haut, le résultat est un infini ;
- ⋆ Underflow : signale un dépassement de capacité vers le bas, le résultat est un dénormalisé ou un zéro signé ;
- ⋆ Inexact : signale que le résultat a dû être arrondi.

A la fin d'un calcul, l'utilisateur peut vérifier les bits des différentes exceptions pour en déduire les éventuels problèmes qui se sont présentés au cours du calcul.

## 1.4 Les NaNs

Lorsqu'un calcul impossible apparaît, il faut pouvoir en rendre compte à l'utilisateur. On renvoie alors un NaN (*Not a Number*). Par défaut, l'obtention d'un NaN ne lance pas d'exception sauf dans certains cas. On parle alors de SNaNs (*Signaling* NaN). C'est l'exception Invalid qui est lancée.

### 1.4.1 Ce qu'impose la norme IEEE-754

Par exemple, si au cours d'un calcul on tente d'extraire la racine carrée d'un nombre négatif, le résultat renvoyé sera un NaN. En fait, seule une classe d'opérations dites *invalides* peut renvoyer des NaNs.

Ces opérations sont :

- ⇒ $\sqrt{x}$ avec $x < 0$ ;
- ⇒ $\infty - \infty$ ;
- ⇒ $0 \times \infty$ ;
- ⇒ $0/0$ ;
- ⇒ $\infty/\infty$ ;
- ⇒ $x$ mod 0 avec $x$ quelconque ;
- ⇒ $\infty$ mod $x$ avec $x$ quelconque.

### 1.4.2 Les cas non-mentionnés

Dans l'idéal il faudrait renvoyer un NaN pour certaines autres opérations, non mentionnées par la norme IEEE-754, parmi lesquelles celles du type :

⇒ $\sin(\infty)$ ;
⇒ $x^y$ avec $x < 0$ et $y$ non entier ;
⇒ $\log(x)$ avec $x < 0$ ;
⇒ $\mathrm{Arcsin}(x)$ avec $|x| > 1$.

On trouvera de plus amples informations dans [5]. Par exemple, il pourrait être intéressant de préciser certaines formes comme $\log(\pm 0) = -\infty$ ou encore $(\pm 0)^x = \infty$ pour $x < 0$ pair, ...

### 1.4.3 Règles de calcul sur les NaNs

Lorsqu'une exception INVALID est lancée, le calcul se poursuit. On a donc dû définir des règles arithmétiques sur les NaNs de même que sur les nombres normaux. Comme un NaN peut en fait être n'importe quel nombre, notamment l'infini, un calcul sur un NaN renvoie un NaN. Pour les opérations de base, c'est toujours justifié.

Cependant, dans certains cas, ce n'est pas obligatoire. Par exemple, si l'utilisateur implémente une fonction $f(x,y) = \sqrt{x^2 + y^2}$. Une implémentation naïve renverra un NaN pour $f(\mathrm{NaN}, \infty)$. Pourtant, on peut constater que le resultat est en fait toujours $+\infty$ (voir [5]). La gestion des NaNs permet donc à l'utilisateur de tester ses entrées pour éventuellement faire disparaître un NaN dans un calcul.

On a également dû préciser les résultats des opérations de test avec des NaNs. Le résultat de $x = x$ où $x$ est un NaN doit renvoyer `false`. Cependant, un compilateur qui optimise les comparaisons va remplacer $x = x$ par `true` et donc interdire la gestion correcte des NaNs. Les compilateurs qui optimisent le code ainsi violent la norme IEEE 754. Heureusement, les compilateurs modernes ne font plus ce genre d'erreurs.

# Chapitre 2

# Les expansions

Si on multiplie des nombres quelconques $a$ et $b$, le nombre de décimales du résultat est souvent supérieur à ceux de $a$ et $b$. En fait, un résultat plus précis a été obtenu par Dekker et Knuth, indépendamment l'un de l'autre : le produit de deux flottants peut toujours s'écrire en une somme de deux flottants.

Cette remarque permet d'affirmer qu'on est toujours capable de faire une multiplication exacte, à condition de stocker le résultat dans non plus un seul mais deux flottants consécutifs. Ainsi, la possibilité pour un réel d'être représenté par une somme de flottants est conservée par somme et produit. Cela permet donc de faire du calcul exact. C'est ce concept qui est à la base du calcul avec des expansions.

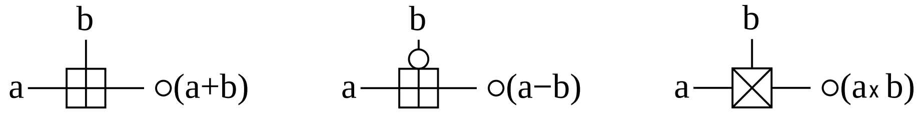
## 2.1 Addition et multiplication exactes

### 2.1.1 Notations

La norme IEEE 754 impose à une unité de calcul en virgule flottante de renvoyer un produit ou une somme de flottants arrondi correctement.

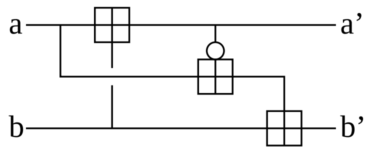
Afin de simplifier les notations, nous noterons dans toute la suite $\circ(x)$ le flottant résultant de l'arrondi correct du réel $x$. Si $a$ et $b$ sont des nombres représentables exactement en machine, c'est-à-dire $a = \circ(a)$ et $b = \circ(b)$, alors la norme impose aux unités de calcul de renvoyer $\circ(a \times b)$ comme résultat du produit des flottants $a$ et $b$.

