Contribution à l'arithmétique des ordinateurs

Y. Herreros

THESE

présentée par

Yvan Herreros

pour obtenir le titre de DOCTEUR
de l'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
(Arrêté ministériel du 23 novembre 1988)

(Spécialité : Mathématiques Appliquées)

CONTRIBUTION A L'ARITHMETIQUE DES ORDINATEURS

Date de soutenance : le 4 octobre 1991

Composition du jury :
J. Della-Dora (président)
M. Cosnard (rapporteur)
J. Vuillemin (rapporteur)
T. Lang
J.M. Muller (directeur de thèse)

Thèse préparée au sein du LABORATOIRE DE MODELISATION ET DE CALCUL.

Sommaire

Je tiens avant toutes choses à remercier ici les membres du jury :

Jean Della Dora, pour l'honneur et le plaisir qu'il me fait en acceptant de présider ce jury, ainsi que pour m'avoir accueilli au sein du LMC. Toutes les discussions que j'ai pu avoir avec lui m'ont beaucoup apporté.

Michel Cosnard, qui a bien voulu être rapporteur de ce travail et dont j'ai également pu apprécier la gentillesse à TIM3 d'abord, puis au LIP qu'il dirige avec bienveillance.

Jean Vuillemin, qui a rapporté cette thèse et dont les critiques ont grandement contribué à l'amélioration de ce document.

Tomas Lang, qui me fait l'honneur de faire partie du jury.

et enfin Jean-Michel Muller, mon directeur de thèse, pour m'avoir supporté tout au long de ce travail. Il a eu besoin de toute son expérience de chef d'orchestre pour me diriger en évitant les fausses notes pour moi et l'ulcère pour lui. Il a su m'apporter le goût de l'arithmétique des ordinateurs et de Mozart.

Je tiens aussi à remercier tous les membres du LMC pour les trois années que je viens de passer en leur compagnie avec une attention particulière pour :

Jean-Yves Blanc, d'abord pour m'avoir présenté Sylvie, puis pour les nombreux après-midi de travail en ville.

Hervé Frydlender, Gilles Villard, et Didier Wenzek pour les cigarettes de 8h, 8h30 et 9h plus toutes les autres.

Jean-Laurent Philippe pour son élégance et sa gentillesse.

Philippe Michallon et Nathalie Revol pour le Risk et le Sun3 couleur.

Je tiens enfin à remercier les membres du LIP et en particulier l'équipe SAAO :

Jean Duprat, Assurancetourix de l'équipe.

Sylvanius Kla, qui n'Ivoirien mais n'en pense pas moins.

Jean-Claude Bajard qui aura la faiblesse de rire à ce qui prècède

Hong-Jin Hie et Mario Aguilar, forçats du mode on-line.

Xavier Merrheim et Chistophe Mazenc, aNormaliens de choc.

Introduction

1. Qu'est ce que l'arithmétique des ordinateurs ?

L'arithmétique des ordinateurs est l'art "d'apprendre à compter aux ordinateurs". Cela recouvre deux problèmes primordiaux pour l'informatique, à savoir la représentation de grandeurs numériques et leur manipulation (opérations arithmétiques). En effet, le terme anglais *computer science* indique clairement l'utilisation principale de l'ordinateur et bien que le terme français "informatique" se veuille moins restrictif, "l'information" reste le plus souvent représentée par des données numériques.

Le premier problème est donc celui de la représentation des nombres, la façon de réaliser des opérations arithmétiques est en effet intimement liée au mode de représentation.

Le problème du calcul des opérations arithmétiques de base se découpe en deux cas distincts : le calcul sur des entiers de taille bornée ou de taille arbitraire. Le premier cas imposera en principe des solutions matérielles alors que le second demandera plutôt des solutions logicielles.

Le calcul des fonctions dites élémentaires, algébriques (racine carrée) ou transcendantes (trigonométriques ou hyperboliques) est aussi une branche de l'arithmétique des ordinateurs. On peut, comme pour les opérations arithmétiques, faire une distinction selon la taille des nombres à traiter ou plutôt selon la précision avec laquelle on désire le résultat. Si pour des nombre de taille (ou de précision) bornée, il existe des solutions matérielles élégantes (l'algorithme CORDIC par exemple [WA71]), pour calculer les fonctions élémentaires avec des réels de précision arbitraire, on est obligé de passer soit par des développements en série, soit par des méthodes itératives (itération de Newton ou itération Arithmético-Géométrique) qui se ramènent à des suites d'opérations arithmétiques de base.

2. Les problèmes

Les valeurs que l'on a à manipuler sont le plus souvent réelles ou complexes et on se ramène en fait à des calculs sur des entiers.

Les besoins principaux des utilisateurs de calcul portent sur deux points principaux et malheureusement souvent contradictoires, la vitesse du calcul et sa précision, le temps de calcul dépendant en effet directement de la longueur des nombres à traiter. Il faut cependant noter que les progrès dans l'intégration des circuits ont permis de concilier ces deux exigences dans une certaine limite : pour les tailles de nombres traditionnellement utilisées en calcul scientifique (32 ou 64 chiffres binaires), on commence maintenant à utiliser des opérateurs quasiment optimaux en temps dans la conception des unités arithmétiques (additionneurs de Brent-Kung ou multiplieurs de Wallace).

Quels que soient le volume et le type de calcul que l'on veut réaliser on se ramènera presque toujours aux quatre opérations arithmétiques de base : addition, soustraction, multiplication, division. Les valeurs que l'on a à manipuler sont le plus souvent réelles ou complexes et on se ramène toujours à des calculs sur des entiers. Ce sont donc les opérations arithmétiques de base sur les entiers qu'il faut réaliser le plus rapidement, possible et pour cela, on doit choisir des modes de représentation qui les facilitent.

Pour ce qui est du calcul sur des entiers de petite taille, les problèmes concernant l'arithmétique de base sont en grande partie résolus. Le codage pratiquement universellement employé pour les entiers est la numération de position en base 2. Ce choix est justifié aussi bien par des raison technologiques qu'algorithmiques (meilleure précision de la représentation, compacité du codage ...) et ne semble pas devoir être remis en question, du moins en ce qui concerne le calcul par voie matérielle.
On connait des relations liant la surface des multiplieurs et le temps mis pour réaliser la multiplication ([BK81],[VUI83]) et on connait des architectures atteignant l'optimum ([PV81]) en complexité temps-surface. On sait depuis longtemps réaliser des multiplieurs ([WA64], [DA65]) ou additionneurs ([BK82]) en temps logarithmique. On connait aussi des méthodes efficaces de calcul de quotients en temps linéaire ([HW81] et [PV90]). Comme on est arrivé à des temps de calculs quasiment optimaux, pour améliorer les performances, il faut découper les opérations de base en sous tâches afin de les enchaîner plus efficacement. On arrive ainsi au principe de *pipeline* utilisé sur les ordinateurs dits vectoriels et particulièrement bien adapté aux calculs d'algèbre linéaire. Pour ce type d'opérateurs on connait également les relations liant le temps et la surface [VUI83].
Les représentations sur un nombre fixe de chiffres sont de loin les plus employées pour les applications classiques de calcul scientifique, surtout les représentations flottantes qui sont relativement standardisées (formats flottants IEEE, DEC ou IBM par exemple). En ce qui concerne les représentations en virgule fixe, les seules différences tiennent au nombre de chiffres (32 ou moins en général) sur lesquels on code les valeurs, ce nombre dépendant de l'application.
Les améliorations envisageables maintenant pour les opérations arithmétiques en base 2 sur un nombre limité de bits semblent plutôt être d'ordre technologique.

Par contre le calcul sur des grands entiers reste délicat, on est obligé de recourir à des méthodes logicielles, et la base 2 ne s'impose plus avec autant d'évidence. pour repousser ces problèmes, on peut mixer des approches matérielles et logicielles [SBV90] ou bien, en adoptant de nouvelles représentations, travailler en série sur les chiffres..

La première méthode logicielle de calcul du produit de grands entiers en temps subquadratique ne date que de 1962 [KO62] et reste la seule réellement utilisée. Des méthodes pratiquement optimales en complexité [SS71] sont d'un intérêt pratique discutable.
On pense à effectuer les calculs en-ligne (en série poids forts en tête) sur les chiffres depuis 1977 ([IRW77], [ET77], ...). Ce mode de calcul est l'objet de développements récents. Il s'agit en fait du principe du *pipeline* poussé jusqu'au niveau du chiffre et qui permet ainsi d'enchaîner les calculs de façon efficace et de réaliser des opérateurs très compacts ou de traiter de plus grands nombres que par les méthodes matérielles classiques. En contrepartie, cela impose de représenter autrement les nombres (dans des systèmes dits à "chiffres signés"). Ce mode de calcul implique d'autre part une certaine latence entre l'entrée des opérandes et la sortie des résultats (ce retard est appelé le délai).

Enfin, les calculs sur les complexes ont toujours été réalisés en séparant les parties réelles et imaginaires des nombres. Cette représentation est la plus économique en occupation mémoire dans le cas d'une représentation non redondante. Cependant elle n'est pas *a priori* la mieux adaptée aux calculs sur les nombres complexes, et il serait intéressant de trouver un mode de représentation plus spécifique.

3. Description de la thèse

Le but de cette thèse est de donner quelques éléments de réponse aux problèmes présentés ci dessus.

Le chapitre 1 fait les rappels nécessaires sur les manières habituelles de représenter et manipuler les nombres réels. On redéfinit la notion de base et de chiffres. On rappelle les avantages et inconvénients des différentes représentations traditionnellement utilisées. On présente l'algorithme d'Avizienis qui permet, sous certaines conditions, de réaliser des additions sans propagation de retenue, puis on montre comment réaliser la même opération avec des hypothèses moins restrictives (redondance minimale).
Le chapitre 2 traite du problème du produit de grands entiers. Après une présentation des algorithmes classiques, on rappelle comment des techniques venant du calcul numérique (Transformées de Fourier Discrètes) peuvent être adaptées aux calculs exacts (transformations arithmétiques) et appliquées au problème du produit d'entiers. On présente brièvement l'algorithme de Schönage et Strassen puis plus en détail un algorithme moins connu mais plus performant en pratique, dû a Pollard (1971). Une implantation de cet algorithme est présentée, ainsi que des comparaisons avec des implantations d'autres algorithmes : bibliothèque BigNum de Serpette, Vuillemin et Hervé ([SVH89]) ou implantation théorique de l'algorithme de Karatsuba. On s'est enfin intéressé au problème de la parallélisation du calcul de la Transformée

de Fourier rapide sur une machine parallèle dont les processeurs sont connectés en hypercube. Les programmes sources en langage C d'une implantation de l'algorithme de Pollard sont donnés en annexe.

Le chapitre 3 aborde le mode de calcul *en-ligne*. On donne des résultats théoriques sur les performances de ce type d'opérateurs en fonction de la base de numération choisie, ce qui nous permet de conclure à l'optimalité de certains opérateurs. On présente enfin des cellules de base qui permettent de construire des opérateurs de calcul en-ligne en base 2.

Enfin, dans le chapitre 4, on s'intéresse à la représentation des complexes. On a voulu étendre l'idée de numération positionnelle à ces nombres. On donne d'abord des résultats sur les valeurs possibles des bases en fonction de l'ensemble de chiffres choisi. Ensuite on présente une représentation originale des complexes qui permet une implantation efficace tant logicielle que matérielle, avec entre autres la possibilité de réaliser des additions de façon parallèle sans propagation de retenues, ou des additions en série, poids forts en tête du même type que celles décrites au second chapitre. On en déduit un multiplieur en ligne, et la faisabilité de la division en ligne. Les additionneurs (parallèle ou en ligne) sont décrits en détails et un multiplieur en ligne pour les complexes est présenté.

I. Représentation des nombres

I.1 Introduction

Nous allons dans ce chapitre présenter les systèmes classiques de représentation des nombres. La première contrainte que l'on peut imposer à un système de numération est de permettre de représenter tous les nombres. Dans le cas des réels, on sait répondre à ce problème si la base est entière, dans le cas des complexes on a moins de résultats si on veut les représenter directement sans passer par les réels pour représenter séparément la partie réelle et la partie imaginaire.
La deuxième contrainte est de pouvoir effectuer des calculs sur les valeurs représentées. En pratique on a surtout besoin de réaliser facilement additions et multiplications.

I.2 Définitions.

On veut représenter tous les nombres complexes. Un *système de numération* sera la donnée d'un ensemble fini D d'éléments de C et d'une suite (e_i) $(i \in Z)$ d'éléments de C de modules croissants vérifiant: $\forall k \in Z$, $\sum_{i \le k} |e_i|$ est borné. D sera nommé l'ensemble des *chiffres* et (e_i) sera appelée *base discrète*. Dans le cas où e_i est de la forme B^i $(B \in C)$ B est appelé la *base* du système de numération. On représente par la suite (d_i) $(d_i \in D)$ le nombre égal à $\sum_{i \in Z} d_i.e_i$. Si $i<j$ (resp. $i>j$), on dit que d_j est un *chiffre* de *poids* supérieur (resp. inférieur) à d_i. Ces notations nous permettront de retrouver des écritures plus ou moins classiques des réels, et d'introduire les résultats du chapitre 4.

I.3 Cas particuliers

I.3.A Base entière positive, chiffres entiers

C'est la classe de systèmes de numération la plus intéressante pour représenter les réels et la seule utilisée en pratique.

I.3.A.a Numération simple de position

C'est le cas particulier où $D=\{0,\ldots,B-1\}$ et $e_i=B^i$ ($B \in \mathbb{N}^*$ $B \ge 2$). On retrouve l'écriture classique des réels positifs en base B. Ce système permet de représenter tous les réels positifs ou nuls. Pour représenter des valeurs négatives, deux méthodes sont classiquement utilisées :

- représentation *signe-valeur absolue* : c'est la plus simple à comprendre, il suffit de rajouter aux chiffres un symbole représentant le signe.

Le principal inconvénient de cette notation est que pour additionner deux valeurs, on doit réaliser une addition ou une soustraction selon les signe des opérandes.
Cette représentation des valeurs négatives est traditionnellement utilisée pour les mantisses des réels en virgule flottante.

• complément à la base (on suppose B pair) : on veut représenter tous les nombres de l'intervalle $[-\frac{B^n}{2}, \frac{B^n}{2}[$. Il suffit en fait de les représenter dans $[0,B^n]$ modulo B^n. Tout nombre de $[0, \frac{B^n}{2}[$ garde son écriture en numération simple de position, et tout nombre x de $[-\frac{B^n}{2}, 0[$ est représenté par $x+B^n$ écrit en numération simple de position.
Pour obtenir l'opposé d'un nombre, on complémente chacun de ses chiffres à B-1 et on ajoute 1 au nombre obtenu.
Exemple : en base 10 sur 4 chiffres, on peut représenter les nombres de -5000 à 4999.
Les nombres compris entre 0 et 4999 sont écrits en numération simple de position. Pour représenter un nombre compris entre -5000 et -1, on l'ajoute à 10000. -1267 est ainsi représenté par 8733.

L'avantage de cette représentation est que l'on traite les valeurs négatives comme les valeurs positives et que les additions et soustractions se font de la même manière qu'en numération simple de position. En contrepartie, les multiplications sont un peu plus délicates à effectuer, et les dépassements de capacité sont plus difficiles à détecter.
En base 2, le complément à 2 est utilisé sur presque tous les ordinateurs pour représenter les entiers signés et les réels en virgule fixe.

I.3.A.b Systèmes d'Avizienis

Avizienis a introduit en 1961 [AV61] toute une classe de systèmes de numération qui englobe la numération simple de position. Certains de ces systèmes sont dits *redondants* en ce sens que des nombres peuvent avoir *plusieurs représentations finies* . La numération simple de position n'est pas un système redondant : on a par exemple 1 = 0,999999... en base 10 mais la deuxième écriture n'est pas finie. Cette redondance peut *a priori* sembler néfaste, car elle entraîne plus de représentations possibles (et donc plus d'occupation mémoire) pour les mêmes nombres représentables et des comparaisons plus délicates à effectuer. En contrepartie, elle permet, ainsi que l'on va le voir, de réaliser sous certaines conditions des additions sans propagation de retenue.
Un système d'Avizienis est défini par la donnée de trois valeurs :
la base B (entier supérieur ou égal à 2), un entier négatif ou nul ω et un entier strictement positif q vérifiant $\omega+q>0$ qui définissent l'ensemble des chiffres $D = \{\omega, \ldots, \omega+q\}$.

On a alors les résultats suivants :

• Si $q < B-1$, on ne pourra pas représenter tous les réels

- Si $q \geq B-1$, on pourra représenter tous les réels si $\omega<0$, et tous les réels positifs si $\omega=0$. si de plus $q \geq B$ le système sera redondant.

Pour démontrer ces résultats, il suffit de faire les démonstrations dans le cas entier avec les puissances positives ou nulles de la base.

Appelons $G(B,\omega,n,q)$ l'ensemble des nombres que l'on peut écrire en base B avec les chiffres dans $\{\omega,\ldots,\omega+q\}$et les puissances positives de B strictement inférieures à n. On a donc :

$$G(B,\omega,n,q) = \left\{ \sum_{i=0}^{n-1} d_i B^i,\ d_i \in \{\omega,\ldots,\omega+q\} \right\}$$

On peut toujours se ramener au cas où $\omega=0$ via une simple translation. En effet,

$$G(B,\omega,n,q) = \left\{ \omega \frac{B^n-1}{B-1} + \sum_{i=0}^{n-1} d_i B^i,\ d_i \in \{0,\ldots,q\} \right\} = G(B,0,n,q)+\omega \frac{B^n-1}{B-1}.$$

Il nous reste à étudier $G(B,0,n,q)$ selon les valeurs de q. Le plus grand élément de $G(B,0,n,q)$ est $q\frac{B^n-1}{B-1}$ et son plus petit élément est 0, donc si tous les éléments de $G(B,0,n,q)$ sont consécutifs, il doit contenir exactement $q\frac{B^n-1}{B-1}+1$ éléments, or on ne peut écrire que $(q+1)^n$ nombres sous la forme $\sum_{i=0}^{n-1} d_i B^i,\ d_i \in \{0,\ldots,q\}$. Donc une condition nécessaire pour que les éléments de $G(B,0,n,q)$ soient consécutifs est que $(q+1)^n$ soit supérieur ou égal à $q\frac{B^n-1}{B-1}+1$, soit $\frac{(q+1)^n-1}{q} \geq \frac{B^n-1}{B-1}$, ce qui implique $q \geq B-1$.

Donc si $q<B-1$ on ne pourra pas représenter d'intervalles en base B avec les chiffres dans $\{\omega,\ldots,\omega+q\}$.

Si $q \geq B-1$ on montre par récurrence sur n que $G(B,0,n,q)$ est égal à $\left\{0, \ldots, q\frac{B^n-1}{B-1}\right\}$. Il suffit de remarquer que $G(B,0,n+1,q) = \{ d_n B^n+x,\ 0 \leq d_n \leq q \text{ et } x \in G(B,0,n,q) \}$ et que si $q \geq B-1$, alors $q\frac{B^n-1}{B-1} \geq B^n-1$.

Donc si $q \geq B-1$, $G(B,\omega,n,q) = \left\{ \omega \frac{B^n - 1}{B-1}, \ldots, (\omega+q) \frac{B^n - 1}{B-1} \right\}$ et on peut donc représenter tous les réels si $\omega<0<\omega+q$ (tous les réels positifs si $\omega=0$).

Additions en temps constant dans les systèmes d'Avizienis.

Avizienis a démontré que dans le cas où $D=\{-a, \ldots, a\}$ avec $a \leq B-1$ et $2a-1 \geq B$ il est possible de réaliser des additions sans propagation de retenue (on peut remarquer que la condition $2a-1 \geq B$ correspond à $q \geq B+1$). Ecrivons $X = \sum x_i.B^i$ et $Y = \sum y_i.B^i$ où les x_i et les y_i sont des éléments de $\{-a, \ldots, a\}$. Pour tout i, $x_i+y_i \in \{-2a, \ldots, 2a\}$.
La condition $a \leq B-1$ implique que $2a<B+(a-1)$ et que donc l'ensemble $\{-2a, \ldots, 2a\}$ est inclus dans $\{-B-(a-1), \ldots, B+(a-1)\}$. La condition $2a-1 \geq B$ implique, elle, que $a-1 \geq B-(a-1)-1$, et que tout entier de $\{-2a,\ldots,2a\}$ peut s'écrire sous la forme $c.B+s$ avec c pris dans $\{-1, 0, 1\}$ et s pris dans $\{-a+1, \ldots, a-1\}$.

Donc pour additionner $X = \sum x_i.B^i$ et $Y = \sum y_i.B^i$ on écrit, pour tout i, $x_i+y_i = c_{i+1}.B+s_i$ où $c_{i+1} \in \{-1, 0, 1\}$ et $s_i \in \{-a+1, \ldots, a-1\}$. On pose alors $t_i = c_i+s_i$ et donc $X+Y = \sum t_i.B^i$ avec t_i dans $\{-a, \ldots, a\}$.
Les calculs des c_{i+i} et des s_i se font en parallèle. Ensuite on calcule en parallèle tous les t_i, et donc le calcul de la somme peut être fait en temps constant.

On peut donner le théorème suivant, plus général :

Théorème : si les nombres sont écrits en base $B \geq 2$ avec les chiffres dans $D=\{\omega, \ldots, \omega+q\}$ où ω et q vérifient $-B+1 \leq \omega \leq 0 < \omega+q \leq B-1$, alors si $q \geq B$, on peut calculer la somme de deux nombres en temps constant.
Algorithme :
Soient 2 entiers X et Y, $X = \sum_{k=0}^{n-1} x_k B^k$ et $Y = \sum_{k=0}^{n-1} y_k B^k$.
Les x_i et les y_i sont dans $\{\omega, \ldots, \omega+q\}$, donc $x_i+y_i \in \{2.\omega, \ldots, 2.\omega+2.q\}$ pour tout i.
On écrit x_i+y_i sous la forme $x_i+y_i = B.c_{i+1}+s_i$ avec c_{i+1} dans $\{-1, 0, 1\}$ tel que pour tout i, $c_i+s_i \in \{\omega, \ldots, \omega+q\}$. On a les différents cas suivants :

- Si $x_i+y_i = \in \{2.\omega, \ldots, \omega-1\}$, on choisit $c_{i+1}= -1$ et $s_i = x_i+y_i+B$.
  On a alors $s_i \in \{\omega+1, \ldots, \omega+q-1\}$ car $2.\omega+B>\omega$ et $B \leq q$.
- Si $x_i+y_i \in \{\omega+1, \ldots, \omega+B-1\}$, on choisit $c_{i+1}=0$ et $s_i = x_i+y_i$.
- Si $x_i+y_i \in \{\omega+B+1, \ldots, 2.\omega+2.q\}$, on choisit $c_{i+1}= 1$ et $s_i = x_i+y_i-B$.
  On a alors $s_i \in \{\omega+1, \ldots, \omega+q-1\}$ car $\omega+q \leq B-1$

Dans ces trois cas $s_i \in \{\omega+1, \ldots, \omega+q-1\}$, et $c_i+s_i \in \{\omega, \ldots, \omega+q\}$ si c_i est dans $\{-1, 0, 1\}$.

• Si $x_i+y_i = \omega$, on a deux choix possibles $c_{i+1}= -1$ et $s_i = B+\omega$ ou $c_{i+1}= 0$ et $s_i = \omega$.
Si $x_{i-1}+y_{i-1} \geq 0$, $c_i = 0$ ou 1, donc on choisit $c_{i+1}= 1$ et $s_i = \omega$.
Si $x_{i-1}+y_{i-1} < 0$, $c_i = -1$ ou 0, donc on choisit $c_{i+1}= -1$ et $s_i = B+\omega$.

• Si $x_i+y_i = \omega+B$, on a deux choix possibles $c_{i+1}= 0$ et $s_i = B+\omega$ ou $c_{i+1}= 1$ et $s_i = \omega$.
Si $x_{i-1}+y_{i-1} \geq 0$, $c_i = 0$ ou 1, donc on choisit $c_{i+1}= 1$ et $s_i = \omega$.
Si $x_{i-1}+y_{i-1} < 0$, $c_i = -1$ ou 0, donc on choisit $c_{i+1}= 0$ et $s_i = B+\omega$.

Tous ces choix garantissent $c_i+s_i \in \{\omega, \ldots, \omega+q\}$ pour tout i, il suffit alors de poser $t_i=c_i+s_i$ pour $1 \leq i \leq n-1$, avec les conventions $c_0=0$, et $s_n=0$ (ce qui entraîne $t_0=s_0$ et $t_n=c_n$).
On a alors $X+Y = T = \sum_{k=0}^{n} t_k.B^k$. Les calculs des couples (c_{i+1}, s_i) puis des t_i se font en parallèle en temps constant, et donc l'addition peut se faire en temps constant en base B avec les chiffres dans $\{\omega, \ldots, \omega+q\}$.
Remarque : La condition d'application de ce théorème est moins restrictive que celle de l'algorithme d'Avizienis, la redondance étant "minimale", c'est pourquoi on doit examiner les chiffres sur deux positions et non plus sur une seule.

I.3.A.c Chiffres binaires signés.

La notation en chiffres binaires signés est un cas particulier des notations d'Avizienis avec B=2 et D={-1, 0 1} (ω=-1 et q=2), et d'après le théorème précédent, on sait effectuer des additions en temps constant dans ce système.
L'algorithme d'addition en temps constant dans ce système est connu depuis 1978 ([CR78]), nous allons le détailler :

On veut additionner $X= \sum x_i.2^i$ et $Y=\sum y_i.2^i$ où les x_i et les y_i sont dans {-1,0,1}. On pose $x_i+y_i = 2.c_{i+1}+s_i$ où les c_{i+1} et les s_i sont dans {-1,0,1}, puis $t_i=c_i+ s_i$, ce qui nous donne $X+Y = \sum t_i.2^i$.
Voici comment on choisit s_i et c_i en fonction de x_i, y_i, x_{i-1} et y_{i-1} :

• Si $x_i+y_i \in \{-2,0,2\}$, le seul choix possible pour s_i est 0, et donc quelle que soit la valeur de la somme $x_{i-1}+y_{i-1}$, $t_i=s_i+c_i$ est dans {-1,0,1}.

• Si $x_i+y_i \in \{-1,1\}$, posons $\delta=x_i+y_i$.
On a deux choix possibles du couple (c_{i+1},s_i) : $(0,\delta)$ ou $(\delta,-\delta)$
Si $x_{i-1}+y_{i-1}$ est du signe de x_i+y_i, c_i ne peut valoir que 0 ou δ. on choisit donc $(c_{i+1},s_i) = (\delta,-\delta)$ et donc $t_i=s_i+c_i$ est dans {-1,0,1}

Si $x_{i-1}+y_{i-1}$ est du signe inverse de x_i+y_i, c_i ne peut valoir que 0 ou $-\delta$. on choisit donc $(c_{i+1},s_i) = (0,\delta)$ et donc $t_i=s_i+c_i$ est dans $\{-1,0,1\}$

On a donc toujours la possibilité de choisir c_{i+1} et s_i en fonction de x_i, y_i, x_{i-1} et y_{i-1} de façon à ce que $t_i = c_i+s_i$ soit dans $\{-1,0,1\}$. Les couples (c_{i+1},s_i) se calculent tous en parallèle puis les t_i se calculent en parallèle. $X+Y = \sum t_i.2^i$ est donc calculé en en temps indépendant de la longueur de X et Y.

I.3.B Base réelle positive, chiffres entiers

Il ne semble pas avoir en général d'intérêt pratique à écrire les nombres avec une base réelle non entière. On montre (voir [MU85]) que si $D = \{0,\ldots, a\}$ alors on a deux cas possibles :

- si $a < B-1$ l'ensemble des nombres représentables est de dimension de Hausdorf $\frac{\log(a+1)}{\log(B)} < 1$, on ne peut donc représenter aucun intervalle.
- si $a \geq B-1$ on peut représenter tout $\mathbb{R}^+$.

I.3.C Base discrète à valeurs réelles positives, chiffres entiers

Le cas des bases discrètes à valeurs réelles positives avec des chiffres entiers a été étudié en détail (voir [MU85]) dans le cas où $e_i = 0$ si $i > 0$. Rappelons les principaux résultats et définitions.

Définition : Une base discrète à valeurs réelles positives est dite base discrète additive d'ordre a si tout nombre de l'intervalle $\left[0,\ a\sum_{i\leq 0} e_i\right]$ peut s'écrire sous la forme $\sum_{i\leq 0} d_i.e_i$ en prenant les d_i dans $\{0,\ldots,a\}$.

Théorème : Une base discrète à valeurs réelles positives est une base discrète additive d'ordre a si et seulement si elle vérifie pour tout i négatif ou nul $e_i \leq a\sum_{j<i} e_j$.

Les bases discrètes additives servent au calcul de certaines fonctions élémentaires (sinus, cosinus, exponentielle,...) par des algorithmes de type CORDIC (voir [VO59] ou [WA71]). Montrons par exemple comment on peut calculer une exponentielle:

on pose $e_i = \ln(1+2^i)$ pour tout i négatif, et $D = \{0,1\}$. C'est une base discrète additive d'ordre 1, donc tout nombre de $\left[0,\ \sum_{i\leq 0} e_i\right]$ peut s'écrire sous la forme $\sum_{i\leq 0} d_i.e_i$ avec d_i dans $\{0,1\}$ et son exponentielle s'écrit donc $\prod_{i\leq 0}(1+2^i)^{d_i}$. Pour plus de détails sur les bases discrètes et leurs applications, on pourra se reporter à [MU85] ou [MU89].

Les représentations sur des bases discrètes ne sont pas utilisées directement. On ne sait pas en général additionner ou multiplier deux nombres écrits dans une base discrète sans repasser par une représentation plus simple. On n'utilise l'écriture d'un nombre sur une base discrète que pour calculer une fonction. Le nombre lui même est conservé dans une représentation plus classique.

I.3.D Base complexe, chiffres complexes

Nous reviendrons plus en détail sur le cas de la représentation des complexes dans le dernier chapitre. Voyons brièvement différents cas possibles :

• Base complexe, chiffres entiers

Diverses représentations de ce type ont été proposées dans le cas où la base est une racine $8^{ème}$ de 16 (de module $\sqrt{2}$). Par exemple l'écriture en base i-1 ou en base $i\sqrt{2}$.

• Base réelle, chiffres complexes

Dans le cas où les chiffres sont dans des ensembles particuliers (l'ensemble des racine $n^{èmes}$ des l'unité plus zéro), on connait des bornes sur la valeur de la base, pour pouvoir représenter tous les complexes.

• Base entière, chiffres complexes

On traite au dernier chapitre le cas de la base 2 avec comme chiffres les racines $6^{èmes}$ de l'unité plus zéro. On peut représenter tous les complexes de façon simple, et de plus on sait réaliser des additions parallèles en temps constant (ou des additions en série poids forts en tête) en utilisant un algorithme dérivé de celui d'Avizienis.

II. Multiplication rapide de grands entiers

II.1 Introduction

Un des principaux problèmes du calcul formel ou par exemple de la cryptographie est de réaliser efficacement le produit de "grands" entiers : de 512 bits pour la cryptographie à plusieurs milliers de chiffres décimaux en calcul formel. L'algorithme élémentaire (celui que nous utilisons pour effectuer les multiplications à la main) demande un temps de calcul proportionnel à n^2 où n est le nombre de chiffres des opérandes.

Depuis 1962, on sait faire mieux : l'algorithme de Karatsuba (que nous allons décrire) a une complexité en $O(n^{1,585...})$ et est effectivement utilisable pour des entiers de taille "moyenne" à "grande".

Pour trouver des algorithmes plus performants, nous allons devoir passer par d'autres outils. Il est connu (et nous le rappellerons) que le produit de grands entiers peut se ramener à un calcul de convolutions d'entiers. On sait également comment un produit de convolution peut être calculé via la Transformée de Fourier. On utilisera l'algorithme de Transformée de Fourier Rapide (FFT) de Cooley et Tuckey qui permet de réaliser une transformée de Fourier de taille n en un temps de l'ordre de n.log(n).

Tous ces calculs peuvent être effectués dans un corps fini, ce qui évitera de travailler dans le champ complexe (avec tous les problèmes de contrôle d'erreur d'arrondi que cela entraîne). Schönague et Strassen ont proposé en 1971 un algorithme de multiplication qui utilise des transformées de Fourier en nombre entiers. Cet algorithme que nous allons brièvement décrire plus loin permet théoriquement de réaliser des produits de nombres arbitrairement grands en temps O(n.log(n).log(log(n))). Nous allons cependant lui préférer un algorithme décrit par Pollard en 1971 ([PO71]) qui est aussi basé sur des transformées de Fourier en nombres entiers, mais qui est plus efficace et simple à implanter : cet algorithme dépasse une hypothétique implantation "parfaite" de l'algorithme de Karatsuba pour des entiers de 40000 chiffres décimaux, les temps de cette implantation "parfaite" étant obtenus en majorant grossièrement les temps de calcul et en ne tenant pas compte du coût de contrôle.

Nous présentons ensuite des méthodes de parallélisation du calcul de la FFT afin d'accélérer encore l'algorithme de Pollard, ainsi qu'une proposition d'architecture dédiée.

II.2 Algorithmes classiques.

II.2.A Algorithme naïf.

Considérons deux entiers $X=\sum_{k=0}^{n-1} x_k.B^k$ et $Y=\sum_{k=0}^{n-1} y_k.B^k$ écrits sur n chiffres en base B ($0\leq x_k,y_k<B$).

On veut obtenir le produit $Z= XY = \sum_{k=0}^{2n-2} B^k \left(\sum_{h+m=k} x_h.y_m \right)$. En posant $z_k= \sum_{h+m=k} x_h.y_m$, il est clair que Z est égal à $\sum_{k=0}^{2n-2} z_k.B^k$.

Il reste à effectuer une propagation de retenue (qui s'effectue en un temps proportionnel à n) pour obtenir Z sous la forme $Z = \sum_{k=0}^{2n-1} z'_k.B^k$ où les z'_k sont inférieurs à B.

II.2.B Algorithme de Karatsuba

Le premier algorithme de multiplication de grands entiers en temps subquadratique semble dû à Karatsuba [KO62], il utilise l'identité suivante :

$$(xB+x')(yB+y') = (B^2+B)xy - (x-x')(y-y')B +(B+1)x'y'.$$

Soient deux entiers $X = \sum_{k=0}^{2n-1} x_i.B^i$ et $Y = \sum_{k=0}^{2n-1} y_i.B^i$,

si on pose $x' = \sum_{k=0}^{n-1} x_i.B^i$, $x = \sum_{k=n}^{2n-1} x_i.B^i$, $y' = \sum_{k=0}^{n-1} y_i.B^i$ et $y = \sum_{k=n}^{2n-1} y_i.B^i$ on a : $X=x.B^n+x'$ et $Y=y.B^n+y'$ et donc en utilisant l'identité précédente on obtient :

$$XY = (B^{2n}+B^n)xy - (x-x')(y-y')B^n +(B^n+1)x'y'.$$

On a donc ramené un produit de deux nombres de 2n chiffres à trois produits de nombres de n chiffres plus des additions (qui se font en temps linéaire). On décompose ainsi récursivement les multiplications à effectuer jusqu'à ce que les opérandes aient moins de n_0 chiffres de long, auquel cas, on applique l'algorithme de son choix que l'on appellera par la suite *algorithme basique*. On obtient ainsi un algorithme récursif pour lequel on note $T_K(p)$ le temps nécessaire à la multiplication de deux nombres de p chiffres. On note T(p) le temps de calcul du produit de 2 nombres de p chiffres par l'algorithme basique. $T_K(p)$ vérifie les relations suivantes :

$$T_K(2n) = 3.T_K(n)+C.n \quad \text{si} \quad 2n>n_0$$

$$T_K(p) = T(p) \text{ si } p \leq n_0.$$

C est constant pour une implantation donnée et correspond au coût des additions et autres opérations annexes.

Les relations précédentes nous donnent $T_K(2n)+C.2n = 3(T_K(n)+Cn)$ si $2n > n_0$, ce qui nous donne $T_K(2^k.n_0)=(T(n_0)+C.n_0)3^k-C.2^k.n_0$, soit posant $\omega = \log_2 3 = \frac{\log 3}{\log 2}$:

$$T_K(n)= \frac{T(n_0)+C.n_0}{n_0^{\omega}} n^{\omega} - C.n \text{ si } n = 2^k n_0.$$

Pour $n>n_0$ quelconque, on a :

$T_K(n) \leq T(n_0.2^{\lceil \log_2(n/n_0) \rceil}) = (T(n_0)+C.n_0).3^{\lceil \log_2(n/n_0) \rceil}-C.n_0.2^{\lceil \log_2(n/n_0) \rceil})$,

et donc $T_K(n) \leq 3.(T(n_0)+C.n_0).3^{\log_2(n/n_0)} = 3. \frac{T(n_0)+C.n_0}{n_0^{\omega}} n^{\omega}$.

On a donc un temps de calcul en $O(n^{\omega}) = O(n^{\log_2(3)}) = O(n^{1,58\ldots})$.

L'algorithme de Karatsuba est intéressant aussi bien en pratique qu'en théorie. En effet, il dépasse rapidement l'algorithme naïf et est donc applicable pratiquement pour des tailles "raisonnables" d'entiers.

On peut généraliser ce résultat : Knuth montre dans [KN81] que si on connait un algorithme pour multiplier deux polynômes de degré d en effectuant m multiplications scalaires, on peut en déduire un algorithme pour multiplier deux nombres de (d+1).n chiffres en faisant m multiplications de nombres de n chiffres. Ceci donne un algorithme de multiplication de grands entiers de complexité $O(n^{\alpha})$ où $\alpha = \frac{\log(m)}{\log(d+1)}$. En pratique, quand on augmente d, la constante multiplicative appliquée au terme de plus fort degré augmente trop vite pour donner des algorithmes utilisables.

II.2.B.a Implantation "idéale" de l'algorithme de Karatsuba

On a vu que le temps de l'algorithme est donné par $T_K(n)= \frac{T(n_0)+C.n_0}{n_0^{\omega}} n^{\omega} - C.n$. Pour C donné, n_0 est choisi de façon à minimiser $F(n)= \frac{T(n)+C.n}{n^{\omega}}$ et ne dépend donc que de C et de T(n), le temps mis par l'algorithme basique pour réaliser le produit de deux entiers de n chiffres. La meilleure implantation de l'algorithme de Karatsuba est obtenue avec la plus petite valeur de C possible et le meilleur algorithme basique possible.

P. Zimmerman à proposé [ZIM89] une formule permettant d'estimer la constante C qui intervient dans le calcul de la complexité de l'algorithme, il obtient :

$$C= 7.t_+ + 4.t_- + 4t_{RAZ} + 2.t_{copie},$$

avec les notations suivantes :

$N.t_+$ est le temps nécessaire à l'addition de deux nombres de N chiffres.

N.t_- est le temps nécessaire à la soustraction de deux nombres de N chiffres.

N.t_{RAZ} est le temps nécessaire à la mise à zéro d'un nombre de N chiffres.

N.t_{copie} est le temps nécessaire à la copie d'un nombre de N chiffres.

En pratique on a $t_- = t_+$.

Ces valeurs peuvent être très facilement estimées en réalisant des boucles de test.

On peut obtenir une minoration plus sévère de C à l'aide de la formule :

$$(x.B^n+x')(y.B^n+y') = B^{2n}xy - ((x-x')(y-y')-xy-x'y')B^n + x'y'.$$

Pour réaliser le produit de deux nombre de taille 2.n, il faut réaliser *au minimum* :

2 soustractions de nombres de taille n : x-x' et y-y'.

2 soustractions de nombres de taille 2.n : (x-x')(y-y')-xy-x'y'.

1 soustraction de nombres de taille au moins 2n : $(B^{2n}xy+x'y')$ - ((x-x')(y-y')-xy-x'y')B^n.

Ce qui veut dire qu'il est *impossible* d'avoir une implantation de l'algorithme de Karatsuba avec une valeur de C inférieure à 8.t_- ou 8 t_+.

On peut de la même façon obtenir une minoration de T(n) en supposant que l'on choisit l'algorithme naïf comme algorithme basique et on obtient $T(n) \geq t_x n^2$ où t_x est le temps du produit de deux chiffres.

Pour obtenir une minoration de temps total de l'algorithme de Karatsuba, on prend pour n_0 la valeur qui minimise $\frac{t_x.n_0^2+C.n_0}{n_0^\omega}$ soit $n_0 = \frac{C(\omega-1)}{t_x.(2-\omega)}$ et on prend 8.t_+ comme valeur de C.

On obtient alors $T_K(n) \geq \frac{t_x.n_0^2+8.t_+.n_0}{n_0^\omega}\, n^\omega - 8.t_+.n$ avec.$n_0 = \frac{8.t_+.(\omega-1)}{t_x.(2-\omega)}$.

Donc si on peut minorer t_+et t_x sur une machine donnée, on peut par la formule précédente minorer le temps de calcul de *toute les implantations de l'algorithme de Karatsuba* sur cette machine. Cette minoration nous donne une implantation *"idéale"* de l'algorithme qui va nous servir à faire les comparaisons avec les algorithmes présentés plus loin.

II.3 Multiplication et transformation de Fourier discrète

II.3.A Multiplication et convolution.

Soient deux suites finies (x_i) et (y_i) (i = 0, ..., n-1)

on appelle *convolution cyclique positive* la suite finie (z_i) (i = 0, ..., n-1) définie par

$$z_i = \sum_{k=0}^{i} x_k.y_{i-k} + \sum_{k=i+1}^{n-1} x_k.y_{n+i-k} = \sum_{h+m\equiv i\ (n)} x_h.y_m$$

Voyons quel est le lien entre le produit de grands entiers et la convolution de suite finies d'entiers. Les deux entiers $X = \sum_{k=0}^{n-1} x_i.B^i$ et $Y = \sum_{k=0}^{n-1} y_i.B^i$ dont on cherche le produit sont écrits sur n chiffres en base B ($0 \le x_k, y_k < B$). On pose $X'=(x_i)$ et $Y'=(y_i)$ (i = 0, ..., 2n-2) où $x_i = y_i = 0$ si $n \le i \le 2n-2$.

Le produit Z= XY s'écrit sous la forme $Z= \sum_{k=0}^{2n-2} B^k \left(\sum_{h+m=k} x_h.y_m \right)$. En posant $z_k= \sum_{h+m=k} x_h.y_m$, on obtient $Z = \sum_{k=0}^{2n-2} z_k.B^k$, et comme $x_i=y_i=0$ si $n \le i \le 2n-2$, on remarque que z_i est égal à $\sum_{h+m \equiv i\ (n)} x_h.y_m$. Donc $Z'=(z_0, \ldots, z_{2n-2})$ est la convolution cyclique positive de X' et Y'.

II.3.B Convolution et Transformée de Fourier Discrète

Montrons qu'un produit de convolution peut se ramener à un calcul de Transformée de Fourier Discrète. Soit une suite (x_i) (d'entiers, de réels ou de complexes). Dans toute la suite de l'exposé, on notera ω_n une racine $n^{ième}$ primitive de l'unité (c'est à dire telle que $(\omega_n)^n = 1$, et $(\omega_n)^k \neq 1$ pour $0< k < n$). On appelle *Transformée de Fourier Discrète* ou DFT (de l'anglais *Discrete Fourier Transform*) de (x_i) la suite $(\hat{x}_i)$ définie par $\hat{x}_i = \sum_{k=0}^{n-1} x_k.\omega_n^{ik}$.

On peut monter facilement la relation : $x_k = \frac{1}{n} \sum_{i=0}^{n-1} \hat{x}_i.\omega_n^{-ik}$. La *Transformée de Fourier Inverse* de la suite (x_i) est donc définie par $\check{x}_i = \frac{1}{n} \sum_{k=0}^{n-1} x_k.\omega_n^{-ik}$.

Une propriété fondamentale des Transformées de Fourier est que si z_i vaut $\hat{x}_i.\hat{y}_i$ alors $\check{z}_i$ vaut $\sum_{h+m \equiv i\ (n)} x_h.y_m$.

En effet :

$$\check{z}_i = \frac{1}{n} \sum_{k=0}^{n-1} z_k.\omega_n^{-ik} = \frac{1}{n} \sum_{k=0}^{n-1} \hat{x}_k.\hat{y}_k.\omega_n^{-ik} = \frac{1}{n} \sum_{k=0}^{n-1} \left(\sum_{h=0}^{n-1} x_h\, \omega_n^{hk} \right) \left(\sum_{m=0}^{n-1} y_m\, \omega_n^{mk} \right).\omega_n^{-ik}$$

$$= \frac{1}{n} \sum_{h=0}^{n-1} \sum_{m=0}^{n-1} x_h\, y_m \left(\sum_{k=0}^{n-1} \omega_n^{k(h+m-i)} \right) = \sum_{h+m \equiv i\ (n)} x_h.y_m$$

On peut donc calculer une convolution cyclique à l'aide de la Transformée de Fourier discrète et de son inverse.

II.3.C Les algorithmes de Transformées de Fourier rapide

La DFT d'une suite de n termes peut être calculée directement à l'aide de la définition précédente en un temps proportionnel à n^2 (il s'agit d'un produit matrice*vecteur). Cela n'apporte rien pour un calcul de convolution qui se fait de façon triviale en $O(n^2)$. Cependant, en 1965, Cooley et Tuckey proposèrent un algorithme efficace, appelé Transformée de Fourier Rapide ou FFT (de l'anglais *Fast Fourier Transform*) pour calculer la Transformée de Fourier Discrète.

Voici les deux principales versions de cet algorithme pour calculer la Transformée de Fourier Discrète quand le nombre n de termes de la suite est une puissance de 2. Ces deux méthodes ont pour noms *décimation en temps* et *décimation en fréquence*. L'origine de ces termes vient de la transformée de Fourier continue qui fait passer d'un signal fonction du temps à sa répartition en fonction des fréquences. On fera une décomposition récursive soit dans l'espace de départ (le temps) soit dans l'espace d'arrivée (les fréquences).

II.3.C.a La décimation en temps

Supposons que n est pair, si i est inférieur à $\frac{n}{2}$, alors

$$\hat{x}_i = \sum_{k=0}^{n-1} x_k.\omega_n^{ki} = \sum_{k=0}^{\frac{n}{2}-1} x_{2k}.\omega_n^{2ki} + \omega_n^i \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1}.\omega_n^{2ki} = \sum_{k=0}^{\frac{n}{2}-1} x_{2k}.\omega_{\frac{n}{2}}^{ki} + \omega_n^i \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1}.\omega_{\frac{n}{2}}^{ki}$$

tandis que

$$\hat{x}_{i+\frac{n}{2}} = \sum_{k=0}^{n-1} x_k.\omega_n^{k(i+\frac{n}{2})} = \sum_{k=0}^{\frac{n}{2}-1} x_{2k}.\omega_n^{2ki} - \omega_n^i \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1}.\omega_n^{2ki} = \sum_{k=0}^{\frac{n}{2}-1} x_{2k}.\omega_{\frac{n}{2}}^{ki} - \omega_n^i \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1}.\omega_{\frac{n}{2}}^{ki}$$

On prend un terme sur deux (d'où le terme de *décimation*) dans l'espace de départ (le temps)
Les calculs de $\hat{x}_i$ et de $\hat{x}_{i+\frac{n}{2}}$ demandent donc l'évaluation des mêmes termes,

donc pour $0 \le i < \frac{n}{2}$:

$$\hat{x}_i = \sum_{k=0}^{\frac{n}{2}-1} x_{2k}.\omega_{\frac{n}{2}}^{ki} + \omega_n^i \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1}.\omega_{\frac{n}{2}}^{ki}$$

$$\hat{x}_{i+\frac{n}{2}} = \sum_{k=0}^{\frac{n}{2}-1} x_{2k}.\omega_{\frac{n}{2}}^{ki} - \omega_n^i \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1}.\omega_{\frac{n}{2}}^{ki}$$

Le calcul d'une DFT de taille n est donc ramené à celui de deux DFT de taille $\frac{n}{2}$. Si l'on note T(n) le temps nécessaire au calcul d'une DFT de taille n, on obtient $T(n) = 2T\left(\frac{n}{2}\right) + C\,n$, où C est une constante. Si n est une puissance de 2 on peut appliquer récursivement cette décomposition et on obtient $T(2^k) = C\,k\,2^k$ (car T(1)=0).
On peut donc écrire (en théorie) une procédure récursive de calcul de FFT, cependant une implantation non récursive est plus facile à mettre en oeuvre et peut s'écrire comme suit en pseudo-code :

```
procedure FFT (Tab,n)    /* décimation en temps */
    d : = 1;
    h : = 2;
    nboucle : = n/2;
    tantque d < n faire
        w : = 1
        pour j : =0 a d-1 faire
            pour i : = 0 à nboucle-1 faire
                aux : = Tab[i*h+j] + w*Tab[i*h+j+d];
                Tab[i*h+j] : = Tab[i*h+j] - w*Tab[i*h+j+d];
                Tab[i*h+j+d] : = aux;
            fait
            w : = w* (ω_n)^(n/h)      /* (ω_n)^(n/h) = ω_h, w = (ω_h)^j */
        fait
        nboucle : = nboucle/2;
        h : = h*2;
        d : = d*2;
    fait
finprocedure
```

Les valeurs ω_h sont en fait stockées dans un tableau. La variable "h" est toujours égale à une puissance de 2 et donc ce tableau est de petite taille (trente éléments pour une FFT d'un milliard de termes). Ce tableau peut être initialisé en début de programme, ou bien on peut entrer les valeurs des ω_h comme constantes.
On remarque que cet algorithme impose un bouleversement de l'ordre des données en entrée. Il faut séparer les coefficients d'indices pairs et impairs récursivement chaque fois que l'on scinde le calcul d'une DFT en deux calculs de DFT de taille moitié. Cette permutation est connue sous le nom de *transformation miroir* car elle revient à inverser les bits de l'écriture binaire des indices des coefficients ainsi que l'on peut le vérifier sur la figure 1 qui représente le graphe de circulation des données lors de l'exécution de l'algorithme. La figure 2 représente l'effet d'un "papillon"

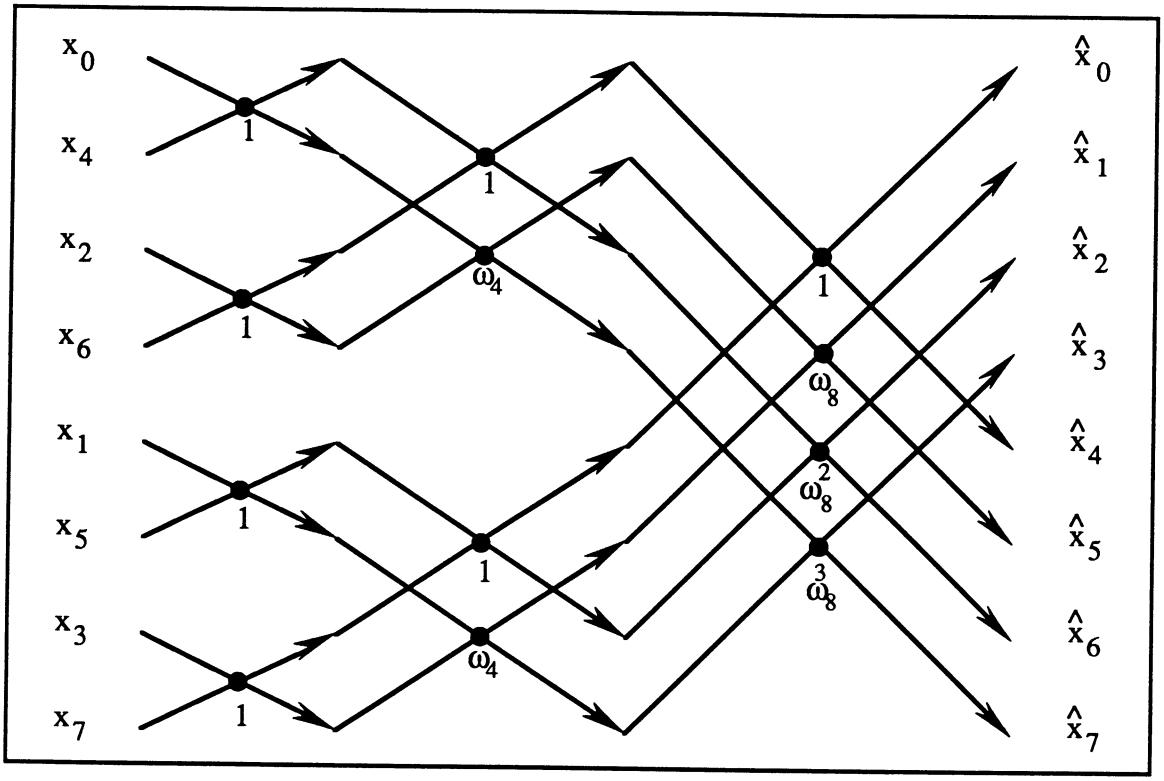


Figure 1 : graphe de circulation des donnée. Décimation en temps.

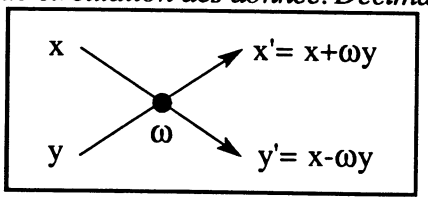


Figure 2 : effet d'un "papillon".

II.3.C.b La décimation en fréquence

On peut écrire $\hat{x}_i = \sum_{k=0}^{n-1} x_k.\omega_n^{ik} = \sum_{k=0}^{\frac{n}{2}-1} \left(x_k + x_{k+\frac{n}{2}} . \omega_n^{i\frac{n}{2}} \right) \omega_n^{ki}$

Si l'on sépare les valeurs d'indices pairs et impairs, on obtient

$$\begin{cases} \hat{x}_{2i} = \sum_{k=0}^{\frac{n}{2}-1} \left(x_k + x_{k+\frac{n}{2}} \right) \omega_n^{2ki} \\ \hat{x}_{2i+1} = \sum_{k=0}^{\frac{n}{2}-1} \left(x_k - x_{k+\frac{n}{2}} \right) \omega_n^{i}\, \omega_n^{2ki} \end{cases} \quad \text{pour } 0 \le i < \frac{n}{2}.$$

On prend un terme sur deux dans l'espace d'arrivée(les fréquences).

On effectue ainsi la FFT de manière récursive dans le même temps que par la décimation en temps, mais dans ce cas les entrées sont normalement ordonnées tandis que les résultats sont

ordonnés selon la transformation miroir, comme le montre le graphe de circulation des données représenté sur la figure 3.

Comme pour la décimation en temps, une écriture récursive de cet algorithme est possible, mais on préfère une implantation non récursive dont voici une écriture en pseudo-code :

```
procedure FFT (Tab,n)    /* décimation en fréquence */
    d : = n/2;
    h : = n;
    nboucle : = 1;
    tantque  d > 0 faire
        w : = 1
        pour j : =0 a d-1 faire
            pour i : = 0 à nboucle-1 faire
                aux : =    Tab[i*h+j] + Tab[i*h+j+d];
                Tab[i*h+j] : = w*(Tab[i*h+j] - Tab[i*h+j+d]);
                Tab[i*h+j+d] : = aux;
            fait
            w : = w* (ω_n)^(n/h)      /* (ω_n)^(n/h) = ω_h, w = (ω_h)^j */

        fait
        nboucle : = nboucle*2;
        h : = h/2;
        d : = d/2;
    fait
finprocedure
```

On peut remarquer que ces deux écritures sont très proches, on a juste modifié le sens de parcours des boucles. Cette similitude vient de la symétrie entre les graphes correspondants aux deux algorithmes. Pour une écriture récursive, on retrouverait une même symétrie : en utilisant la décimation en temps, l'appel récursif se ferait en début de procédure, alors qu'avec la décimation en fréquence, il se ferait à la fin.

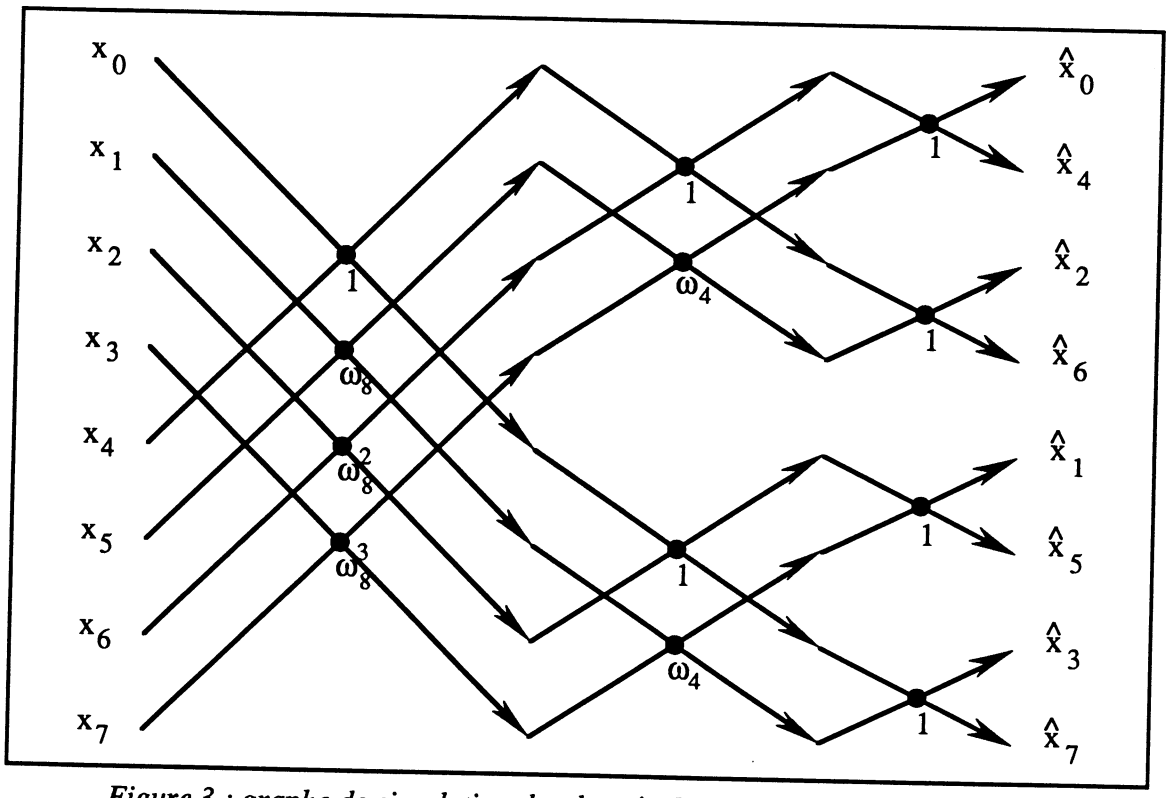


Figure 3 : graphe de circulation des donnée. Décimation en fréquence.

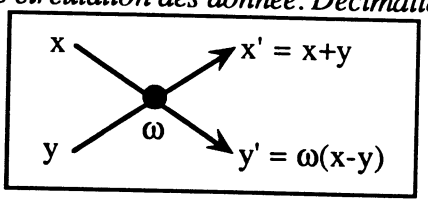


Figure 4 : effet d'un "papillon"

Donc pour réaliser un produit de convolution *en évitant d'avoir à réordonner les données par permutation miroir*, il suffit d'utiliser la décimation en fréquence pour effectuer la transformation directe et la décimation en temps pour effectuer la transformation inverse.

II.4 Transformées de Fourier dans des ensembles finis

Dans l'application des Transformations de Fourier Discrètes à la multiplication, nous avons en entrée des entiers inférieurs à B et en sortie des entiers inférieurs ou égaux à $(n+1)(B-1)^2$ (B est la base de numération utilisée). Il serait peu judicieux d'effectuer tous les calculs dans le champ complexe, d'autant plus que pour être sûr que les erreurs d'arrondi n'influent pas sur le résultat final, il faudrait effectuer tous les calculs sur un assez grand nombre de chiffres.

Strassen et Schönhage ont proposé en 1971 ([SS71]) un algorithme pour le produit de grands entiers qui réalise des DFT en effectuant uniquement des calculs sur des entiers. Cet algorithme donne un temps de calcul pour la multiplication en O(n.log(n).log(log(n))) mais ne s'applique que si la base de numération utilisée est une puissance de 2.

Nous proposons ici de travailler dans des corps finis particuliers qui nous donnent un algorithme utilisable pour différentes bases. Nous présentons d'abord les principes de toutes ces méthodes.

II.4.A Transformations arithmétiques (NTT)

On rappelle quelques définitions et résultats, pour une étude plus approfondie, on peut se référer à [NU80].

L'idée est de faire tous les calculs sur le corps $\frac{\mathbb{Z}}{q\mathbb{Z}}$ des entiers modulo q, où q est premier.

On appelle Transformation arithmétique ou NTT (de l'anglais *Number Theoretic Transform)* de longueur n et de racine g, la transformation sur $\frac{\mathbb{Z}}{q\mathbb{Z}}$ définie par $\hat{x}_i = \sum_{k=0}^{n-1} x_k . g^{ik}$ modulo q.

La transformation inverse lorsqu'elle est définie s'obtient par : $\bar{x}_i = \frac{1}{n} \sum_{k=0}^{n-1} x_k . g^{-ik}$ modulo q.

Si q est premier, toutes ces valeurs existent car g et n admettent des inverses. Si g est une racine nième primitive de l'unité modulo q, la NTT ainsi définie nous permet de calculer un produit de convolution d'entiers modulo q. Une condition nécessaire pour effectuer ces calculs est que n divise (q-1).

Si q n'est pas premier, on peut cependant définir une Transformation arithmétique inverse sur l'anneau $\frac{\mathbb{Z}}{q\mathbb{Z}}$, il suffit que n^{-1} et g^{-1} existent.

Si de plus, $g^n \equiv 1$ (q) et si pour $0 \le t < n$, $\sum_{k=0}^{n-1} g^{tk} \equiv 0$ (q), alors on peut utiliser cette NTT et son inverse pour calculer un produit de convolution modulo q. On peut noter que cette dernière condition est toujours vérifiée si q est premier et si n divise (q-1).

On sait que si a et q sont premiers entre eux, alors $a^{\phi(q)} \equiv 1$ (q) où ϕ est la fonction d'Euler ($\phi(m)$ est le nombre d'entiers plus petits que m et premiers avec m).

Généralement pour réaliser des convolutions d'entiers, on effectue des NTT modulo des nombres de Fermat ou de Mersenne :

• Un nombre de Fermat est un nombre de la forme $F_t=2^{2^t}+1$. Modulo F_t, on peut réaliser des NTT de longueur 2^{t+1} et de racine 2. En effet, $2^{2^t} \equiv -1$ (F_t) donc $2^{2^{t+1}} \equiv 1$ (F_t). De plus l'inverse de 2 modulo F_t est $2^{(2^{t+1}-1)}$ et celui de 2^{t+1} est $2^{(2^{t+1}-t-1)}$.

• Un nombre de Mersenne est un nombre de la forme $M=2^p-1$ où p est premier. Modulo M, on peut réaliser des NTT de longueur p et de racine 2, car $2^p \equiv 1$ (2^p-1) et comme p est premier, p divise $2^p-2 = M-1$. Donc 2 et p admettent un inverse modulo M.

L'algorithme de Schönhage et Strassen utilise des transformations arithmétiques modulo des nombres de Fermat.

Le principal avantage de travailler modulo un nombre de Fermat ou de Mersenne est que comme la racine de la NTT est 2, on n'a aucune "vraie" multiplication ou division à effectuer car les multiplications par des puissances de 2 se ramènent à des décalages si on travaille sur une machine représentant les entiers en base 2. De même, on peut très facilement calculer le reste d'un entier modulo un de ces nombres à l'aide de décalages. En contrepartie, la longueur maximale des NTT permises est de l'ordre du logarithme à base 2 du nombre (de Fermat ou de Mersenne) considéré.

II.4.B Algorithme de multiplication de Schönhage et Strassen

Cet algorithme a été publié en 1971 par Schönhage et Strassen ([SS71]). Il utilise des NTT modulo des nombres de Fermat. On peut en première approximation dire que l'idée de base est de découper un nombre de taille N en à peu près $\sqrt{N}$ nombres de taille $\sqrt{N}$, et de réaliser les produits de convolution modulo un nombre de Fermat supérieur à $\sqrt{N}$. Ce produit de convolution s'effectue à l'aide de l'algorithme de Transformée de Fourier Rapide. Les racines de l'unité modulo un nombre de Fermat sont des puissances de 2. On n'aura donc qu'à peu près $\sqrt{N}$ "vraies" multiplications à effectuer modulo ce nombre de Fermat (qui sera de l'ordre de grandeur de $\sqrt{N}$). La seconde idée de Schönhage et Strassen est que ces produits peuvent être eux mêmes effectués en faisant la même décomposition de manière récursive. Nous allons maintenant étudier cet algorithme plus en détail.

Soient deux suites (x_i) et (y_i) $(i = 0, \ldots, n-1)$
on appelle *convolution cyclique négative* de (x_i) et (y_i) la suite (d_i) définie par

$$d_i = \sum_{k=0}^{i} x_k . y_{i-k} - \sum_{k=i+1}^{n-1} x_k . y_{n+i-k}.$$

On note ω_{2n} une racine $2n^{ème}$ primitive de 1, on a $(\omega_{2n})^n = -1$ et on définit les suites (x'_i), (y'_i) et (d'_i) par $x'_i = (\omega_{2n})^i . x_i$, $y'_i = (\omega_{2n})^i . y_i$ et $d'_i = (\omega_{2n})^i . d_i$.
$(\hat{x}'_i)$, $(\hat{y}'_i)$ et $(\hat{d}_i)$ sont définies par $\hat{x}'_i = \sum_{k=0}^{n-1} x'_k . \omega_n^{ik}$, $\hat{y}'_i = \sum_{k=0}^{n-1} y'_k . \omega_n^{ik}$ et $\hat{d}_i = \hat{x}'_i . \hat{y}'_i$.

la suite (d'_i) vérifie alors $d'_i = \frac{1}{n} \sum_{k=0}^{n-1} \hat{d}_k . \omega_n^{-ik}$.

Si $X = \sum_{k=0}^{n-1} x_k . B^k$ et $Y = \sum_{k=0}^{n-1} y_k . B^k$ on a $(XY) \equiv \sum_{k=0}^{n-1} d_k . B^k \ (B^n+1)$, donc si B est une puissance de 2 et si n est de la forme 2^k, on se ramène à effectuer des calculs modulo 2^k+1.

Algorithme :

On veut multiplier 2 nombres X et Y tels que XY tienne sur n bits où n est une puissance de 2 ($n=2^v$). On multiplie en fait X et Y modulo $F_v = 2^{2^v}+1$ et comme n est pris assez grand, le résultat modulo F_v est le résultat exact.

• on découpe X et Y en $h=2^{\eta}$ chiffres de $m=2^{\mu}$ bits ($\eta+\mu=\nu$) avec $\eta \leq \mu+1$.

$X=\sum_{k=0}^{h-1} X_k.2^{mk}$ et $Y=\sum_{k=0}^{h-1} Y_k.2^{mk}$ où $X_k=\sum_{i=0}^{m-1} x_{mk+i}.2^i$ et $Y_k=\sum_{i=0}^{m-1} y_{mk+i}.2^i$

On a $XY \equiv \sum_{k=0}^{h-1} W_k.2^{mk}\ (2^{2^{\nu}}+1)$ où $W_k=\sum_{i=0}^{k} X_i.Y_{k-i} - \sum_{i=k+i}^{h-1} X_i.Y_{h+k-i}$

• On calcule $W'_k = W_k \bmod (2^{2^{\mu}}+1)$ et $W''_k = W_k \bmod (2^{\eta})$

comme $2^{2^{\mu}}+1$ et 2^{λ} sont premiers entre eux, on peut reconstituer W_k à partir de W'_k et W''_k.

• calcul des W'_k : 2 est racine $(2^{\mu+1})^{ème}$ de l'unité modulo F_{μ} et $\eta \leq \mu+1$, on peut donc calculer les W'_k en réalisant une convolution cyclique négative avec une transformation arithmétique modulo F_{μ}, comme vu ci-dessus, en utilisant l'algorithme de transformée de Fourier rapide. Comme les racines de l'unité qui interviennent dans le calcul sont des puissances de 2, les multiplications par ces racines sont en fait des décalages et se font en temps $O(2^{\mu})$. On n'a en fait que $h=2^{\eta}$ "vraies" multiplications modulo F_{μ} à effectuer. Et pour ces multiplications, *on réapplique récursivement l'algorithme sans avoir à doubler la taille des nombres car on veut effectivement le résultat modulo* F_{μ}. Ou bien, si μ est assez petit, on utilise l'algorithme classique de multiplication.

• calcul des W''_k : ce calcul se fait facilement. En effet, posons $X''_k=X_k \bmod (2^{\eta})$ et $Y''_k=Y_k \bmod (2^{\eta})$. Si on appelle X" le nombre dont les chiffres sont les X''_k en base $2^{3\eta}$ et Y" le nombre dont les chiffres sont les Y''_k en base $2^{3\eta}$, alors les chiffres en base $2^{3\eta}$ de X"Y" sont congrus aux W" modulo 2^{η}. Comme X" et Y" sont chacun des nombres écrits sur $3\eta.2^{\eta}$ bits, X"Y" peut être calculé en temps O(n) en utilisant l'algorithme classique.

• calcul des W_k : on pose d'abord $W'''_k= (2^{2^{\mu}}+1)((W''_k-W'_k) \bmod 2^{\eta})+W'_k$ et ensuite on obtient $W_k = W'''_k$ si $W'''< (k+1)2^{2^{\mu}}$ et $W_k = W'''_k - 2^{\eta}(2^{2^{\mu}}+1)$ sinon

• calcul de XY modulo $(2^{2^{\mu}}+1)$: Comme les W_k peuvent être plus grands que $2^{2^{\mu}}$, on doit effectuer une propagation de retenue qui se fait en temps linéaire.

Le temps total de calcul est en O(n.log(n).log(log(n))). Pour plus de détails sur l'algorithme et sa complexité, on peut se référer à [BMZ86].

Cet algorithme permet de réaliser des multiplications de taille arbitrairement grande, mais son implantation est difficile, et on ne peut l'appliquer que si l'on travaille avec une base qui est une puissance de 2. En pratique, il y a très peu d'implantations complètes de l'algorithme de Schönhage et Strassen dans sa version récursive.

II.4.C Algorithme de Pollard (1971)

On veut réaliser le produit de convolution des suites (x_i) et (y_i) (i=0, ..., n-1) avec x_i et y_i dans [0, B-1]. Chaque terme du produit vérifie donc $\sum_{h+m=i} x_h \cdot y_m < (n+1)(B-1)^2$. Il nous faut donc travailler dans un ensemble qui permette de représenter exactement toutes ces valeurs. Un autre impératif est de pouvoir trouver des racines $n^{ièmes}$ primitives de l'unité avec n qui soit une puissance de 2, pour pouvoir appliquer l'algorithme de Transformée de Fourier Rapide pour le calcul de la NTT.

Comme on l'a vu précédemment si a et q sont premiers entre eux, alors $a^{\phi(q)} \equiv 1$ (q). Donc si q est premier, $\phi(q) = q-1$ et pour tout entier a, $a^{q-1} \equiv 1$ (q) (petit théorème de Fermat). De plus il existe au moins une racine $(q-1)^{ème}$ de l'unité.

Pollard a proposé ([PO71]) de travailler dans le corps $K = \frac{\mathbb{Z}}{p\mathbb{Z}}$ où p est un nombre premier de la forme $k.2^q+1$.

On veut calculer le produit de convolution de deux suites de n entiers inférieurs à B. Si $n=2^m$ avec $m \leq q$ et $(n+1)(B-1)^2 < p$, on peut alors utiliser l'algorithme de FFT sur K et donc réaliser ce produit en temps O(n.log(n)) sans avoir à effectuer de calculs sur le corps des complexes. Sur ce type de corps fini, on peut réaliser des NTT de longueur $2^q = \frac{p-1}{k}$, et on peut donc multiplier des entiers de n chiffres en base B si $n.(B-1)^2 < p$. On n'est donc plus limité à une base égale à 2 ou à une puissance de 2.

La différence fondamentale avec l'algorithme de Schönhague-Strassen est que l'on peut réaliser des NTT dont la longueur est *de l'ordre du nombre premier choisi* et que tous les calcul se font modulo *un nombre premier qui tient dans un mot machine.*

Par exemple si l'on choisit $p=3.2^{30}+1$, tous les calculs peuvent être effectués sur un mot de 32 chiffres binaires. Pour cette valeur de p, on peut réaliser les combinaisons suivantes pour B et n :

$B=2^8$	$n=2^{15}$	
$B=2^4$	$n=2^{23}$	
$B=2$	$n=2^{30}$	(1 milliard de bits)
$B=100$	$n=2^{16}$	
$B=10$	$n=2^{25}$	(32 millions de chiffres décimaux)

Pour le même choix de p, on peut donc travailler avec des bases qui sont des puissances de 2 ou (ce qui peut être plus intéressant) de 10.

Voyons comment on effectue pratiquement le produit de deux entiers en base B en travaillant modulo p :
on a $p=k.2^q+1$, avec $(n+1)(B-1)^2<p$ et $n\leq 2^q$

X occupe m chiffres et Y n chiffres en base B. On suppose que m est supérieur ou égal à n.
Posons $X=\sum_{i=0}^{m-1} x_i.B^i$ et $Y=\sum_{i=0}^{n-1} y_i.B^i$,
on a $p=k.2^q+1$, $(n+1)(B-1)^2<p$ et $n+m\leq 2^q$.

• On cherche la plus petite valeur entière μ vérifiant $m+n\leq 2^\mu$, et on étend les nombres X et Y à 2^μ chiffres en ajoutant des zéros, c'est à dire que l'on écrit :

$$X=\sum_{i=0}^{2^\mu-1} x_i.B^i,\ Y=\sum_{i=0}^{2^\mu-1} y_i.B^i \text{ où } x_i=0 \text{ si } i\geq m \text{ et } y_i=0 \text{ si } i\geq n.$$

• On effectue une transformation arithmétique modulo p sur les chiffres de X et Y ainsi étendus. Cette transformation est réalisée par l'algorithme FFT avec décimation en fréquence.

• On effectue les produits terme à terme sur les vecteurs transformés ce qui nous donne un nouveau vecteur Z'.

• On effectue la transformation inverse modulo p sur Z', ce qui nous donne un vecteur Z dont les composantes sont les convolutions des chiffres des x_i et des y_i.

On a alors $Z=\sum_{i=0}^{2^\mu-1} z_i.B^i$ mais les z_i peuvent être supérieurs ou égaux à B. On réalise une propagation de retenue pour obtenir Z=XY en base B.

Pour réaliser une implantation machine efficace, il est intéressant d'utiliser une valeur de p dont la taille est la plus proche possible de celle du mot machine. Mais cela nous limite pour le choix de B ainsi qu'on l'a vu ci dessus. C'est pourquoi Pollard a proposé d'améliorer la méthode en travaillant modulo *plusieurs nombres premiers tenant dans un mot machine.*

On a d nombres premiers de la forme $p_i=k_i.2^{q_i}+1$ (on suppose $q_1\geq \ldots \geq q_d$). Alors on peut réaliser le produit d'un entier de m chiffres en base B par un entier de n chiffres en base B (on suppose $m\geq n$), si m et n vérifient $(n+1)(B-1)^2<\prod_{i=1}^{d} p_i$ et $n+m\leq 2^{q_d}$.

Pour cela *on réalise indépendamment d convolutions modulo les p_i* (toujours via les NTT et l'algorithme FFT). Et on trouve les valeurs $z_{k,i}=z_k \bmod(p_i)$ pour tous les p_i ($z_k=\sum_{h+m=k} x_h.y_m$). Il

suffit de remarquer que si on connait $z_{k,i} = z_k \bmod(p_i)$ pour $1 \le i \le d$, alors on peut retrouver z_k par le théorème des restes chinois.

On pose $N=\prod_{i=1}^{d} p_i$ et $N_i=\frac{N}{p_i}=\prod_{j\neq i} p_j$. Pour tout i, p_i et N_i sont premiers entre eux et donc, d'après le théorème de Bezout, il existe r_i et s_i tels que $r_i.p_i+s_i.N_i=1$ (comme p_i est ici premier on peut prendre pour s_i l'inverse de N_i modulo p_i et donc $0 \le s_i < p_i$).

Si $x \equiv x_i$ (p_i) on a alors directement $x \equiv \sum_{i=1}^{d} x_i.s_i.N_i$ (N), car $s_i.N_i \equiv 1$ (p_i) et $s_j.N_j \equiv 0$ (p_i) si $j \neq i$.

Pour accélérer le calcul, on peut poser $\xi_i = x_i.s_i \bmod (p_i)$ et on a $x \equiv \sum_{i=1}^{d} \xi_i.N_i$ (N). On remarque qu'ainsi on a toujours $\xi_i.N_i < N$ car $\xi_i < p_i$.

En pratique on choisit comme base la puissance de 2 correspondant à la taille du mot machine ($B=2^{32}$ ou 2^{16} en général). Pour réaliser des produits d'entiers la valeur de d la plus intéressante est donc 3, car $N=p_1.p_2.p_3$ est alors proche de B^3 et on peut réaliser des produits de nombres dont la taille est de l'ordre de B chiffres en base B.

II.4.C.a Complexité :

On pourrait *a priori* croire que la complexité est en O(n.log(n)), mais il ne faut pas oublier que le nombre de nombres premiers nécessaires (les p_i ci dessus) dépend de la longueur des entiers à multiplier : en effet on doit avoir $(n+1)(B-1)^2 < \prod_{i=1}^{d} p_i$ soit $d \ge \frac{\log(n+1)}{\log(B-1)} + 2$ (car $p_i \le B-1$), ce qui nous donne une complexité en $O(n.\log(n)^2)$. En fait, si on utilise 3 nombres premiers tenant sur 32 bits, on peut réaliser des produits d'entiers de 10^9 mots de 32 bits, et on a dans ces limites une complexité pratique en O(n.log(n)).

II.5 Implantation séquentielle

On a réalisé une implantation complète de l'algorithme précédent avec $p_1=3.2^{31}+1$, $p_2=13.2^{28}+1$ et $p_3=29.2^{27}+1$ sur différentes machines 32 bits. On a réalisé une procédure interfacée avec le logiciel BigNum réalisé par B. Serpette, J. Vuillemin et J.C. Hervé ([SVH89]), afin d'effectuer des comparaisons. Notre procédure s'appelle FastMult et réalise à peu près la même opération que la procédure BnnMultiply du logiciel BigNum. Ces deux procédures s'appellent avec les mêmes paramètres :

BnnMultiply(Z,p,X,m,Y,n) calcule Z+X.Y et range le résultat dans Z. p est la longueur de Z, m la longueur de X, et n celle de Y. On impose $p \ge m+n$. Cette procédure peut générer une retenue.

FastMult(Z,p,X,m,Y,n) calcule X.Y et range le résultat dans Z. Cette procédure ne génère pas de retenue.

Il faut noter que la procédure FastMult demande que soit réservée la place de 4 tableaux dont la longueur est la plus petite puissance de 2 supérieure à la longueur du résultat. En effet Pour calculer les produit de convolution modulo p_i il faut deux tableaux pour faire les deux FFT, et il en faut deux autres pour stocker les produits de convolution modulo p_1 et p_2 pendant qu'on calcule modulo p_3. En règle générale si on utilise d nombres premiers différents, il faut réserver d+1 tableaux. On peut remarquer qu'en fait, un seul tableau suffirait si on sauvegardait les calculs intermédiaires sur fichiers.

On n'a pas tenu compte du temps d'allocation de la place nécessaire et on a une fois pour toutes réservé un espace de travail suffisant.

Pour mesurer les temps de calcul, on a tiré au hasard les chiffres en base 2^{32} puis on a lancé les deux procédures à comparer pour le même jeu de données.

On appelle $T_B(p)$ le temps de calcul du produit de deux nombres de p chiffres par la procédure BnnMultiply et $T_F(p)$ le temps mis par la procédure Fastmult.

Les nombres sont écrits en base 2^{32} et on peut donc (en théorie) réaliser des produits de nombres de 2^{27} chiffres en base 2^{32}. On s'est arrêté à 2^{20}.

On présente en annexe les programmes sources en langage C des procédures correspondantes (arithmétique modulo p_i, transformations arithmétiques, restes chinois ...). On donne aussi pour les différentes machines utilisées les sources en assembleur de l'arithmétique modulo p. Ceci donne pratiquement une multiplication de grands entiers "clefs en main", les parties laissées à l'initiative de l'utilisateur potentiel sont liées à sa façon de représenter ou mémoriser les entiers.

Enfin on comparé les temps obtenus avec deux implantations fictives de l'algorithme de Karatsuba vue au paragraphe II.2.

Une première implantation donne un temps de calcul $T_K(n) = \frac{T_B(n_0)+C_Z.n_0}{{n_0}^{\omega}} n^{\omega} - C_Z.n$, où $T_B(p)$ est le temps de calcul par la procédure BnnMultiply, et C_Z est estimé par la formule donnée par P.Zimmermann [ZIM89] ($\omega=\log_2 3 = 1{,}58...$) :

$$C_Z = 7.t_+ + 4.t_- + 4.t_{RAZ} + 2.t_{copy} = 11.t_+ + 4.t_{RAZ} + 2.t_{copy}$$

n_0 est choisi de façon à minimiser la valeur $\frac{T_B(n_0)+C_Z.n_0}{{n_0}^{\omega}}$.

On a aussi regardé ce que donne l'implantation *"idéale"* présentée au paragraphe II.2, dont on sait qu'elle est plus rapide que toutes les implantations possibles de l'algorithme de Karatsuba.

$$T_K(n) = \frac{t_x.{n_0}^2+8.t_+.n_0}{{n_0}^{\omega}} n^{\omega} - 8.t_+.n, \text{ où } n_0 = \frac{8.t_+(\omega-1)}{t_x.(2-\omega)}.$$

On a donc pour chacune des machines testées deux implantations fictives de l'algorithme de Karatsuba : une "raisonnablement optimiste" qui reprend les estimation de P.Zimmermann et

utilise les temps de BnnMultiply et une implantation *idéale* dont les temps minorent ceux de toutes les implantations possibles.

II.5.A Machines cibles

Les machines utilisées étaient des Sun à base de microprocesseurs MC68020, MC68030 ou SPARC. Les 68030 possèdent dans leur jeu d'instruction la multiplication 32 bits par 32 bits et la division 64 bits par 32 bits, ce qui permet de réaliser facilement les opérations modulo p (si p tient sur 32 bits). Dans le cas où ces instructions ne sont pas accessibles, on réalise les opérations modulo p en langage de haut niveau (langage C dans notre cas) avec bien entendu une chute de performance.
On a donc ainsi réalisé une version optimisée en assembleur 68020 pour Sun3 et une version entièrement en langage C qu'on a testé sur Sun4. Cela nous a permis de faire des comparaisons intéressantes avec le logiciel BigNum qui existe en version optimisée en assembleur (entre autre pour machines à base de 68030) ou entièrement en C. On a donc comparé les versions en assembleur sur Sun3 et les versions en C sur Sun4.
Les estimations des constantes qui interviennent dans l'étude de complexité de l'algorithme de Karatsuba on donné les résultats ci dessous. Les temps sont donnés en secondes.

	$C_Z=11.t_+ + 4.t_{RAZ} +2.t_{copy}$	$8.t_+$	t_x	$n_0=\frac{8.t_+(\omega-1)}{t_x.(2-\omega)}$	$\frac{t_x.n_0^2+8.t_+.n_0}{n_0^\omega}$
MC 68030	$2,1.10^{-5}$	10^{-5}	$3,3.10^{-6}$	4,27	$1,02.10^{-5}$
SPARC	$1,5.10^{-5}$	8.10^{-6}	3.10^{-6}	3,76	$8,9.10^{-6}$

II.5.B Résultats pratiques

Les différents tableaux et schémas qui suivent donnent les résultats des mesures effectuées.
On peut d'abord remarquer que l'on a obtenu des résultats semblables sur les deux machines, tant qualitativement que quantitativement : les versions de BnnMultiply et Fastmult en langage C sur Sun4 ont des vitesses comparables à celles de leurs implantations respectives optimisées sur Sun3.
On a retrouvé les complexités attendues : à savoir $O(n^2)$ pour BnnMultiply et $O(n.\log(n))$ pour FastMult, ce qui nous a permis d'extrapoler les mesures pour des très grandes tailles d'entiers.

Pour les "petits" nombres (moins de 4000 bits sur Sun4 et moins de 8000 bits sur Sun3) l'algorithme classique est supérieur.
Ensuite l'algorithme de Pollard prend le pas et permet d'aller jusqu'à des tailles très importantes d'entiers : le produit de 2 entiers de 4 millions de bits (1 million de chiffres décimaux) se fait en 4 minutes et le produit de 2 entiers de 32 millions de bits (10 millions de chiffres décimaux) se ferait en 37 minutes si on disposait de la taille mémoire suffisante (32 Mo).

Pour les deux implantations fictives de l'algorithme de Karatsuba, on peut d'abord remarquer qu'entre l'implantation "parfaite" et l'implantation "raisonnable" basée sur les estimations de P.Zimmerman, on trouve un facteur 2 sur Sun3 et un facteur 3 sur Sun4. De plus il est peu probable qu'une implantation réelle atteigne les performance de l'implantation que nous avons qualifiée de "raisonnable". Sur un exemple réel d'implantation de l'algorithme de Karatsuba (en lisp) sur VAX 785 [ZIM89], les temps de calcul mesurés sont :

$$T_K(n) = (6{,}2.n^{\omega} - 13{,}5.n).10^{-5},$$

alors que les estimations de P.Zimmermann donnent :

$$T_K(n) = (1{,}2.n^{\omega} - 5{,}4.n).10^{-5}.$$

Le rapport entre les performances mesurées et théoriques est donc de 5 dans ce cas précis.

Le choix de comparer notre implantation de l'algorithme de Pollard à des implantations fictives de l'algorithme de Karatsuba est donc parfaitement légitimé par le calcul et l'expérience.
Il apparaît que sur les deux machines testées, l'algorithme de Pollard (effectivement implanté) prend le pas sur l'hypothétique implantation "parfaite" de l'algorithme de Karatsuba quand la taille des entiers à multiplier dépasse 8000 mots machines (80000 chiffres décimaux). Face à une implantation réelle de l'algorithme de Karatsuba, on prend le pas encore plus vite : si l'implantation réelle va seulement 4 fois plus lentement que l'implantation "parfaite" on gagne à partir de nombres de 5000 chiffres décimaux.
IL faut cependant remarquer que ces tailles de nombres ne sont pas courantes et que donc la plage d'utilisation de l'algorithme de Pollard est limitée aux "grands" nombres.

La table ci-dessous donne les approximations des temps de calcul pour les différents algorithmes testés sur les deux types de machines. Les temps sont exprimés en secondes.

	BnnMultiply	Fastmult	Karatsuba "raisonnable"	Karatsuba "parfait"
MC 68030	$(4+3{,}9.n).n.10^{-6}$	$(4{,}2+0{,}83.\log_2 n).n.10^{-4}$	$(1{,}85.n^{\omega} - 2{,}1.n).10^{-5}$	$(1{,}02.n^{\omega} - 1{,}0.n).10^{-5}$
SPARC	$(4{,}4+0{,}9.n).n.10^{-5}$	$(4{,}5+0{,}77.\log_2 n).n.10^{-4}$	$(2{,}64.n^{\omega} - 1{,}5.n).10^{-5}$	$(8{,}9.n^{\omega} - 8.n).10^{-6}$

les tables qui suivent donnent les détails des temps mesurés ou extrapolés. Sur les schémas, on a indiqué les temps de l'implantation "raisonnable" de l'algorithme de Karatsuba.

N= nombre de mots	n = nombre de bits	Fastmult $T_F(N)$	BnnMul. $T_B(N)$	Karatsuba $N_0 = 16$	$\frac{T_B(N)-C_Z.N}{N^\omega}$	Karatsuba "parfait"
1	32		3,162e-5		5,262e-5	
2	64		5,666e-5		3,289e-5	
4	128		1,266e-4		2,340e-5	*5,180e-5*
8	256	5,499e-3	3,610e-4		1,959e-5	*1,954e-4*
16	512	0,012	1,166e-3	*1,166e-3*	1,854e-5	*6,663e-4*
32	1024	0,027	4,333e-3	*3,834e-3*	2,059e-5	*2,159e-3*
64	2048	0,059	0,016	*0,013*	2,332e-5	*6,797e-3*
128	4096	0,129	0,065	*0,038*	3,109e-5	*0,021*
256	8192	0,283	0,267	*0,117*	4,145e-5	*0,064*
512	16384	0,600	1,000	*0,354*	5,135e-5	*0,196*
1024	32768	1,300	4,067	*1,073*	6,923e-5	*0,592*
2048	65536	2,733	16,233	*3,242*	9,185e-5	*1,787*
4096	131072	5,850	65,231	*9,770*	1,229e-4	*5,381*
8192	262144	12,366	261,423	*29,397*	1,640e-4	*16,186*
16384	524288	26,149	*1048,57*	*88,364*		*48,640*
32768	1,049e+6	55,06	*4194,30*	*265,454*		*146,093*
65536	2,097e+6	115,478	*16777,21*	*797,049*		*438,608*
131072	4,194e+6	243,857	*67108,86*	*2392,615*		*1316,529*
262144	8,389e+6	*506,724*	*268173,31*	*7180,599*		*3950,899*
524288	1,681e+7	*1057,489*	*1,073e+6*	*21547,302*		*11855,317*
1048576	3,355e+7	*2203,058*	*4,292e+6*	*64656,624*		*35573,234*

Figure 5 : Temps sur Sun3 (68030) $C_Z = 2{,}1.10^{-5}$, les valeurs en italique sont extrapolées.

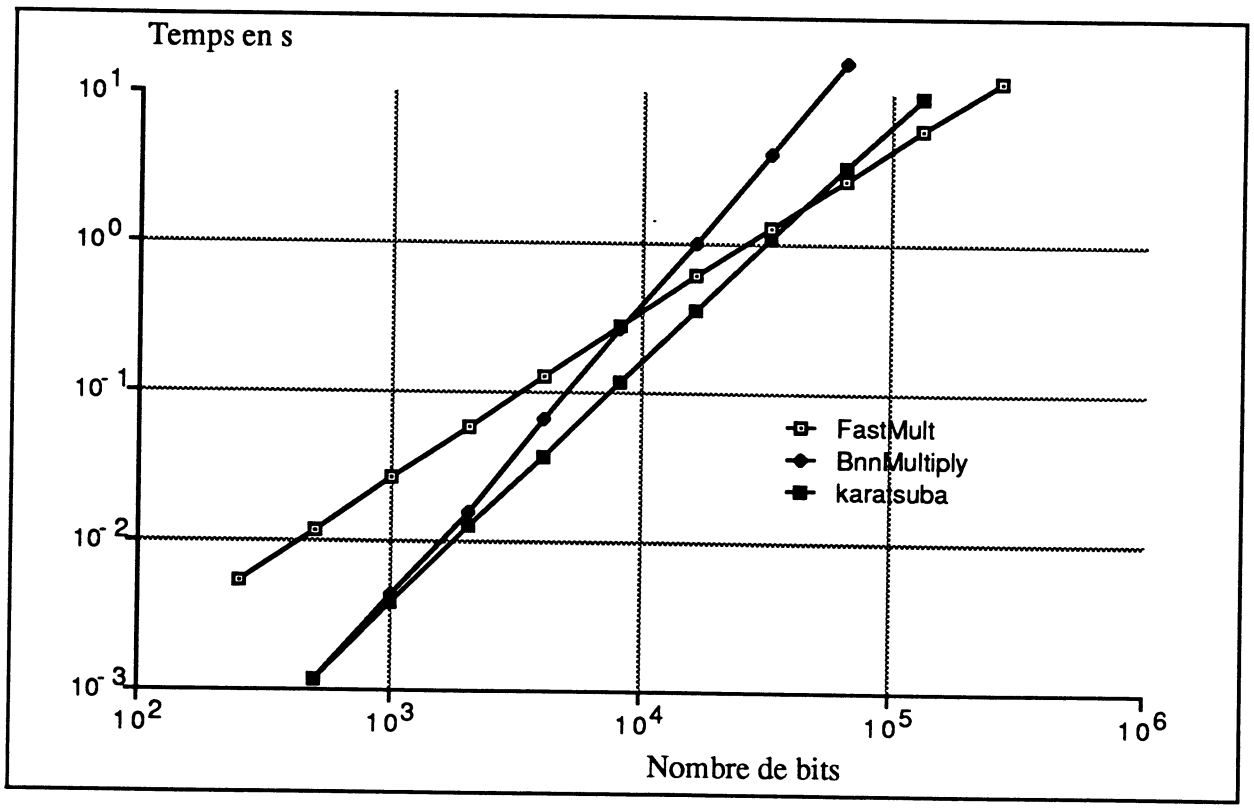


Figure 6 : croissances comparées des temps de calcul sur Sun3.

N= nombre de mots	n = nombre de bits	Fastmult $T_F(N)$	BnnMul. $T_B(N)$	Karatsuba $N_0 = 4$	$\frac{T_B(N)-C_Z.N}{N^\omega}$	Karatsuba "parfait"
1	32		1,6e-5		3,100e-5	
2	64		5,01e-5		2,670e-5	
4	128		1,78e-4	*1,780e-4*	2,644e-5	*4,810e-5*
8	256	5,333e-3	6,45e-4	*5,939e-4*	2,833e-5	*1,763e-4*
16	512	0,013	2,5e-3	*1,902e-3*	3,382e-5	*5,930e-4*
32	1024	0,026	0,010	*5,946e-3*	4,115e-5	*1,907e-3*
64	2048	0,058	0,038	*0,018*	5,349e-5	*5,977e-3*
128	4096	0,133	0,150	*0,056*	6,949e-5	*0,018*
256	8192	0,267	0,600	*0,170*	9,204e-5	*0,056*
512	16384	0,567	2,367	*0,513*	1,206e-4	*0,171*
1024	32768	1,217	9,516	*1,547*	1,614e-4	*0,518*
2048	65536	2,617	38,032	*4,654*	2,148e-4	*1,561*
4096	131072	5,533	153,444	*13,995*	2,888e-4	*4,698*
8192	262144	12,050	*606,208*	*42,046*		*14,129*
16384	524288	25,582	*2441,216*	*126,261*		*42,453*
32768	1,049e+6	*52,462*	*9732,096*	*379,054*		*127,498*
65536	2,097e+6	*109,904*	*38928,38*	*1137,656*		*382,754*
131072	4,194e+6	*229,900*	*155713,53*	*3414,082*		*1148,831*
262144	8,389e+6	*479,986*	*622854,14*	*10244,212*		*3447,543*
524288	1,681e+7	*999,817*	*2,492e+6*	*30736,568*		*10344,726*
1048576	3,355e+7	*2080,375*	*9,967e+6*	*92222,855*		*31040,151*

Figure 7 : Temps sur Sun4 (Sparc) $C_Z = 1{,}5.10^{-5}$ les valeurs en italique sont extrapolées.

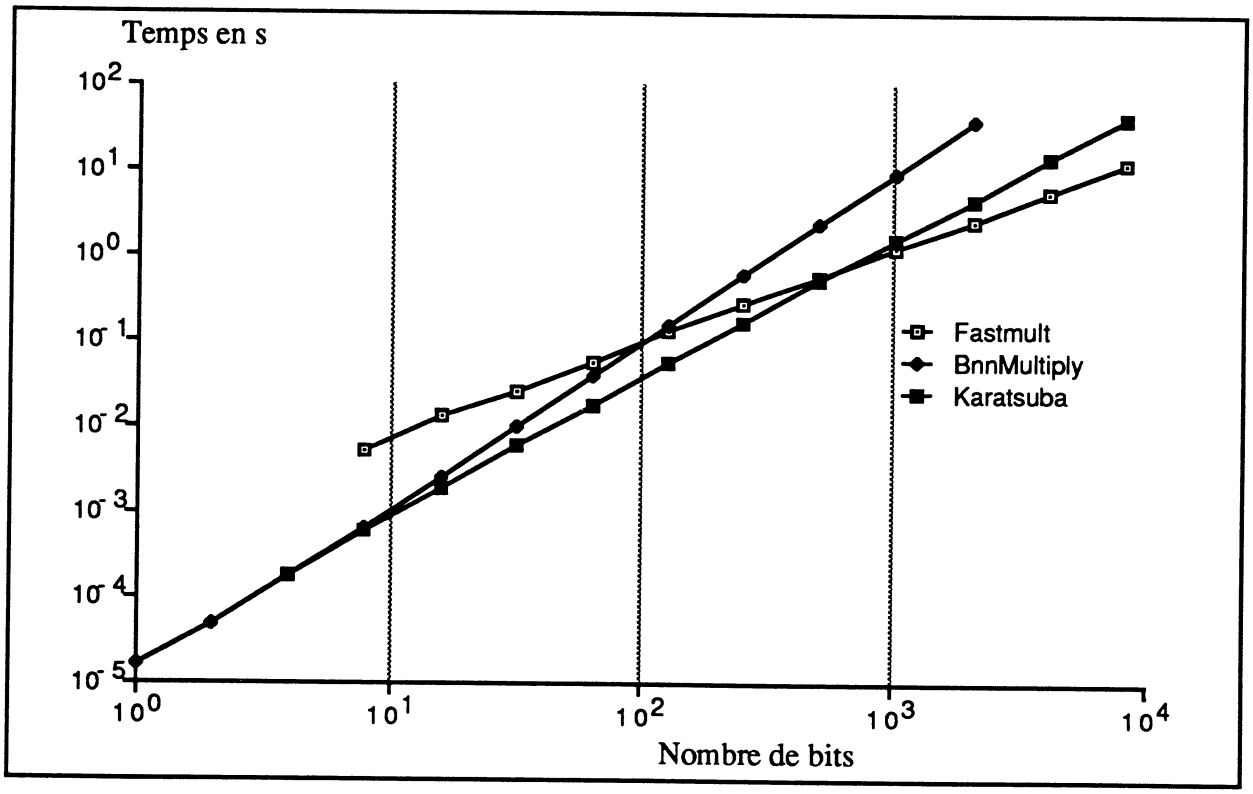


Figure 8 : Figure 6 : croissances comparées des temps de calcul sur Sun4.

II.6 Implantation parallèle

Pour l'implantation parallèle, on s'est limité à paralléliser la partie "sensible", à savoir le produit de convolution de deux vecteurs d'entiers modulo p. On a donc réalisé une implantation parallèle des transformations arithmétiques modulo p (où $p=k.2^q+1$ et $p<2^{32}$) calculées à l'aide de l'algorithme FFT.

II.6.A Machine cible

On disposait d'une machine parallèle FPS T40 de Floating Point System, avec 32 Transputers T414 disposés en hypercube.
Il s'agit d'une machine à mémoire distribuée dans laquelle chaque processeur ne peut accéder directement qu'à son espace mémoire propre. Chaque processeur a aussi la possibilité d'échanger des données avec les processeurs voisins en utilisant des primitives de communication. Les processeurs sont numérotés de sorte que si deux processeurs sont voisins sur l'hypercube leurs numéros (appelés codes de Gray) respectifs ne diffèrent que d'un bit. Différents exemples sont donnés figure 9.
Tous les processeurs reçoivent le même programme. Afin de pouvoir exécuter des instructions différentes, il ont la possibilité de connaître leurs codes de Gray respectifs, ceux-ci étant rangés dans une variable que nous appellerons **numéro**.

Pour l'application spécifique du calcul de la FFT on n'a utilisé que la primitive de communication suivante :
comm(Numéro du bit, Départ, Arrivée, @départ, @arrivée, Longueur du message)
Lorsqu'un processeur exécute cette instruction, il réalise une communication avec son voisin dans la $d^{ième}$ direction (d est le numéro du bit qui diffère).
Les paramètres départ et arrivée valent 0 ou 1 (arrivée = 1-départ). La communication se fait du processeur dont le $d^{ième}$ bit vaut départ vers l'autre. Le processeur émetteur envoie alors les valeurs commençant à l'adresse @départ et le processeur récepteur les range à partir de l'adresse @arrivée. La communication ne peut se faire que si les deux processeurs exécutent en même temps cette primitive. Le temps d'une communication de n mots machine d'un processeur à un de ses voisin est de $T_{start} + n.T_{com}$ (T_{start} est le temps d'initialisation de la communication).

Le Transputer dispose d'instructions microprogrammées pour réaliser des produits 32 bits par 32 et des divisions 64 bits par 32. Un produit modulo p demande de l'ordre d'une centaine de cycles, et les temps de communication seront en grande partie négligeables devant les temps de calcul.

II.6.B Parallélisation de la FFT sur un hypercube

On va présenter des algorithmes parallèles de calcul de FFT sur un hypercube de diamètre k à $P=2^k$ processeurs.

Pour l'analyse de l'accélération, nous appellerons T_{pap} le temps d'un produit "papillon" et T_{mul} le temps d'un produit simple. Dans notre implantation sur Transputer, T_{pap} et T_{mul} sont proches. T_{start} et T_{com} ont été définis précédemment.

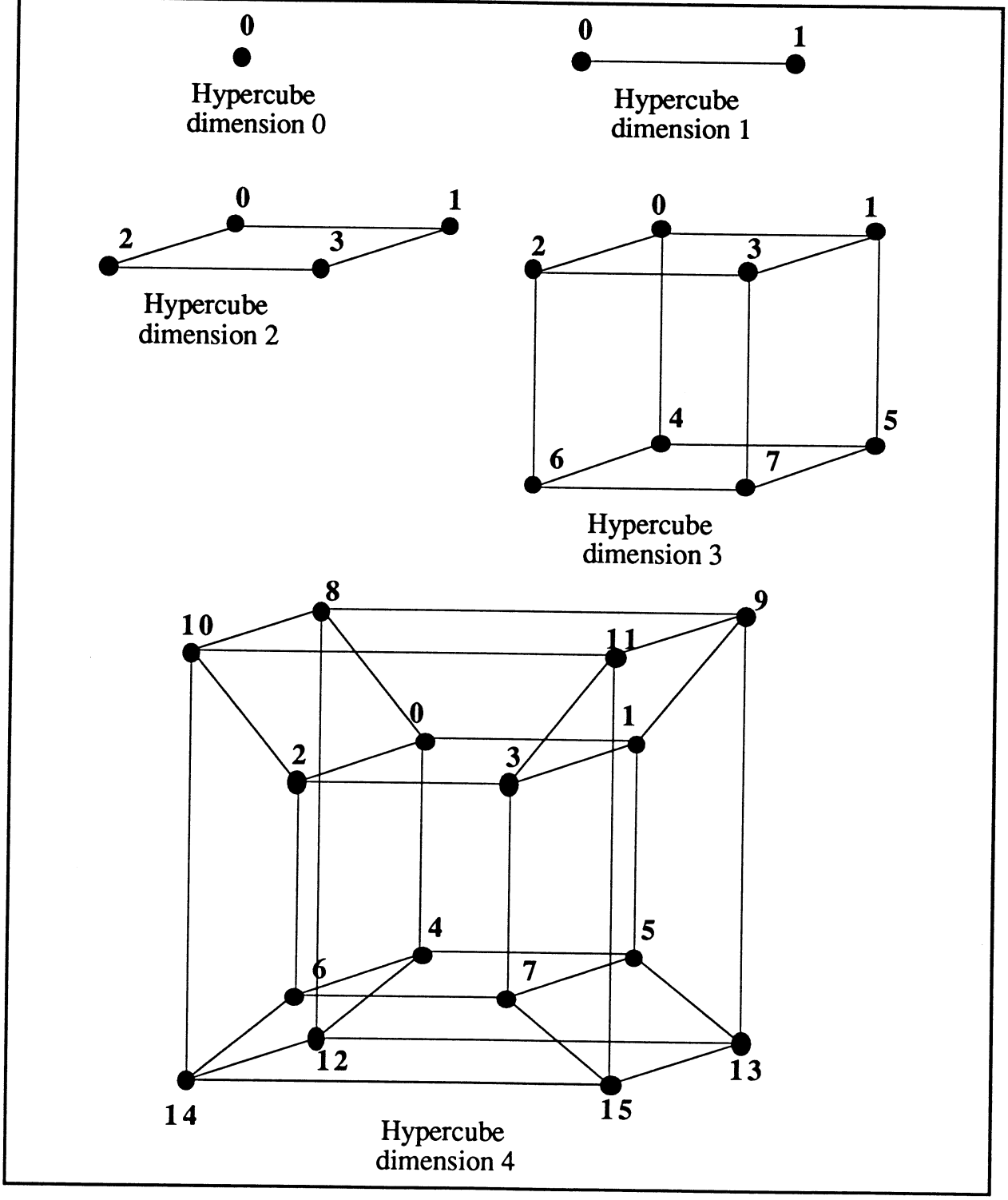


Figure 9 : hypercubes et codes de Gray.

II.6.B.a Modifications de l'algorithme séquentiel.