Dans les schémas, on utilisera les opérateurs addition, soustraction et multiplication usuels sur les flottants, notés comme suit :



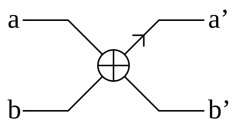
### 2.1.2 Addition exacte de deux flottants

Considérons tout d'abord deux réels $a$ et $b$ d'ordres de grandeur proches. Les unités de calcul en virgule flottante peuvent déterminer le flottant le plus proche de la somme exacte $a+b$. Il s'agit de $\circ\,(a+b)$. Pour obtenir la somme exacte, il nous faut calculer l'erreur commise. On va donc déterminer le réel $a+b\ -\ \circ\,(a+b)$. On obtient alors une écriture exacte du réel $a+b$, décomposée en deux flottants, ne se recouvrant pas.




On a représenté ci-contre la méthode dans le cas $|a|>|b|$. On a ici $a' = \circ\,(a+b)$ et $a'+b' = a+b$. Les deux composantes $a'$ et $b'$ obtenues ne se superposent pas du tout.

Il existe un algorithme similaire pour traiter l'addition dans le cas général que nous ne développerons pas ici. On a représenté ci-contre le schéma que nous utiliserons par la suite pour une addition, renvoyant la grande composante dans $a' = \circ\,(a+b)$ et le reste dans $b'$. Le code source est inséré en annexe B.1.1.




### 2.1.3 Produit exact

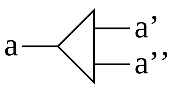
Si on écrit un flottant sous la forme d'une somme de puissances de 2, on voit que ces puissances varient dans un intervalle de la longueur $N$ de la mantisse. Le produit de deux flottants $a$ et $b$ s'écrit donc en une somme de puissances de 2, et ces puissances varient dans un intervalle de deux fois la longueur $N$ de la mantisse.

En effet, soient deux flottants $a = 2^e \sum_{i=0}^{N-1} d_i 2^i$ et $b = 2^{e'} \sum_{j=0}^{N-1} d'_j 2^j$.

Alors : $\qquad ab = 2^{e+e'} \sum_{k=0}^{2N-1} d''_k 2^k \qquad$ avec $\qquad d''_k = \sum_{i+j=k} d_i d'_j$

En séparant les décimales $d_0, \ldots, d_{N-1}$ et $d_N, \ldots, d_{2N-1}$, on obtient une écriture du produit sous la forme d'une somme de deux flottants sans recouvrement.

En fait, il existe un algorithme permettant de faire ce calcul de façon plus efficace, et surtout plus facile à implémenter :




On renvoie tout d'abord l'arrondi du produit des deux flottants. Pour calculer le reste, on coupe tout d'abord chaque flottant en deux. L'opérateur coupant un flottant $a$ en un grand flottant $a'$ et un petit $a''$ a été représenté ci-dessous. Son code source est inséré en annexe B.1.2.

Ensuite, on calcule les quatre sous-produits, qui tiennent chacun entièrement dans un flottant. Des additions ou soustractions bien ordonnées entre l'arrondi initial et ces sous-produits conduisent alors au reste. On a représenté à la figure 2.1 le schéma récapitulatif de cet algorithme.

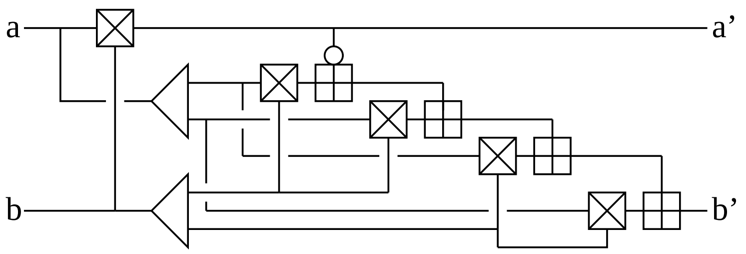
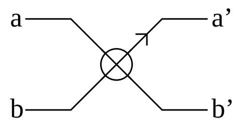



Fig. 2.1 – Principe du produit exact de deux flottants

On a représenté ci-contre le schéma que nous utiliserons par la suite pour un produit de deux flottants $a$ et $b$, renvoyant la grande composante dans $a' = \circ(a \times b)$ et le reste dans $b'$. On a $a \times b = a' + b'$. Le code source est inséré en annexe B.1.3.




## 2.2 Définitions

### 2.2.1 Expansions et composantes

On appelle **expansion** un nombre écrit sous la forme $x_0 + \ldots + x_{n-1}$ où les $x_i$ sont des nombres flottants appelés **composantes**. On impose aux composantes d'être triées en ordre décroissant de leur valeur absolue. Leur signe peut par contre être quelconque.

La définition des expansions impose la présence d'un zéro entre chaque composante. Les différentes composantes ne peuvent donc pas se recouvrir. Cette propriété peut paraître étrange dans la mesure où il est facile de trouver des nombres ne la vérifiant pas. Mais on voit sur l'exemple suivant, qu'il est toujours possible de se ramener à cette écriture.

### 2.2.2 Un zéro entre chaque composante ?

Considérons des flottants dont la mantisse s'écrit sur 3 bits. On cherche à représenter le nombre binaire 11111 par une expansion. On pose donc $x_0 = 11100 = 1,11 \times 2^4$. Cependant, si on prend $x_1 = 11 = 1,1 \times 2^1$ comme seconde composante, la propriété qui impose un 0 entre chaque composante n'est pas vérifiée.

En remarquant que $11 = 100 - 1$, on décompose $x_1$ en un flottant de l'ordre de $x_0$ et un autre flottant bien séparé de $x_0$. On réinjecte la première partie dans $x_0$ et on laisse la seconde seule dans $x_1$. On pose donc $x_0 = 11100 + 100 = 100000 = 1,00 \times 2^6$ et $x_1 = -1$. La propriété est alors vérifiée.

### 2.2.3 Pseudo-expansions et calcul en ligne

La technique précédente peut bien-sûr s'adapter à n'importe quel cas susceptible de se présenter. Cependant, elle est lourde à implémenter. On travaille désormais plutôt sur des **pseudo-expansions**. C'est-à-dire qu'on s'autorise un recouvrement de quelques derniers bits d'une composante par les premiers de la suivante.

Cela simplifie beaucoup les algorithmes, et il suffit d'une opération simple à la fin des calculs pour réécrire proprement le résultat, sans recouvrement. Cependant, ce n'est pas leur principal intérêt.

Le **calcul en ligne** consiste à utiliser au fur à mesure les composantes en entrée et à sortir immédiatement les composantes pertinentes. On n'a pas besoin de faire tous les calculs intermédiaires pour commencer à obtenir le résultat. On ne fait entrer et on ne calcule que ce qui est nécessaire pour obtenir la précision désirée en sortie.

Cependant, cette technique impose de travailler en commençant par les grandes composantes et interdit des opérations de réarrangement des résultats intermédiaires. L'utilisation des pseudo-expansions est alors indispensable.

## 2.3 Passage en taille fixe

Munis des addition et multiplication exactes, il nous est maintenant possible d'ajouter ou de multiplier des pseudo-expansions entre elles, pour fournir une pseudo-expansion en résultat.

Dans toute la suite, l'entier $n$ représentera le nombre de composantes des expansions manipulées, quand il sera fixé.

Lorsqu'on réalise le produit de deux expansions de taille $n$, on doit calculer les produits de chaque composante de la première expansion, avec toutes celles de la seconde. A chaque fois, on obtient deux composantes. Finalement, le résultat s'écrit sur $n \times n \times 2 = 2n^2$ composantes. Et même si leur sommation puis leur réorganisation pour minimiser les recouvrements va réduire ce nombre, le nombre de décimales obtenues reste très supérieur à celui du départ.

Comme la précision nécessaire n'est jamais infinie, il peut être intéressant de limiter le nombre de composantes des expansions obtenues. Nous allons donc expliciter les algorithmes de calcul sur des expansions de taille fixe.

# Chapitre 3

# Opérations sur les expansions

Les algorithmes que nous allons présenter dans la suite de ce chapitre travaillent *bits de poids fort en tête*. Ils utilisent tout d'abord les grandes composantes entrantes, et produisent également d'abord des grandes composantes.

## 3.1 Addition

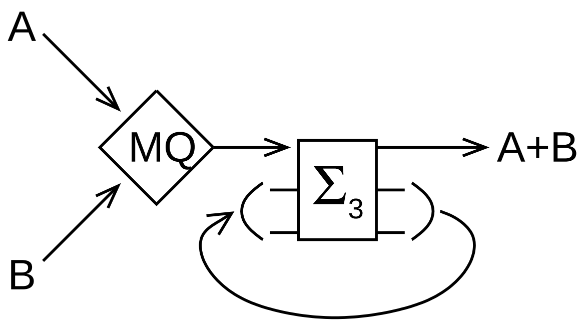



FIG. 3.1 – Principe de l'addition en ligne

Le principe de base de l'addition en ligne (figure 3.1) est de prendre continuellement des composantes en entrée, et d'en sortir dès qu'elles sont pertinentes. On utilise un opérateur MQ (*Merge Queue*) afin de trier les composantes des deux expansions en ordre décroissant de leur valeur absolue.

Ensuite, l'opérateur $\Sigma_3$ (figure 3.2) se charge de fusionner trois composantes et de produire en sortie trois composantes sans recouvrement.

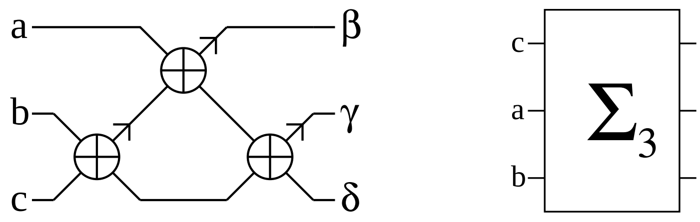



FIG. 3.2 – L'opérateur $\Sigma_3$

Il suffit alors de tester le nombre de composantes nulles en sortie pour en déduire la pertinence de la plus grande : si la troisième composante ($\delta$) est non-nulle, on sait que la première ($\beta$) ne sera quasiment plus modifiée par les composantes entrantes (l'ordre décroissant l'assure). $\beta$ est donc pertinente. On peut donc la sortir, c'est une composante du résultat, et $\delta$ et $\gamma$ sont accumulées en attendant le prochain passage dans $\Sigma_3$. Dans le cas contraire, on accumule $\beta$ et $\gamma$.

Ainsi, des composantes pertinentes sortent au fur et à mesure de l'entrée des composantes de deux expansions à additionner. L'algorithme s'arrête lorsqu'on a fait entrer entièrement les deux expansions, ou quand on a obtenu le maximum autorisé de composantes en sortie.

L'implantation de cet algorithme en taille fixe reste assez proche du code de la bibliothèque en taille quelconque. Le principal changement qui intervient est un test du nombre de composantes sorties à chaque fois qu'une nouvelle est produite. Le code source de l'algorithme est accessible en annexe B.2.1.