Reprenons l'algorithme de FFT par décimation en temps, vu précédemment, et voyons ce que désignent les variables par rapport au graphe de circulation des données.

```
procedure FFT (Tab,n)    /* décimation en temps */
    d : = 1;
    h : = 2;
    nboucle : = n/2;
    tantque  d < n faire
        w : = 1
        pour j : = 0 à d-1 faire
            pour i : = 0 à nboucle-1 faire
                aux : = Tab[i*h+j] + w*Tab[i*h+j+d];
                Tab[i*h+j] : = Tab[i*h+j] - w*Tab[i*h+j+d];
                Tab[i*h+j+d] : = aux;
            fait
            w : = w* (ωn)^n/h      /* (ωn)^n/h = ωh, w = (ωh)^j */
        fait
        nboucle : = nboucle/2;
        h : = h*2;
        d : = d*2;
    fait
finprocedure
```

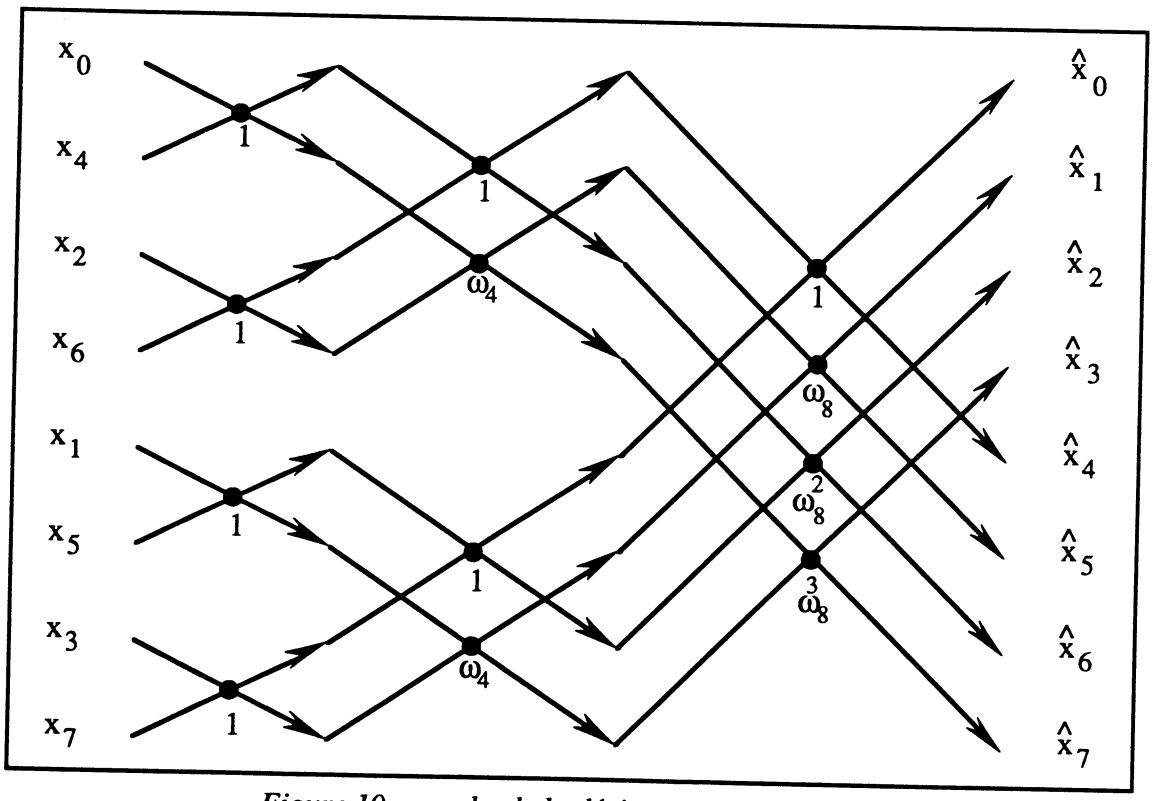


Figure 10 : graphe de la décimation en temps.

Dans la boucle principale (sur d), chaque passage correspond à un étage. A chaque étage d'une FFT de taille n, on effectue en fait les $\frac{n}{h}$ derniers étages d'une FFT de taille h, et tous les produits "papillon" du même étage se font sur des données distantes de d positions.
Les deux boucles intérieures peuvent être inversées. la boucle "i" correspond à des "sous-FFT" du même étage et la boucle "j" correspond à des données d'une même "sous-FFT" : "i" varie de 0 à "nboucle"-1, la variable "nboucle" indiquant le nombre de "sous-FFT" indépendantes. : "j" varie de 0 à "d"-1,la variable "d" étant la moitié de la longueur des "sous-FFT".
En plaçant la boucle "i" à l'intérieur, on calcule une seule fois $(\omega_h)^j$ dans un étage pour j donné.
L'algorithme de décimation en fréquence s'analyse de la même façon.
Voyons ce que donne la procédure écrite en inversant les deux boucles internes : cette deuxième version est moins performante pour une implantation séquentielle car on calculera plusieurs fois les mêmes valeurs $(\omega_h)^j$.
On aura $\frac{n}{2}\log(n)$ produits papillon et n-1 produits simples dans la première version contre $\frac{n}{2}\log(n)$ produits papillon et $\frac{n}{2}\log(n)$ produits simples dans la deuxième.

```
tantque  d < n faire
    pour i : = 0 à nboucle-1 faire
        w : = 1
        pour j : = 0 à d-1 faire
            aux : = Tab[i*h+j] + w*Tab[i*h+j+d];
            Tab[i*h+j] : = Tab[i*h+j] - w*Tab[i*h+j+d];
            Tab[i*h+j+d] : = aux;
        w : = w* (ωn)n/h      /* (ωn)n/h = ωh, w = (ωh)j */
        fait
    fait
    nboucle : = nboucle/2;
    h : = h*2;
    d : = d*2;
fait
```

Pour paralléliser la FFT, on parallélise la boucle la plus interne ce qui nous conduit à deux méthodes différentes, qui proviennent des deux écritures séquentielles de l'algorithme. Dans ces deux méthodes, la boucle "d" s'exécute séquentiellement et les communications éventuelles entre processeurs se font entre deux passages dans la boucle.
Nous rappelons que nous voulons paralléliser l'ensemble FFT-FFT inverse pour réaliser un produit de convolution, avec la décimation en fréquence pour la FFT et la décimation en temps pour la FFT inverse (en fait on calcule deux FFT et une FFT inverse mais l'analyse est la même).

II.6.B.a.1 Boucle "i" à l'intérieur

A chaque passage dans la boucle "d", on associe à un processeur un "sous étage" complet de FFT. Un même sous étage peut être calculé par plusieurs processeurs ce qui introduit une redondance dans les calculs, ainsi qu'on va le voir.
L'algorithme parallèle obtenu est illustré par le schéma de la figure 11 dans le cas d'un hypercube à 8 processeurs numérotés de 0 à 7. Chaque ligne correspond à un étage de la FFT ou de la FFT inverse et représente la répartition des données et des calculs par processeur. Les calculs se font du haut vers le bas. La ligne représente en fait le tableau des données. Les traits verticaux indiquent le découpage du tableau entre les processeurs, les numéros sur la ligne indiquent les numéros des processeurs travaillant sur la partie correspondante du tableau. Les communications entre processeurs sont représentées par les flèches verticales. Le numéro du lien correspond à la direction dans laquelle communiquent les processeurs (ou au numéro du bit).

Ainsi la première ligne correspond au premier passage dans la boucle "d" lors de la FFT par décimation en fréquence : tous les processeurs effectuent les mêmes calculs sur le vecteur complet qui est donc répliqué dans toutes les mémoires locales.
Sur la deuxième ligne le vecteur est découpé en deux parties : les processeurs 0 à 3 travaillent sur la première moitié et les processeurs de 4 à 7 sur la seconde moitié.
Sur la troisième ligne le vecteur est découpé en 4, et enfin à de la quatrième ligne, on à ramené notre FFT de taille n à 8 FFT de taille n/8.
La partie centrale correspond à la fin de la FFT par décimation en fréquence et au début de la FFT inverse par décimation en temps. Tous les processeurs travaillent alors en parallèle sur des données différentes et il n'y a pas besoin de communications.
Les dernières lignes du schéma correspondent à la fin de la FFT inverse par décimation en temps. Comme on l'a vu, lorsque l'on fait la décimation en temps, on reconstruit le résultat d'une FFT à partir des résultats de deux FFT de taille moitié. Si ces deux FFT on été calculées sur deux processeurs différents, il faut communiquer les résultats pour achever le calcul. C'est pourquoi cette partie comprend des communications entre chaque ligne.

On donne plus loin une écriture en pseudo-code des versions parallèles de la FFT par décimation en fréquence et de la FFT inverse par décimation en temps.

On voit que l'efficacité n'est pas maximale, car de nombreux calculs sont effectués plusieurs fois. En effet, au $k^{ème}$ étage de la FFT, on n'a que 2^k sous-étages et donc tous les processeurs ne travaillent pas. Par contre, on minimise les communications. Cet algorithme impose de plus que chaque processeur ait en mémoire toutes les données au départ. L'architecture en hypercube est bien adaptée, car les communications se font de voisin à voisin.

On peut déjà remarquer que dans la partie avec communication, des processeurs effectuent le même calcul en même temps, et que donc cet algorithme n'est surement pas optimal.

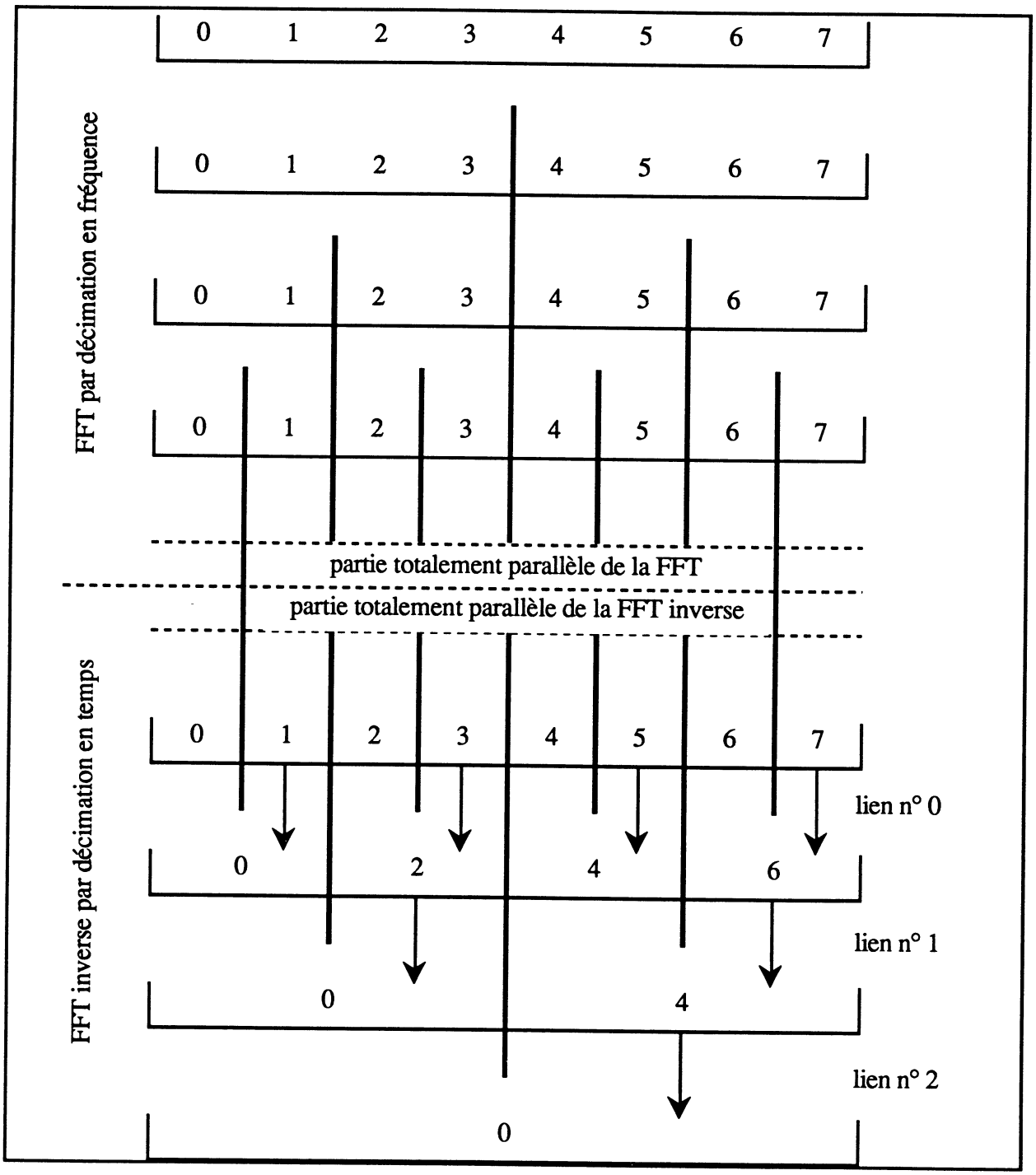


Figure 11 : répartition des données entre les processeur set graphe de communications dans le cas du 3cube (p=8).

Algorithme parallèle de Transformée de Fourier discrète par décimation en fréquence (entrées normales, sorties miroir).
Boucle "i" à l'intérieur.
Tous les processeurs ont le même programme et chacun peut connaître sa place par la variable "numero" qui lui donne son code de gray.

```
procedure fftpar(Tab,n)
    prof : = k;  /* on a P = 2^k processeurs */
    debut : = 0;
    h : = n;
    d : = n/2;
    nboucle : =1;
    tantque d>0 faire
        w : =   1;
        pour j : = 0 à d-1 faire
            pour i : =0 à nboucle-1 faire
                m : = debut+i*h+j;
                aux : =    Tab[m] + Tab[m+d];
                Tab[m] : = w*(Tab[m] - Tab[m+d]);
                Tab[m+d] : = aux;
            fait
            w : = w* (ω_n)^(n/h)      /* (ω_n)^(n/h) = ω_h, w = (ω_h)^j */
        fait
        prof : = prof-1;
        si prof<0
        alors nboucle : =nboucle*2; finsi
        sinon
            si (numero mod (P*h/n))≥ P*d/n
            alors debut : = debut +d; finsi
        finsi
        h : = h/2;
        d : = d/2;
    fait
finprocedure
```

Sur un hypercube à k dimension ($P=2^k$ processeurs), tant que l'on a pas réparti toutes les "sous-FFT" sur des processeurs différents, chaque processeur n'exécute qu'une seule fois la boucle "j" et c'est pourquoi la variable "nboucle" n'est pas modifiée tant que la répartition entre processeurs n'est pas entièrement faite, ce qui demande k passages dans la boucle "d". Ceci explique l'utilisation de la variable "prof" (pour "profondeur dans la récursivité") qui est initialisée à k et décrémentée à chaque passage dans la boucle "d".

Algorithme parallèle de Transformée de Fourier inverse par décimation en temps (entrées miroir, sorties normales).
Boucle "i" à l'intérieur.

```
procedure fftpinv(Tab,n) /* n = 2q */
    prof : = k-q  /* on a P=2k processeurs */
    debut : = numero*(n/P);
    h : = 2;
    d : = 1;
    nboucle : = n/P;
    tantque d<n faire
        si prof<0 alors nboucle : =nboucle/2; finsi
        prof : = prof+1
        w : =   1;
        pour j : = 0 à d-1 faire
            pour i : =0 à nboucle-1 faire
                m : = debut+i*h+j;
                aux : =    Tab[m] + Tab[m+d];
                Tab[m] : = w*(Tab[m] - Tab[m+d]);
                Tab[m+d] : = aux;
            fait
            w : = w* (ωn)n/h      /* (ωn)n/h = ωh, w = (ωh)j */
        fait
        si prof>=0
        alors
            comm(prof,1,0,debut,debut+d,1);
            si (numero mod (P*h/n))≥ P*d/n
            alors retour; finsi
        finsi
        h : = h*2;
        d : = d*2;
    fait
    pour i : =0 à n-1 faire Tab[i] : = tab[i]/n; fait
finprocedure
/*****************************************************/
```

• Le temps de calcul de la FFT est $T_{FFT} = \left(\frac{n}{2P}(\log(n)-\log(P)) + n - \frac{n}{2P}\right)T_{pap} + n.T_{mul}$

$= \frac{n}{2p}\left[(\log(n)-\log(P)+2P-1)T_{pap}+2P.T_{mul}\right]$.

On calcule deux FFT et on effectue ensuite $\frac{n}{P}$ produits terme à terme par processeur.

• Le temps de calcul de la FFT inverse est $T_{FFT} + \left(n - \frac{n}{P}\right)T_{com} + \log(P)T_{start}$,

ce qui nous donne un temps total en parallèle de :

$$T_{par} = \frac{3n}{2P}\left[(\log(n)-\log(P)+2P-1)T_{pap}+2\left(P+\frac{1}{3}\right)T_{mul}+ \frac{2}{3}(P-1)T_{com} + \frac{2P}{3n}\log(P)T_{start}\right].$$

• Le temps de calcul des deux FFT plus la FFT inverse plus les produits terme à terme en séquentiel est $T_{seq}= 3\left(\frac{n}{2}\log(n)T_{pap}+n\,T_{mul}\right)+n.T_{mul} = \frac{3n}{2}\left(\log(n)T_{pap}+\frac{8}{3}\,T_{mul}\right)$

On définit l'accélération comme le temps du meilleur algorithme séquentiel divisé par le temps de l'algorithme parallèle.

$$\text{accélération} = P.\ \frac{\log(n)T_{pap}+\frac{8}{3}\,T_{mul}}{(\log(n)-\log(P)+2P-1)T_{pap}+2\left(P+\frac{1}{3}\right)T_{mul}+ \frac{2}{3}(P-1)T_{com} + \frac{2P}{3n}\log(P)T_{start}}$$

On voit que même en négligeant les temps de communication, pour avoir une accélération proche de P, il faut que P soit petit devant log(n). Or pour les valeurs courantes de n et P, cette condition n'est pas réalisée. En tenant aussi compte des temps de communication, il faut que T_{com} soit petit devant $\log(n).T_{pap}$ pour que l'efficacité soit proche de 1. On peut aussi remarquer qu'il est possible de superposer les calculs et les communications. Cette version parallèle est dérivée du meilleur algorithme séquentiel et est asymptotiquement optimale, mais ici P=32 et l'accélération croît très lentement : en ne tenant pas compte des temps de communication en en posant $T_{pap}=T_{mul}$, pour avoir une accélération de 16 avec 32 processeurs il faut que n soit supérieur à 2^{150}!

II.6.B.a.2 Boucle "j" à l'intérieur

On cherche à ce que tous les processeurs aient toujours la même charge de travail, et qu'ils communiquent et calculent tous en même temps. Ici le vecteur est dès l'origine découpé et réparti entre les processeurs. Si on a $P=2^k$ processeurs, le vecteur est découpé en 2P blocs. Tous les processeurs travaillent en parallèle et effectuent les mêmes calculs mais chacun sur sa partie du vecteur. Ensuite, tous les processeurs communiquent en même temps. Les

communications se font en parallèle de voisin à voisin sur l'hypercube. A chaque étage, toutes les communications se font dans la même direction sur l'hypercube.

On remarque que maintenant, chaque processeur ne conserve dans sa mémoire propre qu'un $P^{ème}$ du vecteur total ce qui permet, avec la même machine, de réaliser des convolutions sur des vecteurs P fois plus longs.

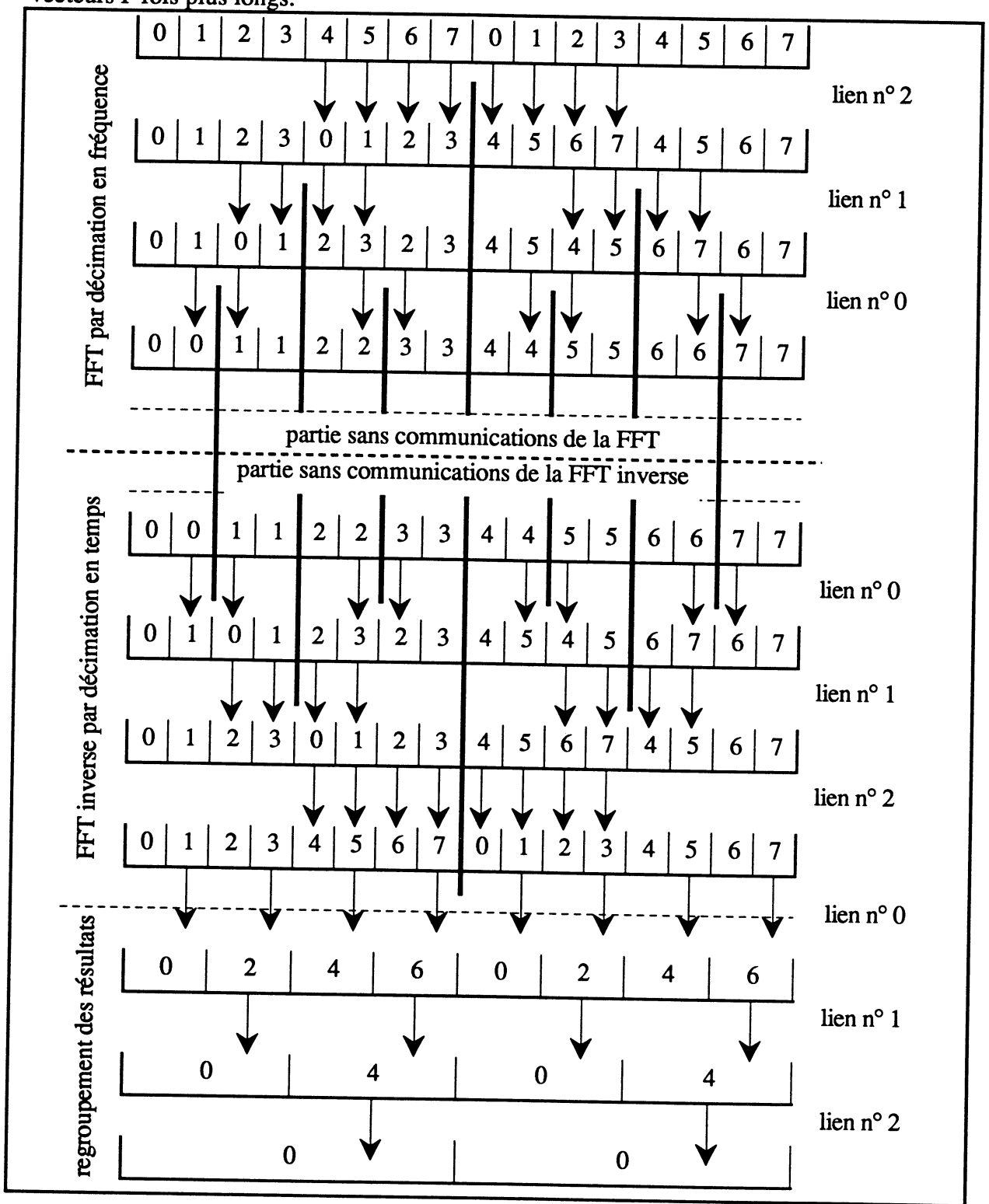


Figure 12 : répartition des données entre les processeurs et graphe de communication dans le cas du 3cube (p=8).

Algorithme parallèle de Transformée de Fourier discrète par décimation en fréquence (entrées normales, sorties miroir).
La partie sans communications est réalisée en mettant la boucle "i" à l'intérieur.
La partie avec communications est réalisée en mettant la boucle "j" à l'intérieur.

```
procedure fftpar(Tab,n)
    prof : = k; /* on a P = 2^k processeurs */
    debut : = numero*n/(2*P);
    h : = n;
    d : = n/2;
    h'= n/(2*P);
    tantque prof≥0 faire
        w : =  w(P*h/n)^numero;
        pour j : = 0 à h'-1 faire
            m : = debut+j;
            aux : =   Tab[m] + Tab[m+d];
            Tab[m] : = w*(Tab[m] - Tab[m+d]);
            Tab[m+d] : = aux;
            w : = w* (ωn)^n/h     /* (ωn)^n/h = ωh, w = (ωh)^j */
        fait
        h : = h/2;
        d : = d/2;
        prof : = prof-1;
        si prof≥0
        alors
            comm(prof,1,0,debut,debut+d,4*h');
            comm(prof,0,1,debut+1,debut+d,4*h');
            si (numero mod (P*h/n))≥ P*d/n
            alors debut : =debut+d; finsi
        finsi
    fait
    nboucle : = 2;
    tantque  d > 0 faire
        w : = 1
        pour j : =0 à d-1 faire
            pour i : = 0 à nboucle-1 faire
                m : = debut+i*h+j;
                aux : =   Tab[m] + Tab[m+d];
                Tab[m] : = w*(Tab[m] - Tab[m+d]);
                Tab[m+d] : = aux;
            fait
            w : = w* (ωn)^n/h     /* (ωn)^n/h = ωh, w = (ωh)^j */
        fait
        nboucle : = nboucle*2;
        h : = h/2;
        d : = d/2;
    fait
finprocedure
```

Algorithme parallèle de Transformée de Fourier inverse par décimation en temps (entrées miroir, sorties normales).
La partie sans communications est réalisée en mettant la boucle "i" à l'intérieur.
La partie avec communications est réalisée en mettant la boucle "j" à l'intérieur.

```
procedure fftpinv(Tab,n) /* n = 2q */
    prof : = k-q  /* on a P=2^k processeurs */
    debut : = numero*(n/P);
    h : = 2;
    d : = 1;
    h'= n/(2*P);
    nboucle : = n/P;
    tantque prof<0 faire
        nboucle : = nboucle/2;
        prof : = prof+1;
        w : =   1;
        pour j : = 0 à d-1 faire
            pour i : =0 à nboucle-1 faire
                m : = debut+i*h+j;
                aux : =    Tab[m] + Tab[m+d];
                Tab[m] : = w*(Tab[m] - Tab[m+d]);
                Tab[m+d] : = aux;
            fait
            w : = w* (ωn)^n/h      /* (ωn)^n/h = ωh, w = (ωh)^j */
        fait
        h : = h*2;
        d : = d*2;
    fait
    tantque d<n faire
        comm(prof,1,0,debut+d/2,debut-d/2,h');
        comm(prof,0,1,debut,debut+d,h');
        w : =   w(P*h/n)^numero;
        si (numero mod (p*h/n))≥ p*d/n
        alors debut : = debut-d/2; finsi
        pour j : = 0 à h'-1 faire
            m : = debut+j;
            aux : =    Tab[m] + Tab[m+d];
            Tab[m] : = w*(Tab[m] - Tab[m+d]);
            Tab[m+d] : = aux;
            w : = w* (ωn)^n/h      /* (ωn)^n/h = ωh, w = (ωh)^j */
        fait
        h : = h*2;
        d : = d*2;
    fait
    pour i : =0 à n-1 faire Tab[i] : = tab[i]/n; fait
finprocedure
/*****************************************************/
```

Dans la partie médiane, où tous les processeurs travaillent en parallèle sans aucune communication, on applique l'algorithme le plus performant en séquentiel (boucle i à l'intérieur).
Un autre avantage de cette méthode sur la précédente est que chaque processeur n'a besoin de garder dans sa mémoire locale qu'un $p^{ème}$ du vecteur, on peut donc traiter des problèmes de plus grande taille.

Ici les temps de la FFT et de la FFT inverse sont les mêmes :

$$T_{FFT} = \frac{n}{2P}(\log(n)T_{pap} + \frac{n}{2P}\log(p)T_{mul} + \frac{n}{P}T_{mul} + \frac{n}{P}\log(P)T_{com} + \log(P)\,T_{start}$$

Si on ne veut pas rapatrier tous les résultats sur un seul processeur, on a $T_{par}=3.T_{FFT}+\frac{n}{P}T_{mul}$,
soit $T_{par}=\frac{3n}{2P}\left[\log(n)T_{pap} + \left(\log(P)+\frac{8}{3}\right)T_{mul} + 2.\log(P)T_{com} + \frac{2P}{n}\log(P)T_{start}\right]$,
et donc l'accélération est :

$$P\,\frac{\log(n)T_{pap}+\frac{8}{3}T_{mul}}{\log(n)T_{pap} + \left(\log(P)+\frac{8}{3}\right)T_{mul} + 2.\log(P)T_{com} + \frac{2P}{n}\log(P)T_{start}}.$$

Il suffit que n soit grand devant P pour que l'accélération tende vers P.
Si on désire que tous les résultats soient rapatriés sur un processeur, on a alors

$$T'_{tot} = T_{tot} + \left(n - \frac{n}{P}\right)T_{com} + 2.\log(P)T_{start} = T_{tot} + \frac{n}{P}\left((P-1)T_{com} + \frac{2P}{n}.\log(P)T_{start}\right)$$
$$= T_{tot} + \frac{3n}{2P}\left(\frac{2}{3}(P-1)T_{com} + \frac{4P}{3n}.\log(P)T_{start}\right)$$

ce qui nous donne l'accélération suivante :

$$P\,\frac{\log(n)T_{pap}+\frac{8}{3}T_{mul}}{\log(n)T_{pap} + \left(\log(P)+\frac{8}{3}\right)T_{mul} + \left(\frac{2}{3}(P-1)+2.\log(P)\right)T_{com} + \frac{10.P}{3.n}\log(P)T_{start}}.$$

Il faut maintenant que $\log(n).T_{pap}$ soit grand devant $P.T_{com}$ pour que l'accélération tende vers P.
Cette deuxième méthode est très largement supérieure à la première, ce qu'ont confirmé les expérimentations.

II.6.C résultats pratiques

On dispose d'une machine à 32 processeurs, utilisée dans sa configuration maximale. On a été limité dans nos tests par la place mémoire disponible sur chaque processeur (1 Mo à partager entre code et données).

Il est possible -comme on l'a déjà remarqué- de réaliser des convolutions 32 fois plus grandes, avec la deuxième implantation (boucle "j" à l'intérieur), chaque processeur n'ayant à connaître à chaque instant que sa partie des données. On ne l'a pas fait car les résultats correspondant s'interpolent facilement et ne présentent pas d'intérêt pratique (on ne ne dispose pas de références pour les comparaisons)

II.6.C.a Temps

Les courbes de la figure 13 indiquent les temps de calcul d'un produit de convolution (deux FFT et une FFT inverse) pour l'implantation séquentielle et les deux implantations parallèles présentées.

Comme on s'y attendait, la deuxième implantation (boucle "j" à l'intérieur) a donné de bien meilleurs résultats que la première. On obtient un temps de 5s pour une convolution de 2^{17} termes, ce qui correspond à un produit de deux nombres de 2^{16} chiffres en base 2^8 soit deux nombres de $2^{19} = 524288$ bits.

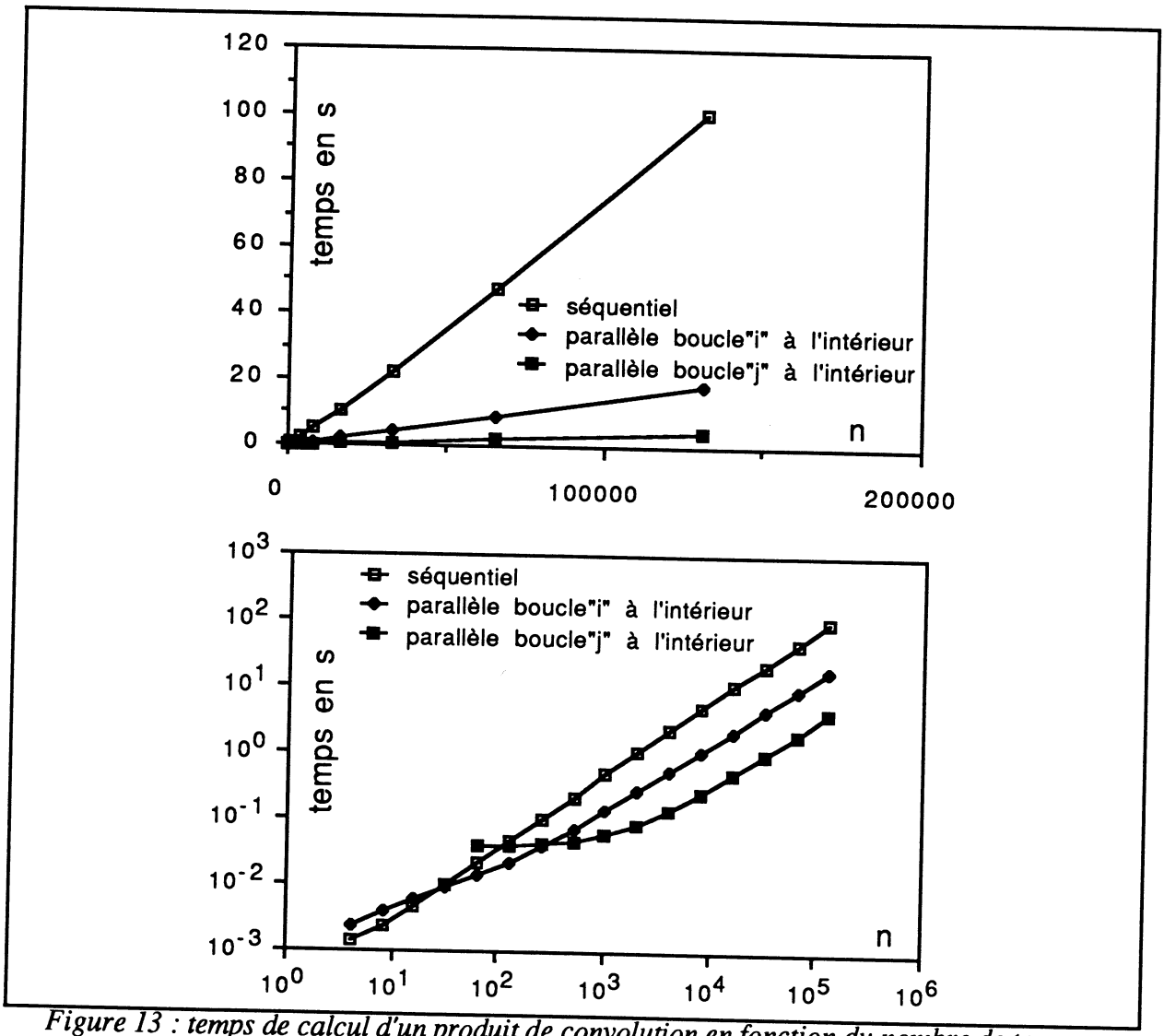


Figure 13 : temps de calcul d'un produit de convolution en fonction du nombre de termes.

II.6.C.b accélération

On avait un nombre donné de processeurs (32), on s'est limité à calculer l'accélération en fonction du nombre de termes du produit de convolution. Le fait que les temps de calculs soient grands devant les temps de communication a permis d'obtenir de très bonnes accélérations pour la deuxième implantation parallèle : l'accélération atteint 25 avec la deuxième méthode, contre 6 avec la première méthode.

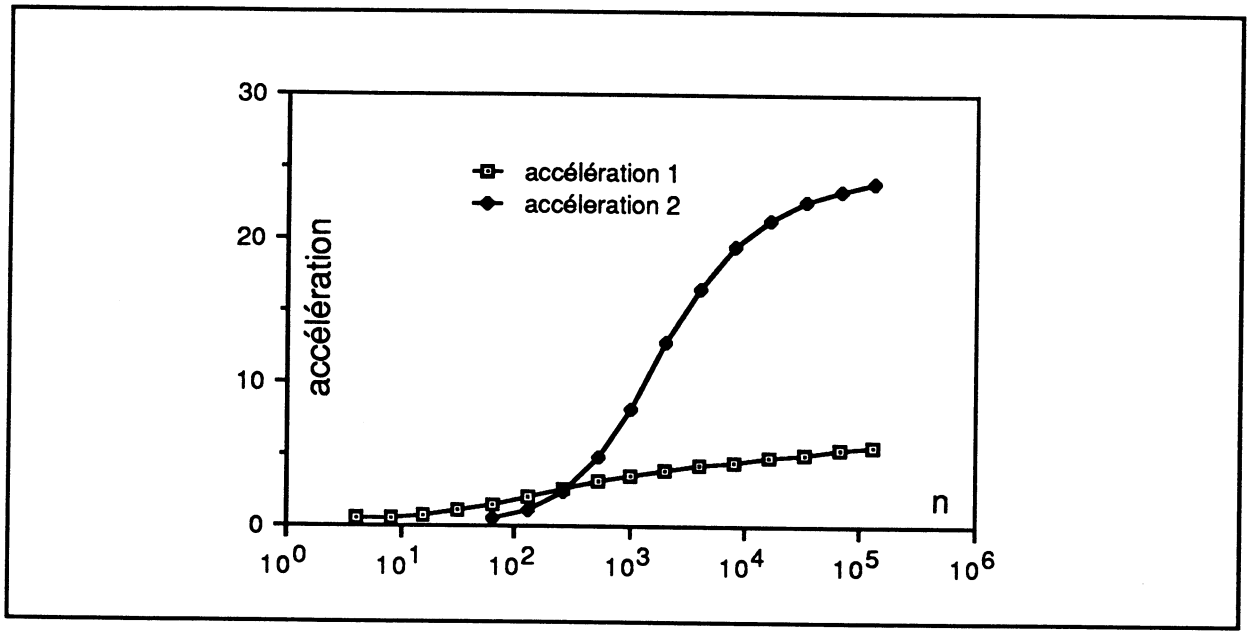


Figure 14 : accélération en fonction du nombre de termes (pour 32 processeurs).

II.7 Une architecture pour l'implantation de l'algorithme de Pollard

Cet algorithme présente l'avantage de pouvoir facilement être réalisé par voie matérielle. On suppose qu'on utilise un seul nombre premier p. Si l'on veut réaliser une unité qui fonctionne comme coprocesseur d'une unité centrale, il faut :

• Une unité de répartition qui découpe les opérandes de façon optimale en fonction de la base dans laquelle ils sont écrits (une puissance de 2 ou de 10 en pratique) et de leur taille. On peut tabuler les valeurs des découpages optimaux afin d'accélérer le traitement. Ce découpage se fait en série quel que soit le sens d'introduction des données.

• Trois (ou deux) bancs mémoire pour ranger les opérandes traités par l'unité précédente et le résultat.

• Une ou plusieurs unités arithmétiques spéciales qui travaillent modulo p

• Une ou plusieurs unités d'adressage et un microprogramme pour réaliser la FFT. Si l'on examine les algorithmes de FFT détaillés précédemment, on se rend compte que l'unité d'adressage et le microprogramme sont simples à réaliser, il suffit de deux ou trois compteurs. La même unité d'adressage peut piloter plusieurs FFT en parallèle.

• Une unité qui réalise la propagation finale de retenue et remet le résultat au format voulu. Cela peut être réalisé par tout microprocesseur dont le bus de données est assez large pour représenter p.

II.7.A Unité d'adressage

Revoyons l'algorithme FFT tel qu'on l'a décrit précédemment : on a trois boucles imbriquées

• une boucle principale sur la variable "d" qui est liée avec les deux variables "h" et "nboucles" par les relations h=2d et nboucle*h=n où n est la taille de la FFT à réaliser. Toutes

ces variables sont toujours égales à des puissances de 2. On passe log(n) fois dans cette boucle principale.

- deux boucles imbriquées qui correspondent à un étage dans le schéma de la FFT

  sur "i" variant entre 0 et nboucle-1

  sur "j" variant entre 0 et d-1

  à l'intérieur de ces deux boucles, on réalise un produit "papillon" entre les éléments d'indice i*l+j et i*l+j+d.

Voyons comment sont écrits en base 2 les valeurs i*h+j et i*h+j+d

$n=2^{\nu}$, $h=2^{\eta}$, $nboucle=2^{\nu-\eta}$, $i \in \{0, \ldots, 2^{\nu-\eta}-1\}$, $j \in \{0, \ldots, 2^{\eta-1}-1\}$

pour réaliser les deux boucles internes on peut soit utiliser deux compteurs (un pour "j" et un pour "i") soit un seul compteur qui donne un chiffre sur ν-1 bits dans lequel on intercale un 0 ou un 1 à la η-1ème position (ce qui correspond à mettre la boucle "j" à l'intérieur). La boucle "d" est réalisée en faisant varier η (la position où on incorpore le 0 ou le 1).

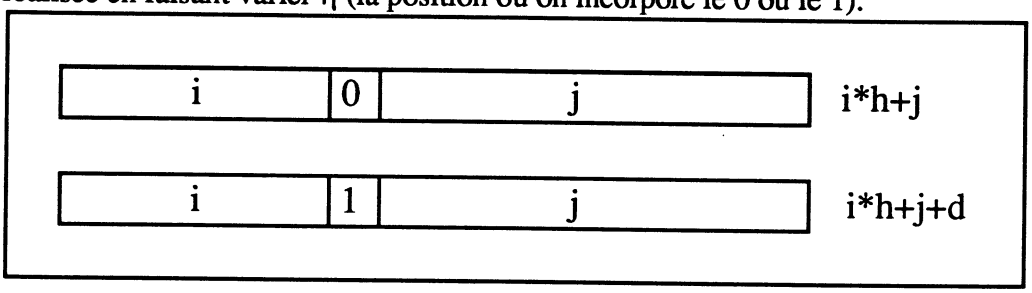


*Figure 15 : écriture binaire des indices i*h+j et i*h+j+d.*

Si on effectue une décimation en temps, η varie de 0 à ν-1, et si on effectue une décimation en fréquence, η varie de ν-1 à 0.

II.7.B Réalisation matérielle d'un multiplieur modulo p

On a $2^{q-1} \leq p < 2^{q}$. On suppose que l'on connait les valeurs $2^k \bmod (p)$ pour $q \leq k \leq 2q-1$.

Alors, si $A.B=C=\sum_{i=0}^{2q-1} c_i.2^i$, on a $A.B \bmod (p) = \left(\sum_{i=0}^{q-1} c_i.2^i + \sum_{i=q}^{2q-1} c_i.(2^i \bmod (p)) \right) \bmod (p)$

On se retrouve avec q+1 termes à sommer ce qui peut se faire en un temps logarithmique et nous ramène d'un nombre de l'ordre de p^2 a un nombre de l'ordre de $p.\log_2(p)$. On peut réappliquer ce procédé jusqu'à ce que l'on obtienne bien A.B mod (p). Le temps total pour la multiplication modulo p sera donc à peu près le double de celui demandé pour une simple multiplication entière.

II.7.C Proposition d'architecture

On propose une architecture "maximale" (voir la figure 16) avec trois bancs mémoire, trois unités arithmétique modulo p et deux unité d'adressage et de contrôle de FFT. Cette disposition permet de chaîner les opérations.

Remarque : si on utilise plusieurs nombres premiers, il suffit de répliquer la même architecture autant de fois qu'il y a de nombres premiers et d'ajouter une unité qui effectue l'algorithme des restes chinois avant la propagation de retenue.

II.7.D Ressource bloquante.

Il apparait donc que la ressource bloquante pour cet algorithme n'est pas l'unité arithmétique modulo p mais simplement le bus de données. En effet chaque produit papillon demande 4 accès mémoire (2 en lecture et 2 en écriture).
Si on suppose que l'on effectue un produit papillon modulo p en moins de 4 cycles ou que l'on peut faire ces produits en pipeline, et si on effectue un et un seul accès mémoire (lecture ou écriture) par cycle, on peut alors faire recouvrir les calculs et les accès mémoire, et réaliser le produit de deux nombres de 2^q chiffres en exactement $(12q+16).2^q$ cycles en tenant compte de le propagation de retenue finale, des conversions initiales et finales et du fait que les NTT sont de longueur 2^{q+1}.

Par exemple si $p=3.2^{30}+1$ et si la fréquence est de 20 MHz on peut multiplier 2 nombres de 10^6 (2^{20}) chiffres décimaux ou hexadécimaux en à peu près 12s avec une occupation mémoire totale de 6 Mmots soit 24 Moctets.

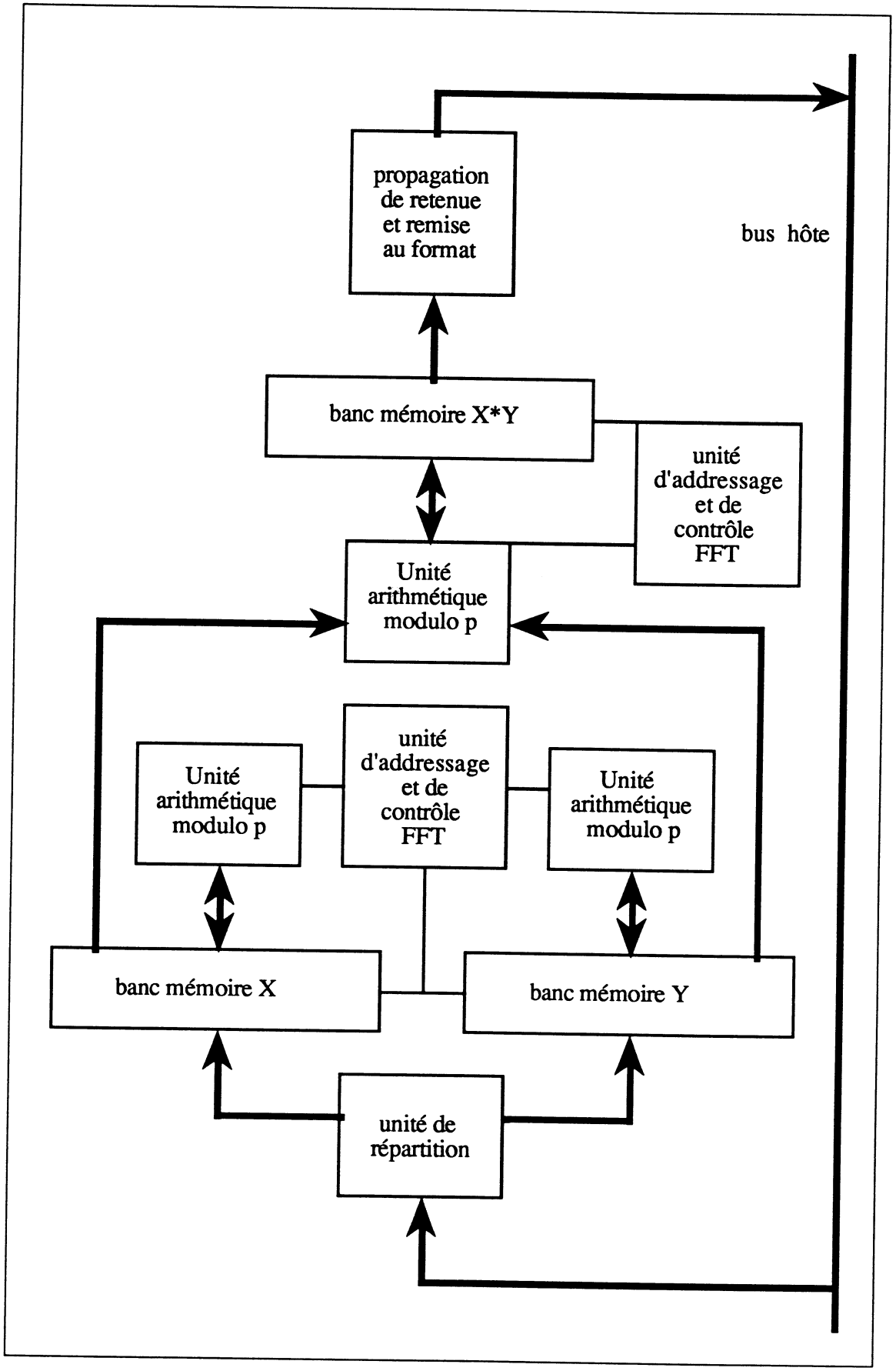


Figure 16 : proposition d'architecture maximale.

III. Calculs *en-ligne*

III.1 Résumé

On présente quelques résultats théoriques concernant les algorithmes de calcul *en ligne* :bornes inférieures et supérieures de délais et propriétés des fonctions calculables par de tels algorithmes. Une conséquence de ces résultats est que toute fonction monotone et de classe C1 sur [-1, 1] est calculable en ligne.

III.2 Introduction

La transmission des données en série permet de simplifier le *layout* de nombreux circuits (en traitement du signal par exemple, [PRI86]), et d'améliorer les performances en communication et calcul, car elle autorise le chaînage des opérations au niveau du chiffre. On peut transmettre en premier les chiffres de poids fort (mode MSB, de l'anglais *Most Significant Bit)* ou de poids faible (mode LSB, de l'anglais *Least Significant Bit*). La transmission LSB peut sembler la plus naturelle car dans toutes les opérations arithmétiques, la transmission des retenues se fait des poids faibles vers les poids forts. Cependant, des opérations comme la division sont impossibles à effectuer en série en mode LSB : en effet on ne peut pas connaître le chiffre de poids faible d'un quotient tant que l'on ne connait pas tous les chiffres du dividende et du diviseur. On peut aussi remarquer qu'en général, lorsqu'on fait de gros calculs, on tronque les résultats à une précision donnée et que ce sont bien entendu les chiffres de poids faible que l'on tronque. Une transmission LSB des chiffres n'a dans ce cas aucun intérêt car on sort d'abord les chiffres que l'on ne désire pas conserver.

Ici, nous allons étudier quelques aspects du mode bit série MSB, aussi appelé mode *On-line* ([IRW77], [ET77], ...) que nous traduirons par mode *"en ligne"*. Comme les retenues se propagent toujours des poids faibles vers les poids forts, les nombres doivent être écrits dans un système de numération qui limite la propagation des retenues. On utilise donc les systèmes d'Avizienis présentés au premier chapitre, avec un ensemble ensemble de chiffres symétrique autour de 0. Ceci nous permet de traiter indifféremment des nombres négatifs ou positifs.

En 1977, Ercegovac et Trivedi ont présenté un algorithme de division en ligne, basé sur l'algorithme de division non restaurante, et un algorithme de multiplication en ligne. Depuis de nombreux algorithmes en ligne ont été découverts (un état de l'art en 1984 est donné par Ercegovac dans [ER84] et des algorithmes récents peuvent être trouvés dans [EL85], [LS87], [EL87b], [ET87] et [DM89]). Les opérateurs en ligne sont caractérisés par un *délai* (le nombre δ de chiffres des opérandes que l'opérateur doit avoir reçu pour pouvoir faire sortir le premier

chiffre du résultat) et une *période* (le temps τ qui s'écoule entre l'arrivée de deux chiffres consécutifs). La période dépend de la technologie, de l'algorithme ainsi que de son implantation, alors que le délai ne dépend que de l'algorithme utilisé. Le résultat d'une opération sur p chiffres avec un délai δ et une période τ est obtenu en un temps τ $(p+\delta)$. Il est possible d'enchaîner des opérations en ligne, le délai global étant la somme des délais des différents opérateurs (en supposant que la période est la même). Dans l'exemple de la figure 1 le délai total est de $2\delta_{mult} + \delta_{add} + \delta_{cos}$, donc p chiffres du résultat peuvent être calculés en un temps $(p + 2\delta_{mult} + \delta_{add} + \delta_{cos})\tau$.

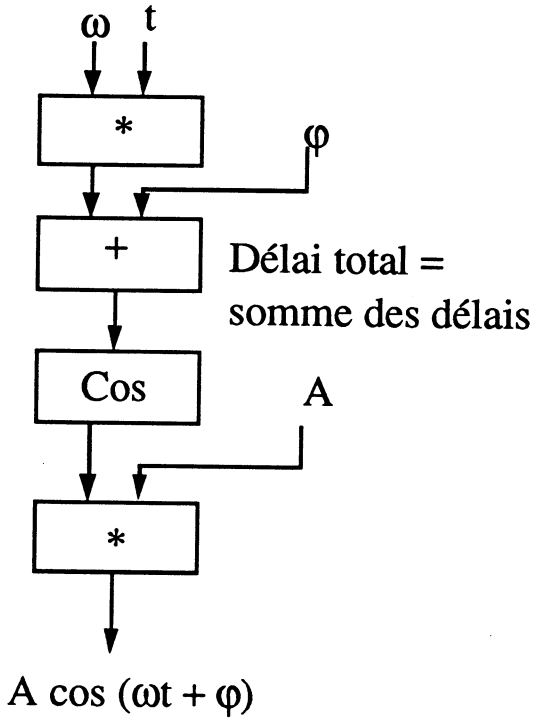


Figure 1 : chaînage au niveau du chiffre

Nous allons nous intéresser aux délais en en donnant des bornes inférieures pour le calcul de fonctions à dérivées continues (ces bornes inférieures sont intéressantes car elles sont atteintes pour certaines opérations arithmétiques). Pour les fonctions monotones d'une variable, on donne une borne supérieure pour le plus petit délai (d'intérêt purement théorique car l'algorithme correspondant à ces bornes demande de pouvoir calculer la fonction inverse en temps constant).

III.3 Système à chiffres signés : rappels et notations

Pour pouvoir réaliser des additions en mode série MSB, il faut pouvoir s'affranchir des propagations de retenue dans les additions. On utilise donc les systèmes d'Avizienis qui autorisent des additions en temps constant. Dans tout ce chapitre, les nombres sont écrits en base B avec les chiffres dans {-a, ..., a}. Si B est égal à 2, alors a=1, sinon on choisit a

vérifiant les deux conditions vues au premier chapitre, à savoir $a \leq B-1$ et $2.a \geq B+1$. On saura donc toujours effectuer des additions en temps constant dans ces systèmes de représentation.

Dans toute la suite de ce chapitre, si on connait une écriture d'un nombre x de [-1, 1] sous la forme $x=\sum_{i=1}^{\infty} x_i B^{-i}$ on notera $x^{(p)}=\sum_{i=1}^{p} x_i B^{-i}$.

Il faut bien remarquer que si on connait deux écritures de x, $x=\sum_{i=1}^{\infty} x_i B^{-i}$ et $x=\sum_{i=1}^{\infty} x'_i B^{-i}$, comme les systèmes utilisés sont redondants, on n'a pas en général égalité entre $x^{(p)}$ et $x'^{(p)}$ où $x^{(p)}=\sum_{i=1}^{p} x_i B^{-i}$ et $x'^{(p)}=\sum_{i=1}^{p} x'_i B^{-i}$.

III.4 Résultats théoriques

III.4.A Bornes générales pour les délais *en ligne*

On calcule des fonctions de l'intervalle [-1, 1] dans l'intervalle [-1, 1]. Les définitions suivantes sont les mêmes que celles d' Ercegovac et Trivedi [ET77].

Définition 1. Soit f une fonction de [-1, 1] dans [-1, 1]. Un algorithme pour calculer f a un délai δ s'il calcule y_p à l'aide uniquement de $x_1, x_2, \ldots, x_{p+\delta}, y_1, y_2, \ldots, y_{p-1}$, où $x = 0.x_1x_2x_3\ldots$ et $y = 0.y_1y_2y_3\ldots$ sont écrits en chiffres signés, et $y = f(x)$. Une fonction f est calculable en ligne s'il existe un algorithme avec un délai δ *fini et indépendant de la longueur des opérandes* pour calculer f.

Nous allons maintenant nous intéresser aux propriétés plus intrinsèques des fonctions, on va pour cela définir les *délais des fonctions*.

Définition 2. Le *délai absolu* δ_{abs} (f) d'une fonction f est la plus petite valeur de δ telle que pour toute valeur de p et pour tout $x \in [-1, 1]$, les p premiers chiffres d'une représentation redondante de f(x) puissent être déduits des $p+\delta$ premiers chiffres de toute représentation redondante de x.

La redondance des représentations en chiffres signés pose un problème : supposons qu'un algorithme donne pour une valeur donnée de p, les p premiers chiffres $y_1, y_2, \ldots, y_p$ d'une représentation de f(x) à partir des $p+\delta$ premiers chiffres d'une représentation de x. Si cet algorithme est utilisé pour calculer les p+1 premiers chiffres $y'_1, y'_2, \ldots, y'_{p+1}$ de f(x), on n'a

pas nécessairement $y'_i = y_i$ (ni $y'^{(i)}=y^{(i)}$) pour tout i. La définition qui suit tient compte de ce problème.

Définition 3. Le *délai pratique* δ_{pra} (f) d'une fonction f est la plus petite valeur de δ telle que y_p puisse être déduit de $x_1, x_2, \ldots, x_{p+\delta}, y_1, y_2, \ldots, y_{p-1}$, où $x = 0.x_1x_2x_3\ldots$ et $y = 0.y_1y_2y_3\ldots$ sont écrits sous forme redondante, et $y = f(x)$. C'est le plus petit délai que puisse avoir un algorithme qui calcule f.

Cette définition est plus pratique pour l'étude des algorithmes en ligne que la définition 2, mais malheureusement, elle donne un délai qui semble plus difficile à évaluer. On verra plus tard qu'en fait, ces deux délais sont égaux.

Théorème 1. Une fonction calculable en ligne est continue sur]-1, 1[.

Démonstration (en base 2).
Supposons que f est calculable en ligne avec un délai δ. Considérons un nombre $x \in]-1, 1[$, et soit $y = f(x)$. Montrons que f est continue au voisinage de x. Observons d'abord qu'il existe une représentation binaire redondante de x, $0.x_1x_2x_3\ldots$, qui ne se termine pas par une infinité de 1 (ou de $\bar{1}$), par exemple la représentation binaire non redondante de x.
Pour un entier donné p, considérons les nombres suivants :

$$A = 0.x_1x_2x_3\ldots x_{p+\delta}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\ldots$$
$$B = 0.x_1x_2x_3\ldots x_{p+\delta}11111111\ldots$$

Tous les nombres de l'intervalle [A, B] admettront une écriture qui commence par $0.x_1x_2x_3\ldots x_{p+\delta}$,. Comme f est calculable en ligne avec un délai δ, si $y_1, y_2, \ldots, y_p$ sont les premiers chiffres de l'écriture de $f(0.x_1x_2x_3\ldots x_{p+\delta})$ obtenus par un algorithme en ligne de délai δ, ce même algorithme donnera les mêmes premiers chiffres pour tous les nombres f(s) où $s \in [A, B]$ si on lui fournit en entrée l'écriture de s commençant par $x_1, x_2, \ldots, x_{p+\delta}$. Donc pour tout $s \in [A, B]$, $| f(s) - f(x) | \leq 2^{-p+1}$. On en déduit :

$\forall \varepsilon > 0 \ \exists \alpha$ (p est choisi afin que $2^{-p+1} < \varepsilon$, $\alpha = \min \{|B-x|, |A-x|\}$) :
$|x-s| < \alpha \Rightarrow |f(x)-f(s)| < \varepsilon$.
Donc f est continue en x et le Théorème 1 est démontré. ◊

Ce théorème n'est pas vrai en écriture non redondante. Considérons l'exemple suivant (le nombre $0.a_1a_2a_3\ldots$est écrit en écriture binaire non redondante) :

$$f(0.a_1a_2a_3\ldots) = 0.a_10a_20a_30\ldots$$

Cette fonction n'est pas continue mais est calculable en ligne.

III.4.A.1. Lien entre délai pratique et délai absolu

Nous avons vu que le délai pratique correspond plus à la notion de calcul en ligne, mais que le délai absolu est plus facile à borner théoriquement. Nous allons maintenant montrer que ces deux valeurs sont identiques, et que si on connait le délai absolu d'une fonction, on sait réaliser (du moins théoriquement) un algorithme calculant cette fonction en ligne avec un délai égal au délai absolu.

Théorème 2.
Si f est calculable en ligne, alors $\delta_{abs}(f)$ et $\delta_{pra}(f)$ sont égaux.

Par la suite, on notera cette valeur $\delta(f)$.

Démonstration.
On travaille en base B avec les chiffres dans {-a, ..., a}.

Il est évident que $\delta_{abs}(f)$ est inférieur ou égal à $\delta_{pra}(f)$. Montrons que $\delta_{abs}(f) \geq \delta_{pra}(f)$. Notons $\delta = \delta_{abs}(f)$. Pour tout x, on peut déduire les p premiers chiffres d'une représentation redondante $y'^{(p)}$ de y = f(x) à partir des $p+\delta$ premiers chiffres d'une représentation de x. On peut proposer l'algorithme suivant de délai δ pour calculer en-ligne les chiffres y_i d'une représentation de y:

Supposons que l'on ait déjà calculé $y^{(p)}=0.y_1y_2 \ldots y_p$ à partir de $x^{(p+\delta)}=0.x_1x_2 \ldots x_{p+\delta}$. On déduit p+1 chiffres $y'^{(p+1)}=0.y_1'y_2'\ldots y_{p+1}'$ d'une représentation de y à partir de $x^{(p+\delta+1)}=0.x_1x_2 \ldots x_{p+\delta+1}$.

On a alors les relations suivantes :

$$y^{(p)} - \sum_{i>p} a.B^{-i} = y^{(p)} - \frac{a.B^{-p}}{B-1} \leq y \leq y^{(p)} + \frac{a.B^{-p}}{B-1}, \text{ et}$$

$$y'^{(p+1)} - \frac{a.B^{-p-1}}{B-1} \leq y \leq y'^{(p+1)} + \frac{a.B^{-p-1}}{B-1}.$$

On désire calculer y_{p+1} tel que $y^{(p+1)} - \frac{a.B^{-p-1}}{B-1} \leq y \leq y^{(p+1)} + \frac{a.B^{-p-1}}{B-1}$.

On a les trois cas suivants :

- $y'^{(p+1)} \geq y^{(p)} + a.B^{-p-1}$.

Cela entraîne $y \geq y^{(p)}+a.B^{-p-1} - \frac{a.B^{-p-1}}{B-1}$.

D'autre part, on sait que $y \leq y^{(p)}+ \frac{a.B^{-p}}{B-1} = y^{(p)}+a.B^{-p-1} + \frac{a.B^{-p-1}}{B-1}$. On a donc l'encadrement suivant : $y^{(p)}+a.B^{-p-1} - \frac{a.B^{-p-1}}{B-1} \leq y \leq y^{(p)}+a.B^{-p-1} + \frac{a.B^{-p-1}}{B-1}$ et on peut donc choisir $y_{p+1}=a$.

- $y'^{(p+1)} \leq y^{(p)}-a.B^{-p-1}$

Par symétrie, on peut choisir $y^{(p+1)} = -a$.

- $y^{(p)}- a.B^{-p-1} < y'^{(p+1)} < y^{(p)}+ a.B^{-p-1}$.

$y'^{(p+1)}$, et $y^{(p)}$ ne peuvent prendre que des valeurs discrètes. Donc $y'^{(p+1)}$ est de la forme $y^{(p)}+\alpha.B^{-p-1}$ où α est entier entre -a+1 et a-1.

D'autre part, comme a est strictement inférieur à B, $y'^{(p)}=0$, $y_1'y_2'\ldots y_p'$ ne peut prendre que l'une des trois valeurs suivantes : $y^{(p)}-B^{-p}$, $y^{(p)}$ et $y^{(p)}+B^{-p}$.

Si $y'^{(p)} = y^{(p)}-B^{-p}$, on peut prendre $y_{p+1} = y'_{p+1}-B$.
en effet, on a $y'^{(p)} = y^{(p)}-B^{-p} < y^{(p)}- a.B^{-p-1} < y'^{(p+1)}=y'^{(p)}+y'_{p+1}.B^{-p-1}$,
donc $B-a<y'_{p+1}$ et donc $-a< y'_{p+1}-B$.
Si $y'^{(p)} = y^{(p)}$, on peut prendre $y_{p+1} = y'_{p+1}$.
Si $y'^{(p)} = y^{(p)}-B^{-p}$, on peut prendre $y_{p+1} = y'_{p+1}+B$.

En résumé, si $y^{(p)}- a.B^{-p-1} < y'^{(p+1)} < y^{(p)}+ a.B^{-p-1}$, on peut toujours choisir y_{p+1} tel que $y'^{(p+1)}$ soit égal à $y^{(p+1)}$, et donc $y^{(p+1)}$ est bien le début d'une écriture de y.

Donc si les p premiers chiffres d'une représentation redondante de f(x) peuvent être déduits des $p+\delta$ premiers chiffres de toute représentation redondante de x, on peut en déduire un algorithme de calcul de f de délai δ, et donc $\delta_{abs}(f) \geq \delta_{pra}(f)$.

III.4.A.2. Borne inférieure sur le délai en fonction de la dérivée

Il semble logique que le délai d'une fonction variant rapidement soit plus grand que celui d'une fonction variant lentement, les cas limites sont ceux des fonctions constantes qui se calculent avec un délai nul et des fonction non continues qu'on ne peut pas calculer en ligne, comme on l'a vu précédemment. On va donc essayer de relier le délai d'une fonction à la valeur de sa dérivée. Les trois théorèmes qui suivent répondent à ce problème dans le cas de fonctions d'une ou plusieurs variables.

Théorème 3.

Si une fonction f calculable en ligne a une dérivée continue, alors $\lceil \log_B \underset{[-1, 1]}{\text{Max}} |f'| \rceil \leq \delta(f)$.

Démonstration.

Supposons que l'on veuille calculer f avec un délai $\delta < \lceil \log_B \underset{[-1, 1]}{\text{Max}} |f'| \rceil$, en base B, avec les chiffres dans{-a, -a+1, ..., a-1, +a}. Comme δ est entier, on a : $\delta < \log_B (\max |f'|)$, donc $\max |f'| > B^{\delta}$.

Comme f ' est continue, il existe un intervalle I tel que pour tout x de I, $|f'(x)| > B^{\delta}$. Il existe p chiffres $x_1, x_2, x_3, \ldots, x_{p+d}$ tel que α et β soient dans I, où :

- $\alpha = 0.x_1x_2\ldots x_{p+\delta}0\bar{a}\bar{a}\bar{a}\bar{a}\bar{a}\ldots = 0.x_1x_2x_3\ldots x_{p+\delta} - a.\, B^{-\delta} \sum_{i=p+2}^{\infty} B^{-i}$
- $\beta = 0.x_1x_2\ldots x_{p+\delta}0aaaaa\ldots = 0.x_1x_2x_3\ldots x_{p+\delta} + a.\, B^{-\delta} \sum_{i=p+2}^{\infty} B^{-i}$

$f(\beta) - f(\alpha)$ vaut $(\beta-\alpha)f'(s)$, où $s \in [\alpha, \beta]$, donc :

$$|f(\beta) - f(\alpha)| > \left[2aB^{-\delta} \sum_{i=p+2}^{\infty} B^{-i} \right] r^{\delta} = 2a \sum_{i=p+2}^{\infty} B^{-i}$$

Donc pour tout $y_1, y_2, \ldots, y_p, y_{p+1}$:

$$0.y_1y_2\ldots y_{p+1}aaaaa\ldots - 0.y_1y_2\ldots y_{p+1}\bar{a}\bar{a}\bar{a}\bar{a}\bar{a}\ldots = 2a \sum_{i=p+2}^{\infty} B^{-i}.$$

On en déduit que l'information "$x_{p+\delta+1} = 0$" (i.e. "$x \in [\alpha, \beta]$") n'est pas suffisante pour calculer les chiffres $y_1, y_2, \ldots, y_p, y_{p+1}$ de f(x), et que l'on ne peut pas calculer f avec un délai δ si $\delta < \lceil \log_B \underset{[-1, 1]}{\text{Max}} |f'| \rceil$.

Théorème 4.

Si f a une dérivée continue et est monotone, alors $\delta(f) \leq \lceil \log_B \underset{[-1, 1]}{\text{Max}} |f'| \rceil + 1$.

Démonstration.

comme précédemment, on note $d = \lceil \log_B \underset{[-1, 1]}{\text{Max}} |f'| \rceil + 1$, on a donc $\underset{[-1, 1]}{\text{Max}} |f'| \leq B^{d-1}$.

On suppose encore que l'on a déjà calculé les p-1 premiers chiffres $0.y_1y_2y_3\ldots y_{p-1}$ d'une représentation de f(x). Les chiffres $0.x_1x_2x_3\ldots x_{p+d}$ de x sont connus. On suppose encore que f est croissante (la démonstration est similaire si f est décroissante)

Appelons I l'intervalle $\left[x^{(p+d)} - \frac{a.B^{-p-d}}{B-1}, x^{(p+d)} + \frac{a.B^{-p-d}}{B-1}\right]$ dans lequel on est sûr que x se trouve. On a $y \in \left[y^{(p-1)} - \frac{a.B^{-p+1}}{B-1}, y^{(p-1)} + \frac{a.B^{-p+1}}{B-1}\right]$.

Evaluons $h = f(x^{(p+d)} + \frac{a.B^{-p-d}}{B-1}) - f(x^{(p+d)} - \frac{a.B^{-p-d}}{B-1})$. h est positif car f est croissante (h est la longueur de l'intervalle f(I)).
Comme $\underset{[-1,\,1]}{\text{Max}} |f'| \leq B^{d-1}$, on obtient $0 \leq h \leq \frac{2.a.B^{-p-1}}{B-1}$.

Soit α dans $\{-a,.., a-1\}$ on pose $y^+ = y^{(p-1)} + \alpha B^{-p} + \frac{a.B^{-p}}{B-1}$ la plus grande valeur que peut atteindre y si son $p^{ème}$ chiffre est α et $y^- = y^{(p-1)} + (\alpha+1)B^{-p} - \frac{a.B^{-p}}{B-1}$ la plus petite valeur que peut atteindre y si son $p^{ème}$ chiffre est $\alpha+1$.
Pour tout α, on a $y^+ > y^-$ car $y^+ - y^- = (\frac{2a}{B-1} - 1)B^{-p} > 0$, car par hypothèse, si B=2 alors a=1 et sinon $2a \geq B+1$. On remarque que $y^+ - y^-$ est indépendant de α. C'est en fait la longueur le l'intervalle sur lequel on peut choisir indifféremment α ou $\alpha+1$ comme $p^{ème}$ chiffre.
Si h est inférieur à $y^+ - y^-$ alors on est sûr que I n'est pas à cheval sur une zone où on est obligé de choisir α comme $p^{ème}$ chiffre et une autre zone où on est obligé de choisir $\alpha+1$ comme $p^{ème}$ chiffre. On peut donc alors trouver un α dans $\{-a,.., a\}$ tel qu'on soit sûr que l'intervalle I soit inclus dans l'intervalle $\left[y^{(p-1)} + \alpha B^{-p} - \frac{a.B^{-p}}{B-1}, y^{(p-1)} + \alpha B^{-p} + \frac{a.B^{-p}}{B-1}\right]$.

On a bien $h \leq y^+ - y^-$ car $\frac{2.a.B^{-p-1}}{B-1} \leq (\frac{2a}{B-1} - 1)B^{-p}$. En effet, $\frac{2a}{B} \leq 2a-B+1$, car
- si B>2, on a $\frac{2a}{B} \leq 2 \leq 2a-B+1$ ($a \leq B-1$ et $2a \geq B+1$ par hypothèse).
- si B=2, alors a=1 et $\frac{2a}{B} = 1 = 2a-B+1$.

Donc on peut bien trouver α tel que y soit dans $\left[y^{(p-1)} + \alpha B^{-p} - \frac{a.B^{-p}}{B-1}, y^{(p-1)} + \alpha B^{-p} + \frac{a.B^{-p}}{B-1}\right]$, et si on prend cet α comme valeur de y_p, alors $y_1, y_2, \ldots, y_p$ sont les p premiers chiffres d'une écriture de y=f(x).

Le théorème est donc démontré.

Il est intéressant de remarquer que si f n'a pas de dérivée continue, il peut être impossible de calculer f en ligne, même si f est continue. $f(x) = \sqrt{x}$ en est un exemple, car le $p^{ème}$ chiffre de la racine 0.00...01... (p-1 zéros) de x = 0.0000...01 (2p-1 zéros) dépend du $2p^{ème}$ chiffre de

x. Cependant, en virgule flottante, $\sqrt{x}$ devient calculable. Un autre exemple est la fonction de Peano suivante :

$f(0.x_1x_2x_3x_4x_5\ldots) = 0.x_1x_3x_5x_7x_9\ldots$ où les nombres sont écrits sous forme binaire non redondante. Cette fonction est continue mais n'a pas de dérivée et n'est pas calculable en ligne car le $p^{ème}$ chiffre du résultat dépend du $2p^{ème}$ chiffre de l'entrée.

Nous allons étudier rapidement le cas des fonctions de 2 variables. Les résultats suivants peuvent être facilement étendus au cas de n variables.

Théorème 5.

Soit f(x, y) une fonction de 2 variables calculable en ligne :

- $\delta_{abs}(f) = \delta_{pra}(f) = \delta(f)$
- Si $\partial f/\partial x$ et $\partial f/\partial y$ existent et sont continues, alors

$$\delta(f) \geq \left\lceil \log_B \text{Max}\left(\left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right|\right)\right\rceil$$

Démonstration.

- $\delta_{abs}(f) = \delta_{pra}(f) = \delta(f)$: la démonstration est identique à celle du théorème 2.
- Supposons que $\partial f/\partial x$ et $\partial f/\partial y$ existent et soient continues. Par souci de simplification, nous supposerons que la base est 2 (la démonstration est similaire en base B). On veut calculer f en ligne avec un délai $\delta < \left\lceil \log_B \text{Max}\left(\left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right|\right)\right\rceil$. Comme δ est un entier, on a :

$$\delta < \left(\log_2 \text{Max}\left(\left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right|\right)\right)$$

$$\text{d'où } 2^\delta < \text{Max}\left(\left|\frac{\partial f}{\partial x}\right| + \left|\frac{\partial f}{\partial y}\right|\right)$$

Comme $\partial f/\partial x$ et $\partial f/\partial y$ sont continues, il existe un domaine $J = [A, B]\times[C, D]$, $A<B$, $C<D$, tel que pour tout $(x, y) \in J$, pour tout $(z, t) \in J$, $\left|\frac{\partial f}{\partial x}\right|(x, y) + \left|\frac{\partial f}{\partial y}\right|(z, t)$ est strictement supérieur à 2^δ, et un domaine I, $J \supseteq I$, $I = [E, F]\times[G, H]$, $E<F$, $G<H$, tel que les signes de $\partial f/\partial x$ et $\partial f/\partial y$ ne changent pas sur I. Il existe $x_1, x_2, \ldots, x_{p+\delta}$ et $y1, y_2, \ldots, y_{p+\delta}$ tels que I contient $[\alpha, \beta]\times[\gamma, \tau]$, avec :

- $\alpha = 0.x_1x_2x_3...x_{p+\delta}0\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}...$
- $\beta = 0.x_1x_2x_3...x_{p+\delta}011111...$
- $\gamma = 0.y_1y_2y_3...y_{p+\delta}0\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}...$
- $\tau = 0.y_1y_2y_3...y_{p+\delta}011111...$

Notons $(\lambda, \mu) = (\gamma, \tau)$ si $\frac{\partial f}{\partial x}$ et $\frac{\partial f}{\partial y}$ ont le même signe sur I, et (τ, γ) si leurs signes sont opposés.

$f(\beta, \mu) - f(\alpha, \lambda) = (\beta-\alpha).\frac{\partial f}{\partial x}(s_1, t_1) + (\mu-\lambda).\frac{\partial f}{\partial y}(s_2, t_2)$, où :

$$\alpha \leq s_1, s_2 \leq \beta \qquad \lambda \leq t_1, t_2 \leq \mu$$

Donc : $|f(\beta, \tau) - f(\alpha, \gamma)| > 2^{-p-\delta}.2^{\delta} = 2^{-p}$.

Donc l'information "$x_{p+\delta+1} = y_{p+\delta+1} = 0$" n'est pas suffisante pour calculer z_{p+1}.

Le théorème 5 est donc démontré.

III.4.B Application aux fonctions élémentaires

Nous allons maintenant appliquer les résultats précédents aux fonctions mathématiques les plus communes. Les bornes inférieures sur les délais obtenues à l'aide des théorèmes 3 et 5 sont présentées dans la figure 2. Comme on travaille dans l'intervalle [-1, 1], on considère 2 cas :

– Un dépassement est possible : le résultat n'est pas valide si $|f(x)|$ (ou $|f(x, y)|$) est plus grand que 1(cas de l'addition).

– Pour éviter les dépassements, au lieu de calculer f, on calcule f/K, où K est une puissance de la base supérieure à la valeur maximale de f sur [-1, 1]. Cela conduit à des bornes inférieures "artificielles" pour les délais, car quand f (x) (ou f(x, y)) est dans [-1, 1], on introduit des zéros "artificiels" en tête du résultat.

Une remarque intéressante est que pour l'addition et la multiplication en base r>2, la borne inférieure pour le délai est atteinte (on va voir le cas de l'addition en base 2). Les bornes théoriques présentées sont donc assez fines. Le théorème suivant montre que le délai de l'addition en base 2 est 2.

Théorème 6. en base 2, le délai de l'addition ($\delta(+)$) vaut 2.

Démonstration. On verra plus loin que l'on connait des algorithmes de délai 2 pour l'addition [GHM89], il nous suffit de prouver que $\delta(+) > 1$. Dans ce but, supposons que l'on puisse effectuer une addition en ligne avec un délai 1.
Soient x et y deux éléments de]-1, 1[. Supposons qu'à partir des p+1 premiers chiffres ($0.x_1x_2...x_{p+1}$ et $0.y_1y_2...y_{p+1}$) de x et y, on ait obtenu les p premiers chiffres d'une représentation de $z = x+y = z_0.z_1z_2...z_p$. On définit :

- $\alpha = 0.x_1x_2x_3...x_{p+2}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}...$
- $\beta = 0.x_1x_2x_3...x_{p+2}11111...$
- $\gamma = 0.y_1y_2y_3...y_{p+2}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}...$
- $\tau = 0.y_1y_2y_3...y_{p+2}11111...$

$x \in [\alpha, \beta]$, et $y \in [\gamma, \tau]$, donc $z \in I_1 = [\alpha+\beta, \gamma+\tau]$. Supposons que x_{p+2} et y_{p+2} suffisent à calculer z_{p+1}. Les nombres qui s'écrivent avec $z_0.z_1z_2...z_pz_{p+1}$ comme p+1 premiers chiffres sont ceux de l'intervalle $I_2 = [z_0.z_1...z_{p+1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}..., z_0.z_1...z_{p+1}111...]$, il faut donc que I_1 soit inclus dans I_2. Comme les longueurs de I_1 et I_2 sont identiques et égales à 2^{-p}, les deux intervalles doivent être confondus, donc leurs milieux sont égaux, donc $z_0.z_1...z_{p+1}$ vaut $\frac{1}{2}(\alpha+\beta+\gamma+\tau) = 0.x_1x_2x_3...x_{p+2} + 0.y_1y_2y_3...y_{p+2}$. C'est impossible, si par exemple, $x_{p+2} = 0$ et $y_{p+2} = 1$, car dans ce cas 2^{p+2} $(0.x_1x_2x_3...x_{p+2} + 0.y_1y_2y_3...y_{p+2})$ est impair alors que $2^{p+2}(z_0.z_1z_2...z_pz_{p+1})$ est pair.

Considérons maintenant le problème de la division (et de l'inversion). On suppose que les nombres sont représentés en virgule flottante et que la mantisse est représenté sous forme redondante. Un nombre, de mantisse m, peut être :

- Normalisé si $\frac{1}{r} \leq |\ m\ | < 1$
- Quasi-normalisé si $\frac{1}{r^2} \leq |\ m\ | < 1$
- Pseudo-normalisé si $\frac{1}{r^p} \leq |\ m\ | < 1$, avec $p \geq 3$.

En fait on ne considère que les nombres normalisés et quasi-normalisés. En pratique, il est très difficile d'assurer qu'un nombre flottant redondant est normalisé (alors que c'est immédiat en notation non redondante : un nombre est normalisé si et seulement si le premier chiffre de sa mantisse est non nul), mais l'examen des 2 chiffres de poids forts suffit à vérifier qu'il est quasi-normalisé (et la quasi-normalisation d'un nombre flottant peut être facilement réalisée en-ligne). Donc si les entrées sont en notation non redondante, on considère le cas normalisé, sinon, on considère la cas quasi-normalisé. Dans [LS87], Lin et Sips ont présenté un

algorithme d'inversion dont le délai est, pour des nombres écrits en base 2, de 3 pour des nombres normalisés et de 4 à 5 pour des nombres quasi-normalisés; Pour des nombres écrits en base r, r≥4, on obtient comme délai, 1 pour des entrées normalisées et de 2 à 3 pour des entrées quasi-normalisées. On déduit des résultats de Lin et Sips, et de la table suivante, que pour des nombres en base r (r≥4), Le délai minimum pour l'inversion est de 1 pour des nombres normalisés et de 2 ou 3 pour des nombres quasi normalisés. Cela implique que le délai pour la division en base r (r≥4), est de 2 pour les nombres normalisés et de 3 ou 4 pour les nombres quasi normalisés.

f(x) ou f(x, y)	intervalle	Max |f '|	$\lceil \log_2 \text{Max}\|f'\| \rceil$	$\lceil \log_B \text{Max}\|f'\| \rceil$
sin(x)	[-1, 1]	1	0	0
cos(x)	[-1, 1]	sin 1	0	0
e^x	[-1, 1]	e	2	2
$e^{x/4}$ (base 2)	[-1, 1]	$\frac{e}{4}$	0	0
$e^{x/B}$ (B>2)	[-1, 1]	$\frac{e}{B}$	0	0
ln(x)	[-1, 1]	e	2	1 si B>2
x+y	[-1, 1]	2	1	1
$\frac{x+y}{B}$	[-1, 1]	$\frac{2}{B}$	0	0
xy	[-1, 1]	2	1	1
$\frac{1}{By}$	$[\frac{1}{B}, 1]$	B	1	1
$\frac{1}{B^2y}$	$[\frac{1}{B^2}, 1]$	B^2	2	2
$\frac{1}{B^py}$	$[\frac{1}{B^p}, 1]$	B^p	p	p
$\frac{x}{By}$	$[\frac{1}{B}, 1]$	1+B	2	2
$\frac{x}{B^2y}$	$[\frac{1}{B^2}, 1]$	$1+B^2$	3	3
$\frac{x}{B^py}$	$[\frac{1}{B^p}, 1]$	$1+B^p$	p+1	p+1
$\sqrt{x}$	$[B^{-p}, 1]$	$0.5B^{p/2}$	$\lceil p/2 - 1 \rceil$	$\lceil p/2 - \log_B 2 \rceil$

Figure 2 bornes inférieure des délais pour le calcul en ligne de fonction classiques.

(pour les fonctions de 2 variables, |f ' | représente | ∂f/∂x | + | ∂f/∂y |).

III.5.Opérateurs pour le calcul en ligne

III.5.A Représentation des nombres

Notation. On écrira les nombres en base 2 et en chiffres signés. Chaque chiffre signé binaire (CSB) x (qui vaut $\bar{1}$, 0 ou 1) est représenté par un couple (x^+, x^-) de chiffres binaires (+/-) tels que $x = x^+ - x^-$. $\bar{1}$ est donc représenté par (0, 1), 1 par (1, 0) et 0 par (0, 0) ou (1, 1). Pour changer le signe d'un CSB, il suffit de permuter (ou de complémenter) les deux bits qui le représentent. Ceci permettra de changer le signe d'un nombre en temps constant. Cette notation est aussi appelée Borrow-Save (BS) par analogie avec la notation Carry-Save.

III.5.B Les opérateurs

Nous allons présenter des opérateurs de base qui permettent de traiter des nombres écrits dans le système décrit ci-dessus.

III.5.B.1 L'opérateur PPM

Cet opérateur est la cellule de base permettant de construire des additionneurs ou des multiplieurs. On peut l'utiliser aussi bien pour construire un additionneur totalement parallèle travaillant en temps constant qu'un additionneur série en mode MSB ou LSB. Cette cellule est, comme on va le voir, semblable à la cellule *full-adder* utilisée pour construire des additionneurs ou multiplieurs en écriture binaire classique.

L'addition est basée sur le fait que l'on calcule successivement une "retenue" négative et une "retenue" positive. Tout l'algorithme sera décrit au niveau du *bit*..
Notons $((a^+_{n-1}, a^-_{n-1}) \dots (a^+_0, a^-_0))$ et $((b^+_{n-1}, b^-_{n-1}) \dots (b^+_0, b^-_0))$ les nombres que l'on désire additionner, et $((s^+_n, s^-_n) \dots (s^+_0, s^-_0))$ leur somme. Définissons des quantités booléennes c^+_i et c^-_i par :

$$a^+_i + b^+_i - a^-_i = 2\ c^+_{i+1} - c^-_i, \qquad \text{et } c^+_0 = c^-_n = 0.$$

Les termes s^+_i et s^-_i s'obtiennent par :

$$-(b^-_i + c^-_i - c^+_i) = -(2\ s^-_{i+1} - s^+_i), \qquad s^-_0 = 0 \text{ et } s^+_n = c^+_n.$$

On a donc besoin d'effectuer toujours la même opération : calculer en fonction de 3 bits x, y et z deux bits t et u tels que $2t - u = x + y - z$. Ceci peut s'obtenir par les fonctions booléennes :

$$u = x \oplus y \oplus z \qquad t = \text{Maj}(x, y, \bar{z}) = xy + x\bar{z} + y\bar{z}$$

Pour calculer ces fonctions, nous utiliserons une cellule élémentaire appelée PPM (de *Plus Plus Moins*), très semblable – comme l'indique la figure 3 – aux cellules élémentaires "FA" (*Full Adder*) des additionneurs classiques. On définit aussi la cellule MMP (de *Moins Moins Plus*), mais comme ces deux cellules sont en fait identiques on les appellera toutes les deux cellules PPM. On indiquera toujours sur les schéma les signes des entrées et des sorties ce qui supprime les risques de confusion entre t et u. De plus (pour les cas où on utilise conjointement des cellules FA et PPM) on impose que la sortie t soit toujours à gauche de u.

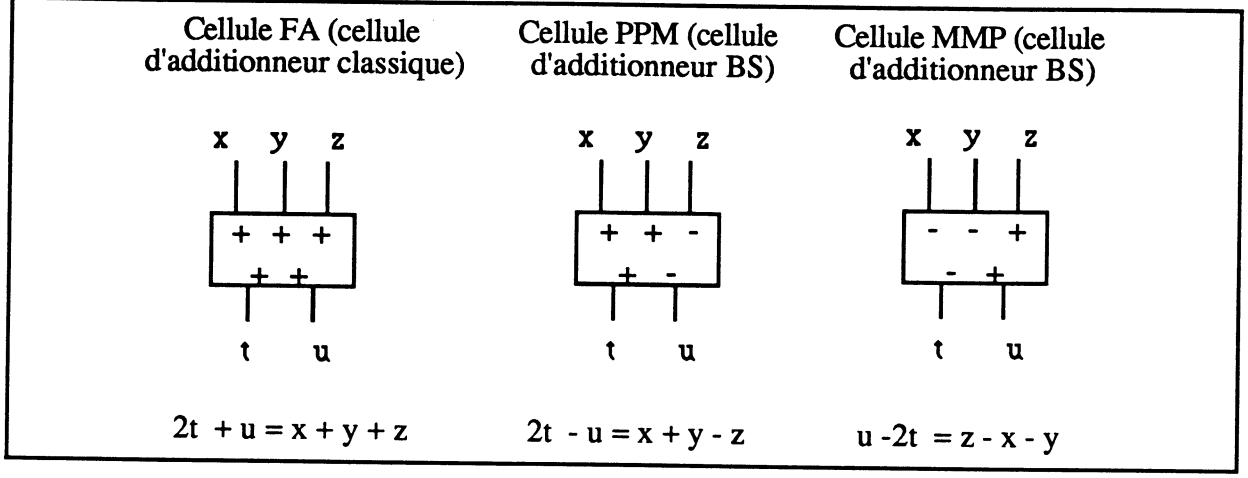


Figure 3 : l'opérateur PPM.

Additionneur parallèle

Voyons comment on peut construire un additionneur parallèle en temps constant à l'aide de cellules PPM :

Soient deux nombres A et B à additionner. $A=\sum_{i=0}^{n-1} a_i 2^i$ et $B=\sum_{i=0}^{n-1} b_i 2^i$ où les a_i et les b_i sont dans l'ensemble{-1, 0, 1}. On écrit $a_i=a_i^+ - a_i^-$ et $b_i=b_i^+ - b_i^-$ où a_i^+, a_i^-, b_i^+ et b_i^- sont dans {0, 1}.

Le circuit décrit figure 4 est un additionneur parallèle. Ecrivons les relations liant les différentes variables pour un étage de l'additionneur.

On a $-2x_{i+1}^- + x_i^+ = b_i^+ - b_i^- - a_i^-$ et $2s_{i+1}^+ - s_i^- = x_i^+ - x_i^- + a_i^+$, ce qui nous donne

$2s_{i+1}^+ - s_i^- - 2x_{i+1}^- = b_i^+ - b_i^- - a_i^- + a_i^+ - x_i^-$.

Donc, si on pose $s_0^+ = x_0^- = 0$ et $s_n^- = x_n^-$ on obtient $S=\sum_{i=0}^{n} s_i 2^i = A+B$ avec $s_i = s_i^+ - s_i^-$.

Cet additionneur calcule donc bien la somme S=A+B, il est presque identique à un additionneur Carry-save en base 2 construit avec des cellules FA

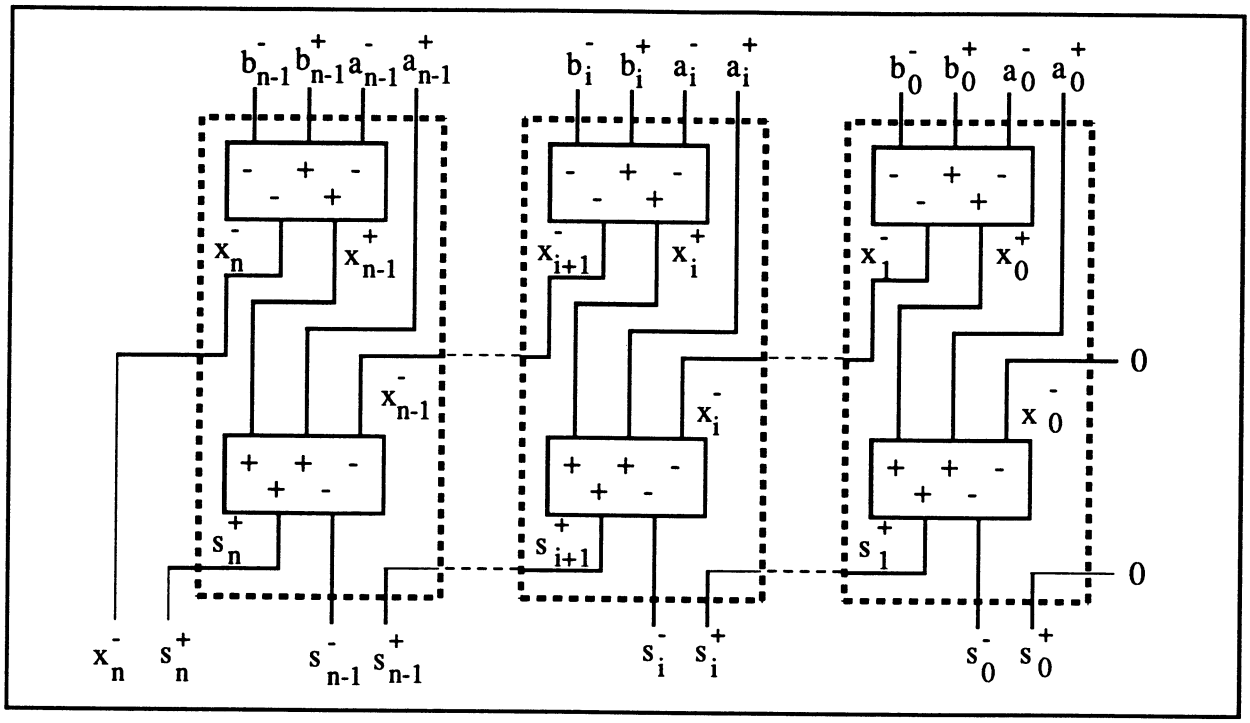


Figure 4 : un additionneur parallèle.

Additionneur à retenue conservée à trois entrées.

Cet additionneur reçoit en entrée trois entiers A, B, et C et fournit deux entiers S et S' tels que A+B+C = S+S'. C'est l'analogue en chiffres signés des additionneurs "carry-save" en écriture classique. En combinant un additionneur parallèle et un additionneur à retenue conservée on peut construire un additionneur à trois entrées dans lequel un signal traverse au plus trois cellules PPM.

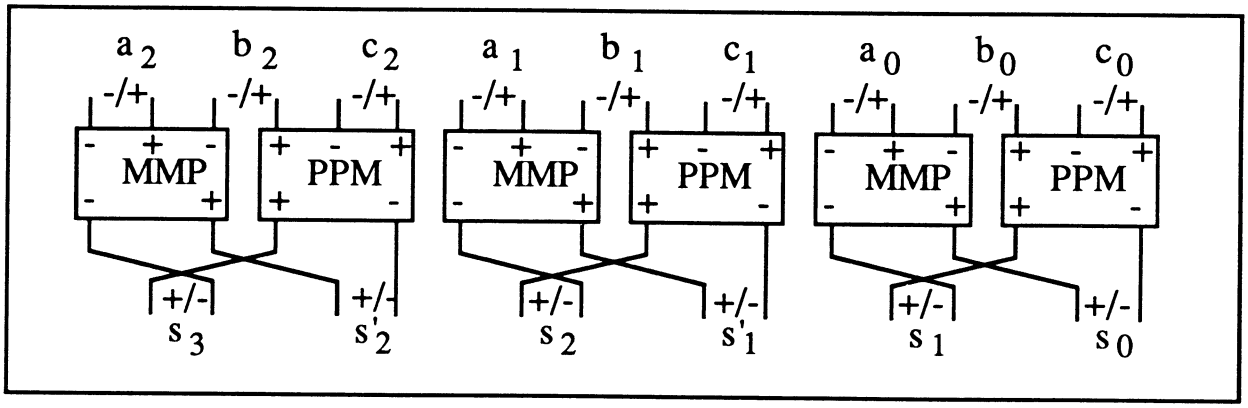


Figure 5 : un additionneur à retenue conservée (3 donne 2).

Addition série poids forts en tête.

Comme précédemment, on veut additionner deux nombres $A=\sum_{i=0}^{n-1} a_i 2^i$ et $B=\sum_{i=0}^{n-1} b_i 2^i$ où les a_i et les b_i sont dans l'ensemble{-1, 0, 1}. On écrit $a_i=a_i^+ - a_i^-$ et $b_i=b_i^+ - b_i^-$ où a_i^+, a_i^-, b_i^+ et b_i^- sont dans {0, 1}.

Pour construire un additionneur en-ligne, on va utiliser un étage de l'additionneur parallèle dont les sorties de poids faible seront retardées, ce qui donne le circuit décrit figure 6. Les cellules appelées D sont des délais d'un cycle (bascule), rappelons les relations liant les différentes valeurs :

$4s^+_{i+2} - 2s^-_{i+1} = 2x^+_{i+1} - 2x^-_{i+1} + 2a^+_{i+1}$ et $-2x^-_{i+1} + x^+_i = b^+_i - b^-_i - a^-_i$, ce qui nous donne

$4s^+_{i+2} - 2s^-_{i+1} = 2a^+_{i+1} - a^-_i + b^+_i - b^-_i + 2x^+_{i+1} - x^+_i$.

Si a^+_i, a^-_i, b^+_i et b^-_i sont nul pour i= -1 et i=n, alors on a $x^-_0 = x^+_{-1} = x^-_{n+1} = x^+_n = 0$ et par conséquent $s^+_{n+1} = s^+_0 = 0$.

Si on pose $S = \sum_{i=0}^{n-1} (s^+_i - s^-_i)2^i$, on a $S = \sum_{i=-1}^{n-1} (4s^+_{i+2} - 2s^-_{i+1})2^i - s^+_{n+1} + s^+_0$.

Or $s^+_{n+1} = s^+_0 = 0$, donc $S = \sum_{i=-1}^{n-1} (4s^+_{i+2} - 2s^-_{i+1})2^i$, que l'on peut réécrire comme suit :

$S = \sum_{i=-1}^{n-1} (2a^+_{i+1} - a^-_i + b^+_i - b^-_i + 2x^+_{i+1} - x^+_i)2^i$, soit encore

$S = \sum_{i=0}^{n-1} (a^+_i - a^-_i + b^+_i - b^-_i)2^i + x^+_n 2^n - \frac{1}{2}x^+_{-1} = \sum_{i=0}^{n-1} (a^+_i - a^-_i + b^+_i - b^-_i)2^i = A+B.$

Le circuit réalise bien la somme de deux nombres écrits en chiffres binaires signés avec un délai 2.

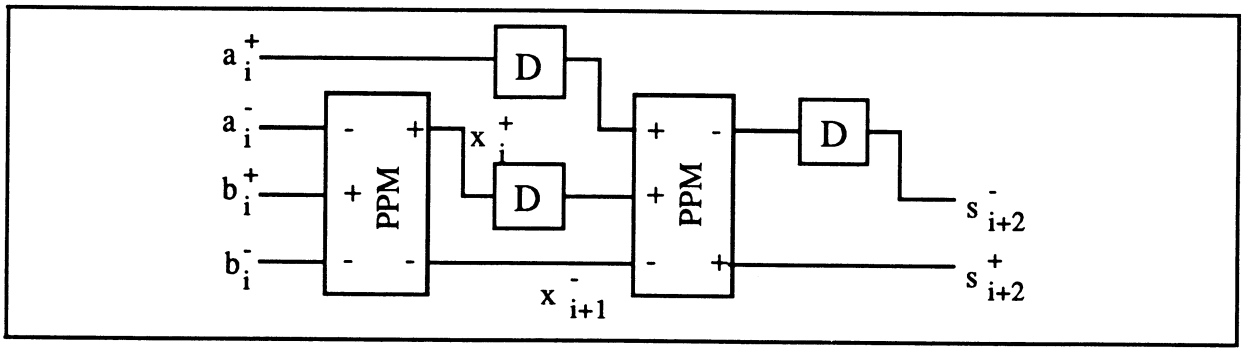


Figure 6 : un additionneur en ligne

En associant cet additionneur avec un additionneur à retenue conservée, on construit un additionneur en ligne à trois entrées de délai 3 (décrit plus loin).

III.5.B.2 Multiplieur CSB

Cette cellule réalise le produit de 2 CSB

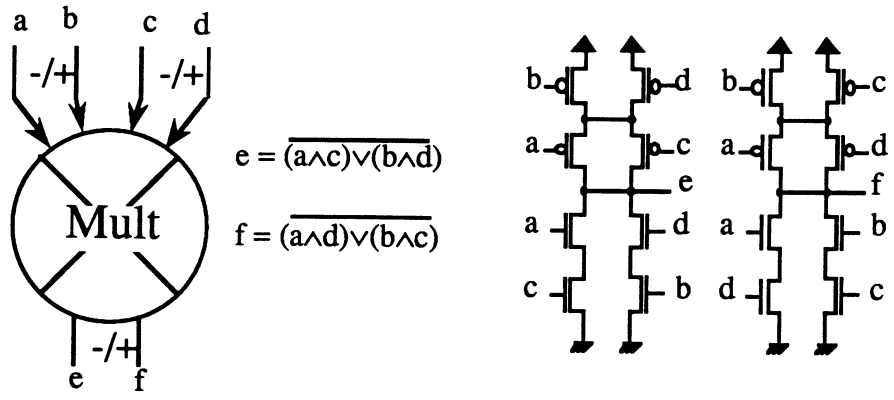


Figure 7 : cellule élémentaire de multiplieur.

La cellule multiplieur CSB va nous servir à construire des multiplieurs parallèles ou série de nombres écrits en écriture binaire redondante.

Multiplieur en ligne.

On veut multiplier deux nombres A et B écrits en chiffres signés binaires sous la forme :

$$A=\sum_{k=1}^{n} a_k 2^{-k},\ a_k=a_k^+ - a_k^- \in \{-1, 0, 1\}.$$

$$B=\sum_{k=1}^{n} b_k 2^{-k},\ b_k=b_k^+ - b_k^- \in \{-1, 0, 1\}.$$

Notons $A^{(i)}=\sum_{k=1}^{i} a_k 2^{-k}$, $B^{(i)}=\sum_{k=1}^{i} b_k 2^{-k}$ et $P^{(i)}=A^{(i)}.B^{(i)}$.

On a $A^{(i+1)} = A^{(i)} + a_{i+1}2^{-i-1}$ et $B^{(i+1)} = B^{(i)} + b_{i+1}2^{-i-1}$,

soit $P^{(i+1)} = P^{(i)} + A^{(i)}.b_{i+1}2^{-i-1} + B^{(i+1)}.a_{i+1}2^{-i-1}$

Si on pose $\Pi^{(i)} = P^{(i)}.2^i$ on obtient $\Pi^{(i+1)} = 2.\Pi^{(i)} + A^{(i)}.b_{i+1} + B^{(i+1)}.a_{i+1}$.

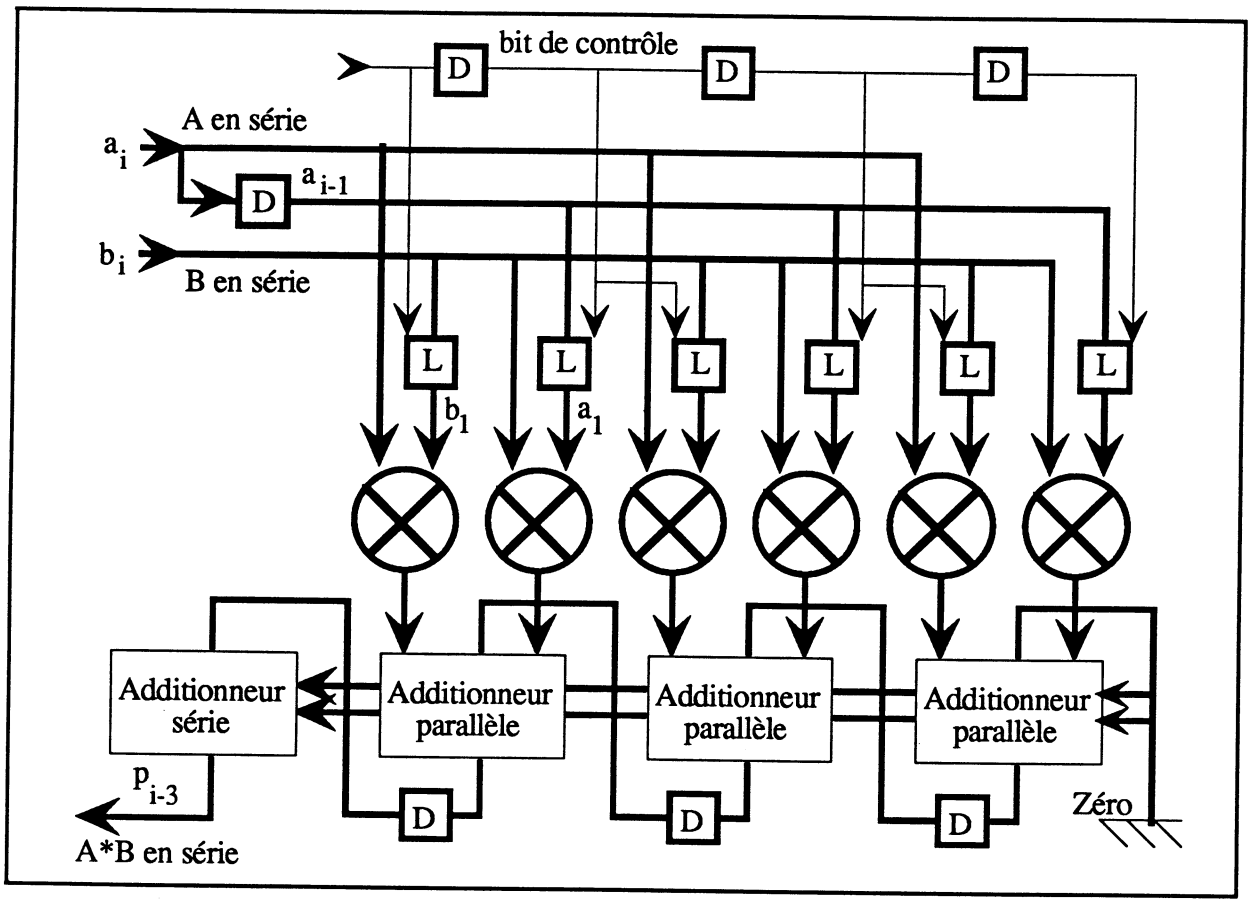


Figure 8 : multiplieur en ligne.

Le circuit de la figure 8 décrit un multiplieur de délai 3, les connexions en gras transportent des CSB, les cellules L sont des tampons pour mémoriser A et B. Avant le calcul, elles sont toutes mises à zéro. Pendant le calcul, un bit de contrôle (jeton) circule à l'envers pour charger en série les CSB de A et B. Le détail d'un étage d'additionneur est donné figure 10. Le séquencement du calcul de A*B est détaillé dans la table de la figure 9.