## 3.2 Multiplication

La multiplication implantée naïvement imposerait de faire tous les produits partiels de composantes des expansions entrantes. Le tri de ces produits serait trop coûteux.

Ici, on utilise (figure 3.3) une queue de priorité (PQ) dont chaque file est constituée de tous les produits partiels de composantes d'une expansion par une seule de l'autre. C'est l'opérateur PP (*partial products*) qui se charge de ces calculs.

On peut ainsi entrer tous les produits partiels dans l'ordre décroissant de leur valeur absolue. Il faut noter qu'on entre des couples de flottants, car un produit de flottants ne peut pas toujours s'écrire sur un seul flottant (voir 2.1.3).

On utilise ensuite un opérateur $\Sigma_5$ chargé de réorganiser les cinq entrées et de sortir cinq composantes ordonnées, sans recouvrement, éventuellement nulles, et tel que la somme des composantes en entrée est égale à la somme des composantes en sortie. On teste alors le nombre de composantes nulles en sortie pour en déduire la pertinence des premières.

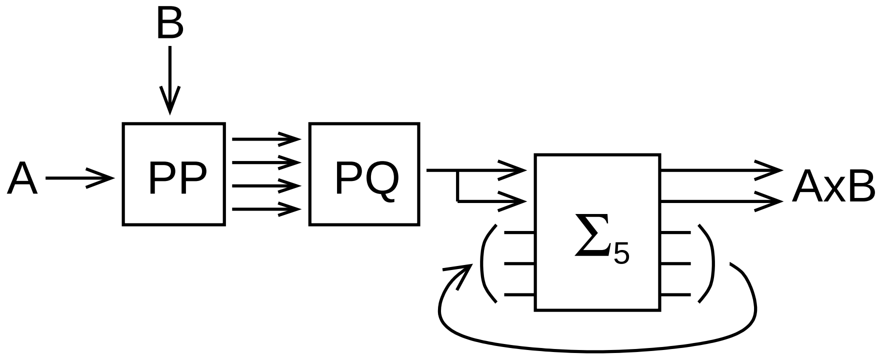



FIG. 3.3 – Principe de la multiplication en ligne

Implanter cet algorithme en taille fixe conduit à adapter légèrement les structures de données utilisées en taille quelconque. Ensuite, comme pour l'addition, on effectue un test du nombre de composantes sorties et, le cas échéant, l'algorithme s'arrête. Le code source de l'algorithme est accessible en annexe B.2.2.

## 3.3 Division

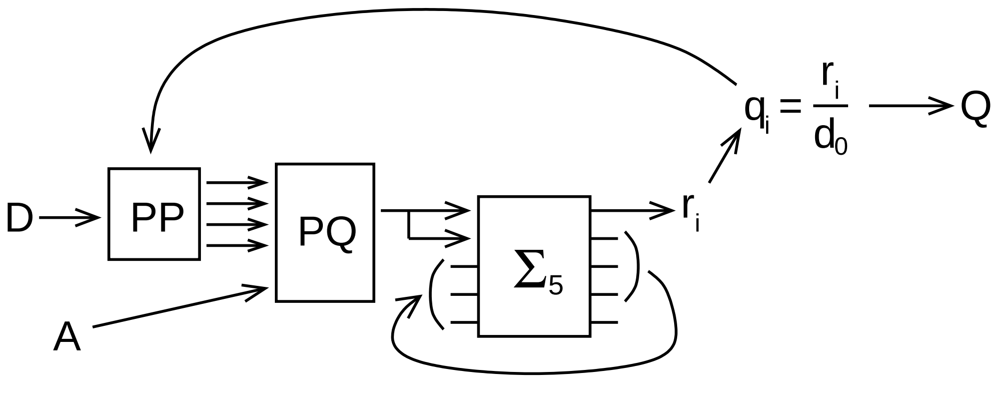



FIG. 3.4 – Principe de la division en ligne

On note ici (figure 3.4) $D = d_0 + \ldots + d_{n-1}$ le diviseur, $d_0$ sa plus grande composante, $A$ le dividende et $Q = q_0 + \ldots + q_{n-1}$ le quotient. $r_i$ représente le reste obtenu après la $i$-ème étape.

Pour diviser deux expansions, on va raisonner de la même façon qu'un enfant pose une division entière. A chaque étape, on regarde la partie prédominante du reste, on la divise par le diviseur, et on en déduit le nouveau reste. Ici, la division par le diviseur va être remplacée par la division par sa composante principale. Ainsi, à chaque étape, on a seulement une division de flottants à effectuer.

Chaque étape de l'algorithme consiste donc à évaluer $r_i$, la composante pertinente du reste. On effectue pour cela plusieurs itérations dans les opérateurs PP, PQ et $\Sigma_5$ vus plus haut. On divise ce reste par $d_0$, la grande composante du diviseur. On en déduit une composante du quotient. On réinjecte alors cette composante dans PP avant de débuter une nouvelle étape. Il faut noter que la composante n'est pas tout à fait exacte car on ne divise pas par le diviseur mais par sa plus grande composante. On produit donc une petite erreur, qui sera corrigée par l'étape suivante. Ainsi, à chaque étape une composante est produite.

Dans le cas de la division, le travail en taille fixe prend un grand intérêt car il permet d'assurer la terminaison des algorithmes. En effet, en taille quelconque, il est très facile de trouver des divisions qui conduisent à un nombre infini de composantes. Ici, dès qu'on a suffisamment de composantes, l'algorithme s'arrête. En taille quelconque, on serait obligé d'imposer un point d'arrêt arbitraire pour éviter que des divisions non exactes fassent tourner l'algorithme indéfiniment. Ici, il suffit de tester le nombre de composantes produites pour stopper l'algorithme au bon moment. Le code source de l'algorithme est accessible en annexe B.2.3.