Cycle	Bits	Multiplicande	Multiplieur	Entrées de l'additionneur parallèle		
nomb.	entrants			*(plus 2 fois la ligne précédente)*		
0		0 0 0	0 0 0	0	0	0
1	a_1 b_1	0 0 0	b_1 0 0	a_1b_1	0	0
2	a_2 b_2	a_1 0 0	b_1 b_2 0	$a_1b_2+a_2b_1$	a_2b_2	0
3	a_3 b_3	a_1 a_2 0	b_1 b_2 b_3	$a_1b_3+a_3b_1$	$a_2b_3+a_3b_2$	a_3b_3

*Figure 9 : calcul de A*B.*

Finalement le produit est calculé comme suit :

$8P = ((((a_1b_1)*2 + a_1b_2 + a_2b_1)*2 + a_2b_2 + a_1b_3 + a_3b_1)*2 + a_2b_3 + a_3b_2)*2 + a_3b_3$

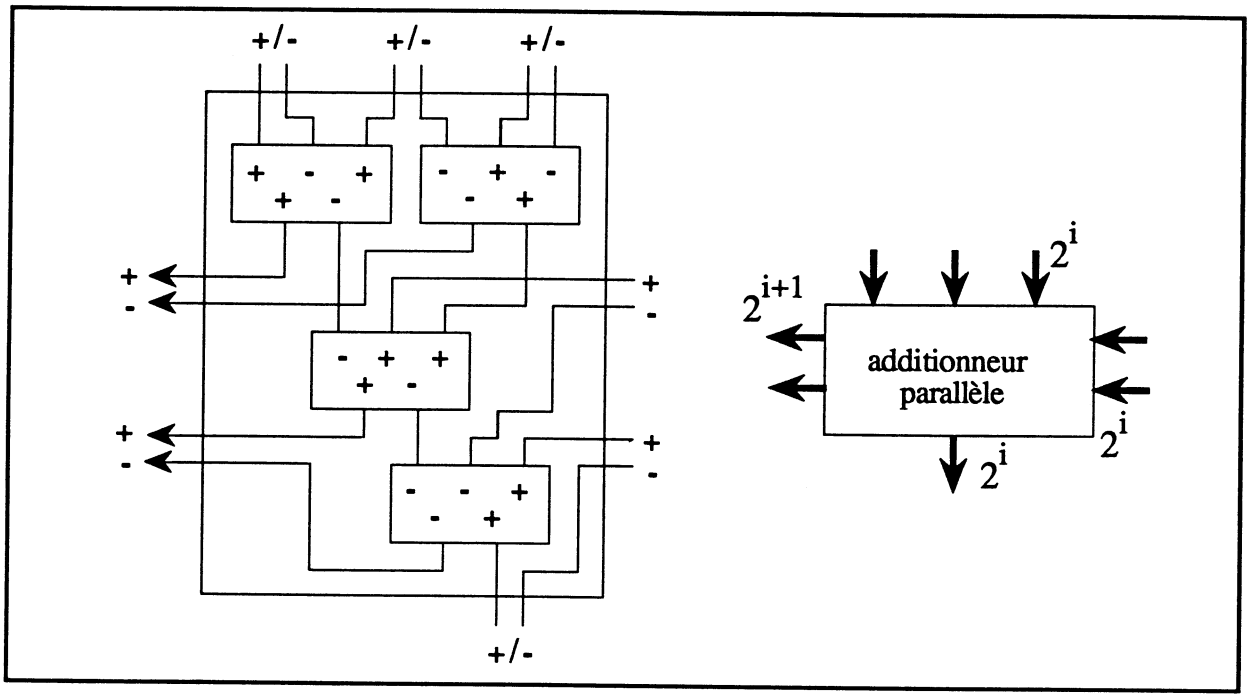


Figure 10 : détail d'un étage de l'additionneur parallèle.

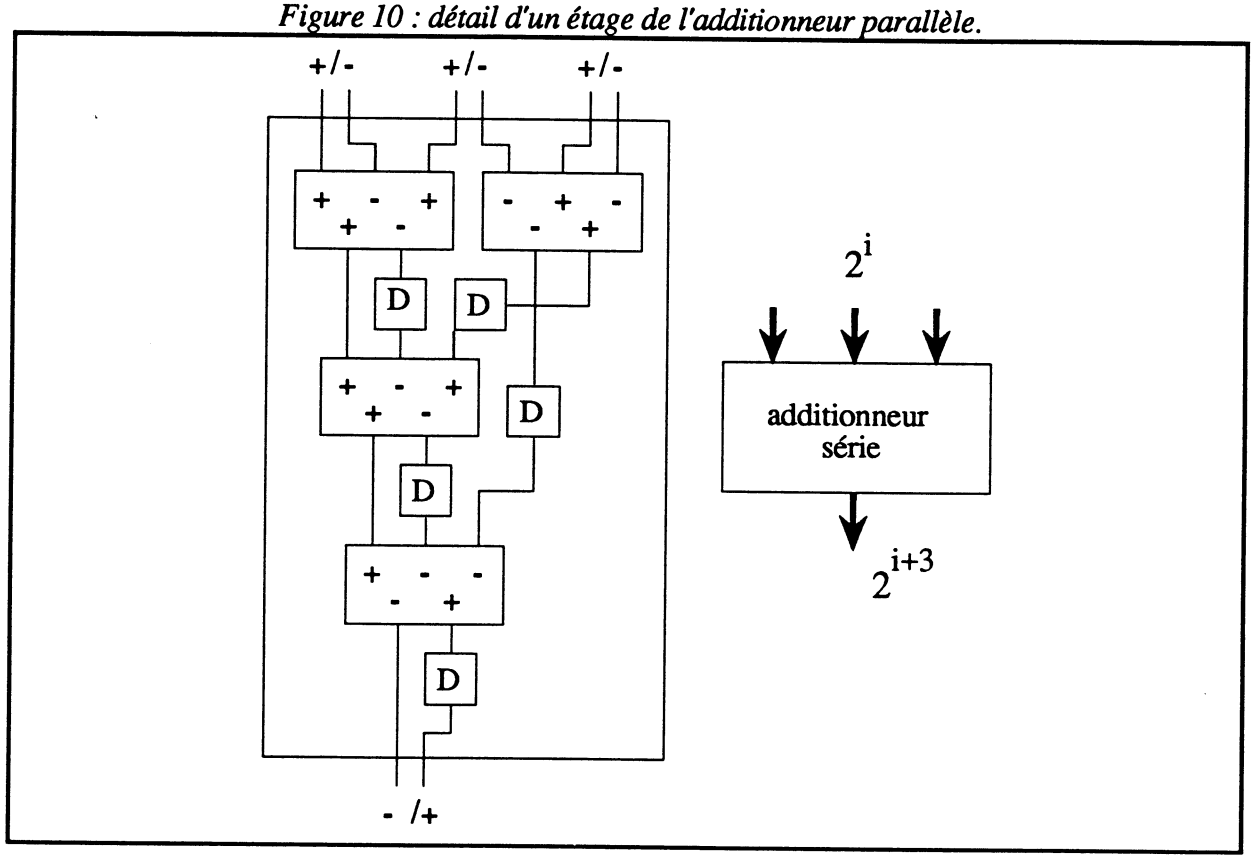


Figure 11: détail de l'additionneur série à trois entrées.

Implantation.

Le circuit de la figure 8 est constitué de trois étages identiques (détaillés figure 10). En assemblant un plus grand nombre d'étages, on peut obtenir un multiplieur de la taille voulue

sans avoir à modifier le schéma car chaque étage ne communique qu'avec ses voisins immédiats. Une implantation fine de ces cellules demande 32 portes et 152 transistors par étage ([GHMP 89]). Ainsi un multiplieur de 2000 CSB (600 chiffres décimaux) demanderait 300k transistors tout en restant très régulier, ce qui est parfaitement réalisable.

III.5.C Calcul du maximum de deux nombres

Supposons que l'on désire évaluer en mode *on-line* le maximum $y = 0.y_1y_2y_3y_4\ldots$ de p nombres $x^{(1)}, x^{(2)}, \ldots, x^{(p)}$. Le nombre $x^{(j)}$ est supposé écrit en chiffres signés binaires sous la forme :

$$x^{(j)} = 0.x_1^{(j)}x_2^{(j)}x_3^{(j)}x_4^{(j)}\ldots, x_k^{(j)} \in \{\bar{1}, 0, 1\}$$

Nous allons donner un algorithme permettant de calculer $y = 0.y_1y_2y_3y_4\ldots$ avec un délai égal à zéro. Il s'agit de lire en parallèle les nombres à comparer, à raison d'un chiffre par étape. Le plus grand des nombres constitués par les chiffres déjà reçus, appelé maximum courant, est déterminé à chaque étape et comparé aux autres nombres. Les nombres trop distants du maximum courant sont exclus de la compétition.

Une remarque préliminaire s'impose : le premier chiffre de y, soit y_1, sera déduit des premiers chiffres $x_1^{(j)}$ des nombres $x^{(j)}$. Il est tout à fait naturel (et dans certains cas, nécessaire) de choisir y_1 égal au plus grand des $x_1^{(j)}$. Toutefois, le nombre $x^{(k)}$ égal au maximum des $x^{(j)}$ a peut-être un premier chiffre $x_1^{(k)}$ inférieur à l'un des $x_1^{(j)}$ (que l'on songe par exemple au fait que le nombre $a = 0.0111$ est strictement supérieur à $b = 0.1\overline{111}$, alors que son premier chiffre est inférieur au premier chiffre de b). Une conséquence de ceci est que l'écriture du maximum des $x^{(j)}$ que fournira l'algorithme ne coïncide pas forcément avec l'écriture de ce maximum fournie en entrée de l'algorithme.

L'algorithme consiste à repérer à l'étape i le maximum courant, c'est à dire à évaluer un indice $maxcour \in \{1, 2, \ldots, p\}$ tel que le nombre constitué des i premiers chiffres de $x^{(maxcour)}$ est le plus grand des nombres constitués des i premiers chiffres des $x^{(k)}$. Considérons la différence entre le nombre constitué des i premiers chiffres de $x^{(maxcour)}$ (c'est à dire le maximum courant) et le nombre constitué des i premiers chiffres de $x^{(k)}$, si cette différence est supérieure ou égale à 2^{i-1}, il est facile de se persuader que $x^{(k)}$ ne peut pas être supérieur à $x^{(maxcour)}$ (même si tous les chiffres ultérieurs de $x^{(k)}$ valaient 1 et si tous les chiffres ultérieurs de $x^{(maxcour)}$ valaient $\bar{1}$, $x^{(maxcour)}$ et $x^{(k)}$ seraient au mieux égaux). A chaque étape on définit des termes $d^{(k)}$ égaux à :

• 0 si la différence entre le nombre constitué des i premiers chiffres de $x^{(maxcour)}$ et celui constitué des i premiers chiffres de $x^{(k)}$ est nulle.

• 1 si cette différence est égale à 2^{-i}

• $+\infty$ sinon.

Il n'y aura plus besoin de considérer dans le déroulement ultérieur de l'algorithme les $x^{(k)}$ tels que $d^{(k)}$ vaut $+\infty$.

Définissons une fonction R(x) par R(x) = x si $x \in \{0, 1\}$, R(x) = $+\infty$ sinon. La première étape de l'algorithme consiste à calculer les termes suivants :

• Maxcour = j tel que $\forall k \in \{1, 2, \ldots, p\}$, $x_1^{(j)} \geq x_1^{(k)}$.

• $y_1 = x_1^{(maxcour)}$

• $\forall k \in \{1, 2, \ldots, p\}$, $d^{(k)} = R\ [y_1 - x_1^{(k)}]$

A l'étape i de l'algorithme, supposons que l'on ait déjà calculé :

$$0.y_1y_2y_3 \ldots y_i = 0.x_1^{(maxcour)}x_2^{(maxcour)}x_3^{(maxcour)} \ldots x_i^{(maxcour)}$$

(N.B. L'égalité ci-dessus est une égalité des *nombres*, pas forcément des chaînes de chiffres qui les représentent). Nous recevons à ce moment là les termes $x_{i+1}^{(k)}$.

Posons, pour k = 1, 2, ..., p :

$$\alpha^{(k)} = 2d^{(k)} + [x_{i+1}^{(maxcour)} - x_{i+1}^{(k)}]$$

La différence entre le nombre constitué des i+1 premiers chiffres de $x^{(maxcour)}$ et le nombre constitué des i+1 premiers chiffres de $x^{(k)}$ est par construction infinie ou du signe de $\alpha^{(k)}$. Par conséquent, deux cas peuvent se produire :

1) Si pour tout k = 1, 2, ..., p, $\alpha^{(k)} \geq 0$, alors :

• maxcour reste inchangé.

• $y_{i+1} = x_{i+1}^{(maxcour)}$

• $\forall k \in \{1, 2, \ldots, p\}$, $d^{(k)} = R\ [\alpha^{(k)}]$.

2) S'il existe un terme $j \in \{1, 2, \ldots, p\}$ tel que $\alpha^{(j)} = \min_k \alpha^{(k)} < 0$, alors :

• maxcour = j.

• $y_{i+1} = x_{i+1}^{(j)}$

• $\forall k \in \{1, 2, \ldots, p\}$, $d^{(k)} = R\ [\alpha^{(k)} - \alpha^{(j)}]$.

En effet, dans ce cas, à l'étape précédente, bien que maxcour fût différent de j, les nombres constitués des i premiers chiffres de y et des i premiers chiffres de $x^{(j)}$ étaient forcément égaux. Il est donc naturel d'utiliser le $i+1^{ème}$ chiffre de $x^{(j)}$ pour l'écriture de y.

Le tableau suivant donne un exemple d'exécution de cet algorithme. On cherche le maximum de :

$x^{(1)} = 0.1\bar{1}\bar{1}1\bar{1}1\bar{1}1$
$x^{(2)} = 0.00110\bar{1}10$
$x^{(3)} = 0.1\bar{1}\bar{1}010\bar{1}\bar{1}$
$x^{(4)} = 0.001100\bar{1}1$

i	maxcour	$\alpha^{(1)}$	$\alpha^{(2)}$	$\alpha^{(3)}$	$\alpha^{(4)}$	$d^{(1)}$	$d^{(2)}$	$d^{(3)}$	$d^{(4)}$	y_i
1	1					0	1	0	1	1
2	1	0	1	0	1	0	1	0	1	$\bar{1}$
3	1	0	0	0	0	0	0	0	0	$\bar{1}$
4	1	0	0	1	0	0	0	1	0	1
5	2	0	$\bar{1}$	0	$\bar{1}$	1	0	1	0	0
6	4	0	0	1	$\bar{1}$	1	1	∞	0	0
7	4	2	0		0	∞	0	∞	0	$\bar{1}$
8	4		1		0	∞	1	∞	0	1

On trouve bien le maximum, soit $x^{(4)}$, avec un délai zéro mais avec une écriture différente, qui est $0.1\bar{1}\bar{1}100\bar{1}1$.

III.7 Conclusion

On a donné des résultats de complexité qui permettent de déterminer exactement (ou de borner) les délais en ligne des fonctions arithmétiques. Cela nous permet de montrer que beaucoup d'opérateurs en ligne connus sont optimaux en délai. On propose des cellules de base (cellule PPM et multiplieur CSB) qui permettent de réaliser simplement des opérateurs en ligne fondamentaux (additionneurs et multiplieurs) qui occupent une petite surface.

Ce mode de calcul n'est pas d'usage universel pour le calcul scientifique, on retrouve les mêmes limitations que pour la notation des réels en virgule fixe : il faut connaître a priori l'ordre de grandeur des valeurs qui peuvent apparaître. De plus, le calcul avec des opérateurs en ligne, est plus lent que le calcul avec des opérateurs classiques si on n'exploite pas la possibilité d'enchaîner des nombreuses opérations au niveau du bit.

On peut donner deux applications importantes pour lesquelles le mode en ligne peut apporter de réels progrès :

- Le traitement du signal en basses et moyennes fréquences (de l'ordre du MHz) : un filtre numérique demande plusieurs additionneurs et multiplieurs chaînés et rebouclés et on réalise toujours les mêmes opérations sur des données qui arrivent en série.
- Le calcul en grande précision : le mode en ligne permet de réaliser des additionneurs de surface constante et des multiplieurs de taille linéaire en la longueur des opérandes. On peut donc calculer en ligne sur des entiers de grande taille sans passer par des solutions logicielles.

IV. Représentation polygonale des complexes

IV.1 Introduction

Les nombres complexes sont utilisés dans de nombreuses branches de l'informatique (traitement du signal etc.). Il convient donc de les représenter aussi efficacement que possible. La première idée qui vient à l'esprit est de représenter le complexe a+i.b par le couple de réels (a,b).

Cette représentation n'a pas que des avantages : en effet les ensembles de nombres représentables avec a et b pris chacun dans un intervalle (les rectangles du plan complexe de cotés parallèles aux axes) ne sont pas stables par multiplication : par exemple l'ensemble E des complexes a+i.b avec a et b dans [-1,1] est représenté ci dessous ainsi que l'ensemble des produits des éléments de E.

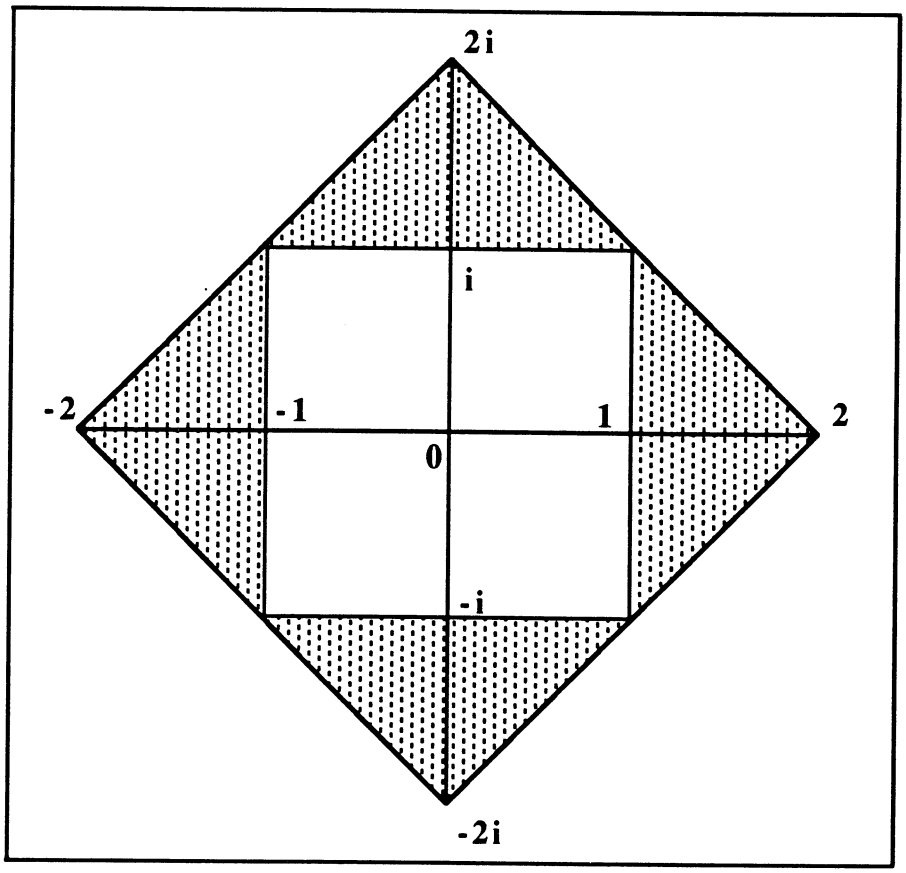


*Figure 1 : E et E*E.*

en fait, il n'existe pas de couple (a,b) de réels positifs tels que les ensembles $([-a,a]+i[-b,b])^2$ et $k([-a,a]+i[-b,b])$ soient égaux avec k réel.

Un autre inconvénient est que dans ce type de notation, les produits de nombres complexes ne peuvent pas se faire chiffre à chiffre. Différentes autres représentations ont été proposées :

IV.1.A Base $i\sqrt{2}$ et chiffres dans {0,1}

Cette représentation n'est en fait pas très éloignée de la représentation classique a+ib : on augmente juste la granularité selon l'axe des imaginaires d'un facteur $\sqrt{2}$. Les ensembles de nombres représentables avec le même nombre de chiffres étant les mêmes, à une affinité près, on se retrouvera confronté aux même problèmes.

IV.1.B Base i-1 et chiffres dans {0,1} ([KN81])

Cette représentation permet d'écrire tous les éléments de Z[i] (les entiers de Gauss) de façon unique.

Si $X=A+i.B \in Z[i]$, on peut trouver eux entiers a et b de façon unique tels que l'on puisse écrire $A+i.B=(i-1)(a+i.b)+d$ avec d dans {0, 1}.
En effet, $A+i.B=(i-1)(a+i.b)+d$ entraîne $a = \frac{B-A+d}{2}$ et $b = \frac{-A-B+d}{2}$, et comme a et b doivent être entiers, on peut choisir d de façon unique dans {0, 1}car B-A et -A-B ont la même parité.
Donc tout entier X=A+iB de Z[i] s'écrit de façon unique sous la forme (i-1)(a+ib)+d avec d dans {0, 1} et a+i.b dans Z[i], ce qui impose que si un entier de Z[i] admet une écriture en base i-1 avec les chiffres dans {0, 1}, alors cette écriture est unique.
Pour trouver une écriture d'un entier de Gauss, on peut procéder comme suit :
on pose $X_0=X=A_0+i.B_0$.
On écrit alors pour tout k de façon unique $X_k = A_k+i.B_k = (i-1)(A_{k+1}+i.B_{k+1})+d_k$, et pour tout n entier, on a de manière unique $X_0 = X_n.(i-1)^n+\sum_{k=0}^{n-1} d_k.(i-1)^k$.

Il suffit de montrer que ce processus est fini, c'est à dire qu'il existe n tel que $X_n = 0$.

On peut pour cela remarquer que :

$$|X_{k+1}|^2 = A_{k+1}^2 + B_{k+1}^2 = \left(\frac{B_k-A_k+d_k}{2}\right)^2+\left(\frac{-A_k-B_k+d_k}{2}\right)^2 = \frac{B_k^2+(A_k-d_k)^2}{2},$$

et que par conséquent,

$$2(|X_k|^2 - |X_{k+1}|^2) = 2(A_k^2 + B_k^2 - A_{k+1}^2 - B_{k+1}^2) = 2A_k^2 + 2B_k^2 - B_k^2 -(A_k-d_k)^2$$
$$\text{soit } 2(|X_k|^2 - |X_{k+1}|^2) = B_k^2 + (A_k + d_k)^2 - 2d_k^2.$$

Donc si $|A_k|>2$ ou $|B_k|>1$, alors on a $|X_{k+1}|<|X_k|$ et le module des X_k décroît strictement tant que $|A_k|>2$ ou $|B_k|>1$, ce qui nous garantit qu'il existe n fini tel que $|A_n|\leq 2$ et $|B_n|\leq 1$. Et comme on le

voit sur la figure 2, tous les éléments de {-2, ..., 2}+i.{-1, 0, 1} sont représentable en base i-1 avec les puissances de i-1 inférieures ou égales à 7.
Ceci entraîne que tout entier de Z[i] admet une écriture unique finie en base i-1 avec les chiffres dans {0, 1}.

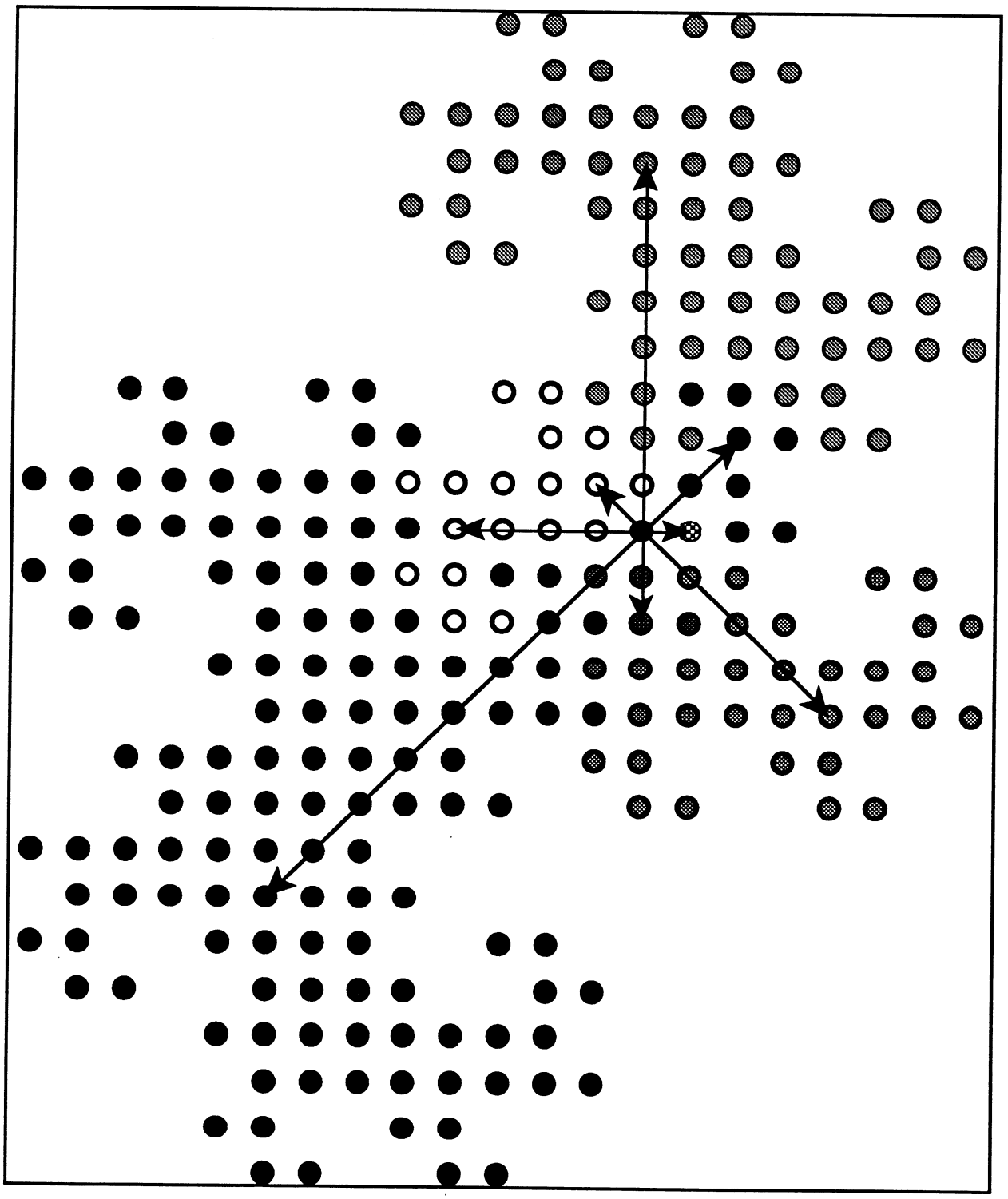


Figure 2 : entiers de Gauss représentables en base i-1 avec les puissances de i-1 inférieures ou égales à 7 et les chiffres dans {0, 1}.

IV.1.C Base i-1 et chiffres dans {-1, 0, 1}

Proposition : on peut réaliser des additions totalement parallèles sans propagation de retenue en base i-1 avec les chiffres dans {-1, 0, 1}.

Remarquons d'abord que, puisque $(i-1)^4 = -4$, si on regroupe les chiffres par 4, écrire les nombres en base i-1 avec les chiffres dans {-1, 0, 1} est équivalent à écrire les nombres en base -4 avec les chiffres dans l'ensemble $G_4 = \left\{ \sum_{k=0}^{3} d_k (i-1)^k,\ d_k \in \{-1, 0, 1\} \right\}$. Comme de plus G_4 est symétrique par rapport à zéro, écrire en base -4 avec les chiffres dans G_4 est équivalent à écrire en base 4 avec les chiffres dans G_4.

En effet si $X = \sum_{k=0}^{4n-1} x_k \beta^k = \sum_{k=0}^{n-1} X_k (-4)^k = \sum_{k=0}^{n-1} (-1)^k X_k 4^k$, où $X_k = \sum_{j=0}^{3} x_{j+4k} \beta^k \in G_4$.

donc $-X_k = \sum_{j=0}^{3} (-x_{j+4k} \beta^k) \in G_4$.

Or G_4 recouvre {-1, ..., 3} + i.{-1, ..., 3}= G'_4 et tout élément de G_4 peut être écrit comme somme de 5 éléments de G'_4 ainsi qu'on le voit dans la figure ci-contre où G_4 et G'_4 sont représentés : les points de G'_4 sont en gris.

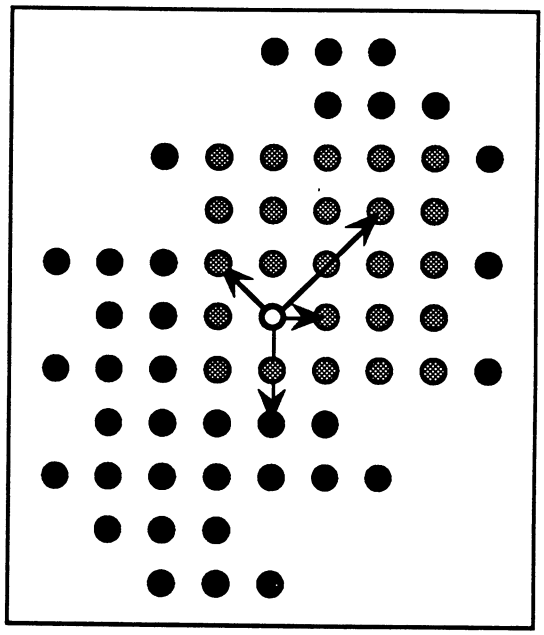


Figure 3 : G_4 et G'_4.

D'après les résultats du premier chapitre, on sait effectuer une addition sans propagation de retenue en base 4 avec les chiffres dans l'ensemble {-1, ..., 3}, on pourra donc effectuer une addition en temps constant en base (i-1) avec les chiffres dans {-1, 0,1}. on procède comme suit :

on veut additionner X et Y, $X = \sum_{k=0}^{4n-1} x_k \beta^k$ et $Y = \sum_{k=0}^{4n-1} y_k \beta^k$.($\beta$ = i-1)

On pose $X=\sum_{k=0}^{n-1}(-1)^k X_k 4^k$ avec $X_k=\sum_{j=0}^{3} x_{j+4k}\beta^k$, et $Y=\sum_{k=0}^{n-1}(-1)^k Y_k 4^k$ avec $Y_k=\sum_{j=0}^{3} y_{j+4k}\beta^k$.

comme les $X_k$et les Y_k peuvent être écrits comme somme de 5 éléments de G'_4, calculer X+Y revient à faire la somme de 10 nombres écrits en base 4 avec les chiffres dans G'_4.

On peut maintenant remarquer qu'écrire en base 4 avec les chiffres dans G'_4 est équivalent à écrire séparément la partie réelle et la partie imaginaire en base 4 avec les chiffres dans l'ensemble {-1, ..., 3}et on sait réaliser une addition en temps constant dans ce système. On écrit en temps constant X+Y sous la forme suivante :

$X+Y=\sum_{k=0}^{n+3} S_k 4^k = \sum_{k=0}^{n+3}(-1)^k S_k(-4)^k$ et comme $s_k \in G'_4$, S_k et $-S_k$ sont dans G_4. Il suffit maintenant d'écrire en temps constant les Sk sous la forme $S_k=\sum_{j=0}^{3} s_{j+4k}\beta^k$, ce qui peut se faire par une simple lecture de table.

Cependant cet algorithme est trop compliqué pour pouvoir en faire une implantation matérielle satisfaisante. L'écriture séparée des parties imaginaires et réelles en base 4 avec les chiffres dans l'ensemble {-3, ...,3}, est plus compacte et plus pratique.

IV.1.D Autre approche possible

Les représentations que nous venons de voir utilisent une base complexe et des chiffres entiers, voyons ce que donnerait l'approche duale, à savoir utiliser une base réelle et des chiffres complexes : c'est ce que nous faisons dans la suite de ce chapitre : on s'intéresse au cas d'une représentation avec une base réelle dans laquelle les chiffres sont les racines $n^{èmes}$ de l'unité plus zéro. On appellera ce type de système *représentation polygonale*. Dans une première partie, on étudie le cas de la base 2, puis on essaie de donner des bornes sur les valeurs possibles de la base.

Ensuite on s'intéresse plus particulièrement à la représentation hexagonale binaire qui donne un codage efficace des nombres complexes et qui permet de réaliser des additions en temps constant.

On propose enfin des algorithmes matériels de calcul en-ligne en représentation hexagonale binaire.

IV.2 Représentation polygonale binaire

Nous allons maintenant travailler en base 2, les chiffres utilisés seront les racines $n^{ièmes}$ de l'unité plus zéro, soit l'ensemble

$$D_p = \{0, 1, \omega_n, (\omega_n)^2, \ldots, (\omega_n)^{p-1}\} \text{ où } \omega_n = e^{\frac{2i\pi}{n}}$$

Chaque fois qu'il n'y aura pas de confusion possible, on notera $\omega = \omega_n$ pour simplifier les écritures. De même, on notera $\alpha = \frac{2i\pi}{n}$.

On va chercher pour quelles valeurs de n on peut représenter tous les nombres complexes.

IV.2.A n =1

On retrouve l'écriture classique en base 2, avec les chiffres dans {0,1}, dite "numération simple de position". Nous ne reviendrons pas dessus; on rappelle juste qu'elle permet de représenter tous les réels positifs ou nuls.

IV.2.B n = 2

comme pour p=1, on retombe sur quelque chose de classique : c'est l'écriture en base 2 en "chiffres signés". On peut représenter tous les réels.

IV.2.C n =3

L'ensemble des points représentables a une structure fractale, la dimension fractale semble être 2 mais on ne peut représenter aucun disque centré sur zéro. Pis encore, la somme ou le produit de 2 nombres représentables peut ne pas être représentable (même sur un nombre infini de chiffres)

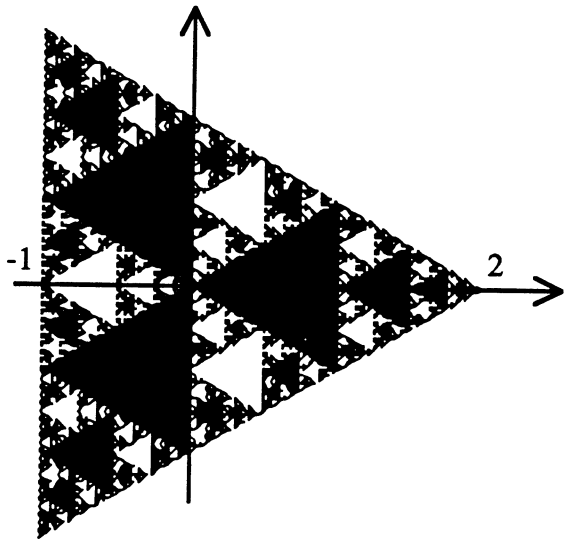


Figure 4 : nombres représentables en base 2 sous la forme $0.a_1a_2a_3\ldots$ avec les chiffres 0, 1, j et j^2.

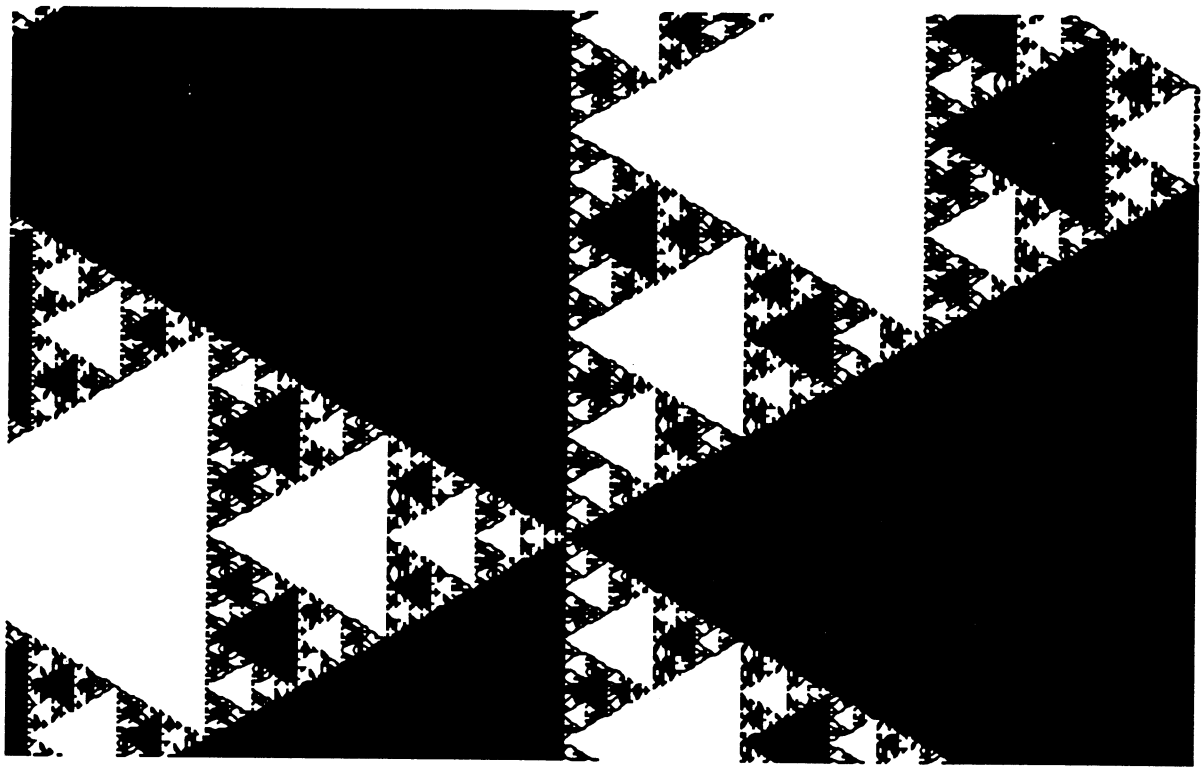


Figure 5 : agrandissement de la figure 3 mettant en évidence la structure fractale de l'ensemble.

IV.2.D $n \geq 4$

Si n est supérieur ou égal à 4, on peut représenter tous les nombres complexes. Pour montrer ceci, on construit un algorithme permettant de trouver l'écriture d'un complexe donné, en s'inspirant de l'algorithme d'écriture d'un réel en base 2. On va par des considérations géométriques chercher sous quelles conditions cet algorithme peut s'appliquer.

Comme on peut déduire de façon triviale l'écriture de $2^k.X$ de l'écriture de X en base 2 si k est entier, on limite sans perte de généralité la démonstration à la décomposition d'un complexe appartenant à $E(D_n)$, l'enveloppe convexe de D_n (c'est le n-gone de rayon 1).

Algorithme : (on décompose $X \in E(D_n)$)

$X_0 = X$

si X_i est dans $E(D_n)$, on choisit d_{i+1} dans D_n tel que $2.X_i - d_{i+1}$ soit dans $E(D_n)$, (A)

et on pose alors $X_{i+1}=2.X_i - d_{i+1}$.

On a toujours $X=\sum_{i=1}^{k} d_i 2^{-i} + X_k 2^{-k}$, et comme X_k est dans $E(D_n)$, $|X_k 2^{-k}|$ tend vers 0 quand k tend vers l'infini et donc $X=\sum_{i\geq 1} d_i 2^{-i}$ et les d_i sont les chiffres d'une écriture de X.

On voit que l'algorithme ne fonctionne que si l'étape (A) est possible, ce qui revient à dire que la réunion des n+1 n-gones de rayon 1 centrés sur les éléments de D_n recouvre le n-gone de rayon 2 centré en 0. En effet, si X_i est dans $E(D_n)$, $2.X_i$ est dans le n-gone de rayon 2. Et donc comme on le voit sur les figures 7 et 8, on peut toujours choisir d_i tel que $2.X_i - d_i$ soit dans $E(D_n)$. Il suffit de prendre le d_i qui minimise le module de $2.X_i - d_i$. Ce choix n'est d'ailleurs pas toujours unique.

Cela impose $\pi - \frac{2\pi}{n} \geq \frac{\pi}{2}$ (comme on le voit ci dessous), ce qui est réalisé si $p \geq 4$

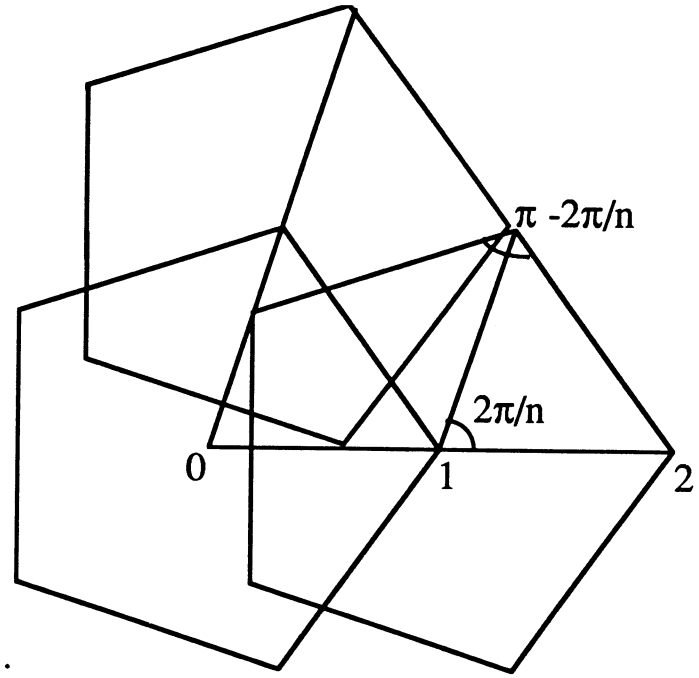


Figure 6 : condition de recouvrement.

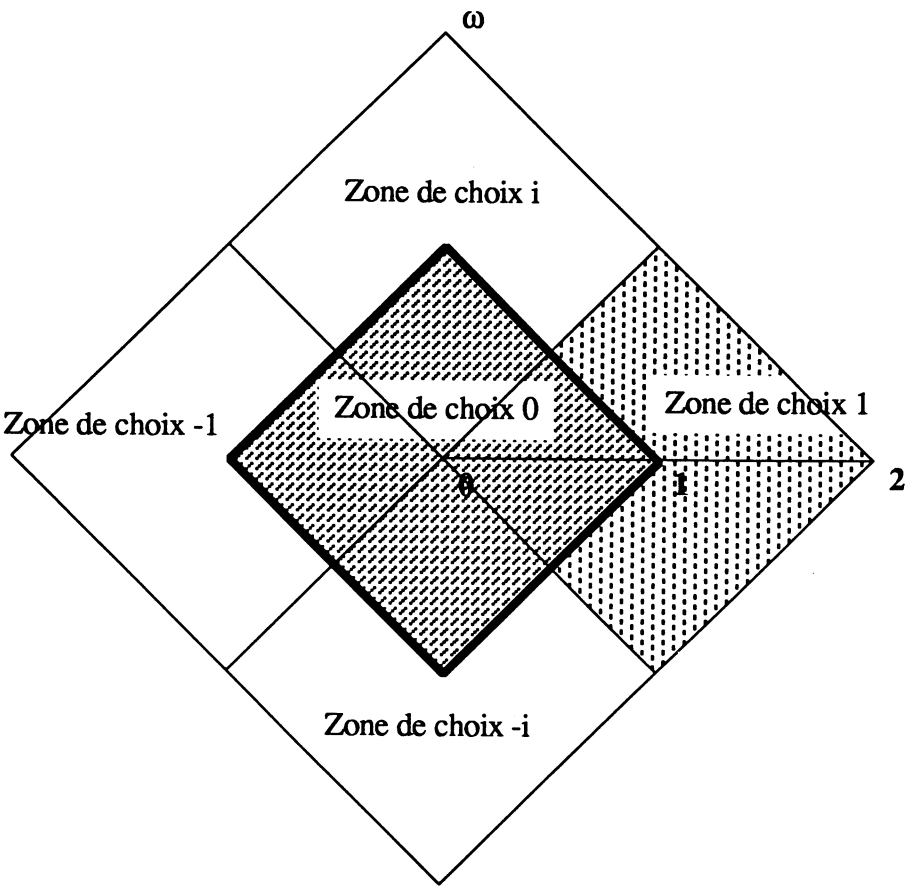


Figure 7 : choix de d_i en fonction de la valeur de $2X_i$ pour n=4.

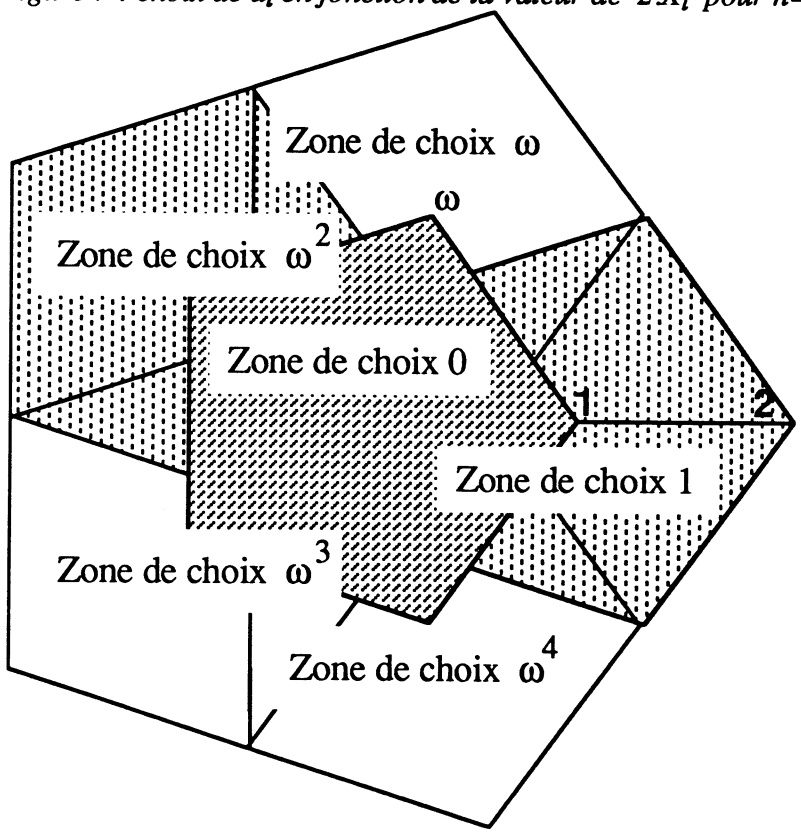


Figure 8 : choix de d_i en fonction de la valeur de $2X_i$ pour n=5.

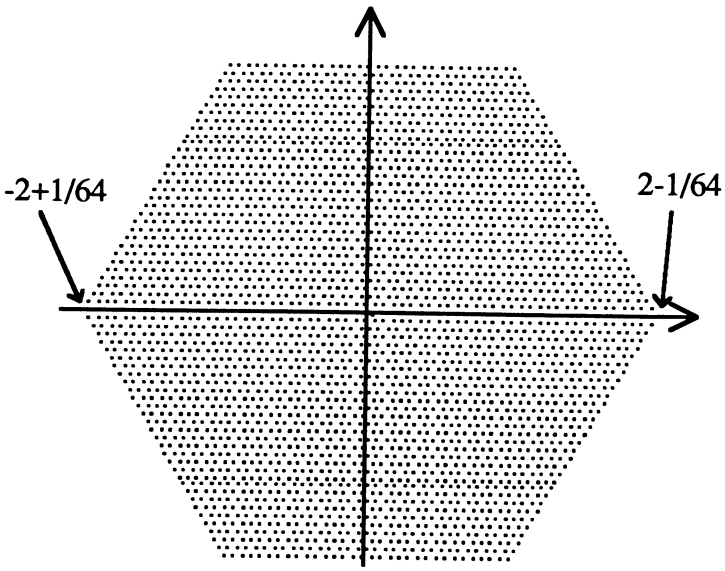


Figure 9 : ensemble $G_6(1;-6)$ en base 2.

On notera $G_n(i)$ l'ensemble des nombres représentables avec des chiffres dans D_n et les puissances de B strictement inférieures à i.
On notera $G_n(i,j)$ l'ensemble des nombres représentables avec des chiffres dans D_n et les puissances de B strictement inférieures à i et supérieures ou égales à j.

IV.3 Base limite d'une représentation polygonale

On peut se poser la question suivante : en prenant les chiffres dans D_n ($n \geq 4$), pour quelles valeurs de B pourra-t-on représenter tous les nombres complexes ?
On peut reformuler la question comme suit : pour quelles valeurs de B $G_n(0)$ contient il un disque de rayon non nul centré en 0 ?

Pour trouver un minorant des valeurs admissibles de B on cherchera des algorithmes d'écriture des complexes. On peut d'ailleurs remarquer que l'algorithme vu précédemment apporte un élément de réponse en effet, si $B \leq 2$, les n+1 n-gones de rayon 1 centrés sur les éléments de D_n recouvrent le n-gone de rayon B centré en 0.

Pour trouver un majorant des valeurs admissibles de B, on utilisera le résultat suivant :
Lemme 1 : $G_n(i)$ est inclus dans le n-gone de rayon $\frac{B^i}{B-1}$.

Démonstration : il suffit de montrer que $G_n(0)$ est inclus dans le n-gone de rayon $\frac{1}{B-1}$.
En effet, soit x un élément de $G_n(0)$. Il peut s'écrire $x = \sum_{i\geq 0} x_i B^{-i}$, $x_i \in D_n$. Il est donc barycentre des éléments de D_n car $x = \sum_{\delta \in D_n} \delta . P(\delta)$ où $P(\delta) = \sum_{x_i = \delta} B^{-i}$.

La somme des coefficients vaut $\sum_{i\geq 0} B^{-i} = \frac{1}{B-1}$. x est donc dans l'enveloppe convexe des points $\frac{\delta}{B-1}$ où δ est dans D_n. C'est justement le n-gone de rayon $\frac{1}{B-1}$.

Ce résultat nous montre que pour B > 3, quel que soit n, on ne pourra représenter aucun disque centré en 0. Les n+1 ngones de rayon $\frac{1}{2}$ centrés sur les points de D_n sont tous disjoints.

Des essais numériques ainsi que l'expérience de la numération usuelle (chiffres entiers) nous ont amené à énoncer la conjecture suivante :
Conjecture : pour tout n ≥ 3, il existe une valeur limite $B_0(n)$ que l'on notera plus simplement B_0, telle que si $B < B_0$ on peut représenter tous les complexes en base B avec les chiffres dans D_n et si $B > B_0$ on ne peut représenter aucun disque centré en 0.
De plus $B_0(n)$ tend vers 3 quand n tend vers l'infini.

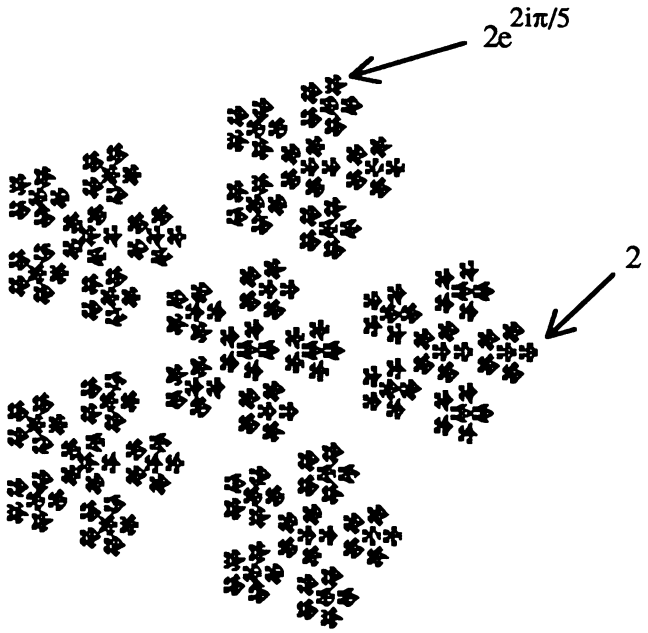


Figure 10 : ensemble des nombres représentables en base 3 avec comme chiffres 0 et les racines 5èmes de l'unité.

IV.3.A Minorant de B_0

Pour minorer B_0, on essaie d'écrire tous les complexes en base B avec les chiffres dans D_n. Pour cela, on veut utiliser un algorithme de décomposition d'un complexe semblable à l'algorithme vu précédemment (en IV.2.D). On cherche d'abord pour quelles valeurs de B on peut trouver un réel positif r tel que la réunion des n+1 disques de rayon r centrés sur les points de D_n recouvre le disque de rayon B.r centré en zéro. On en déduit ensuite un algorithme d'écriture des complexes.

Proposition 1 : Si $B \leq 1+2\cos\left(\frac{2\pi}{n}\right)$ alors la réunion des disques de rayon $r = \frac{1}{2.\cos(\alpha/2)}$ centrés sur les éléments de D_n recouvre le disque de rayon B.r centré en zéro ($\alpha = \frac{2\pi}{n}$).

Démonstration :

r est le module de l'orthocentre H de $\{0,1,\omega\}$ ($\omega = e^{i\alpha}$) ($r = |H| = |H-1| = |H-\omega|$). Les disques de rayon r centrés en 0,1 et ω recouvrent donc le triangle $\{0,1,\omega\}$.Notons H' le symétrique de H par rapport à la droite passant par 1 et ω. la réunion des disques de rayon r centrés sur les éléments de D_n recouvre le disque de rayon $r' = |H'|$.

Notons L et L' les projections de H et H' sur l'axe des réels parallèlement à $1-\omega$ (voir figure 11). $\frac{r'}{r}$ est égal à $\frac{L'}{L}$.

$L'-1 = 1-L$, soit $L' = 2-L$. Donc $\frac{L'}{L} = \frac{2}{L} - 1$, et puisque L est égal à $\frac{1}{2.\cos(\alpha/2)^2}$, on trouve que $\frac{L'}{L} = 4.\cos(\alpha/2)^2 - 1 = 1+2.\cos(\alpha)$. Donc $r' = r(1+2.\cos(\alpha))$.

Si $B \leq 1+2.\cos(\alpha)$, alors $B.r \leq r'$, et le disque de rayon B.r centré en 0 est recouvert par la réunion des disques de rayon r centrés sur les éléments de D_n.

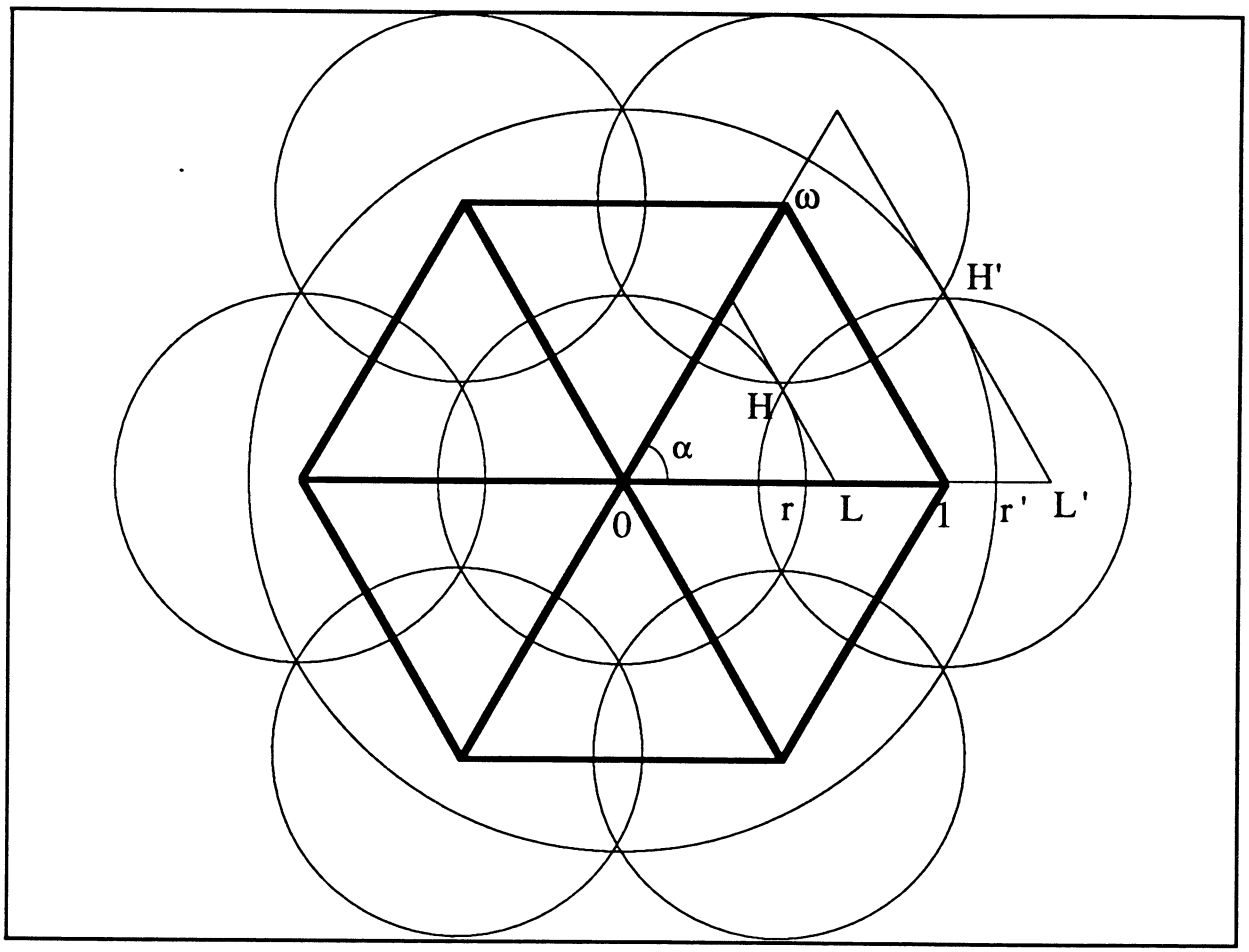


Figure 11 : recouvrement du disque de rayon r' par les disques de rayon r.

Proposition 2 : Si $B \leq 1+2\cos\left(\frac{2\pi}{n}\right)$ alors on peut représenter tous les nombres complexes en base B avec les chiffres dans D_n.

Il suffit de montrer qu'on peut écrire tous les nombres complexes contenus dans le disque de rayon B.r.
Soit X tel que $|X| \leq B.r$, on va construire une suite (d_i) d'éléments de D_n tels que $X=\sum_{i\geq 0} d_i B^{-i}$

Posons $X_0=X$ et $r = \frac{1}{2.\cos(\alpha/2)}$.

Comme le disque de rayon B.r centré en 0 est recouvert par les disques de rayon r centrés sur les éléments de D_n, si $|X_i| \leq B.r$ on peut choisir d_i tel que $|X_i - d_i| \leq r$ et donc le module de $B.(X_i - d_i)$ est inférieur ou égal à B.r. Posons alors $X_{i+1}= B.(X_i - d_i)$. Les suites (X_i) et (d_i) ainsi définies vérifient :

$$X_{k+1} = B^{k+1}.(X - d_0 - d_1.B^{-1} - \ldots - d_k.B^{-k}), \text{ soit } X - d_0 - \ldots - d_k.B^{-k} = \frac{X_{k+1}}{B^{k+1}}.$$

Donc $|X - d_0 - \ldots - d_k.B^{-k}| \leq \frac{r}{B^k}$, donc $X=\sum_{i\geq 0} d_i B^{-i}$.

On a bien construit une décomposition de X, donc X est représentable en base B.
La proposition 2 est donc démontrée.

Corollaire : Si la conjecture est vraie, alors $B_0 \leq 1+2\cos\left(\frac{2\pi}{n}\right)$

Remarque :

On a essayé de déterminer numériquement pour différentes valeurs de n, la plus grande valeur de la base telle que l'on puisse représenter le point H. Pour n pair, on a trouvé des valeurs limites supérieures à la borne précédente et des décompositions régulières. Ces expérimentations nous ont conduit au résultat suivant :

Proposition 3 :
Si n est pair, il est possible de trouver $B>1+2\cos(\alpha)$ tel que l'on puisse écrire H en base B.

Démonstration : on sépare les cas n=4.k et n=4.k+2.

Remarquons d'abord que $H + \frac{H}{\omega} = 1$. On a donc $H = \frac{\omega}{1+\omega}$

cas n=4.k : $\omega^k = i$

On va chercher B tel que l'on puisse écrire H sous la forme suivante

$$H = \frac{1}{B} + \frac{\omega}{B^2} + \ldots + \frac{\omega^{k-1}}{B^k} + \frac{\omega^k}{B^k}\left[\frac{1}{B} + \frac{1}{B^2} + \ldots\right]$$

Cela va nous donner :

$$\frac{\omega}{1+\omega}=\frac{1}{B}+\frac{\omega}{B^2}+\ldots+\frac{\omega^{k-1}}{B^k}+\frac{\omega^k}{B^k}\left[\frac{1}{B}+\frac{1}{B^2}+\ldots\right]=\frac{1}{B}\frac{1-\frac{\omega^k}{B^k}}{1-\frac{\omega}{B}}+\frac{\omega^k}{B^k(B-1)}$$

soit $\frac{\omega}{1+\omega}=\frac{1}{B}\frac{B^k-\omega^k}{B-\omega}+\frac{\omega^k}{B^k(B-1)}$

Ce qui est équivalent à $\frac{\omega.B^k}{1+\omega}=\frac{B^{k+1}-\omega^{k+1}-B^k+\omega^k}{B^2-(1+\omega)B+\omega}$,

c'est à dire $\omega B^{k+2}-(1+\omega)^2B^{k+1}+(1+\omega+\omega^2)B^k+\omega^k(\omega^2-1)=0$,

soit : $B^k(B-1)[\omega B-(1+\omega+\omega^2)]+\omega^k(\omega^2-1)=0$.

On peut réécrire cette expression pour obtenir des coefficients fonctions de α ce qui donne :

$B^k(B-1)\left[B-(1+\omega+\frac{1}{\omega})\right]+\omega^k\left(\omega-\frac{1}{\omega}\right)=0$, ou encore $B^k(B-1)\left[B-(1+2.\cos(\alpha))\right]-2.\sin(\alpha)=0$

Donc pour pouvoir écrire H sous la forme voulue, il faut et il suffit que B soit racine du polynôme ci-dessus.

On voit que ce polynôme a une racine réelle supérieure a $1+2\cos(\alpha)$.

On en déduit qu'il existe $B>1+2\cos(\alpha)$ tel que l'on puisse écrire H en base B. avec les chiffres dans D_n.

<u>cas n= 4.k+2</u> :

On va chercher B tel que l'on puisse écrire H sous la forme suivante

$$\frac{\omega}{1+\omega}=\frac{1}{B}+\frac{\omega}{B^2}+\ldots+\frac{\omega^{k-1}}{B^k}+\frac{\omega^k}{B^k}\left[\frac{1}{B}+\frac{\omega}{B^2}+\frac{1}{B^3}+\frac{\omega}{B^4}+\ldots\right],$$

soit $\frac{\omega}{1+\omega}=\frac{1}{B^k}\frac{B^k-\omega^k}{B-\omega}+\frac{\omega^k}{B^{k-1}(B^2-1)}+\frac{\omega^{k+1}}{B^k(B^2-1)}$ que l'on réécrit comme suit :

$\frac{\omega}{1+\omega}=\frac{1}{B^k}\frac{B^k-\omega^k}{B-\omega}+\frac{\omega^k}{B^{k-1}(B^2-1)}\left(1+\frac{\omega}{B}\right)$, ou bien $\frac{\omega}{1+\omega}=\frac{1}{B^k}\frac{B^k-\omega^k}{B-\omega}+\frac{\omega^k(B+\omega)}{B^k(B^2-1)}$,

ou encore $\frac{\omega}{1+\omega}=\frac{B^{k+2}-\omega^{k+2}-B^k+\omega^k}{B^k(B-\omega)(B^2-1)}$ ce qui donne :

$\omega B^{k+3}-(1+\omega+\omega^2)B^{k+2}-\omega B^{k+1}+(1+\omega+\omega^2)B^k+(1+\omega)\omega^k(\omega^2-1)=0$

On obtient finalement $B^k(B^2-1)(\omega B-(1+\omega+\omega^2))+(1+\omega)^2\omega^k(\omega-1)=0$, que l'on va réécrire pour obtenir des coefficients fonctions de α.

$$B^k(B^2-1)\left(B-\left(1+\omega+\frac{1}{\omega}\right)\right)+\left(2+\omega+\frac{1}{\omega}\right)\omega^{k+\frac{1}{2}}\left(\omega^{\frac{1}{2}}-\frac{1}{\omega^{\frac{1}{2}}}\right)=0$$

ou encore $B^k(B^2-1)(B-(1+2\cos(\alpha)))-(2+2\cos(\alpha))2\sin(\frac{\alpha}{2})=0$

De la même façon que dans le cas précédent, pour pouvoir écrire H sous la forme voulue, il faut et il suffit que B soit racine du polynôme ci-dessus, polynôme qui admet une racine réelle supérieure à $1+2\cos(\alpha)$.
On en déduit aussi qu'il existe $B>1+2\cos(\alpha)$ tel que l'on puisse écrire H en base B. avec les chiffres dans D_n.

IV.3.B Majorant de B_0

Ainsi que nous l'avons dit, 3 est un majorant de B_0 (si B_0 existe). Nous allons donner des majorations plus fines de B_0 en fonction de n. Ces majorations utilisent le fait que l'ensemble $G_n(0)$ des nombres complexes représentables avec les puissances négatives de la base est inclus dans le n-gone de rayon $\frac{1}{B-1}$.

• Majoration grossière : on a montré que $G_n(0)$ est inclus dans un n-gone de rayon $\frac{1}{B-1}$, Donc *a fortiori* dans le disque de même rayon, ce qui va nous donner la majoration suivante.

Proposition 4 : si $B > 1+2.\cos(\alpha/2)$ alors on ne peut représenter aucun disque centré en 0.

Démonstration :
Pour cela, nous allons montrer que l'orthocentre H de $\{0, 1,\omega\}$ n'est pas représentable.
On remarque immédiatement que $H\notin G_n(0)$ car $|H| = \frac{1}{2.\cos(\alpha/2)} > \frac{1}{B-1}$.
De plus $|H|=|1-H|=|\omega-H|\leq|\omega^d-H|$ pour tout d entier, donc $H\notin G_n(1)$.
Nous allons montrer que pour tout $k>0$ et pour tout d entier (ω^dB^k-H) n'est pas dans $G_n(k)$, ce qui implique que $H\notin G_n(k+1)$ et donc que H n'est pas représentable.

D'après le lemme 1, $G_n(k)$ est inclus dans le n-gone de rayon $\frac{B^k}{B-1}$ et donc si $|\omega^dB^k-H| > \frac{B^k}{B-1}$, alors $(\omega^dB^k-H)\notin G_n(k)$.

$$|\omega^dB^k-H|^2= \left|B^k(\cos(d.\alpha)+i.\sin(d.\alpha))-\frac{1}{2}(1+i.tg(\alpha/2))\right|^2$$

$$= B^{2k}\cos^2(d.\alpha) - B^k\cos(d.\alpha)+\frac{1}{4}+B^{2k}\sin^2(d.\alpha) - B^k\sin(d.\alpha).tg(\alpha/2)+\frac{1}{4}tg^2(\alpha/2)$$

$$= B^{2k} - B^k\frac{\cos((d-\frac{1}{2})\alpha)}{\cos(\alpha/2)}+\frac{1}{4.\cos^2(\alpha/2)} > B^{2k} - \frac{B^k}{\cos(\alpha/2)}+\frac{1}{4.\cos^2(\alpha/2)} = \left(B^k-\frac{1}{2.\cos(\alpha/2)}\right)^2.$$

donc $|\omega^d B^k - H| - \frac{B^k}{B-1} > B^k - \frac{1}{2.\cos(\alpha/2)} - \frac{B^k}{B-1} = B^k\frac{B-2}{B-1} - \frac{1}{2.\cos(\alpha/2)}$.

Par hypothèse, $B>1+2.\cos(\alpha/2)>1$, et de plus $\frac{X-2}{X-1}$ est croissant pour $X>1$.

Donc $|\omega^d B^k - H| - \frac{B^k}{B-1} > B\frac{B-2}{B-1} - \frac{1}{2.\cos(\alpha/2)} > \frac{(1+2.\cos(\alpha/2))(2.\cos(\alpha/2)-1)}{2.\cos(\alpha/2)} - \frac{1}{2.\cos(\alpha/2)}$

$= \frac{4.\cos(\alpha/2)^2 - 2}{2\cos(\alpha/2)} = \frac{\cos(\alpha)}{\cos(\alpha/2)}$.

Donc $|\omega^d B^k - H| - \frac{B^k}{B-1} > \frac{\cos(\alpha)}{\cos(\alpha/2)} \geq 0$ si $\alpha \leq \frac{\pi}{2}$ soit $n \geq 4$.

Donc comme pour tout d entier et pour tout $k>0$ $(\omega^d B^k - H) \notin G_n(k)$ et donc on ne pas écrire H en base B.

On ne peut pas non plus écrire $B^k H$ et on ne peut donc représenter aucun disque centré en 0.

IV.3.B.a Majoration plus fine (si n est pair et n≥6)

• première Majoration : on a montré que $G_n(0)$ est inclus dans un n-gone de rayon $\frac{1}{B-1}$, ce qui nous donne une première majoration de B_0, en remarquant que pour pouvoir représenter tous les éléments du n-gone de rayon 1 centré en 0, il faut pouvoir recouvrir ce n-gone par les n+1 n-gones de rayon $\frac{1}{B-1}$ centrés sur les éléments de D_n.

Proposition 5 : si n est pair et B strictement supérieur à $2+\cos(\frac{2\pi}{n})$, on ne peut représenter aucun disque centré en 0.

Démonstration : $G_n(0)$ est inclus dans le n-gone de rayon $\frac{1}{B-1}$. Si l'orthocentre H de $\{0,1,\omega\}$ n'est pas contenu dans ce n-gone, alors H n'est pas dans $G_n(0)$, et comme $H=\frac{\omega}{1+\omega}$, $H-1=-\frac{H}{\omega}$ et $H-\omega = -\omega H$ ne sont pas dans $G_n(0)$ non plus.

De plus on remarque que si H n'est pas dans le n-gone de rayon $\frac{B^k}{B-1}$ centré en B^k alors $\omega^d H$ n'y est pas non plus et donc $B^k - \omega^d H$ n'est pas dans $G_n(k)$ et donc $\omega^d B^k - H$ n'est pas dans $G_n(k)$.

Il suffit donc de montrer que si $B>2+\cos(\alpha)$ $(\alpha = \frac{2\pi}{n})$ alors :

i $H \notin G_n(0)$

ii $H - B^k \notin G_n(k)$ pour $k \geq 2$.

i Ainsi qu'on le voit figure 12, si $\frac{1}{B-1} < \frac{1}{2\cos(\alpha/2)^2}$, H n'est pas dans $G_n(0)$.

Or $\frac{1}{B-1} < \frac{1}{2\cos(\alpha/2)^2} \Leftrightarrow B>1+2\cos(\alpha/2)^2 \Leftrightarrow B>2+\cos(\alpha)$.

Donc si $B>2+\cos(\alpha)$ alors H n'est pas dans $G_n(0)$.

ii De même, si $\frac{B-2}{B-1} > 1-\frac{1}{2\cos(\alpha/2)^2}$, alors $B^k\frac{B-2}{B-1} > 1-\frac{1}{2\cos(\alpha/2)^2}$ et $H-B^k \notin G_n(k)$.

Or $\frac{B-2}{B-1} > 1-\frac{1}{2\cos(\alpha/2)^2} \Leftrightarrow \frac{1}{B-1} < \frac{1}{2\cos(\alpha/2)^2} \Leftrightarrow B>2+\cos(\alpha)$.

Donc si $B>2+\cos(\alpha)$ alors $H-B^k$ n'est pas dans $G_n(k)$.

Donc si $B>2+\cos(\alpha)$ alors H n'est pas représentable et par homothétie B^kH non plus.

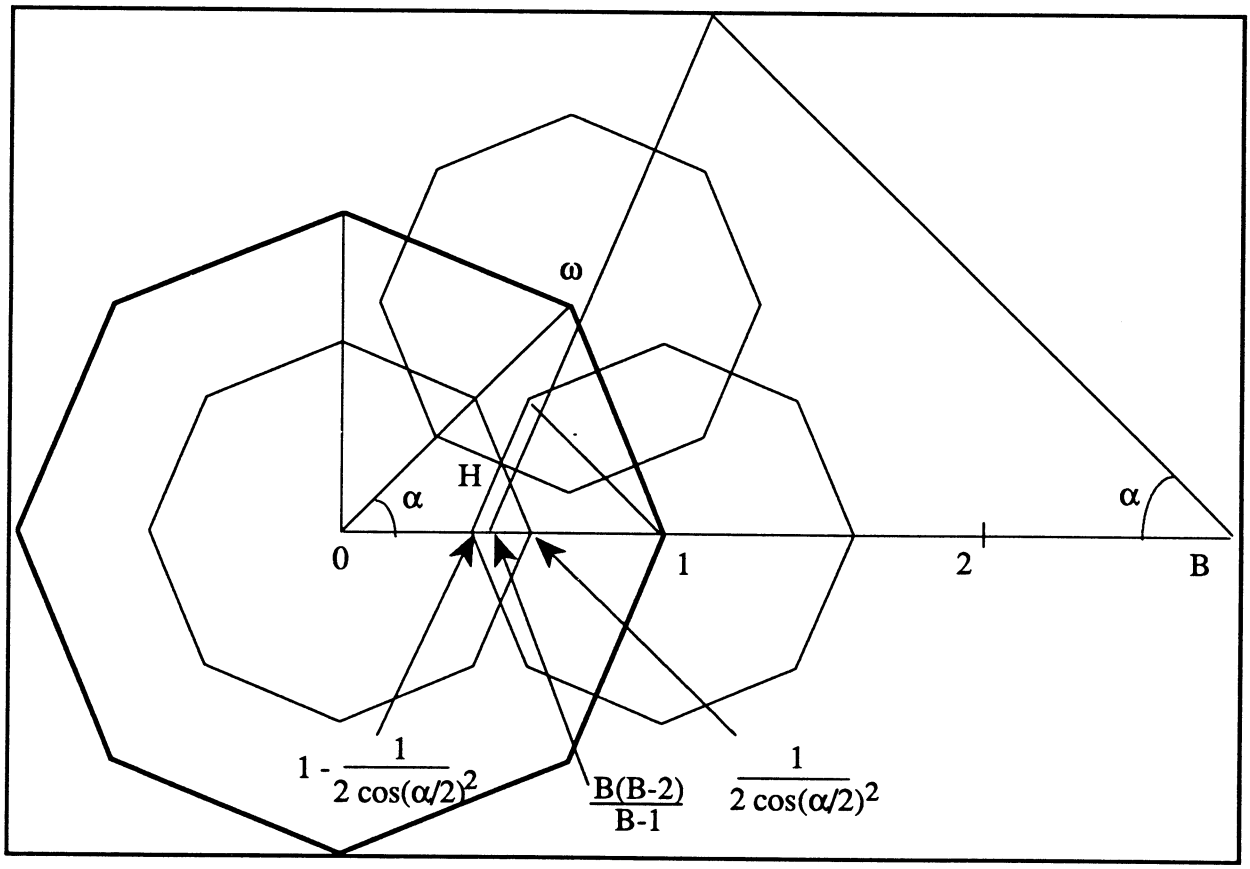


Figure 12 : recouvrement de H.

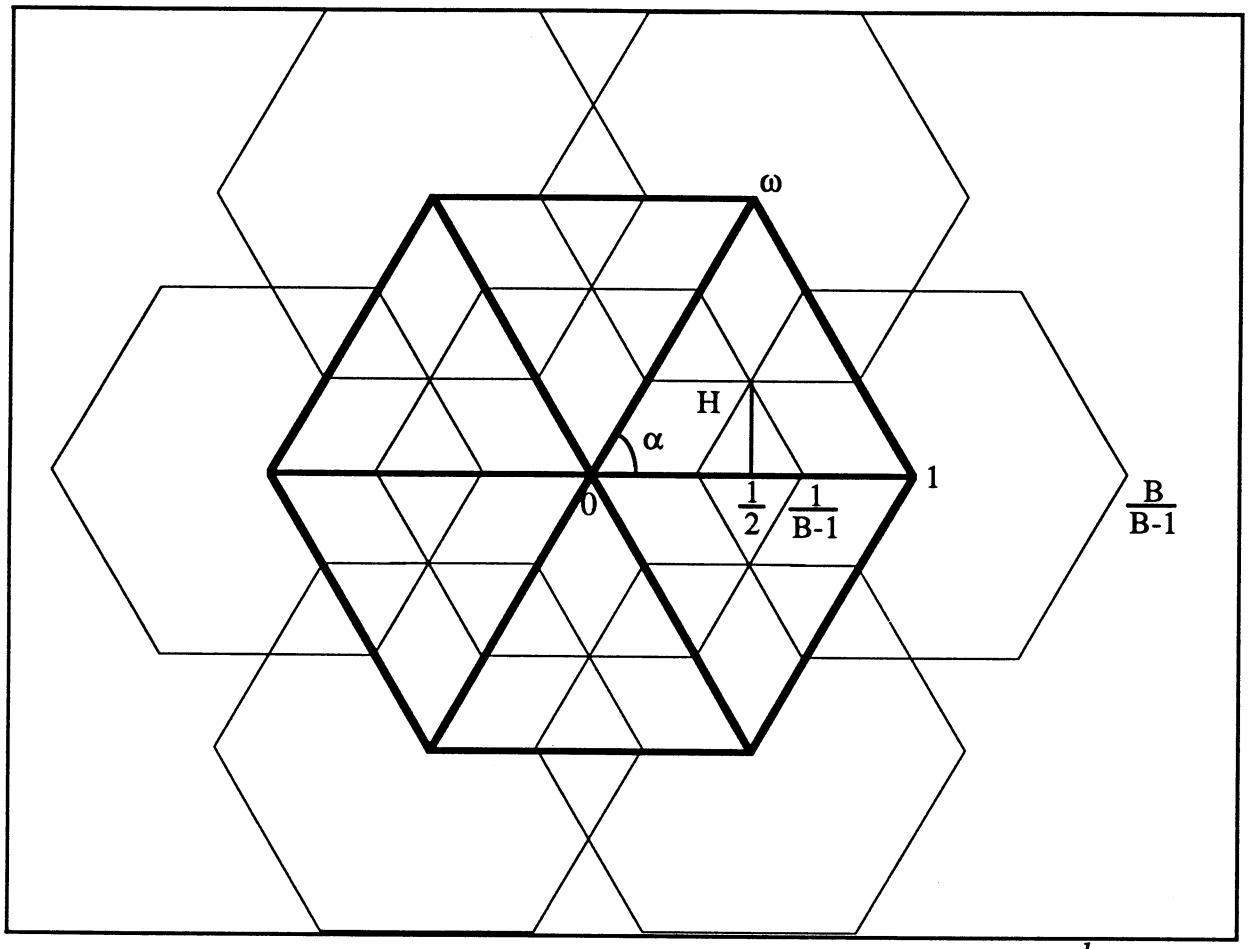


Figure 13 : recouvrement du n-gone de rayon 1 par n+1 n-gones de rayon $\frac{1}{B-1}$.

• Majoration plus fine : On utilise le fait que si $G_n(0)$ est inclus dans un n-gone de rayon $\frac{1}{B-1}$ centré en 0, alors $G_n(1)$ est inclus dans la réunion de n+1 n-gones centrés sur les points de D_n, ce qui va nous donner une majoration plus fine. Cela revient à recouvrir le n-gone de rayon B par des n-gones de rayon $\frac{1}{B-1}$ centrés sur les éléments de la forme $d_0+d_1.B$ où d_0 et d_1 sont dans D_n. Le point qui va poser problème est maintenant le point B.H.

Proposition 6 :

Si BH n'est pas élément de la réunion des n+1 n-gones de rayon $\frac{1}{B-1}$ centrés sur les éléments de D_n, alors H n'est pas représentable en base B.

Démonstration :

Si BH n'est pas élément de la réunion des n+1 n-gones de rayon $\frac{1}{B-1}$ centrés sur les éléments de D_n, alors BH n'est pas représentable en base B avec les puissance négatives ou nulles de B. Par homothétie H n'est pas représentable en base B avec les puissance strictement négatives de B, et d'après ce qu'on a vu précédemment, on ne peut pas représenter H en base B.

Voyons ce que cela entraîne sur B. La figure 14 détaille la partie du plan qui nous intéresse.

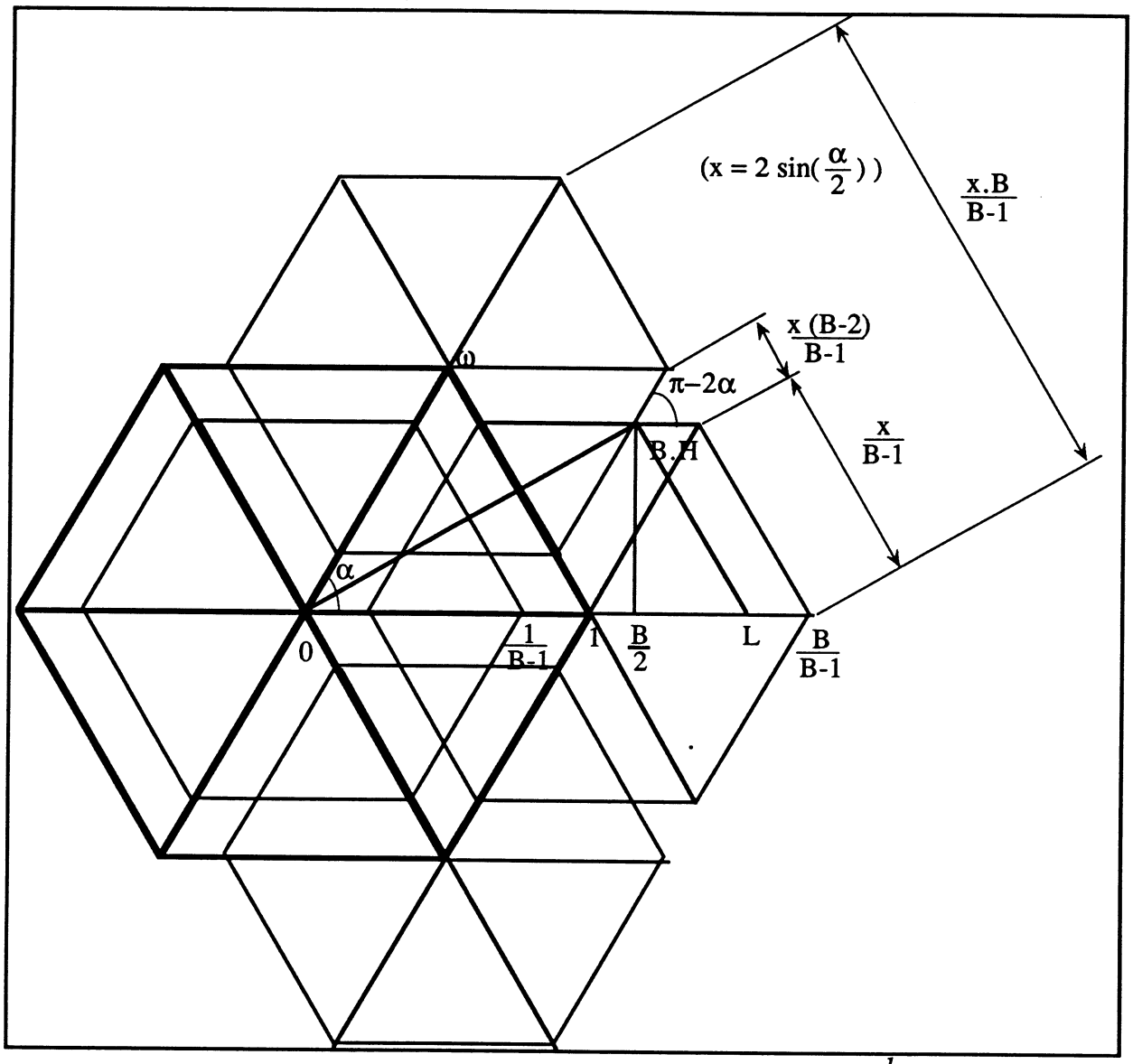


Figure 14 : recouvrement de H par des n-gones de rayon $\frac{1}{B-1}$.

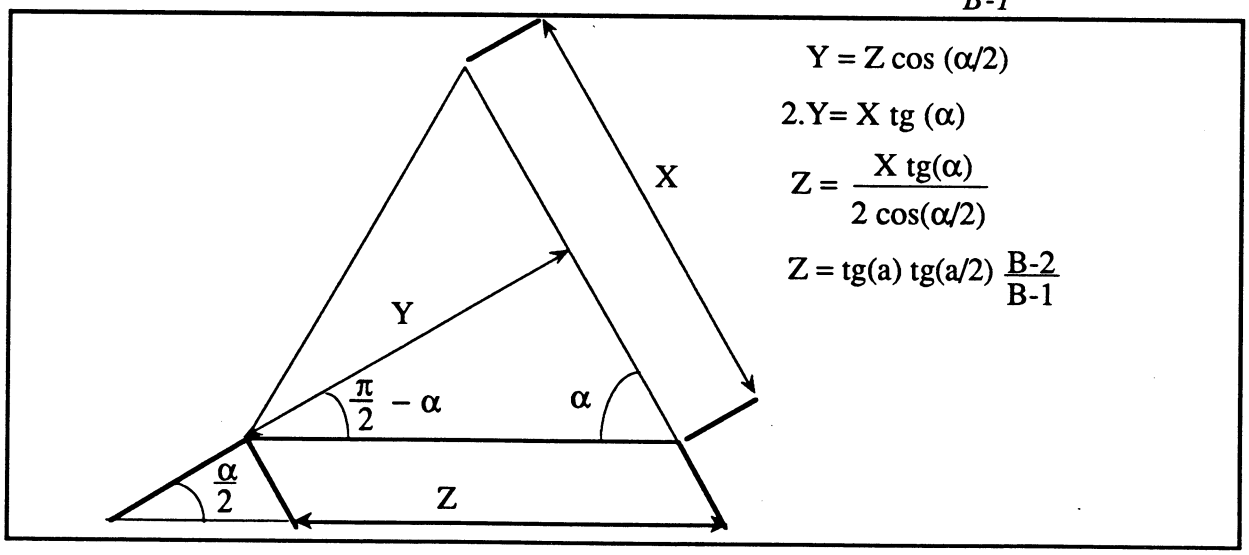


Figure 15 : détail de la figure 14.

BH n'est pas représentable si

$\frac{B}{B-1} - \frac{B-2}{B-1} tg(\alpha)\, tg(\frac{\alpha}{2}) < \frac{B}{2\cos(\frac{\alpha}{2})^2} = L$ ce qui nous donne

,

$$B(B-1) > B\left(2\cos\left(\frac{\alpha}{2}\right)^2\right) - (B-2)\,2\cos\left(\frac{\alpha}{2}\right)\sin\left(\frac{\alpha}{2}\right) tg(\alpha),$$

soit $B^2 - B - B(1+\cos(\alpha)) + (B-2)\sin(\alpha)\, tg(\alpha) > 0;,$

soit $P(B) = B^2 - B\,(2+\cos(\alpha)-\sin(\alpha)\, tg(\alpha))) - 2\sin(\alpha)\, tg(\alpha) > 0$

Comme $P(2+\cos(\alpha)) = \sin(\alpha)^2 > 0$, les racines de P sont inférieures à $2+\cos(\alpha)$, on a donc bien une majoration plus fine de B_0. Explicitons la racine :

$$P(B) > 0 \Leftrightarrow \cos(\alpha)B^2 - B\,(2\cos(\alpha)+\cos(\alpha)^2-\sin(\alpha)^2) - 2\sin(\alpha)^2 > 0$$

$$\Leftrightarrow \cos(\alpha)B^2 - B\,(2\cos(\alpha)+2\cos(\alpha)^2-1) + 2 - 2\cos(\alpha)^2 > 0$$

donc si $B > B_1 = \frac{2\cos(\alpha)+2\cos(\alpha)^2 -1+\sqrt{4\cos^4(\alpha)+4\cos(\alpha)+1}}{2.\cos(\alpha)}$, on ne peut représenter aucun disque centré en 0.

On peut réécrire notre condition à l'aide de ω, ce qui donne :

$$\frac{1}{2}(\omega+\frac{1}{\omega})\,B^2 - B\,(\omega+\frac{1}{\omega}+\frac{1}{2}(\omega^2+\frac{1}{\omega^2}))+\frac{1}{2}(\omega^2+\frac{1}{\omega^2})>0$$

$$\text{soit } (\omega^3+\omega)B^2 - B\,(\omega^4+2\omega^3+2\omega+1) + \omega^4-2\omega^2+1 > 0$$

Remarque : pour n=6 et n=8, la borne supérieure donnée ci dessus correspond à la valeur de B pour laquelle on a exhibé une décomposition de H.

• si n=6 : B_0 est majoré par la plus grande racine du polynôme ci-dessous

$$(\omega^3+\omega)B^2 - B\,(\omega^4+2\omega^3+2\omega+1) + \omega^4-2\omega^2+1 = \omega^2 B^2 - \omega^2 B - 3\,\omega^2 = \omega^2 (B^2 - B - 3)$$

et on a vu que l'on peut exhiber une décomposition de H si B est racine du polynôme suivant :

$$B(B^2-1)(\omega B-(1+\omega+\omega^2)) +(1+\omega)^2\omega(\omega-1)$$

cette décomposition est $H=\frac{1}{B}+\frac{\omega}{B^2}+\frac{\omega^2}{B^3}+\frac{\omega}{B^4}+\frac{\omega^2}{B^5}+\frac{\omega}{B^6}+\frac{\omega^2}{B^7}+\dots$

or $B(B^2-1)(\omega B-(1+\omega+\omega^2))+(1+\omega)^2\omega(\omega-1)$ est divisible par $B^2 - B - 3$.

en effet :

$B(B^2-1)(\omega B-(1+\omega+\omega^2))+(1+\omega)^2\omega(\omega-1)$

$= \omega B^4 - (1+\omega+\omega^2)B^3 - \omega B^2 - (1+\omega+\omega^2)B + \omega^4+\omega^3 - \omega^2 - \omega = \omega(B^4 - 2 B^2 -B^2+ 2B - 3)$

$= \omega (B^2 - B - 3)(B^2 - B + 1)$

Donc $B = \frac{1+\sqrt{13}}{2} = 2.30277\dots$ (solution de $B^2 -B -3 =0$) est un majorant de B_0 et on peut représenter H en base B. Donc, $\frac{1+\sqrt{13}}{2}$ semble être un bon candidat pour B_0 dans le cas n=6

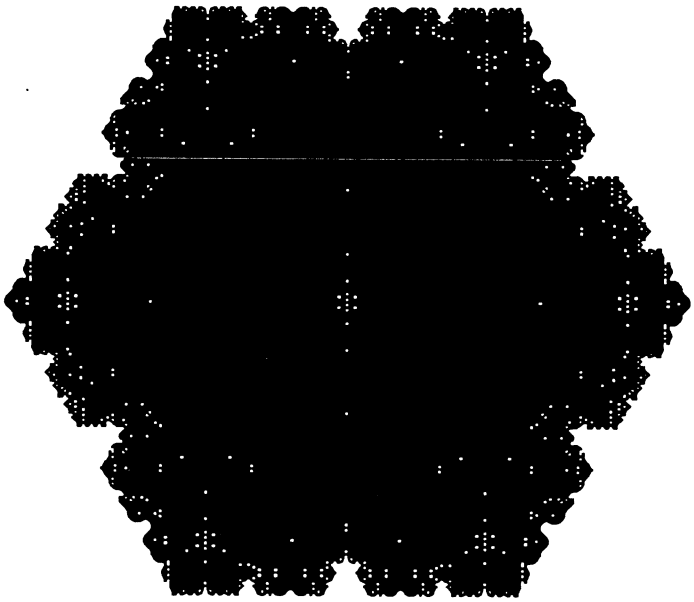


Figure 16 : ensemble des nombres représentables avec les puissances négatives de la base pour B=2.302 avec les chiffres dans D_6.

• Si n=8 : La majoration fine de B_0 donne comme majorant la plus grande racine du polynôme

$(\omega^3+\omega)B^2 - B(\omega^4+2\omega^3+2\omega+1) + \omega^4-2\omega^2+1 = (\omega^3+\omega)B^2 - 2B(\omega^3+\omega) -2\omega^2$

$= \sqrt{2}\ \omega^2 (B^2 -2B - \sqrt{2})$,

et on sait exhiber une décomposition de H pour B racine de $B^2(B-1)(\omega B-(1+\omega+\omega^2))+\omega^2(\omega^2-1)$

La décomposition est : $H=\frac{1}{B}+\frac{\omega}{B^2}+\frac{\omega^2}{B^3}+\frac{\omega^2}{B^4}+\frac{\omega^2}{B^5}+\dots$

Or $B^2(B-1)(\omega B-(1+\omega+\omega^2))+\omega^2(\omega^2-1)$ est divisible par $B^2 -2B - \sqrt{2}$. En effet :

$B^2(B-1)(\omega B-(1+\omega+\omega^2)) +\omega^2(\omega^2-1) = \omega B^4 - (1+\omega)^2 B^3 + (1+\omega+\omega^2)B^2 + \omega^4 - \omega^2$

$= \omega(B^4 - (2+\sqrt{2})^2 B^3 + (1+\sqrt{2} B^2 - \sqrt{2}) = \omega (B^2 -2B - \sqrt{2})(B^2 + \sqrt{2} B +1)$

on a vu une décomposition de H pour $B = 1+\sqrt{1+\sqrt{2}} = 2.55...$
Donc $B = 1+\sqrt{1+\sqrt{2}} = 2.55...$ (solution de $B^2 -2B - \sqrt{2} =0$) est un majorant de B_0 et on peut représenter H en base B. Donc, $1+\sqrt{1+\sqrt{2}}$ semble être un bon candidat pour B_0 dans le cas n=8.

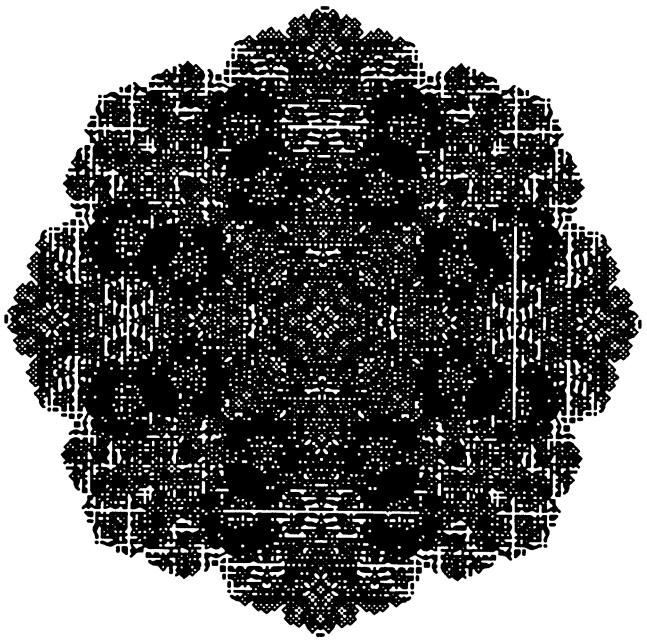


Figure 17 : ensemble des nombres représentables avec les puissances négatives de la base pour B=2.55 avec les chiffres dans D_8.

Remarque : pour n pair et $B \leq 3$, alors l'ensemble $G_n(0)$ des points représentables avec les puissances de B strictement négatives est connexe.
Démonstration : On va montrer que tout nombre de $G_n(0)$ est relié à 0.
Soit un nombre x de $G_n(0)$, $x = \sum_{i>0} x_i.B^{-i}$. On va poser $y_0 = 0$ et $y_k = \sum_{i=1}^{k} x_i.B^{-i}$.
On peut représenter tous les nombres du segment reliant y_i à y_{i+1}. Comme $y_{i+1}=y_i+x_i.B^{-i}$, il suffit de montrer que le segment reliant 0 a $x_i.B^{-i}$ est inclus dans $G_n(-i+1)$. Comme x_i est dans D_n il suffit de se ramener au cas $x_i = 1$ et de montrer que $[0,B^{-i}] \subset G_n(-i+1)$. $-1 \in D_n$ car n est pair, et donc {-1,0,1} est inclus dans D_n. Or B est inférieur ou égal à 3 et on sait alors que l'on peut représenter tous les chiffres de $[0,B^{-i}]$ en base B avec les chiffres dans {-1,0,1} et les puissances de B inférieures ou égales à -i. par conséquent, l'intervalle $[0,B^{-i}]$ est inclus dans

$G_n(-i+1)$. Si l'on multiplie tous les éléments de $[0,B^{-i}]$ par x_i, on obtient que le segment reliant 0 à x_iB^{-i} est inclus dans $G_n(-i+1)$. Donc le segment reliant y_i à y_{i+1} est inclus dans $G_n(0)$ et la réunion des segments reliant les y_i successifs est incluse dans $G_n(0)$ et relie 0 à x.
On a bien montré que tout nombre de $G_n(0)$ est relié à 0 et que par conséquent $G_n(0)$ est connexe.

IV.3.B.b Majoration plus fine (si n est impair)

On utilise les mêmes arguments que pour le cas pair, cependant comme les n-gones ne sont plus symétriques par rapport à 0, la construction géométrique faite quand n est pair ne s'applique plus.

Proposition 7 : si $B> 1+\cos\left(\frac{\alpha}{2}\right)+\cos\left(\frac{\alpha}{2}\right)^2$, on ne peut pas représenter tous les complexes.

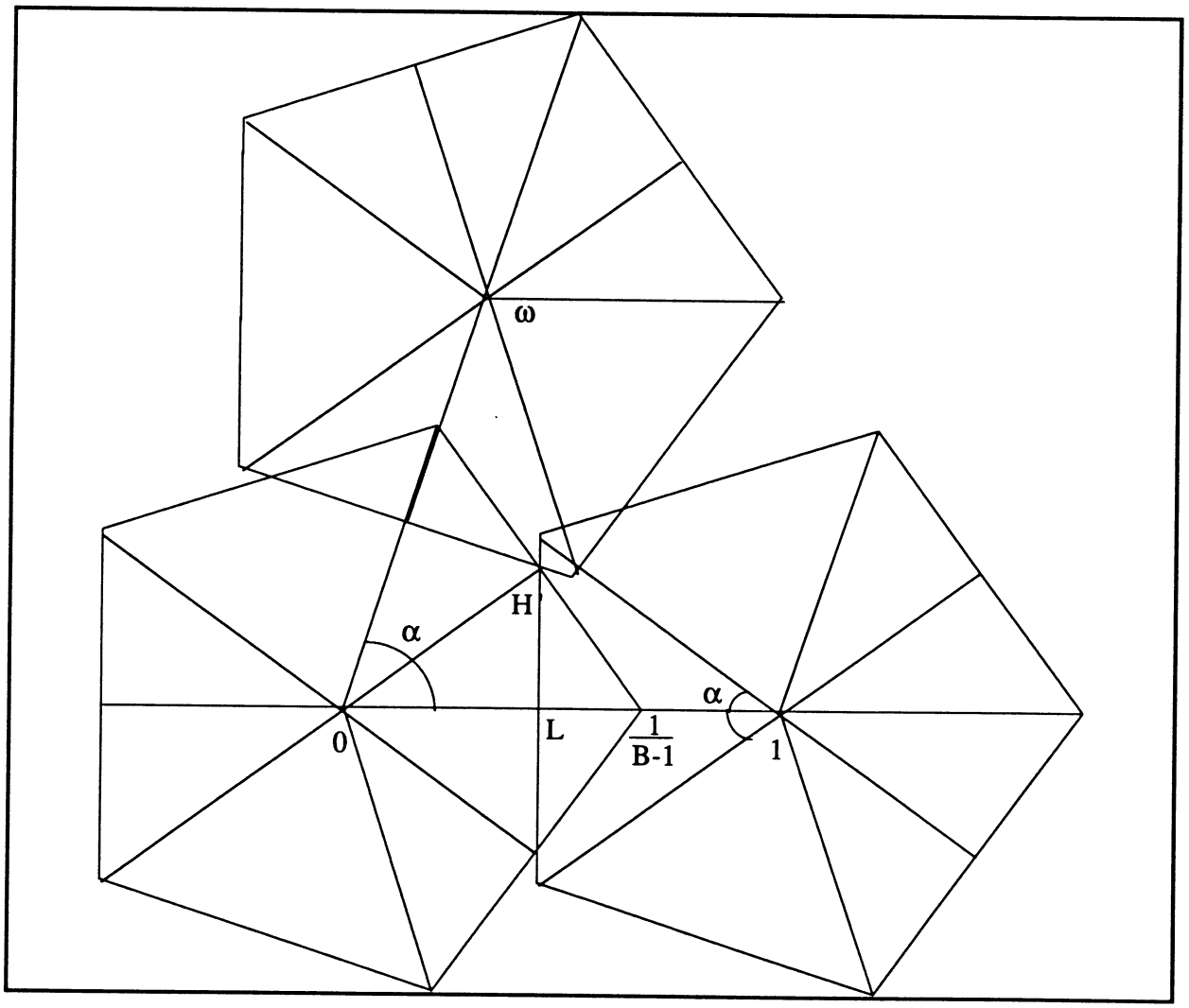


Figure 18 : recouvrement par des n-gones (cas impair).

Considérons le point H' sur la bissectrice de 1 et ω tel que le module de H' soit égal à 1-Re(H'). H' est représenté figure 18. On a $L+\frac{L}{\cos(\alpha/2)}=1$ soit $L=\frac{\cos(\alpha/2)}{\cos(\alpha/2)+1}$.

On voit que si $B^k-\cos(\alpha/2)\frac{B^k}{B-1}>L$ alors B^k-H' ne peut pas être dans $G_n(k)$ pour $k\geq 0$.

Or $B^k-\cos(\alpha/2)\frac{B^k}{B-1}=B^k(1+\frac{\cos(\alpha/2)}{B-1})>1+\frac{\cos(\alpha/2)}{B-1}$

et $1+\frac{\cos(\alpha/2)}{B-1}>L=\frac{\cos(\alpha/2)}{\cos(\alpha/2)+1}$ est équivalent à $B>1+\cos\left(\frac{\alpha}{2}\right)+\cos\left(\frac{\alpha}{2}\right)^2$.

Donc $H'\notin G_n(k)$ si $k\geq 1$.

de plus, si $B>1+\cos\left(\frac{\alpha}{2}\right)+\cos\left(\frac{\alpha}{2}\right)^2$, on a $\frac{\cos^2(\alpha/2)}{B-1}<\frac{\cos(\alpha/2)}{\cos(\alpha/2)+1}$.= L. et donc $H'\notin G_n(0)$.

Donc si $B>1+\cos\left(\frac{\alpha}{2}\right)+\cos\left(\frac{\alpha}{2}\right)^2$, H' n'est pas représentable et on ne peut représenter aucun disque centré en 0.

IV.3.C Conclusion

Pour tout n on a exhibé une valeur $B_{inf}(n)$ telle que si $B \leq B_{inf}(n)$ on peut représenter tout les complexes. De même, on a trouvé une valeur $B_{sup}(n)$ telle que si $B > B_{sup}(n)$ on ne peut représenter aucun disque centré en 0. Si la conjecture faite au début du paragraphe IV.3 est vraie, c'est à dire s'il existe B_0 tel que si $B<B_0$ on peut représenter tous les complexes et si $B>B_0$ on ne peut représenter aucun disque centré en 0, alors $B_{inf}(n) \leq B_0 \leq B_{sup}(n)$ et B_0 tend vers 3 quand n tend vers l'infini.

IV.4 Représentation hexagonale binaire

Nous allons maintenant nous intéresser à l'écriture en base 2 avec les chiffres pris dans D_6. Nous allons voir que cette écriture nous donnera un codage compact et élégant des nombres complexes avec de plus la possibilité de réaliser des additions en temps indépendant de la longueur des opérandes, ou des additions en-ligne (en série, poids forts en tête).

Posons H(1) = D_6. Nous noterons ensuite H(n) l'ensemble des points qui s'écrivent comme somme de n éléments de H(1). Si on note H(m)+H(n) = { x+y, x ∈ H(m), y ∈ H(n) } on a immédiatement H(m)+H(n)=H(n+m).

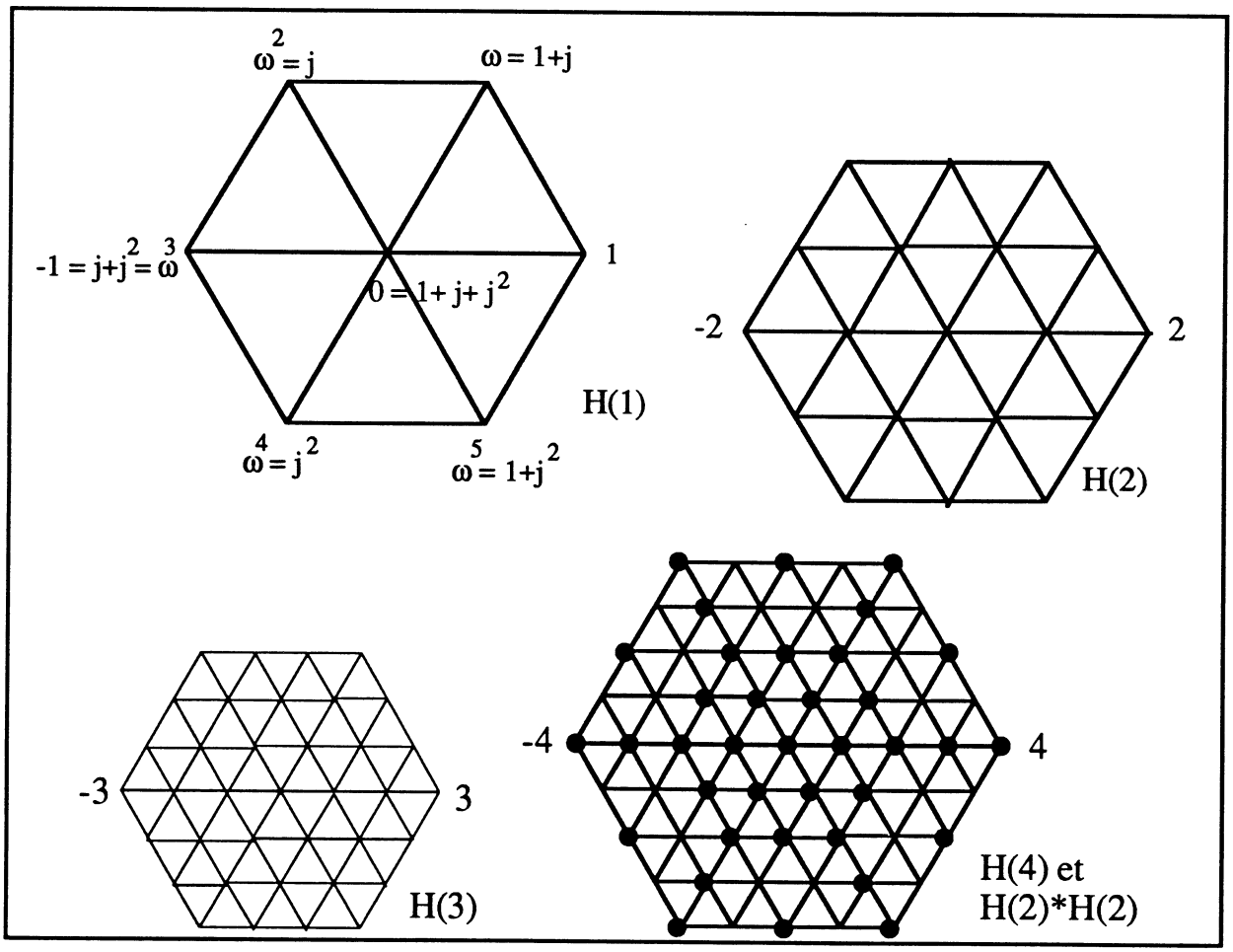


Figure 19 : exemples d'ensemble H(n).