## 3.4 Racine carrée

Même si je n'ai pas eu le temps d'implémenter un algorithme, j'ai pu envisager une méthode de calcul de racine carrée.

En utilisant le fait que le développement limité au premier ordre de $\sqrt{1+x}$ est $1 + \frac{x}{2}$, on peut calculer une nouvelle approximation de la racine carrée d'une expansion à partir de la précédente et de l'erreur correspondante.

Si après $k$ étapes de calcul, on suppose connaître $b_0 + \cdots + b_{k-1}$ une valeur approchée de $\sqrt{a_0 + \ldots + a_{n-1}}$, on note $\epsilon$ l'erreur $(a_0 + \ldots + a_{n-1}) - (b_0 + \ldots + b_{k-1})^2$. On peut alors constater que la composante suivante sera à peu près $b_k = \frac{\epsilon}{2b_0}$.

En prenant $b_0 = \sqrt{a_0}$ et en utilisant les opérateurs proches de ceux de la division, on peut implémenter facilement cet algorithme. Ici aussi, le travail en taille fixe permet d'assurer la terminaison de l'algorithme, contrairement au cas en taille quelconque.

# Chapitre 4

# Précision et efficacité

## 4.1 Précision des algorithmes

### 4.1.1 Addition

L'implantation de l'addition et de la multiplication fait qu'au cours de leur l'exécution, la somme des composantes pertinentes sorties, des composantes restant en entrée et les parties accumulées dans $\Sigma_3$ ou $\Sigma_5$ est constante et égale à la valeur théorique du résultat.

Si le résultat exact peut-être stocké sur le nombre de composantes qu'on manipule, le résultat obtenu par les algorithmes sera exact. Sinon, il subsistera une petite erreur. Cette erreur est la somme des composantes accumulées dans $\Sigma_3$ ou $\Sigma_5$, et des composantes restant en entrée.

Considérons l'addition de deux expansions $A = a_0 + \ldots + a_{n-1}$ et $B = b_0 + \ldots + b_{m-1}$. Soit $a_i$ la dernière composante entrée de $A$, et $b_j$ celle de $B$. Supposons que $b_j$ est entrée après $a_i$.

Plaçons-nous au moment où l'algorithme termine car la $n$-ième composante du résultat vient d'être produite. Cette composante est nécessairement la sortie $\beta$ de $\Sigma_3$. Notons $\beta = (-1)^s \times 1, ...... \times 2^e$.

Comme les sorties de $\Sigma_3$ ne se recouvrent pas, on a nécessairement $|\gamma| < 2^{e-N}$ et $|\delta| < 2^{e-2N-1}$ ($N$ est le nombre de bits de la mantisse). Or, pour qu'une composante $\delta$ si faible ait été produite en sortie de $\Sigma_3$, il faut que la dernière composante entrée soit suffisamment petite,

Soit $$\boxed{|b_j| < 2^{e-N-1}}$$

Comme $a_{i+1}$ n'est pas entrée, on a également $\boxed{|a_{i+1}| < 2^{e-N-1}}$.

Par ailleurs, on sait que les pseudo-expansions en entrée présentent peu de recouvrement. Soit $D$ le nombre maximal de bits de recouvrements.

$$\forall k \in \{i+2, \ldots, n-1\}, \qquad |a_k| < 2^{e-2N-2+D}$$

Et $$\forall k \in \{j+1, \ldots, m-1\}, \qquad |b_k| < 2^{e-2N-2+D}$$

L'erreur totale s'écrit

$$\begin{array}{rccccccc}\Delta = | & \gamma & + \quad \delta & + (a_{i+1} + \ldots + a_{n-1}) + & b_{j+1} & + (b_{j+2} + \ldots + b_{m-1})\,| \\ < & 2^{e-N} & + 2^{e-2N-1} + & 2^{e-2N-2+D} \times n & + 2^{e-N-1} + & 2^{e-2N-2+D} \times m\end{array}$$

D'où $$\boxed{\Delta < 2^{e-N+1}}$$

L'erreur est donc strictement inférieure au dernier bit du résultat. Cela nous permet d'affirmer que le résultat ainsi obtenu est exact sauf peut-être son dernier bit.

### 4.1.2 Multiplication

Dans le cas de la multiplication, le raisonnement est assez similaire. Cependant, une grosse différence apparaît lorsqu'on évalue les composantes restant en entrée. Il peut en effet rester jusqu'à $n-1$ composantes de l'ordre de $2^{e-N-1}$, alors que dans le cas de l'addition, cela ne peut être le cas que de $a_{j+1}$.

L'erreur commise est alors majorée par

$$2^{e-N} + (n-1) \times 2^{e-N-1} < 2^{e-N} + 2^{e-N-1+\lceil \log_2(n-1) \rceil}$$

Si le nombre de composantes $n$ vaut 5, on obtient $2^{e-N} + 2^{e-N+1}$. Les deux derniers bits du résultat peuvent alors être faux.

### 4.1.3 Division

Au cours de la division, les principales approximations sont dues à la division par la grande composante du diviseur au lieu du diviseur exact. En effet, comme on autorise des recouvrements de quelques bits, la première composante n'est égale au diviseur entier qu'aux derniers bits près. Si on note $D$ le nombre de bits de recouvrement autorisé, le rapport entre la première composante du diviseur et le diviseur entier, est compris entre $1 - 2^{D-N}$ et $1 + 2^{D-N}$ (cela revient à dire que l'erreur relative est majorée par $2^{D-N}$).