Il viendra tout aussi immédiatement H(m)*H(n) = { xy, x ∈ H(m), y ∈ H(n) } ⊂ H(m*n)

Enfin on note H(∞) = H. H est en fait le "maillage" du plan complexe par des triangles équilatéraux de coté de longueur 1. En fait H est un ensemble bien connu aux propriétés intéressantes, ainsi que nous allons le voir

IV.4.A Résultats préliminaires sur H

L'ensemble H est en fait $\mathbf{Z}[j]$ (où $j = e^{\frac{2i\pi}{3}}$). H est égal à $\mathbf{Z} + j\,\mathbf{Z} = \mathbf{Z} + j^2\,\mathbf{Z} = j\,\mathbf{Z} + j^2\,\mathbf{Z}$.

On peut aussi considérer H comme l'ensemble des polynômes de $\mathbb{Z}[X]$ modulo X^2+X+1
Rappelons quelques propriétés de H

- H est un anneau unitaire commutatif.
- H est un anneau euclidien.

Rappel : un anneau A est euclidien s'il existe une fonction λ de $A-\{0\}$ dans IN telle que pour tout (x,y) de A^2 (y non nul), il existe un couple (q,r) de A^2 tel que
$x=y\,q+r$ et ($\lambda(r) < \lambda(y)$ ou $r = 0$)

Démonstration : Il suffit de poser $\lambda(x) = |x|^2 = x\bar{x}$
si $x = a+bj$, $\lambda(x) = |a+bj|^2 = |a+bj^2|^2 = (a+bj)(a+bj^2) = a^2+b^2 - ab$

- Division euclidienne dans H :

Soient x et y deux éléments de H : $x= a+jb$ et $y=a'+jb'$
Il existe q et r de H tels que $x = qy +r$ et $|r|^2 < |y|^2$. En effet :

$$\frac{x}{y} = \frac{a+jb}{a'+jb'} = \frac{(a+jb)(a'+j^2b')}{(a'+jb')(a'+j^2b')} = \frac{aa'+bb'-ab'+(ba'-ab')\,j}{a'^2+b'^2 - a'b'} = A + B\,j$$

on peut trouver 2 entiers α et β tels que $|A - \alpha| \le \frac{1}{2}$ et $|B - \beta| \le \frac{1}{2}$

donc $\left|\frac{x}{y} - (\alpha+\beta j)\right| = (A - \alpha)^2+(B - \beta)^2-(A - \alpha)(B - \beta) \le \frac{3}{4}$

et $\left|x-y(\alpha+\beta j)\right| \le \frac{3}{4}|y| < |y|$

il suffit de poser $q = \alpha + \beta\, j$ et $r = x- qy$.

Le groupe unité de H est formé des racines $6^{èmes}$ de 1.

Rappels :

1) Deux entiers x et y de H sont dits associés si $x=\eta y$ où η fait partie du groupe unité.
2) Un entier x de H est dit premier si tous ses diviseurs sont des unités ou lui sont associés.
3) Comme H est un anneau euclidien, tout entier de H admet une décomposition unique en produit de facteurs premiers.
4) Un entier x de H dont le module au carré est premier dans Z est premier dans H :
en effet si $|x|^2 = p$ où p est premier et si $x = y.z$, alors $|y|^2\,|z|^2 = p$ donc $|y|^2$ ou $|z|^2$ vaut 1 et donc y ou z fait partie du groupe unité. Par conséquent, x est premier.
Par exemple $1+\omega = 1-j$ et $|1+\omega|^2 = 3$. 3 est premier, donc $1+\omega$ est premier

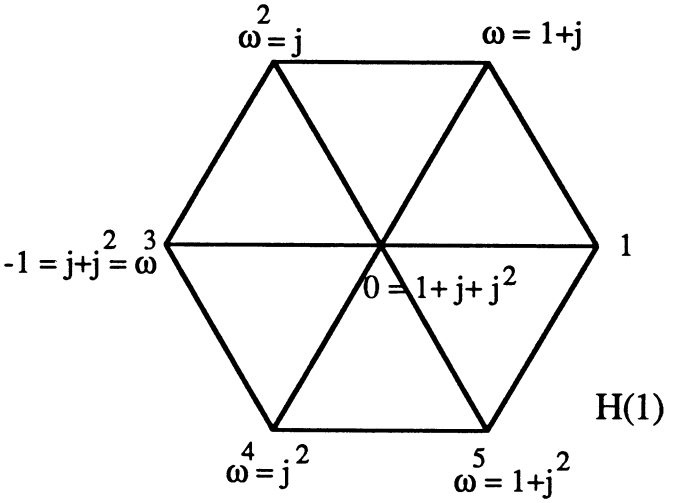
La réciproque est fausse : 2 est premier dans H et $|2|^2 = 4$ n'est évidemment pas premier dans Z

IV.4.B Représentation des éléments de H et de C

Il est immédiat qu'avec n chiffres de H(1) en base 2, on représente $H(2^n - 1)$. On pourra donc représenter tout H en base 2^n avec des chiffres pris dans $H(2^n - 1)$. Et de même, on peut écrire les nombres de H en base B avec les chiffres dans H(B-1). On verra sous quelles conditions on peut écrire tous les éléments de H en base B avec les chiffres dans H(a) où a<B.
De plus, nous allons montrer que sous certaines conditions sur B, on peut réaliser des additions de manière totalement parallèle en un temps indépendant de la longueur des opérandes.
Finalement, on pourra facilement généraliser l'écriture d'entiers de H dans le but d'écrire tous les nombres complexes.

IV.4.B.a Codage, codage minimal.

On peut remarquer que H(1) est en fait $\{0,1\} + j\ \{0,1\} + j^2\ \{0,1\}$. On appelle *Chiffre Hexagonal Binaire* (CHB) un élément de H(1). Le codage des éléments de H(1) est représenté ci-dessous.



De la même façon, si on note $h(a) = \frac{\mathbb{Z}}{(a+1)\mathbb{Z}} = \{0,\ldots,a\}$ ($a \in \mathbb{N}$), on a immédiatement : $H(a) = h(a) + h(a).j + h(a).j^2$.
On obtient de même que H est égal à $\mathbb{N} + j\,\mathbb{N} + j^2\,\mathbb{N}$.
Par extension, pour tout ρ de $\mathbb{R}^+$, on notera $\eta(\rho) = [0,\rho] + j[0,\rho] + j^2[0,\rho]$, et finalement on aura $C = \mathbb{R}^+ + j\,\mathbb{R}^+ + j^2\mathbb{R}^+$ $(= \mathbb{R} + j\,\mathbb{R})$.

Donc un chiffre de H(1) sera codé par trois chiffres binaires, un chiffre de H(a) sera codé par trois chiffres de h(a), un nombre de H par trois entiers de $\mathbb{N}$ et un complexe par trois réels

positifs. Ce codage nécessite trois bits par chiffre au lieu de deux bits par chiffre dans la représentation classique "partie réelle-partie imaginaire". Cependant il permet de réaliser des additions en temps constant ainsi qu'on va le voir. Une addition en temps constant en base 2 avec une représentation "partie réelle-partie imaginaire" en chiffres binaires signés demanderait quatre bits par chiffre. C'est en ce sens que l'on peut qualifier la représentation hexagonale de "compacte".

On peut remarquer qu'écrire séparément la partie réelle et la partie imaginaire en base 4 avec les chiffres dans {-3, ..., 3} est aussi compact, cependant on perd alors une propriété très importante de la notation hexagonale binaire pour laquelle *l'ensemble des chiffres est stable pour la multiplication* ce qui permet de réaliser très simplement des multiplieurs.

Cette représentation présente plusieurs avantages :

- si $d_i = a_i+jb_i+j^2c_i \in H(p)$ alors $-d_i = (p-a_i)+j(p-b_i)+j^2(p-c_i)$
- de même, si $d_i = a_i+jb_i+j^2c_i$ alors $\bar{d}_i = a_i+j^2b_i+jc_i$

Il est donc très facile de trouver l'opposé ou le conjugué d'un nombre écrit dans ce système.

Remarque : Si $x = a+j\,b+j^2\,c$ avec $(a,b,c) \in \mathbb{R}^3$ alors $N(x) = \max(a,b,c) - \min(a,b,c)$ définit une norme sur C.

Ceci vient du fait que comme $1+j+j^2=0$, $x= (a+t)+j(b+t)+j^2(c+t)$ pour tout t de IR. On peut donc définir une représentation canonique de x : c'est celle pour laquelle Min(a,b,c) vaut 0.

Les boules définies par cette norme sont en fait des hexagones. La boule de centre 0 et de rayon ρ est $\eta(\rho)$.

Cette norme vérifie $N(X.Y) \leq N(X).N(Y)$, et $\frac{\sqrt{3}}{2} N(x) \leq |x| \leq N(x)$.

IV.4.B.b Conditions sur a pour pouvoir écrire tous les nombres de H.

On veut écrire tous les nombres de H en base B avec les chiffres dans H(a). Pour cela il suffit que l'on puisse représenter tous les points de H(B), il suffit donc que H(B) soit inclus dans B.H(1) +H(a), c'est à dire que l'hexagone de rayon B soit recouvert par les sept hexagones de rayon a centrés sur les éléments de B.H(1), ce qui est réalisé si $a \geq 2\frac{B}{3}$. Comme H(B) et H(a) sont des ensembles *discrets*, si $a > 2\frac{B}{3} - 1$ on peut bien atteindre tous les points de H(B), ce qui nous donne $3.(a+1)>2.B$. La figure 20 montre les différents cas possibles selon le reste de B modulo 3.

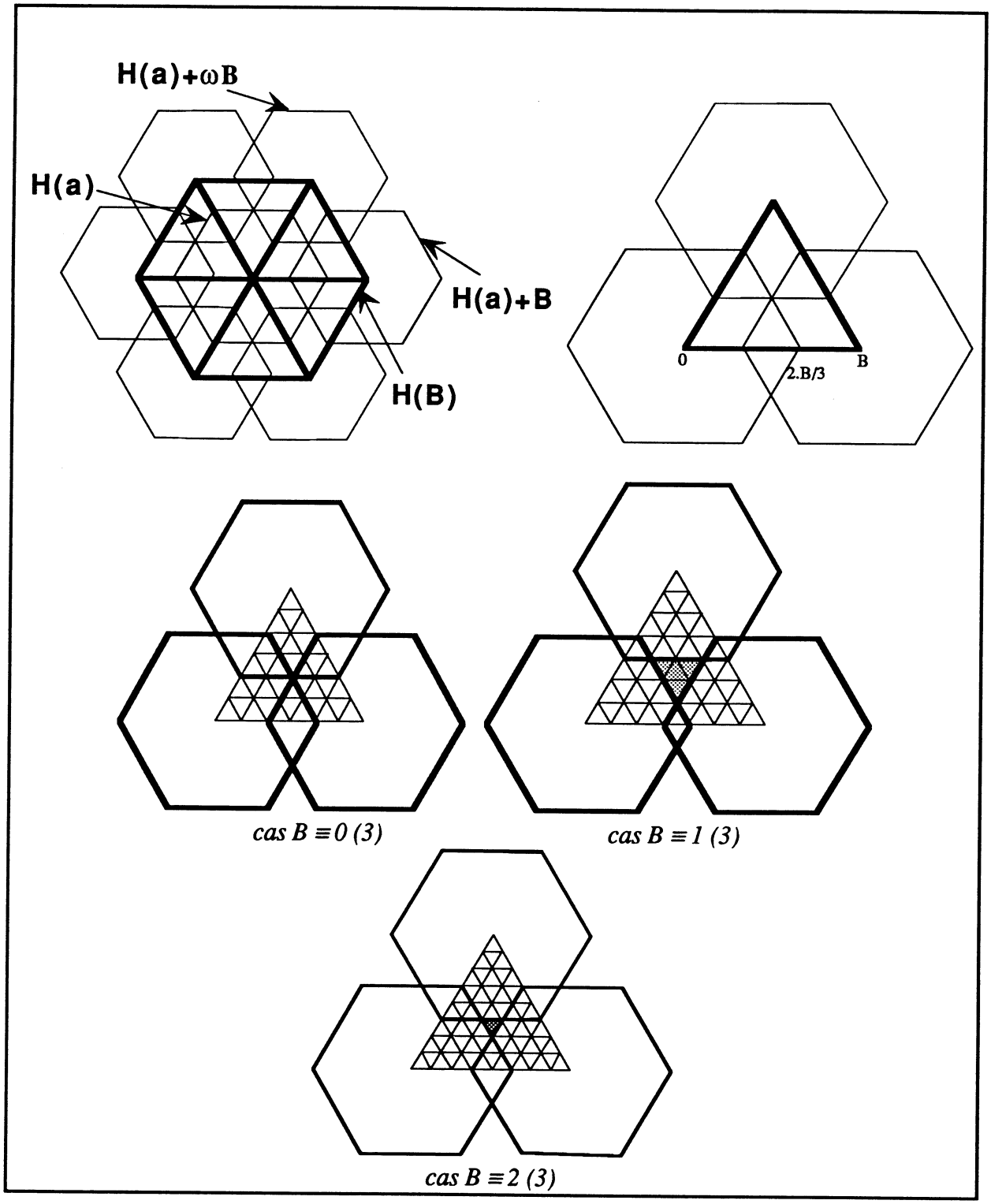


Figure 20 : recouvrement de H(B).

IV.4.C Algorithme d'addition en base B avec des chiffres dans H(a)

Nous allons d'abord présenter un algorithme dérivé de l'algorithme d'Avizienis (vu au premier chapitre) pour additionner les entiers.

Soient X et Y deux éléments de H dont les chiffres sont les x_i et y_i. $X = \sum x_i.B^i$ et $Y=\sum y_i.B^i$. On sait que x_i+y_i est dans H(2a). On désire écrire x_i+y_i sous la forme $x_i + y_i = c_{i+1}.B + s_i$

avec c_{i+1} dans H(1) et les s_i dans H(a-1). Les chiffres de la somme seraient alors $t_i=c_i+s_i$ et seraient dans H(a). Malheureusement, cela ne fonctionne pas, car puisque a<B, on n'a jamais H(2a) inclus dans B.H(1)+H(a-1).

démonstration :

Supposons a<B et 3(a+1)>2.B, montrons que x=aω+a= n'est pas dans B.H(1)+H(a-1).

Il suffit de vérifier que x, B-x, ωB-x, ..., ω^5B-x ne sont pas dans H(a-1).

On n'a en fait à le vérifier que pour B-x et ω.B-x les autres cas sont immédiats. De plus par raison de symétrie, il suffit de le vérifier pour B-x.

B-x= B-aω-a = (B-a) -aω = (B-a) + a.ω^4 n'est pas dans H(a-1)

On voit immédiatement sur le schéma de la figure 21 que a(1+ω)-B est sur la frontière de H(a), il ne peut donc pas appartenir à H(a-1).

En fait on peut dire que a+aω–B = 2a-B +aj +0j^2, et donc N(a+aω-B) = a (N est la norme définie précédemment). Donc a+aω-b ne peut pas appartenir a H(a-1).

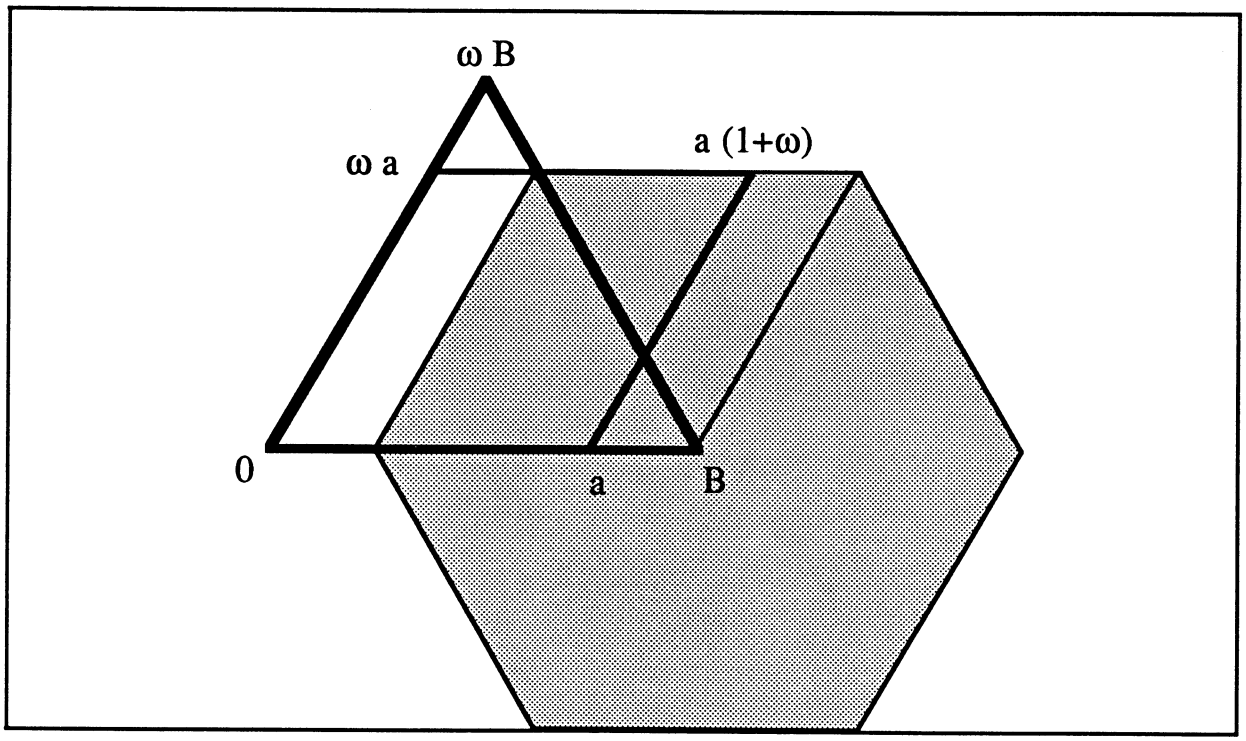


Figure 21 : la norme de B-a(1+ω) est égale à a.

On va donc modifier l'algorithme d'Avizienis pour effectuer des additions sur des entiers écrits en base B avec des chiffres pris dans {-a,...,a}, avec a<B.

Soient x et y deux éléments de {-a,...,a} on va chercher à écrire x+y sous la forme c*B+s où c est dans {-2,...,2} et s dans {-a+2,...,a-2}.

Il faut montrer que tout élément de {-2a,...,2a} peut s'écrire sous cette forme. Pour cela, il suffit que l'on aie a-2 ≥ B-(a-2)-1 ce qui donne 2(a-2) ≥ B+1 c'est à dire 2a ≥ B+3
Comme l'on veut de plus a≤B-1, ces deux conditions ne peuvent être satisfaites conjointement que si B ≥ 5.

Il suffit de modifier simplement cet algorithme pour pouvoir additionner des éléments de H écrits en base B avec les chiffres dans H(a).
Soient x et y deux élément de H(a) écrivons x+y sous la forme c*B+s où c est dans H(2) et s dans H(a-2).

On remarque que pour pouvoir écrire tous les nombres de H(2.B) sous cette forme, il suffit de pouvoir écrire tous les nombres de H(B) sous la forme c*B+s où s est dans H(a-2) et c dans H(1). On a vu précédemment que si 3(a'+1) >2.B, on peut écrire tous les nombres de H en base B avec les chiffres dans H(a'). Il suffit donc que a-2 vérifie cette condition pour que l'on puisse écrire tous les nombres de H(2.B) sous la forme c*B+s et donc *a fortiori* tous les nombres de H(2.a). La condition est : 3((a-2)+1) > 2.B ce qui donne 3(a-1) > 2B.
Comme de plus on veut a ≤ B-1, on obtient 3(B-2) > 2B, ce qui n'est possible que si B est strictement supérieur à 6.
On obtient alors l'algorithme suivant :

Algorithme : (pour additionner 2 nombres écrits en base B avec les chiffres pris dans H(a))
$X = \sum x_i B^i$, $Y = \sum y_i B^i$ $x_i, y_i \in H(a)$
- On trouve c_{i+1} et s_i tels que $x_i + y_i = c_{i+1}*B+s_i$ où $c_{i+1} \in H(2)$ et $s_i \in H(a-2)$
- ensuite, on pose $t_i = c_i+s_i$.
- On a bien $t_i \in H(a)$ et $X+Y= \sum t_i B^i$

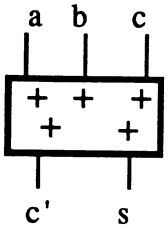
Cet algorithme ne fonctionne donc que pour B>6. Cela ne veut pas dire que l'on ne puisse pas trouver un algorithme d'addition en temps constant pour B≤6 : dans le cas B=2, il suffira de regrouper les chiffres par 3 pour pouvoir effectuer une addition en temps constant, ce qui revient implicitement à travailler en base 8.

IV.4.D Algorithme matériel d'addition

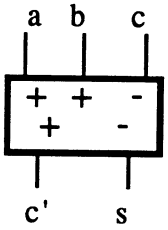
On va montrer comment on peut réaliser un additionneur parallèle en temps constant pour les nombres complexes écrits en représentation hexagonale binaire, puis un additionneur série poids forts en tête. Pour cela on va d'abord donner sans démonstration un additionneur en temps constant, puis voir comment on peut le redessiner sous forme de succession de tranches identiques. Ensuite, on va détailler le fonctionnement de ces tranches et prouver que le circuit réalise bien l'addition de deux complexes. Il suffira ensuite de faire quelques modifications sur ces tranches pour construire un additionneur en-ligne.

IV.4.D.a Addition parallèle en temps constant.

L'additionneur de la figure 23 ([DHK91]) permet de réaliser une addition en temps constant. On utilise le fait que $1+j+j^2 = 0$, donc $1= -j -j^2$, $j = -1-j^2$ et $j^2 = -1-j$. Comme ces relations introduisent des valeurs négatives, on utilise des cellule FA et PPM présentées au chapitre 3 et dont le fonctionnement est rappelé figure 21. Les connexions en gras transportent les retenues issues de ces cellules.



Cellule FA



Cellule PPM

Figure 22 : cellules de base de l'additionneur. Elles calculent $a+b\pm c = 2c'\pm s$.

On regroupe les bits par trois ce qui revient à travailler en base 8. On a une retenue dans H(2) ce qui est normal, car on a vu que l'on ne pouvait pas avoir d'algorithme d'addition en temps constant en ayant des retenues dans H(1).

Cet additionneur peut être redessiné à l'aide de "tranches" (figure 24) chaque tranche ayant en entrée 4 éléments de H(1) (2 éléments à sommer plus deux retenues). Pour cela on effectue une permutation circulaire sur les composantes en 1, j et j^2.

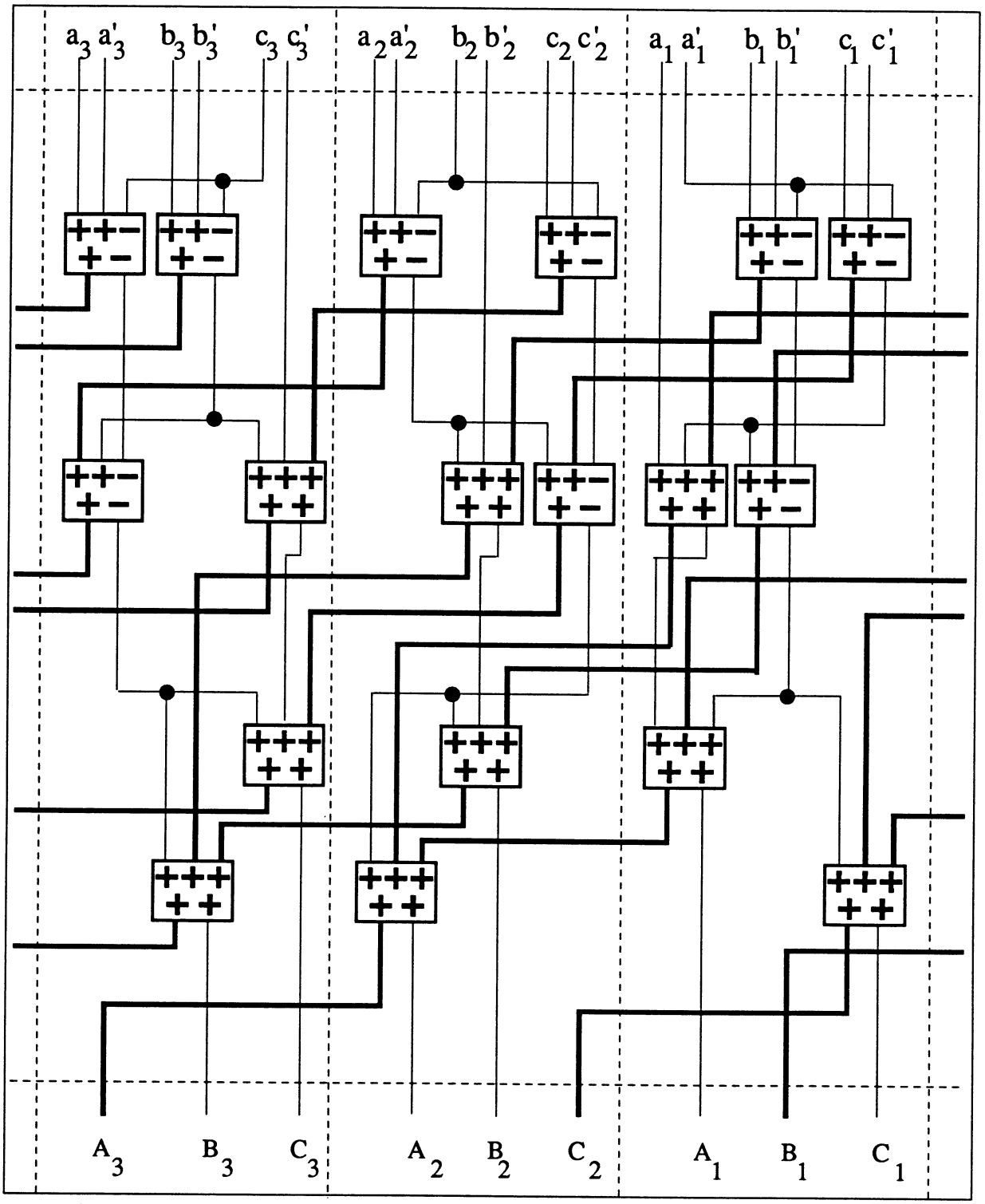


Figure 23 : étage d'additionneur en temps constant.

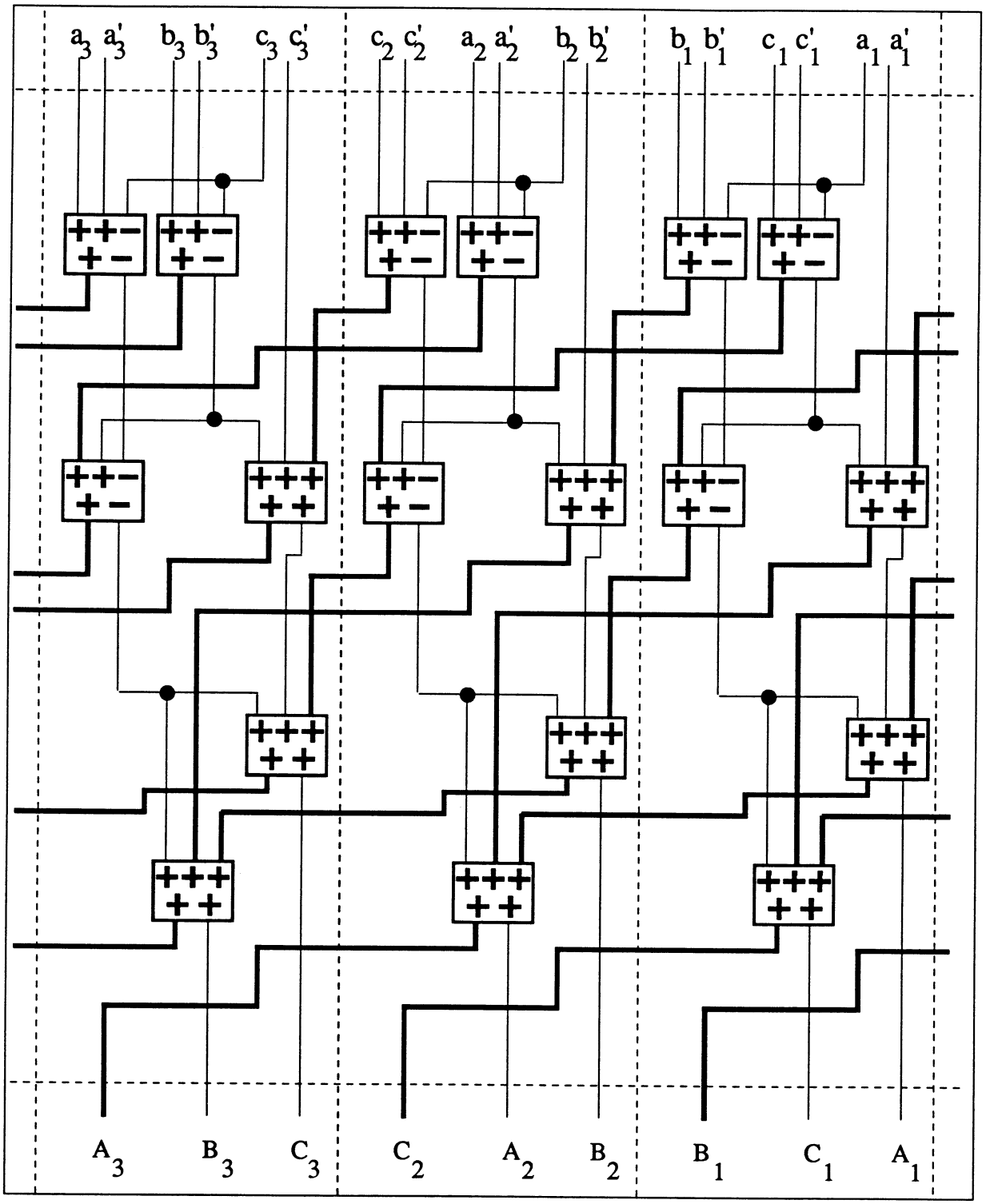


Figure 24 : additionneur en tranches.

L'additionneur de la figure 24 est construit à l'aide des tranches décrites figure 25. Les valeurs X,Y et Z sont portées par les directions respectives j^i,j^{i+1} et j^{i+2}.

Détaillons le fonctionnement des différentes cellules. On a les relations suivantes :

$2C'_6 + Y = \gamma + C_4 + C_5$

$2C'_5 + Z = C_3 + \gamma + \delta$

$2C'_4 + \delta = \beta + z' + C_1$

$2C'_3 - \gamma = C_2 + \beta - \alpha$

$2C'_2 - \beta = y + y' - z$

$2C'_1 - \alpha = x + x' - z$

$X = C_6$

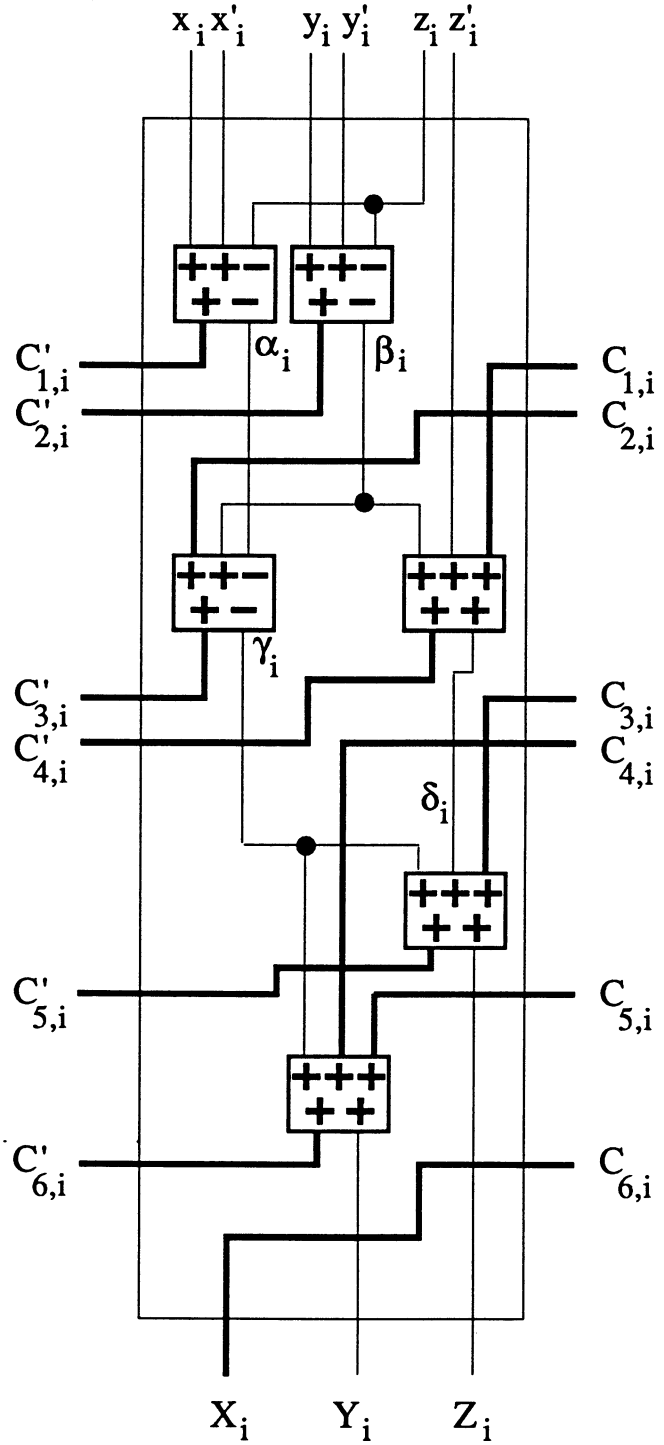


Figure 25 : tranche d'additionneur.

Nous allons réécrire ces équations pour faire disparaître les variables internes. On obtient les trois relations suivantes :

$2C'_1 + 2C'_3 + X = x + x' + C_2 + C_6 + \gamma + \beta - z$

$2C'_2 + 2C'_6 + Y = y + y' + C_4 + C_5 + \gamma + \beta - z$

$2C'_4 + 2C'_5 + Z = z' + C_1 + C_3 + \gamma + \beta$

Ce qui donne, en reportant sur les directions respectives de X,Y et Z :

$(2C'_1 + 2C'_3 + X)\, j^i + (2C'_2 + 2C'_6 + Y) j^{i+1} + (2C'_4 + 2C'_5 + Z) j^{i+2} =$

$(x + x' + C_2 + C_6)\, j^i + (y + y' + C_4 + C_5) j^{i+1} + (z + z' + C_1 + C_3) j^{i+2}$

Une tranche effectue donc la somme de deux entrées et de deux retenues entrantes et donne une somme et deux retenues sortantes.

Voyons comment il faut réorganiser les chiffres des opérandes à l'entrée de l'additionneur. On veut additionner les deux valeurs complexes suivantes :

$$R=\sum_{i=0}^{n-1}(a_i + b_i j + c_i j^2)2^i \text{ et } R'=\sum_{i=0}^{n-1}(a'_i + b'_i j + c'_i j^2)2^i.$$

On va les réécrire en posant :

$(x_i,y_i,z_i) = (a_i,b_i,c_i)$ si $i=3k$

$(x_i,y_i,z_i) = (b_i,c_i,a_i)$ si $i=3k+1$

$(x_i,y_i,z_i) = (c_i,a_i,b_i)$ si $i=3k+2$

ce qui nous donne :

$$R=\sum_{i=0}^{n-1}(a_i + b_i j + cij^2)2^i = \sum_{i=0}^{n-1}(x_i.j^i + y_i.j^{i+1} + c_i.j^{i+2})2^i.$$

Si on fait de même pour R', on obtient :

$$R'=\sum_{i=0}^{n-1}(a'_i + b'_i j + c'ij^2)2^i = \sum_{i=0}^{n-1}(x'_i.j^i + y'_i.j^{i+1} + c'_i.j^{i+2})2^i.$$

Ecrivons maintenant la somme $R+R'=\sum_{i=0}^{n-1}((x_i+x_i')j^i + (y_i+y_i')j^{i+1} + (z_i+z_i')j^{i+2})2^i$

$$=\sum_{i=0}^{n-1}((x_i+x_i'+C_{2,i}+C_{6,i})j^i + (y_i+y_i'+C_{4,i}+C_{5,i})j^{i+1} + (z_i+z_i'+C_{1,i}+C_{3,i})j^{i+2})2^i$$

$$- \sum_{i=0}^{n-1}((C_{2,i}+C_{6,i})j^i + (C_{4,i}+C_{5,i})j^{i+1} + (C_{1,i}+C_{3,i})j^{i+2})2^i$$

$$R+R'=\sum_{i=0}^{n-1}((2C'_{1,i}+2C'_{3,i}+X)j^i +(2C'_{2,i}+2C'_{6,i}+Y)j^{i+1} + (2C'_{4,i}+2C'_{5,i}+Z)j^{i+2})2^i$$

$$- \sum_{i=0}^{n-1}((C_{2,i}+C_{6,i})j^i + (C_{4,i}+C_{5,i})j^{i+1} + (C_{1,i}+C_{3,i})j^{i+2})2^i.$$

Soit encore $R+R'=\sum_{i=0}^{n-1}(X\, j^i +Y\, j^{i+1} + Z\, j^{i+2})2^i$

$$+\sum_{i=0}^{n-1}((C'_{1,i}+C'_{3,i})j^i +(C'_{2,i}+C'_{6,i})j^{i+1} + (C'_{4,i}+C'_{5,i})j^{i+2})2^{i+1}$$

$$- \sum_{i=0}^{n-1}((C_{2,i}+C_{6,i})j^i + (C_{4,i}+C_{5,i})j^{i+1} + (C_{1,i}+C_{3,i})j^{i+2})2^i$$

Les tranches de l'additionneur sont reliées entre elles et on a donc $C'_{k,i}= C_{k,i+1}$. Donc

$$R+R'=\sum_{i=0}^{n-1}(X\,j^i + Y\,j^{i+1} + Z\,j^{i+2})2^i$$

$$+\sum_{i=1}^{n}((C_{1,i}+C_{3,i})j^{i-1} +(C_{2,i}+C'_{6,i})j^i + (C_{4,i}+C_{5,i})j^{i+1})2^i$$

$$-\sum_{i=0}^{n-1}((C_{2,i}+C_{6,i})j^i + (C_{4,i}+C_{5,i})j^{i+1} + (C_{1,i}+C_{3,i})j^{i+2})2^i.$$

Finalement, on obtient $R+R' =\sum_{i=0}^{n-1}(X\,j^i + Y\,j^{i+1} + Z\,j^{i+2})2^i$

$$+((C_{1,n}+C_{3,n})j^{n-1} +(C_{2,n}+C_{6,n})j^n + (C_{4,n}+C_{5,n})j^{n+1})2^n$$

$$-((C_{2,0}+C_{6,0}) + (C_{4,0}+C_{5,0})j + (C_{1,0}+C_{3,0})j^2).$$

Les retenues entrantes $C_{i,0}$ sont forcées à zéro. Il ne reste plus qu'à faire l'addition des retenues sortantes $C_{i,n}$.

Le Circuit de la figure 24 réalise donc bien l'addition de deux complexes écris en notation CHB.

IV.4.D.b Addition en-ligne.

On va construit un additionneur en ligne à l'aide d'une tranche d'additionneur parallèle dont on a rebouclé les retenues sortantes sur les retenues entrantes. Les sorties de cellule FA et PPM qui ne correspondent pas à des retenues sont retardées d'un cycle comme indiqué figure 26.

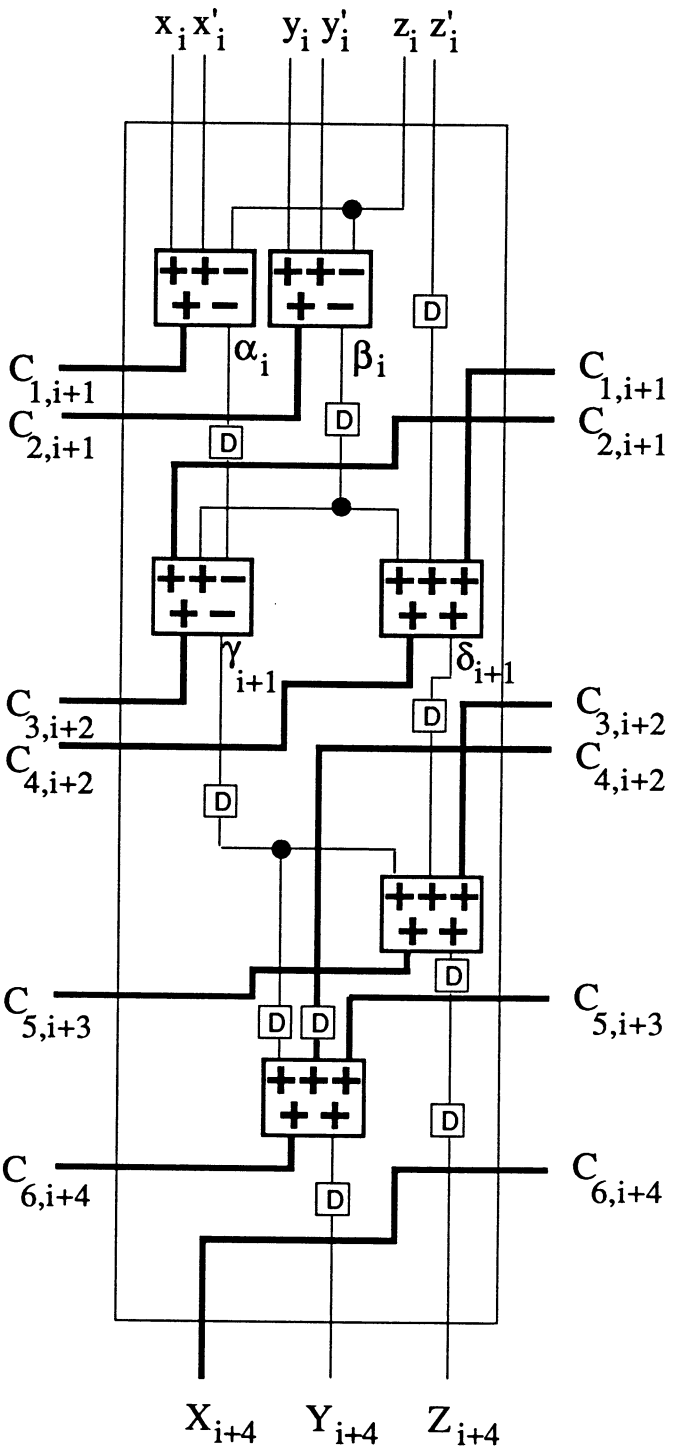


Figure 26 : additionneur en-ligne.

On additionne deux complexes R et R' avec les mêmes notations que précédemment. Si on détaille le fonctionnement des cellules de cet additionneur en-ligne, on trouve les relations suivantes :

$2C_{6,i+4} + Y_{i+3} = \gamma_{i+3} + C_{4,i+3} + C_{5,i+3}$

$2C_{5,i+3} + Z_{i+2} = C_{3,i+2} + \gamma_{i+2} + \delta_{i+2}$

$2C_{4,i+2} + \delta_{i+1} = \beta_{i+1} + z'_{i+1} + C_{1,i+1}$

$2C_{3,i+2} - \gamma_{i+1} = C_{2,i+1} + \beta_{i+1} - \alpha_{i+1}$

$2C_{2,i+1} - \beta_i = y_i + y'_i - z_i$

$2C_{1,i+1} - \alpha_i = x_i + x'_i - z_i$

$X_{i+4} = C_{6,i+4}$

Pour montrer que ce circuit réalise bien la somme des nombres en entrée, nous allons calculer la valeur du nombre $S = \sum_{i=0}^{n+1} (X_i.j^i + Y_i.j^{i+1} + Z_i.j^{i+2}) 2^i$ et montrer qu'il est bien égal à la somme des deux nombres en entrée, pour cela on va d'abord réécrire les relations ci dessus afin de faire disparaître au maximum les variables internes, ce qui nous donne les relations ci dessous :

$16X_{i+4} + 8Y_{i+3} + 4Z_{i+2} = 8\gamma_{i+3} + 8C_{4,i+3} + 8C_{5,i+3} + 4Z_{i+2}$

$= 8\gamma_{i+3} - 4\delta_{i+2}+4\beta_{i+2}+4z'_{i+2}+4C_{1,i+2}+ 4\gamma_{i+2} +4\delta_{i+2}+4C_{3,i+2}$

$= 8\gamma_{i+3}+4\beta_{i+2}+4z'_{i+2}+4C_{3,i+2}+ 4\gamma_{i+2}+4C_{1,i+2}$

$= 8\gamma_{i+3}+ 4\gamma_{i+2}+2\gamma_{i+1}+4\beta_{i+2}+4z'_{i+2}+2C_{2,i+1}+2\beta_{i+1}-2\alpha_{i+1}+4C_{1,i+2}.$

donc
$16X_{i+4}+8Y_{i+3}+4Z_{i+2} = 8\gamma_{i+3}+ 4\gamma_{i+2}+ 2\gamma_{i+1}+4\beta_{i+2}+2\beta_{i+1}+\beta_i$
$+y_i+y'_i-z_i+2x_{i+1}+2x'_{i+1}-2z_{i+1} +4z'_{i+2}.$

On va montrer que si pour $-2\le i<0$ et $n-1<i\le n+1$ on a $x_i=x'_i=y_i=y'_i=z_i=z'_i=0$, alors
$S=\sum_{i=0}^{n+1}(X_i.j^i +Y_i.j^{i+1} +Z_i.j^{i+2})2^i = R+R'$

On suppose que $x_i=x'_i=y_i=y'_i=z_i=z'_i=0$ pour $-2\le i<0$, on a donc $\alpha_i=\beta_i=0$ pour $-2\le i<0$, et $C_{1,i}=C_{2,i}=0$ pour $-1<i\le 0$.
$2C_{3,0}-\gamma_{-1}=C_{2,-1}+\beta_{-1}-\alpha_{-1}=0$ donc $C_{3,0}=\gamma_{-1}=0$. De même, $C_{4,0}=\delta_{-1}=0$.
$2C_{4,-1}+\delta_{-2}=C_{1,-2}+\beta_{-2}-\alpha_{-2}=C_{1,-2}$, donc $C_{4,-1}=0$.
$2C_{5,0}+Z_{-1}=\gamma_{-1}+\delta_{-1}+C_{3,-1}=C_{3,-1}$ donc $C_{5,0}=0$.
$2X_0+Y_{-1}=\gamma_{-1}+C_{4,-1}+C_{5,-1}=C_{5,-1}$ donc $X_0=0$.
$2X_1+Y_0=\gamma_0+C_{4,0}+C_{5,0}=\gamma_0$.
En résumé, si $x_i=x'_i=y_i=y'_i=z_i=z'_i=0$ pour $-2\le i<0$, on a $\beta_{-1}=\beta_{-2}=\gamma_{-1}=0, X_0=0$ et $2X_1+Y_0=\gamma_0$.

On suppose que $x_i=x'_i=y_i=y'_i=z_i=z'_i=0$ pour $n\le i<n+2$,on a donc $\alpha_i=\beta_i=0$ pour $n\le i\le n+2$, et $C_{1,i}=C_{2,i}=0$ pour $n<i\le n+2$.
$2C_{3,n+2}-\gamma_{n+1}=C_{2,n+1}+\beta_{n+1}-\alpha_{n+1}=0$ donc $C_{3,n+2}=\gamma_{n+1}=0$. De même, $C_{4,n+2}=\delta_{n+1}=0$.
$2C_{4,n+1}+\delta_n=C_{1,n}+\beta_n-\alpha_n=C_{1,n}$, donc $C_{4,n+1}=0$.
$2C_{5,n+2}+Z_{n+1}=\gamma_{n+1}+\delta_{n+1}+C_{3,n+1}=C_{3,n+1}$ donc $C_{5,n+2}=0$.
$2X_{n+2}+Y_{n+1}=\gamma_{n+1}+C_{4,n+1}+C_{5,n+1}=C_{5,n+1}$ donc $X_{n+2}=0$.
$2X_{n+3}+Y_{n+2}=\gamma_{n+2}+C_{4,n+2}+C_{5,n+2}=\gamma_{n+2}$.
En résumé, si $x_i=x'_i=y_i=y'_i=z_i=z'_i=0$ pour $n\le i<n+2$, on a $\beta_{n+1}=\beta_n=\gamma_{n+1}=0, X_n=0$ et $2X_{n+3}+Y_{n+2}=\gamma_{n+2}$.

Ecrivons maintenant ce que vaut S :
$S=\sum_{i=0}^{n+1}(X_i.j^i +Y_i.j^{i+1} +Z_i.j^{i+2})2^i$

$$= \sum_{i=-2}^{n-1}(16X_{i+4} +8Y_{i+3} +4Z_{i+2})j^{i+1}2^i - (2X_{n+3}+Y_{n+2})j^n 2^{n+2} - X_{n+2}j^{n+2}2^{n+2} +2X_1j+X_0+Y_0j.$$

On a vu que $X_{n+2}=X_0=0$ et que $2X_1+Y_0=\gamma_0$ et $2X_{n+3}+Y_{n+2}=\gamma_{n+2}$, donc

$$S= \sum_{i=-2}^{n-1}(16X_{i+4} +8Y_{i+3} +4Z_{i+2})j^{i+1}2^i - \gamma_{n+2}j^n 2^{n+2}+\gamma_0j.$$

Soit $S= \sum_{i=-2}^{n-1}(y_i+y'_i-z_i+2x_{i+1}+2x'_{i+1}-2z_{i+1} +4z'_{i+2})j^{i+1}2^i$

$$+\sum_{i=-2}^{n-1}(8\gamma_{i+3}+ 4\gamma_{i+2}+2\gamma_{i+1}+4\beta_{i+2}+2\beta_{i+1}+\beta_i)j^{i+1}2^i - \gamma_{n+2}j^n 2^{n+2}+\gamma_0j.$$

Comme $1+j+j^2=0$ et $\gamma_{n+1}=\gamma_{-1}=\beta_{n+1}=\beta_n=\beta_{-1}=\beta_{-2}=0$, les β_i et les γ_i s'éliminent et on obtient :

$$S= \sum_{i=-2}^{n-1}(y_i+y'_i)j^{i+1}2^i+ \sum_{i=-2}^{n-1}(2x_{i+1}+2x'_{i+1})j^{i+1}2^i - \sum_{i=-2}^{n-1}z_ij^{i+1}2^i - \sum_{i=-2}^{n-1}2z_{i+1}\,j^{i+1}2^i+ \sum_{i=-2}^{n-1}4z'_{i+2}j^{i+1}2^i,$$

et comme pour $-2\le i<0$ et $n-1<i\le n+1$ on a $x_i=x'_i=y_i=y'_i=z_i=z'_i=0$, on obtient finalement

$$S= \sum_{i=0}^{n-1}((x_i+x'_i)j^i+(y_i+y'_i)j^{i+1}+(z_i+z'_i)j^{i+2})2^i =R+R',$$

et on a donc bien réalisé l'addition en ligne de R et R' avec un délai égal à 4.
On peut enchaîner les additions en séparant les nombres à additionner par 2 zéros.

Si on choisit une représentation des complexes de la forme A+i.B avec A et B écrits en chiffres signés, l'addition en ligne se fait alors avec un délai égal à 2.