Le rapport entre le résultat de la division par le diviseur entier et celle par la première composante est alors également compris entre ces deux valeurs. L'erreur sur une division est donc au pire de $D$ bits.

A chaque étape, l'erreur obtenue à l'étape précédente est corrigée. Cependant l'erreur sur dernière composante sortie subsiste. Au final, le résultat de la division est donc exact sauf peut-être les $D$ derniers bits de sa dernière composante. Le phénomène étudié dans le cas de l'addition et de la multiplication semble ici négligeable.

## 4.2 Efficacité comparée

### 4.2.1 Résultats bruts pour 3 composantes

Les tests ont été réalisés avec la librairie `flops`. Il s'agit de 8 modules différents qui calculent les valeurs de $\pi$ et de différentes intégrales par la méthode des trapèzes. Les valeurs obtenues sont données en MegaFlops. Il s'agit donc d'un nombre d'opérations flottantes par seconde. On effectue un certain nombre d'étapes de calculs, chacune étant constituée de 13 additions, 10 multiplications et 1 division.

La librairie originale effectuait uniquement des opérations flottants classiques (sur des doubles). Ainsi, les valeurs renvoyées permettent de juger de l'efficacité de l'architecture testée. On trouvera les valeurs obtenues pour de nombreuses stations différentes à l'adresse suivante :

`http ://performance.netlib.org/performance/html/PDStop.html`

J'ai adapté cette librairie pour pouvoir tester mes algorithmes de calcul sur les expansions, pour différents nombres de composantes. Ensuite, j'ai pu comparer ces valeurs avec celle obtenues pour des calculs sur des flottants normaux, sur différentes architectures.

On a présenté dans le tableau 4.1 les résultats obtenus pour les expansions sur un AMD Athlon K7 à 650 MHz. Il s'agit ici de la première valeur d'efficacité renvoyée par `flops`. Les calculs avec 3 composantes se font donc aussi vite qu'avec des doubles sur un Acorn (modèle datant de 10 ans).

| Nombre de composantes | Performances (MegaFlops) | Modèle équivalent en flottants |
|---:|---:|---|
| 1 | 2,4043 | 68040 à 25 MHz |
| 3 | 0,7975 | Acorn Risc PC 610 |
| 5 | 0,3820 | Atari Falcon (68030 à 20 MHz) |
| 10 | 0,1114 | Sun 3/60 (68020 à 20 MHz) |

FIG. 4.1 – Performances obtenues sur les expansions (AMD K7 à 650 MHz).

### 4.2.2 Évaluation du nombre d'opérations flottantes

Le même K7 650MHz utilisé avec des doubles donne une efficacité de 650 MegaFlops. Les calculs sur 3 composantes sont donc environ 250 fois plus lents. Pour essayer de justifier ceci, j'ai tenté d'évaluer le nombre d'opérations flottantes nécessaires pour effectuer des opérations sur les expansions à trois composantes. Les résultats sont présentés dans le tableau 4.1.

Ces valeurs ne sont que des ordres de grandeurs mais tendent à justifier la relative lenteur des algorithmes manipulant des expansions.

| | Additions | Produits | Divisions | Comparaisons |
|---|---|---|---|---|
| Addition d'expansions | 90 | | | 5 |
| Produit d'expansions | 400 | 60 | | 40 |
| Division d'expansions | 230 | 60 | 3 | 50 |

Tab. 4.1 – Nombre d'opérations flottantes nécessaires aux opérations sur les expansions de taille 3

### 4.2.3 Évolution avec le nombre de composantes

La figure 4.2 représente les performances pour différentes valeurs du nombre de composantes. On a utilisé une échelle logarithmique en ordonnées. Les performances semblent évoluer avec le nombre de composantes élevé à une puissance de l'ordre de 1,6.

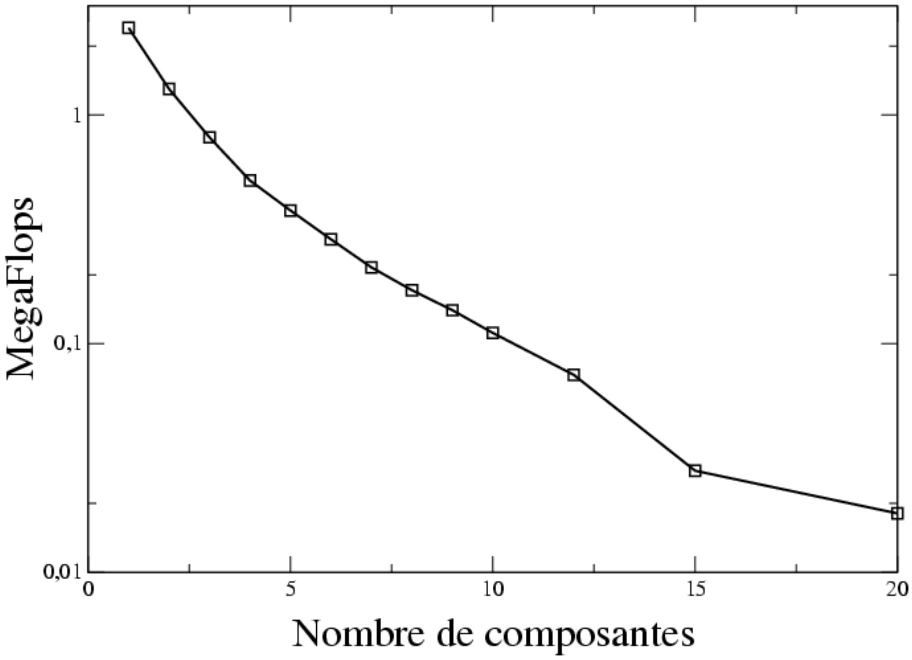



Fig. 4.2 – Évolution des performances avec le nombre de composantes.

# Conclusion

Les expansions constituent une alternative originale aux techniques classiques de multiprécision. L'augmentation de la précision par l'accroissement de l'espace mémoire est ici optimisé afin de ne pas stocker inutilement les zéros intermédiaires. Cependant, cela conduit à une réduction importante de l'efficacité des l'algorithmes.

Les algorithmes initiaux permettaient des calculs avec une précision illimitée. En envisageant des algorithmes travaillant en taille fixe, on restreint la quantité de calculs nécessaires, tandis que la précision reste importante et le calcul en ligne possible.

Comme pour toute librairie de multiprécision, les algorithmes implémentés restent assez lents, mais assurent l'exactitude des résultats, sauf éventuellement pour quelques derniers bits. La précision globale du résultat est donc très grande.

Le passage en taille fixe permet par ailleurs d'assurer la terminaison d'algorithmes tels que la division ou l'extraction de racine carrée, sans que le résultat ne soit nécessairement stockable exactement sur un nombre fini de flottants.

# Bibliographie


[1] D. Goldberg — *What Every Computer Scientist Should Know About Floating-Point Arithmetic* — ACM Computing Survey, 1991.

[2] *IEEE Standard for Binary Floating-Point Arithmetic* — IEEE, 1985.

[3] D.H. Bailey — *A Fortran-90 Based Multiprecision System* — RNR Technical Report, 1995.

[4] D.H. Bailey — *A Portable High Performance Multiprecision Package* — RNR Technical Report, 1993.

[5] W. Kahan — *IEEE Standard 754 for Binary Floating-Point Arithmetic* — 1996.

[6] J.R. Shewchuk — *Adaptative Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates* — 1996.

[7] S. Fortune et C.J. Van Wyk — *Efficient Exact Arithmetic for Computational Geometry* — 1993.

[8] P. Soderquist et M. Leeser — *Division and Square Root : Choosing the Right Implementation* — IEEE, 1997.

[9] M.A. Cornea-Hasegan, R.A. Golliver et P. Markstein — *Correctness Proofs Outline for Newton-Raphson Based Floating-Point Divide and Square Root Algorithms* — IEEE, 1999.

# Annexe A

# Test des différentes implémentations

Les tests suivants ont été réalisés avec 5 composantes.

## A.1 Allocation mémoire

La bibliothèque d'origine calcule en taille quelconque. Les allocations mémoires étaient faites de manière statique. On avait fixé une borne suffisamment grande au détriment de l'espace mémoire. Une allocation dynamique aurait beaucoup mieux convenu mais on sait que cette technique ralentit considérablement le programme.

J'ai mené des tests comparatifs d'allocation statique et dynamique. Les résultats parlent d'eux-même (tableau A.1). L'allocation dynamique rend le programme 2,5 fois plus lent, malgré la faible part de code que les allocations mémoire représentent.

| Allocation | MegaFlops |
|---|---|
| Statique | 0,3820 |
| Dynamique | 0,1636 |

TAB. A.1 – Performances pour différents types d'allocation mémoire

## A.2 Appel des sous-routines

Par ailleurs, le code initial utilisait de nombreuses sous-routines appelées par l'intermédiaires de macros `#define`. Quand on sait que cela impose de grandes précautions dans le passage des arguments et rend le code assez complexe, il peut sembler étonnant de ne pas utiliser des appels de fonctions, quitte à les mettre `inline`.

| Type d'appel | MegaFlops |
|---|---|
| `#define` | 0,3820 |
| Appels `inline` | 0,2451 |
| Appels normaux | 0,1319 |

TAB. A.2 – Performances pour différents types d'appel

En fait, là aussi, des tests (tableau A.2) m'ont permis de justifier pleinement l'utilisation des `#define`.

# Annexe B

# Source de la bibliothèque

On détaille ci-après la partie principale de la bibliothèque développée au cours du stage.

## B.1 Les petites macros

Comme on l'a vu à l'annexe A.2, les petites routines sont implémentées par l'intermédiaire de `#define`. On détaille ici les routines algorithmiquement intéressantes, telles qu'elles ont été écrites par J. Shewchuk.

### B.1.1 Somme exacte de deux flottants

La macro ci-après réalise la somme exacte des flottants $a$ et $b$. Elle met la composante principale dans $x$ et le reste dans $y$.

```
#define Two_Sum(a, b, x, y) { \
  x = (REAL) ((a)+(b));       \
  _bvirt = (REAL) (x - (a));  \
  _avirt = x - _bvirt;        \
  _bround = (b) - _bvirt;     \
  _around = (a) - _avirt;     \
  y = _around + _bround;    }
```

### B.1.2 Couper un flottant en deux

Cette macro coupe le flottant $a$ en deux sous-flottants $ahi$ et $alo$. La variable *splitter* est initialisée au début du programme et dépend de la taille du flottant en mémoire. Le type `REAL` sera en effet `double` ou `long double` suivant l'architecture.

```
#define Split(a, ahi, alo)  {   \
  _e = (REAL) (splitter * (a)); \
  _abig = (REAL) (_e - (a));    \
  ahi = _e - _abig;             \
  alo = (a) - ahi;            }
```

### B.1.3 Produit exact de deux flottants

Enfin, voici comment on écrit dans $x$ et $y$ le produit exact de $a$ et $b$.

```
#define Two_Product(a, b, x, y)    { \
  x = (REAL) ((a)*(b));              \
  Split(a, _ahi, _alo);              \
  Split(b, _bhi, _blo);              \
  _err1 = (x) - (_ahi * _bhi);       \
  _err2 = _err1 - (_alo * _bhi);     \
  _err3 = _err2 - (_ahi * _blo);     \
  y = (_alo * _blo) - _err3;         \
}
```

## B.2 Les algorithmes principaux

### B.2.1 Addition

```
void addition_exp(REAL* a, REAL* b, REAL* r) {
  VARS_SOMME;
  bool fini = false;
  REAL A,B=0.,C=0.,A1,B1,C1;
  int aindex, bindex, rlen=0,i;
  // On cherche ou commencent les bits de poids forts
  for(aindex=NB_COMP-1; aindex>=0 && a[aindex]==0.; aindex--);
  for(bindex=NB_COMP-1; bindex>=0 && b[bindex]==0.; bindex--);
  // Tant qu'un indice est encore valide (sinon, ils vallent -1)
  while ( (!fini) && (aindex+bindex>=-1) ) {
    if ( aindex>=0 ) {
      if ( bindex>=0 ) {
        if ( FABS(a[aindex]) > FABS(b[bindex]) )
          A = a[aindex--];
        else
          A = b[bindex--];
      } else
        A = a[aindex--];
    } else
      A = b[bindex--];
    Inserer3(A,B,C,A1,B1,C1);
    if ( C1 ) {
      B=B1; C=C1;
      Ajoute(rlen,r,A1);
    } else {
      B=A1; C=B1;
    }
  }
  // On recupere les restes
  if ( (!fini) && B ) Ajoute(rlen,r,B);
  if ( (!fini) && C ) Ajoute(rlen,r,C);
  // On reordonne
  RetourneEtZeros(rlen,r);
}
```

### B.2.2 Multiplication

```
void multiplication_exp(REAL* a, REAL* b, REAL* r) {
  VARS_QUEUE;
  bool fini = false;
  REAL A,B,C=0.,D=0.,E=0.,A1,B1,C1,D1,E1;
  int rlen = 0;
  // Initialisation et remplissage de la queue de priorité
  Queue_Priorite Q;
  Q_Init(Q);
  for(int i=0; i<NB_COMP; i++)
    if ( b[i] )
      Q_Push(Q,a,b[i]);
  // Tant qu'on peut faire des pops, on en fait
  while ( (!fini) && (Q.nbpop>0) ) {
    Q_Pop(Q,A,B);
    Inserer5(A,B,C,D,E,A1,B1,C1,D1,E1);
    if ( E1 ) {
      C=C1; D=D1; E=E1;
      Ajoute(rlen,r,A1);
      if ( !fini )
        Ajoute(rlen,r,B1);
    } else if ( D1 ) {
      C=B1; D=C1; E=D1;
      Ajoute(rlen,r,A1);
    } else {
      C=A1; D=B1; E=C1;
    }
  }
  // On recupere les restes
  if ( (!fini) && C ) Ajoute(rlen,r,C);
  if ( (!fini) && D ) Ajoute(rlen,r,D);
  if ( (!fini) && E ) Ajoute(rlen,r,E);
  // On reordonne
  RetourneEtZeros(rlen,r);
}
```

### B.2.3 Division

```
void division_exp (REAL* r, REAL* d, REAL* q) {
  VARS_QUEUE;
  bool fini = false;
  int qlen = 0,i,j;
  REAL A,B,C=0.,D=0.,E=0.,A1,B1,C1,D1,E1;
  Queue_Priorite Q;
  REAL dpf; // Grande composante du diviseur
  for(i=NB_COMP-1; i>=0 && d[i]==0.; i--);
  dpf = d[i];
  // Initialisation et remplissage de la queue de priorité
  // Le reste initial est le dividende
  Q_Init(Q);
  Q_Push(Q,r,1);
  init_number(q);
  // Debut de la boucle
  while ( (!fini) && (Q.nbpop>0) ) {
    Q_Pop(Q,A,B);
    Inserer5(A,B,C,D,E,A1,B1,C1,D1,E1);
    // On teste aussi A/dpf pour eviter d'ajouter un zero (underflow)
    if ( E1 && (A1/dpf) ) {
      C=C1; D=D1; E=E1;
      // On va ajouter deux composantes au quotient
      // et les produits correspondant dans la queue
      Ajoute(qlen,q,A1/dpf);
      Q_Push(Q,d,-A1/dpf);
      // La partie principale du produit a deja ete comptée
      Q.prods[Q.nbf-1][Q.pf[Q.nbf-1]] += A1;
      if ( !fini ) {
        Ajoute(qlen,q,B1/dpf);
        Q_Push(Q,d,-B1/dpf);
        Q.prods[Q.nbf-1][Q.pf[Q.nbf-1]] += B1;
      }
    } else if ( D1 && (A1/dpf) ) {
      C=B1; D=C1; E=D1;
      Ajoute(qlen,q,A1/dpf);
      Q_Push(Q,d,-A1/dpf);
      Q.prods[Q.nbf-1][Q.pf[Q.nbf-1]] += A1;
    } else {
      C=A1; D=B1; E=C1;
    }
    // Si la queue est vide, on verifie que C,D,E sont vides
    // sinon, on les utilise
    if ( (!Q.nbpop) && (!fini) && (C/dpf) ) {
      Ajoute(qlen,q,C/dpf);
      Q_Push(Q,d,-C/dpf);
      Q.prods[Q.nbf-1][Q.pf[Q.nbf-1]] += C;
      C=D; D=E; E=0;
    }
  }
  RetourneEtZeros(qlen,q);
}
```