IV.4.E Passage de IN +j IN +j² IN à Z +j Z

On a vu que l'on écrit les nombres sous la forme $X = \sum x_i2^i$ où $x_i = a_i +j.b_i +j^2.c_i$ et a_i, b_i et c_i sont dans {0,1}. Si on pose $A = \sum a_i2^i$, $B = \sum b_i2^i$ et $C = \sum c_i2^i$ alors $X = A + j.B +j^2.C$. On peut aussi remarquer que comme $j^2= -1-j$, $x_i = a_i +j.b_i +j^2.c_i = (a_i - c_i)+j.(b_i -.c_i) = a'_i+j.b'_i$ avec a'_i et b'_i dans {-1,0,1}.

Cette réécriture se fait en temps constant en parallèle et en ligne avec un délai nul comme on le voit figure 27. Le passage inverse demande de réaliser une addition de complexes et se fait en ligne avec un délai 4.

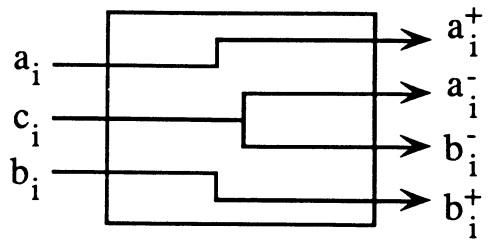


Figure 27 : passage en-ligne de IN +j IN +j² IN à Z +j Z .

IV.4.F Multiplication

La multiplication dans H en base 2 se fera de façon semblable à celle en base 2 sur N. La seule différence est que l'on n'aura pas besoin de réaliser une propagation finale de retenue. Il faut juste être capable de réaliser rapidement le produit de 2 chiffres de H(1).

IV.4.F.a Multiplication de chiffres hexagonaux binaires.

Un chiffre binaire hexagonal s'écrit sous la forme $a+bj+cj^2$ où a,b et c sont dans {0,1}. On peut remarquer que si on représente le complexe $a+bj+cj^2$ par l'entier a+2b+4c, le produit de deux chiffres binaires signés se ramène à un produit d'entiers modulo 7 (figures 24 et 25)

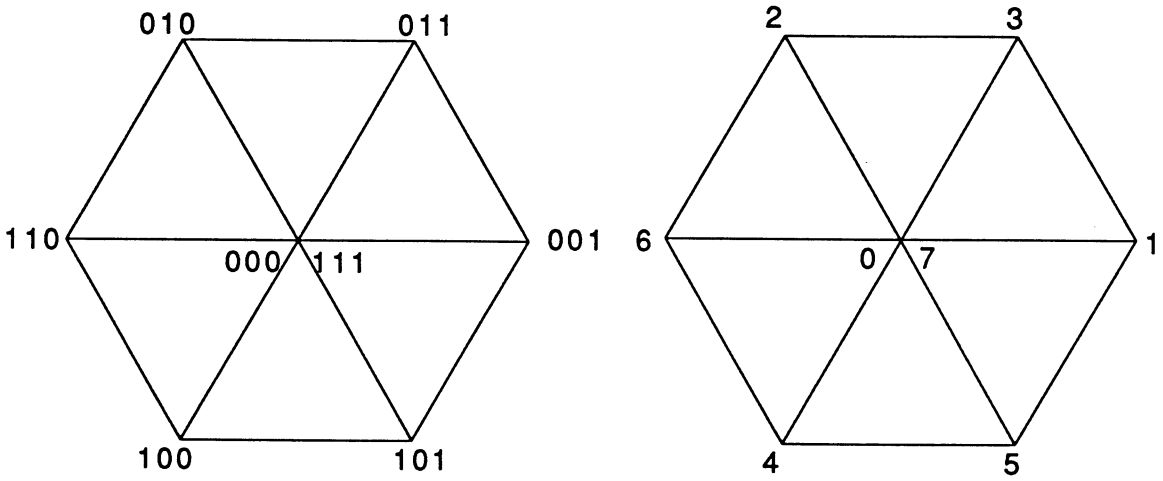


Figure 28 : représentation des CHB par des entiers

	0	1	3	2	4	5	7	6
0	0	0	0	0	0	0	0	0
1	0	1	3	2	4	5	0	6
3	0	3	2	6	5	1	0	4
2	0	2	6	4	1	3	0	5
4	0	4	5	1	2	6	0	3
5	0	5	1	3	6	4	0	2
7	0	0	0	0	0	0	0	0
6	0	6	4	5	3	1	0	1

Figure 29 : table de multiplication des CHB représentés par des entiers.

Un multiplieur de CHB est en fait un simple multiplieur modulo 7.

Cherchons une écriture booléenne de la multiplication de deux CHB, on a $x=a+bj+cj^2$ et $x'=a'+b'j+c'j^2$.

On a donc $x.x'=(a+bj+cj^2)(a'+b'j+c'j^2) = A+Bj+Cj^2$.

Il suffit de trouver les équations logiques de A car comme on a les deux relations

$(c+aj+bj^2)(c'+a'j+b'j^2) = B+Cj+Aj^2$ et $(b+cj+aj^2)(b'+c'j+a'j^2)= C+Aj+Bj^2$, on déduit les équations de B et de C de celle de A.

écrivons la table de vérité de A :

A		a'	0	0	0	0	1	1	1	1
		b'	0	0	1	1	0	0	1	1
a	b	c \c'	0	1	1	0	0	1	1	0
0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0	1	0	0
0	1	1	0	1	0	0	1	1	0	0
0	1	0	0	0	0	0	1	1	0	1
1	0	0	0	0	1	1	0	0	0	1
1	0	1	0	1	1	1	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1	0	0	1

Figure 30 : table de vérité de A.

On obtient l'équation suivante pour A :

$A = a\bar{b}\ a'\bar{c}' + b\bar{c}\ c'\bar{b}' + c\bar{a}\ b'\bar{a}' + \bar{a}b\bar{a}'c' + \bar{b}c\bar{c}'b' + \bar{b}c\bar{b}'a'$.

On en déduit les équations de B et de C :

$B = a\bar{b}\ b'\bar{a}' + b\bar{c}\ a'\bar{c}' + c\bar{a}\ c'\bar{b}' + \bar{a}b\bar{b}'a' + \bar{b}c\bar{a}'c' + \bar{b}c\bar{c}'b'$,

$C = a\bar{b}\ c'\bar{b}' + b\bar{c}\ b'\bar{a}' + c\bar{a}\ a'\bar{c}' + \bar{a}b\bar{c}'b' + \bar{b}c\bar{b}'a' + \bar{b}c\bar{a}'c'$.

IV.4.F.b Multiplication en-ligne de complexes écrits en CHB.

Comme on sait réaliser en-ligne une addition de deux complexes en notation binaire hexagonale, on va réaliser la multiplication en ligne de complexes. Pour cela, on va légèrement modifier le multiplieur en-ligne décrit au chapitre 3. Les connexions en gras transportent ici des CHB et correspondent à trois valeurs binaires, et les cellules délai correspondantes sont en fait des délais sur chacune des composantes.

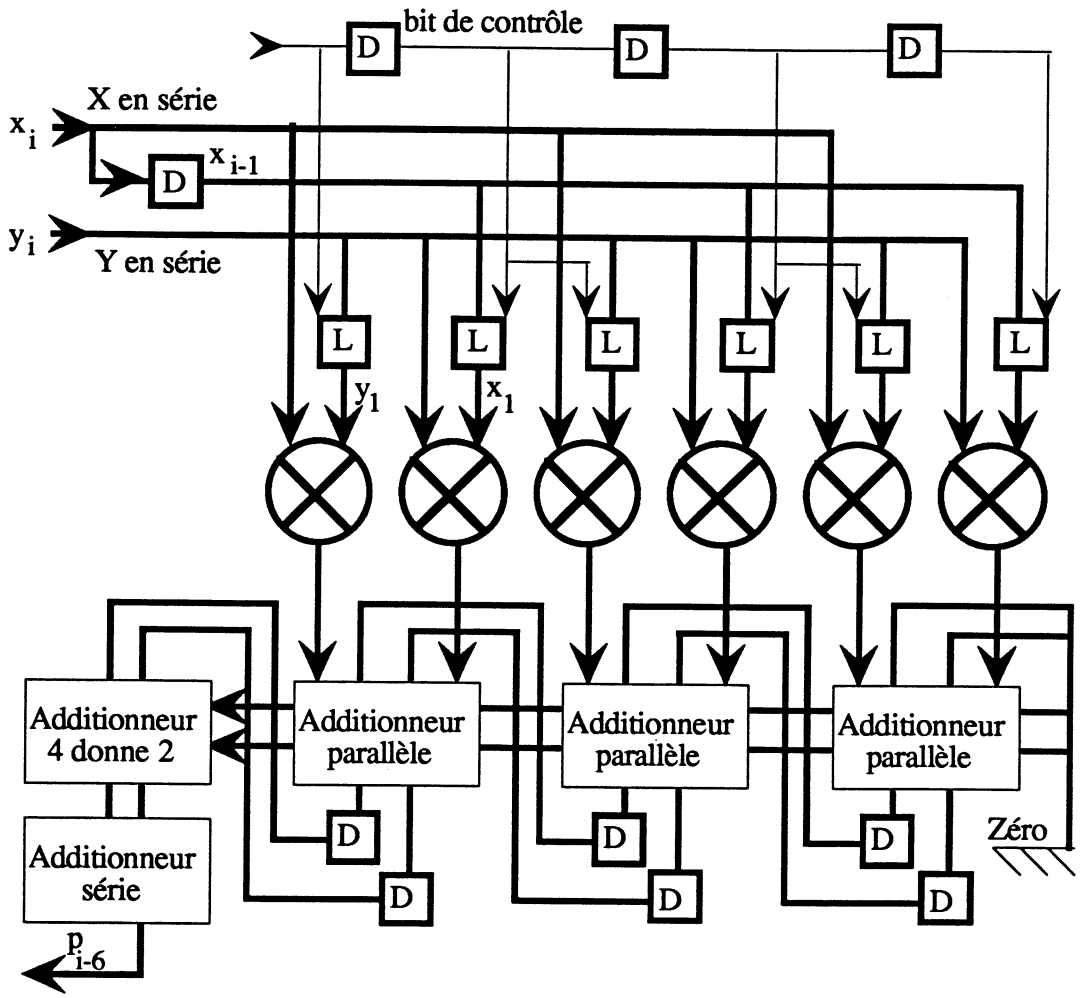


Figure 31 : multiplieur complexe en-ligne.

Pour les étages de l'additionneur parallèle, on travaille séparément sur les composantes en 1,j et j^2 (voir figure 32). De même, pour réaliser l'additionneur série à quatre entrées, on réalise d'abord un additionneur série "4 donne 2" (figure 33) en séparant les composantes avant d'attaquer les entrées de l'additionneur en-ligne décrit précédemment.

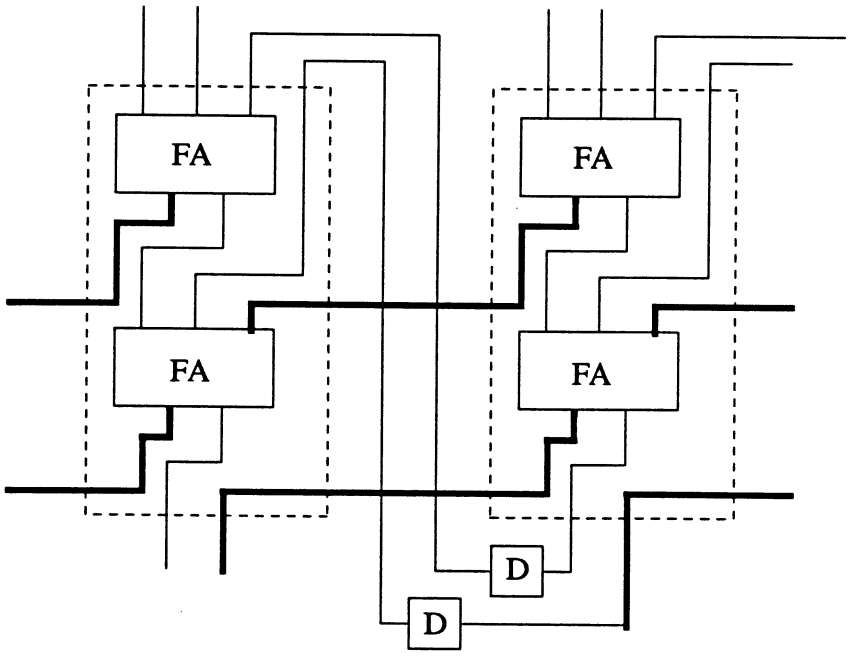


Figure 32 : additionneur parallèle avec retard pour chaque composante.

Le délai total est égal au délai de l'additionneur "4 donne 2" (c'est à dire 2) plus le délai de l'additionneur en-ligne (c'est à dire 4) soit 6.

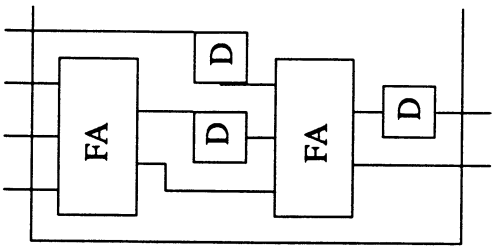


Figure 33 : additionneur série "4 donne 2".

Etudions la surface de chaque tranche.
Chaque tranche contient 2 multiplieurs CHB, un étage d'additionneur parallèle pour chaque composante (en 1, j et j^2), 4 cellules retard CHB et une cellule retard binaire (pour le bit des contrôle), ce qui donne 13 cellules retard binaires, 6 cellules FA (2 par composante comme on le voit figure 33) et 2 multiplieurs CHB. Elle communique à droite et à gauche 7 CHB et une valeur binaire ce qui donne 22 valeurs binaires à communiquer à gauche et à droite.

IV.4.F.c Multiplication en-ligne dans $\mathbf{Z}+i\,\mathbf{Z}$.

Si on utilise une représentation de la forme A+i.B où A et B sont représentés en Chiffres Signés Binaires (CSB), on à deux solutions pour réaliser la multiplication en-ligne : soit effectuer 4 (ou 3) multiplications en-ligne de réels, soit travailler au niveau des puissances de 2 et réaliser des multiplieurs élémentaires pour les nombres de {-1,0,1}+i{-1,0,1}. Nous allons évaluer le délai et la surface demandés par ces deux solutions.

• En utilisant des multiplieurs entiers.
On peut utiliser une des deux relations suivantes :
(A+i.B)(A'+i.B') = (A.A' - B.B') +i.(A.B' + B.A') et
(A+i.B)(A'+i.B') = (A.A' - B.B') +i.((A+A')(B+B')-A.A' - B.B')
Suivant la relation utilisée, on doit donc réaliser 3 ou 4 produits de réels. La solution à 4 multiplications est plus coûteuse en surface mais donne un délai 5 (2 pour les additions plus 3 pour les produits). La solution à 3 multiplications demande elle un délai d'au moins deux unités de plus pour le calcul de A+A' et B+B' sans tenir compte du fait qu'on a trois (et non plus deux) sorties de multiplieur à sommer.
Evaluons maintenant la surface demandée. Chaque tranche d'un multiplieur en CSB demande 4 cellules PPM, 2 multiplieurs CSB et 7 cellules retard binaires. Elle communique à droite et à gauche 7 valeurs binaires.
Donc la surface d'un tel multiplieur pour les complexes serait supérieure à celle d'un multiplieur CHB en-ligne (que l'on utilise 3 ou 4 multiplieurs CSB en-ligne).

• En travaillant au niveau des puissances de 2 avec des multiplieurs élémentaires pour les nombres de {-1,0,1}+i{-1,0,1}.

Chaque multiplieur élémentaire donne en sortie 2 CSB par composante (en 1 et i) ce qui fait que chaque étage d'additionneur parallèle reçoit en entrée par composante au moins 5 CSB (4 sortant des 2 multiplieurs élémentaires de l'étage plus la sortie retardée de l'étage précédent d'additionneur). On présente figure 35 un étage d'additionneur parallèle (pour une composante) avec en entrée 6 CSB (4 venant des multiplieurs et 2 venant des sorties retardées). Le multiplieur complet est décrit figure 34 .

Les traits en gras transportent ici des éléments de {-1,0,1} +i.{-1,0,1} soit 4 valeurs binaires. Chaque tranche demande 16 cellules PPM, 17 cellules retard binaires et 2 multiplieurs élémentaires.

Le délai de l'additionneur "6 donne 2" (pour une composante) est de trois et donc le délai total du multiplieur est de 5.

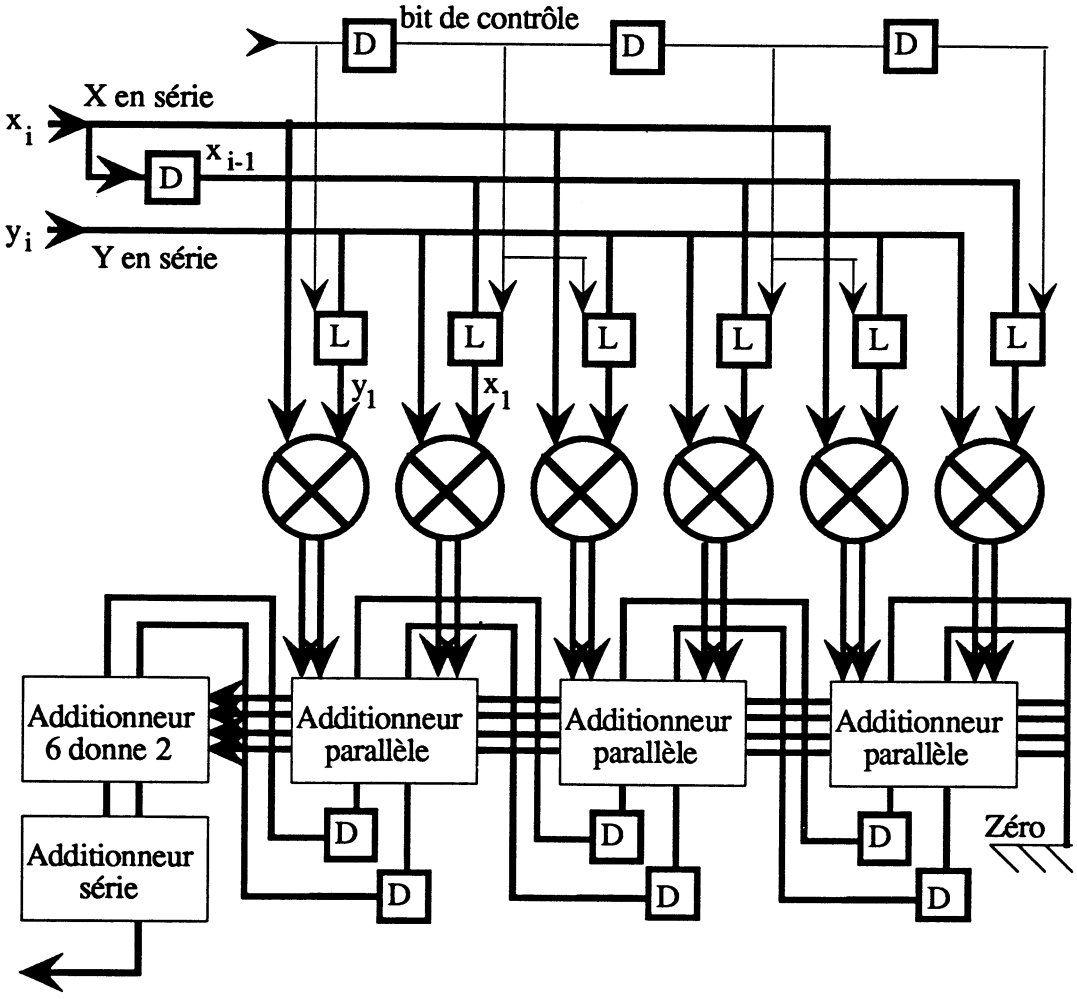


Figure 34 : multiplieur en ligne dans Z + i Z .

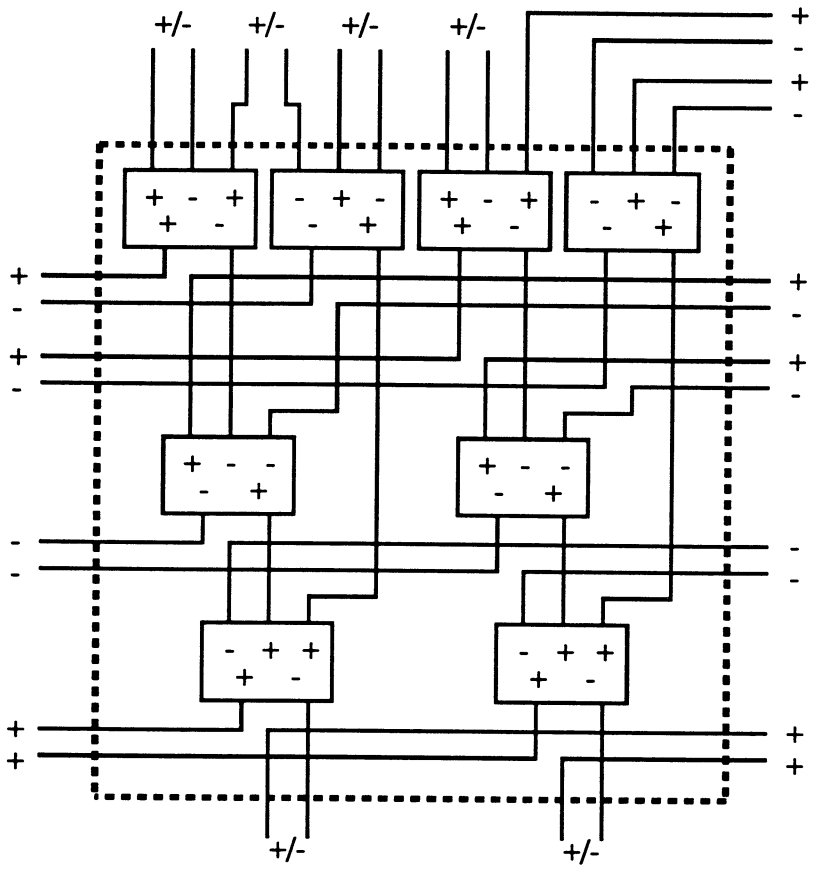


Figure 35 : détail de l'additionneur de la figure 32 pour une composante.

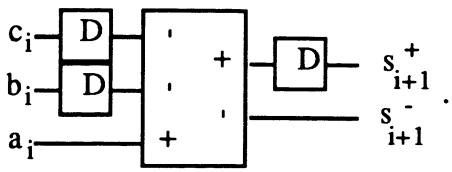
La notation chiffres hexagonaux binaires permet donc de réaliser des multiplications en ligne avec une surface inférieure à celle d'un multiplieur en ligne pour des nombres écrits sous la forme A+i.B où A et B sont écrits en chiffres signés binaires.

IV.4.G Calcul en ligne de la partie réelle

Si on écrit un nombre sous la forme $X=\sum_{k=0}^{n-1} x_k 2^k$ où $x_k=a_k+j.b_k+j^2.c_k$, on peut exprimer sa partie réelle sous la forme $Re(X)=\sum_{k=0}^{n-1}(2.a_k-b_k+c_k)2^{k-1}$. On peut alors facilement réaliser en ligne le calcul de Re(X) à l'aide d'une cellule PPM avec un délai 1.

Si on écrit $2.s^-_{i+1}-s^+_i = a_i - b_{i+1} - c_{i+1}$ alors $Re(X) = \sum_{k=-1}^{n-1}(s^+_i - s^-_i).2^i$, si on prend la précaution de poser $a_{-1}=b_n=c_n=0$.

On obtient le schéma ci contre :



De même, si on a X sous la forme $X=\sum_{k=0}^{n-1}((a_i^+ - a_i^-)+j.(b_i^+ - b_i^-))2^i$, on peut écrire la partie réelle de X sous la forme $Re(X) = \sum_{k=0}^{n-1}((a_i^+ - a_i^-)2^i+(b_i^+ - b_i^-)2^{i-1})$. C'est une simple addition de nombres écrits en chiffres signés binaires, et cette addition se fait avec un délai 2.

IV.4.H Division de complexes

IV.4.H.a Division parallèle.

Pour diviser 2 nombres X et Y, on va utiliser un algorithme semblable à celui de la division pour les réels en base 2, c'est à dire un algorithme itératif de la forme :
$X_{n+1}=2.X_n-d_n.Y$. On remarque que si les d_i sont dans H(1) alors on est obligé de regarder tous les chiffres de X. En effet si $2.X_i$ est très proche de (2j+1)Y, on ne sait pas choisir entre $d_i=j$ et $d_i=j+1$ tant qu'on ne connait pas les valeurs exactes de X_i et Y (donc tant qu'on ne connait pas tous leurs chiffres).

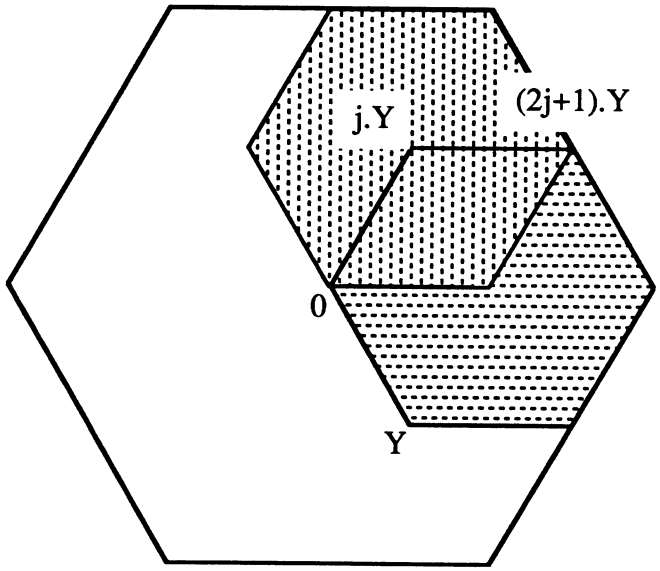


Figure 36 :

Si on prend les d_i dans H(2) alors si X est dans l'hexagone de rayon Y, alors 2X est dans l'hexagone de rayon 2Y et on peut trouver un élément d de H(2) tel que $|2X-d.Y|\leq \frac{1}{\sqrt{3}}|Y|$ car l'hexagone de rayon 2Y est recouvert par des triangles équilatéraux dont les sommets sont dans Y.H(2) et $\frac{1}{\sqrt{3}}|Y|$ est la plus grande distance d'un point intérieur à un triangle équilatéral de coté |Y| à un de ces sommets.

On a vu que $\frac{\sqrt{3}}{2}N(X) \leq |x| \leq N(X)$ pour tout X complexe. La relation $|2X-d.Y| \leq \frac{1}{\sqrt{3}}|Y|$ implique donc qu'il existe d dans H(2) tel que $N(2.X-d.Y) \leq \frac{2}{3}N(Y)$. et on n'a donc pas besoin de connaître tous les chiffres de Y et de X pour trouver d tel que $N(2.X-d.Y) \leq N(Y)$.
On obtient donc l'algorithme suivant pour diviser X par Y si $N(X) \leq N(Y)$:
On pose $X_0=X$,
et $X_{n+1}=2.X_n-d_n.Y$ où d_n est choisi de façon à ce que $N(2.X_n-d_n.Y) \leq N(Y)$.
d_n peut être trouvé en temps constant car H(2) contient un nombre fini (19) d'éléments et on sait qu'il existe δ dans H(2) tel que $N(2.X_n-\delta.Y) \leq \frac{2}{3}N(Y)$. Il suffit donc de regarder un nombre fixe de chiffres de $2.X_n-\delta.Y$ pour être sûr que $N(2.X_n-d_n.Y) \leq N(Y)$.

IV.4.H.b Division en-ligne

Pour réaliser la division en ligne, une première méthode (*a priori* non optimale) est de remarquer que $\frac{x}{y} = \frac{x\bar{y}}{|y|^2}$. Les produits $x\bar{y}$ et $|y|^2$ peuvent être calculés en ligne et comme $|y|^2$ est réel, on peut diviser séparément en ligne les composant en 1,j et j^2 de $x\bar{y}$ par $|y|^2$. $|y|^2$ est une valeur réelle et on peut l'obtenir en utilisant un multiplieur complexe en ligne suivi d'une cellule PPM pour en obtenir la partie réelle comme vu au paragraphe IV.4.G . Les résultats des division en ligne sont en chiffres signés sur chacune des composantes, il nous reste donc à faire une addition en ligne de complexes. Le délai total est la somme des délais d'un multiplieur en-ligne complexe en CHB, d'un diviseur en-ligne en chiffres binaires signés et d'un additionneur en-ligne complexe en CHB.
On peut améliorer cette méthode en remarquant que l'on peut modifier le multiplieur en-ligne pour sortir le résultat dans $\mathbb{Z} + j\,\mathbb{Z}$. Il suffit de réaliser le passage en-ligne de IN +j IN +j2 IN à $\mathbb{Z}$ +j $\mathbb{Z}$ à la sortie des multiplieurs CHB. On utilise alors pour les composantes en 1 et j la partie basse du multiplieur en-ligne en chiffres signés présenté au chapitre 3. Ce qui nous donne un délai de 3 pour la multiplication.

IV.5 représentation hexagonale flottante

On va représenter les complexes comme les réels en virgule flottante. Un complexe sera écrit sous la forme $x = 2^e.\left(\sum_{i=0}^{n} x_i 2^i\right)$ où les x_i sont dans D_6.

e sera noté l'exposant de x, et $m=\sum_{i=0}^{n} x_i 2^i$ sera appelé la mantisse de x. On peut imposer afin de préserver la précision, une contrainte semblable à celle de la normalisation de la représentation en virgule flottante classique, en exigeant que la mantisse vérifie toujours $x_0 \neq -x_1$. Cela revient à imposer que m soit entre l'hexagone de rayon $\frac{1}{2}$ et l'hexagone de rayon 2 (soit $\frac{\sqrt{3}}{4} < |m| < 2$)

comme on le voit ci dessous. On pourrait imposer que la mantisse soit strictement normalisée (c'est à dire comprise entre l'hexagone de rayon 2 et celui de rayon 1) qui donnerait un codage "minimal" de la mantisse avec le chiffre de poids fort non nul. Cela imposerait pour le vérifier de regarder tous les chiffres de la mantisse et de réécrire les nombres : on perdrait alors l'avantage des additions en temps constant.

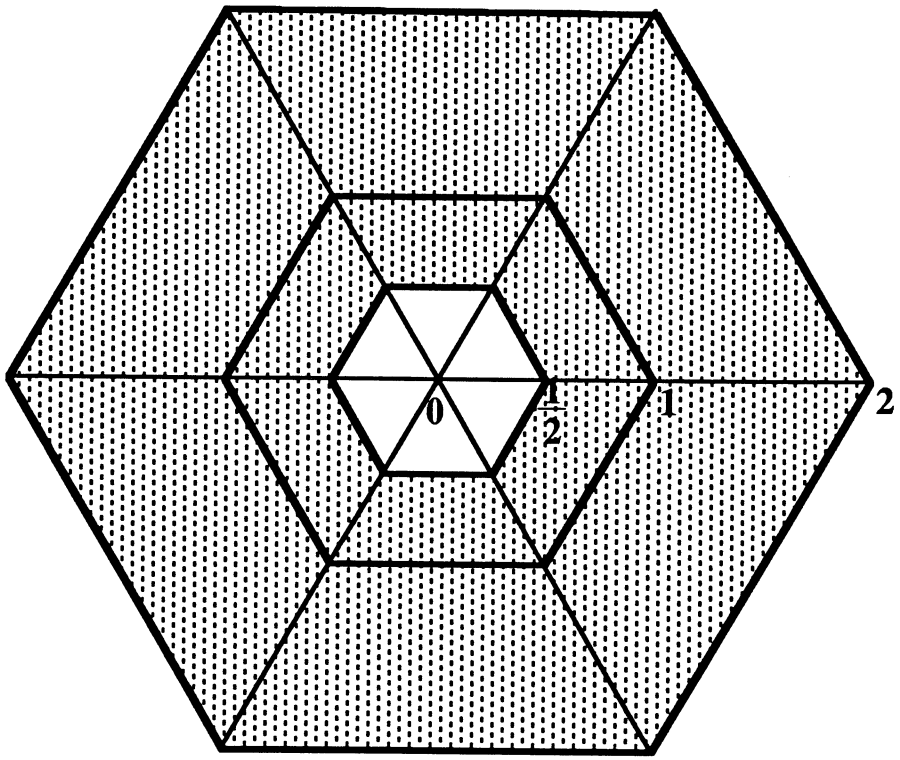


Figure 37 : Ensemble des mantisses pseudo normalisées

problème de l'arrondi :
Pour réaliser un arrondi sur un complexe, il suffit de réaliser le même arrondi dans chacune des trois directions, et comme $1+j+j^2 = 0$, l'arrondi se fera sans biais. Le meilleur arrondi possible est donc de réaliser l'arrondi au plus près sur chaque composante.

IV.6 Conclusion

On a présenté des éléments de réponse aux questions posées au premiers chapitre sur les liens entre l'ensemble des chiffres et la base dans le cas particulier où la base est réelle et les chiffres sont des racine $n^{èmes}$ de l'unité ou zéro. On n'a pas pu comme dans le cas des réels donner de valeur limite pour la base et on a du se contenter d'un encadrement.
La deuxième partie sur la représentation hexagonale binaire des nombres complexes nous donne des résultats plus directement exploitables. On retrouve pour les complexes tous les avantages que donne la représentation binaire signée dans le cas des réels : les additions peuvent être faites en temps constant, l'ensemble des chiffres est stable par multiplication et on peut donc réaliser des multiplieurs en suivant les mêmes schémas que pour les réels.

Ces propriétés nous on permis de décrire des opérateurs de calcul en-ligne pour les nombres complexes : additionneurs et multiplieurs.
De plus, cette notation est plus compacte que la représentation séparée de la partie réelle et de la partie imaginaire en chiffres binaires signés.

Conclusion

On a cherché dans cette thèse à donner quelques éléments de réponse aux deux principaux problèmes de l'arithmétique des ordinateurs : accélérer le calcul des opérations arithmétiques de base et représenter efficacement les nombres. On s'est surtout intéressé au cas des grands entiers et des complexes, avec dans le cas des grands entiers deux approches distinctes, les méthodes logicielles et le calcul en série poids fort en tête.

En ce qui concerne le calcul du produit de grands entiers par voie logicielle, on a montré que les méthodes basées sur les transformations arithmétiques peuvent être implantées efficacement sur des machines classiques et surclasser des méthodes plus classiques (algorithme de Karatsuba) pour des grands nombres (de plus de 40000 chiffres décimaux). Ces algorithmes de transformations arithmétiques peuvent de plus être adaptés beaucoup de problèmes de convolution (convolutions d'images par exemple).

Pour le calcul en ligne, les résultats donnés permettent de conclure à l'optimalité de certains opérateurs en ligne connus : additionneur ou multiplieur.

Enfin, la notation hexagonale binaire proposée pour les complexes semble être bien adaptée car elle permet de travailler sur les complexes de la même manière que l'on travaille sur les réels, et il est facile, ainsi qu'on l'a vu, de concevoir les opérateurs correspondants.

Mais il serait aussi très intéressant d'adapter les architectures aux algorithmes proposés et de réaliser des ordinateurs (ou peut-être dans un premier temps des unités arithmétiques) fonctionnant dans les systèmes décrits : chiffres binaires signés ou chiffres binaire hexagonaux.

Un aspect à développer plus en profondeur serait le calcul en ligne sur des complexes écrits en chiffres hexagonaux signés. On a décrit les opérations de base, mais on pourrait essayer de généraliser les résultats du troisième chapitre aux fonctions de variables complexes.

Bibliographie

[AVI61] Avizienis, *Signed-digit number representations for fast parallel arithmetic*, IRE Transactions on electronic computers, 10, pages 389-400, 1961.

[BK81] R.P. Brent et H.T. Kung, *The area-time complexity of binary multiplication*, Journ. of the ACM, Vol. 28 n°3, Juil. 1981.

[BK82] R.P. Brent et H.T. Kung, *A regular layout for parallel adders*, IEEE Trans. on Computers, Vol. C-31, pages 260-264, Mar. 1982.

[BMZ86] G.Brassard, S. Monet et D. Zuffellato, *Algorithme pour l'arithmétique des grands entiers*, TSI, Vol. 5, n° 2, 1986.

[BO 87] P.T. Balsara, R.M. Owens, *Systolic and Semi-systolic Digit Serial Multipliers*, Proceedings IEEE 8-th Symposium on Computer Arithmetic, Côme, Italie, 1987

[BPT 89] F. Balestro, G. Privat, M.S. Tawfik, *A Bit-Serial Approach to VLSI Implementation of Digital LDI Ladder Filters*, ICASSP'89, Glasgow, Mai 1989.

[CA69] S.A. Cook and S.O. Aandera, *On the minimum computation time of functions*, Trans. of the AMS, 142, 1969, pages 291-314.

[CH73] M. Cappa et V.C. Hamacher, *An augmented iterative array for high-speed binary division*, IEEE Transactions on Computers, Vol. C-22, Fev. 1973.

[CR78] C.Y. Chow et J.E. Robertson, *Logical design of a redundant binary adder*, Proceedings IEEE 4-th Symposium on Computer Arithmetic, 1978

[CT65] J.M.Cooley et J.W.Tuckey, *An algorithm for the machine computing complex Fourier series*, Mathematics of Computation, Vol. 19, 1965, 297-301.

[DA65] L. Dadda, *Some schemes for parallel multipliers*, Alta Frequenza, Vol. 19, pages 349-356, Mar. 1965.

[DHK91] J. Duprat, Y. Herreros and S. Kla, *New redondant representation of complex numbers and vectors*, 10th symposium on computer arithmetic, Grenoble, France, juin 1991.

[DHM89] J. Duprat, Y. Herreros and J.M. Muller, *Some results about on-line computation of functions* 9th symposium on computer arithmetic, Santa Monica, California, Sept. 1989.

[DM89] J. Duprat and J.M. Muller, *The CORDIC algorithm : new results for fast VLSI implementation,* RR LIP-IMAG, Lyon, 1989. Soumis à IEEE Transactions on Computers.

[EL85] M.D. Ercegovac et T. Lang, *A division algorithm with prediction of quotient digits*, 7th Symposium on computer arithmetic, Urbana, Illinois, Juin 1985.

[EL87b] M.D. Ercegovac et T. Lang, *On-line scheme for computing rotation factors*, 8th Symposium on computer arithmetic, Côme, Italie, Mai 1987, IEEE Publ. No 87CH2419-0.

[ER84] M.D. Ercegovac, *On-line arithmetic : an overview*, SPIE Vol. 495, Real time signal processing VII, pages 86-93, 1984.

[ET77] M.D. Ercegovac et K.S. Trivedi, *On line algorithms for division and multiplication*, IEEE Trans. on Computers, Vol. C-26 No 7, pages 681-687, Juil. 1977.

[ET87] M.D. Ercegovac et P.K.G. Tu, *A radix-4 on-line division algorithm*, 8th Symposium on computer arithmetic, Como, Italy, Mai 1987, IEEE Publ. No 87CH2419-0.

[FL70] M.J. Flynn, *On division by functional iteration*, IEEE Transactions on Computers, Vol. C-19, Août 1970.

[GA65] H. Garner, *Number systems and Aritmetic,* Advances in Computers Vol. 6, Academic Press, pages 131-194, 1965.

[GE80] A.L. Grnarov et M.D. Ercegovac, *On the performance of on-line arithmetic*, Proc. 1980 Intern. Conference on parallel processing, IEEE Publ. No 80CH1569-3, pages 55-62, Août.1980.

[GHM89] A. Guyot, Y. Herreros et J.M. Muller, *JANUS, an On-line Multiplier-Divider for Manipulating Large Numbers*, Proceedings IEEE 9th Symposium on Computer Arithmetic, Santa Monica, USA, Sept. 1989

[GHMP89]A. Guyot, Y. Herreros, J.M. Muller et G. Privat, *Redundant Arithmetic Operators in Digital Signal Processing Applications*, IFIP Workshop on Parallel Architectures on Silicon, Grenoble, Déc. 1989

[GU70] H.H. Guild, *Some cellular logic arrays for nonrestoring binary division*, Radio Electron. Eng., Vol. 39, 1970, pages 345-348.

[HP85] M. d'Hoe, M. Pierre, Ph. Deleuze, A. Vandemeulebroecke, P. Jespers et M. Davio, *CASBA: Cryptographic application using signed binary arithmetic*, ESSIRC'85 Sept. 1985.

[HW81] V.C. Hamacher et J. Williams, *A linear-time divider array*, Canadian Electr. Engineering Journal, Vol.6, No 4, 1981.

[HWA79] K. Hwang, *Computer arithmetic principles, architecture and design*, New-York, J. Wiley&Sons Inc., 1979.

[IFR81] G.Ifrah, Histoire universelle des nombres, Ed. Seghers, Paris, 1981.

[IO79] M.J. Irwin et R.M. Owens, *On-line algorithms for the design of pipeline architectures*, 6th symposium on Computer Architecture, Philadelphia, USA, Avr. 1979.

[IO87] M.J. Irwin et R.M. Owens, *Digit-pipelined arithmetic as illustrated by the paste-up system : a tutorial*, IEEE Computer, pages 61-73, Avr. 1987.

[IRW77] M.J. Irwin, *An arithmetic unit for Online computation*, PhD thesis, tech. report UIUCDCS-R-77-873, Dept. of Computer science, university of Illinois, Champaign-urbana, IL 61801, Mai 1977.

[IRW78] M.J. Irwin, *A pipelined processing unit for on-line division*, Proc. 5th symposium on Computer architecture, IEEE Publ. No 78CH1284-9C, pages 24-30, Avr. 1978.

[KN81] D.E.Knuth, *The Art of Computer Programming, Vol. 2: Semi-numerical Algorithms*, seconde édition, Addison-Wesley, 1981.

[KO62] A. Karatsuba et Y. Ofman, *Multiplication of large numbers on an automata*, Dokl. Akad. Nauk. SSSR, 145, 1962, pages 293-294 (en russe).

[KU88] S.Y. Kung, *VLSI Array Processors*, Prentice-Hall, 1988

[KWW89]S.C. Knowles, J.G. Mc Whirter, R.F. Woods et J.V. McCanny, *Bit-level Systolic Architectures for High Performance IIR Filtering,* Journal of VLSI Signal Processing, vol. 1, pages 9-24, 1989.

[LS87] H. Lin et H.J. Sips, *A novel floating-point online division algorithm,* 8th Symposium on computer arithmetic, Como, Italy, Mai 1987, IEEE Publ. No 87CH2419-0.

[LY76] R.F. Lyon, *Two's Complement Pipeline Multipliers*, IEEE Transactions on Communications, vol. COM-24, Avr. 1976.

[MS61] O.L. Mac Sorley ., *High Speed Arithmetic in Binary Computers,* Proc. of IRE, Vol 49, pages 67-91, Jan. 1961.

[MU85] J.M. Muller, *Méthodologie de calcul des fonctions élémentaires*, Thèse de doctorat INP Grenoble, 1985.

[MU89] J.M. Muller, *Aritmétique des ordinateurs*, études et recherches en informatique, Masson 1989.

[NU80] H.J.Nussbaumer, *Fast fourier transform and convolution algorithms*, Springer Series in Information Sciences, Springer-Verlag, 1981.

[PM89] K.K. Parhi et D.G. Messerschmitt, *Pipeline Interleaving and Parallelelism in Recursive Digital Filters*, IEEE Trans on ASSP, vol 37, n° 7, Juil. 1989

[PO71] J.M.Pollard,*The Fast Fourier Transform in a finite field*, Mathematics of Computation, vol. 25, Avr. 71, 365-374.

[PR89] G. Privat et M. Renaudin, *Motion Estimation VLSI Architecture,* ICCD'89, Cambridge, Oct. 1989

[PRI86] G. Privat, *Architectures spécialisées de circuits VLSI pour le traitement du signal*, Thèse de doctorat, Ecole Nationale Supérieure des Télécommunications, France, 1986.

[PV81] F.P. Preparata et J.E. Vuillemin, *Area-Time optimal VLSI networks for computing integer multipliction and discrete Fourier transform,* Proc. ICALP symposium, Haifa, Juillet 1981.

[PV90] F.P. Preparata et J.E. Vuillemin, *Practical Cellular Dividers*, IEEE Trans. Computers,Vol. 39, pages 605-614, Mai 1990.

[RN81] M. Renfors, Y. Neuvo, *The Maximum Sampling Rate of Digital Filters Under Hardware Speed Constraints,* IEEE Trans. Circuits Syst., vol. CAS-28, n° 3, Mar. 1981.

[ROB58] J.E. Robertson, *A new class of digital division methods*, IRE Transactions on electronic computers, Vol. C-7, Sept. 1958.

[ROC89] J.L. Roch, *L'Architecture du Système PAC et son Arithméique Rationnelle*, Thèse de doctorat, TIM3-INPG, Grenoble, France, Déc. 1989.

[SBV90] M. Shand, P. Bertin et J. Vuillemin , *Harware speedup in long integer multiplication,* 2nd Annual ACM Symposium on Parallel Algorithms and Architectures, pp 301-309, 1990..

[SC85] N.R. Scott, *Computer systems and arithmetic*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1985.

[SD88] S. G. Smith, P. B. Denyer, *Serial-Data Computation*, Kluwer, 1988

[SO72] A. Soceneantu et C.I. Toma, *Cellular Logic Array for Redundant Binary Division,* Proc. IEEE, Vol. 119, pages 1452-1456, Oct. 1972.

[SS71] A.Schönhage et V.Strassen, *Schnelle Multiplikation grosser Zahlen*, Computing, 7, 1971, 281-292.

[SVH89] B. Serpette, J. Vuillemin et J.C. Hervé, *A Portable Efficient Package for Arbitrary-Precision Arithmetic*, PRL report 2, Digital Equipment Corp., Paris Research Laboratory.

[TO58] K.D. Tocher, *Technics of multiplication and division for automatic binary computers,* Journ. Appl. Math., Vol. XI, pt 3, pages 364-384, 1958.

[VAN89] A. Vandemeulebroecke et al, *A new carry Free Division Algorithm and Application to A Single Chip 1024 bits RSA Processor*, Proceedings ESSCIRC'89, Vienne, Autriche, Sept. 1989

[VIL88] G. Villard, *Calcul formel et parallélisme, résolution de systèmes linéaires*, Thèse de doctorat, TIM3-INPG, Grenoble, France, Déc. 1988.

[VO59] J. Volder, *The CORDIC computing technique*, IRE Transactions on Computers, Sept. 1959.

[VUI83] J.E. Vuillemin, *A Combinatorial limit to the computing power of the VLSI circuits*, IEEE Trans. on Computers, Vol. C-32 N° 3, Mars 1983.

[WA64] C.S. Wallace, *A suggestion for a fast multiplier*, IEEE Trans. on Electronic Computers, Fev. 1964, pages 14-17.

[WA71] J. Walther, *A unified algorithm for elementary functions*, Joint Computer proceedings, Vol. 38, 1971.

[WF82] S. Waser et M.J. Flynn, *Introduction to arithmetic for digital systems designers*, Holt, Rinehart & Winston, CBS College publishing, New York, 1982.

[WI65] S. Winograd, *On the time required to perform addition*, Journ. of the ACM, Vol. 12, pages 277-285, Avr. 1965.

[WI67] S. Winograd, *On the time required to perform multiplication*, Journ. of the ACM, Vol. 14 n°4, pages 793-802, Oct. 1967.

[ZIM89] P. Zimmermann, *Multiplication rapîde en Le_lisp*, rapport INRIA Oct. 1989.

Annexe : Transformations arithmétiques en C

Arithmétique modulaire en C

```
/*****************************************************/
/*          definition des constantes                */
/*****************************************************/
/*****************************************************/
/* Ni=PBASEj*PBASEk tient sur 2 mots de 32bits       */
/* Si est l'inverse de Ni modulo PBASEi              */
/* DEMIi est l'inverse de 2 modulo PBASEi            */
/* N=PBASE1*PBASE2*PBASE3 tient sur 3 mots de 32bits */
/*****************************************************/

unsigned long PBASE1 = 0xc0000001; /* 3*2**30+1      */
unsigned long DEMI1  = 0x60000001;
unsigned long S1     = 0x999999d4;
unsigned long N1H    = 0xbc800001;
unsigned long N1L    = 0xb8000001;
/*****************************************************/
unsigned long PBASE2 = 0xd0000001; /* 13*2**28+1     */
unsigned long DEMI2  = 0x68000001;
unsigned long S2     = 0x455554e5;
unsigned long N2H    = 0xae000001;
unsigned long N2L    = 0xa8000001;
/*****************************************************/
unsigned long PBASE3 = 0xe8000001; /* 29*2**27+1     */
unsigned long DEMI3  = 0x74000001;
unsigned long S3     = 0xc911114a;
unsigned long N3H    = 0x9c000001;
unsigned long N3L    = 0x90000001;
/*****************************************************/
unsigned long NH     = 0x8d600002;
unsigned long NM     = 0x06800002;
unsigned long NL     = 0x78000001;
/*****************************************************/
unsigned long PBASE,DEMI,SI,*WNK,*WMNK;

/*****************************************************/
/* WNKi[n] racine 2**n ème de 1 modulo PBASEi        */
/* WMNKi[n] est l'inverse de WNKi[n] modulo PBASEi   */
/*****************************************************/
unsigned long WNK1[] = {    0x1 ,    0xc0000000 ,
    0x3c6f986f ,    0x3d771377 ,    0x96990787 ,
    0x3fd4a80 ,     0x26d9e65a ,    0x4d088bd6 ,
    0x74bc4044 ,    0x2d93ac34 ,    0x6e950ac8 ,
    0x45ab3b3c ,    0xc97ec70 ,     0x67620637 ,
    0x56233758 ,    0x53c794bc ,    0x962aa97d ,
    0x6b4fb1ca ,    0x50f7f530 ,    0x1e8119a0 ,
    0xb36523bd ,    0x1ab6df09 ,    0xbdb3bc14 ,
    0x863aacf ,     0x201a6811 ,    0x6d97a293 ,
    0xebf69b7 ,     0xdb2957c ,     0xe8d4a51 ,
    0x3d09 ,        0x7d };
/*****************************************************/
```

```
unsigned long WMNK1[] = {   0x1 ,    0xc0000000 ,
    0x83906792 ,    0x248fd720 ,    0x2c2ecdc1 ,
    0xb8b2a5ce ,    0x2a03bb71 ,    0x2d91ad89 ,
    0xaefba6fd ,    0x920d56a5 ,    0x1fb8c962 ,
    0xa8ac163d ,    0x5eb4be40 ,    0x36cbba5f ,
    0x677a2bea ,    0xb2da7cbf ,    0xa67c9122 ,
    0xa113d5de ,    0x24a9f0a2 ,    0x7a850295 ,
    0x24dc667e ,    0x2d7a6825 ,    0x27fb3db5 ,
    0xae60de57 ,    0x466ffb17 ,    0xc8e3349 ,
    0x4d2fb768 ,    0xb1aaada1 ,    0x6f6749d7 ,
    0x739b025 ,     0x872b020d };
/*****************************************************/
unsigned long WNK2[] = {    0x1 ,    0xd0000000 ,
    0x6dc845de ,    0x3ed776a3 ,    0x5a3c4b22 ,
    0x54e0916f ,    0xc41ff415 ,    0xa678a096 ,
    0x194a2c45 ,    0x73883546 ,    0x2b9eceaa ,
    0x144ecb4f ,    0x3bea94ef ,    0xc09a73d3 ,
    0x3958c13b ,    0x3d6cb0bd ,    0x2b371049 ,
    0xc8dc5f1d ,    0x458e414 ,     0x2d36f378 ,
    0xc9178d98 ,    0x7033afd0 ,    0x442e6110 ,
    0x43e7b70d ,    0x7d8ec0f9 ,    0x837f4441 ,
    0x463cd9c0 ,    0x53027d11 ,    0x1853d3 };
/*****************************************************/
unsigned long WMNK2[] = {0x1 ,       0xd0000000 ,
    0x6237ba23 ,    0x69002491 ,    0x2b2a11b3 ,
    0x246ee36 ,     0x89415c1b ,    0x36aaed69 ,
    0xc794758e ,    0x2d839d09 ,    0x16c606c7 ,
    0x18e4af8c ,    0xc5c91d96 ,    0x9224a3d4 ,
    0xca89b916 ,    0x5e650183 ,    0x3b6dfab7 ,
    0x9f5f8551 ,    0x8ab5d70f ,    0x3262fbef ,
    0x684aa20e ,    0x81f3230d ,    0x537588f ,
    0x4872c5fd ,    0x7bbab461 ,    0x7337e333 ,
    0x648c6c0d ,    0xb50ee6ca ,    0xa2f68959 };
/*****************************************************/
unsigned long WNK3[] = {    0x1 ,    0xe8000000 ,
    0x2fb4d4ff ,    0x8e78b9a7 ,    0xb40b786f ,
    0x6fbb43a7 ,    0xd70cc98 ,     0x2ae72e95 ,
    0x8d174b30 ,    0x618f4d08 ,    0xdca3eb78 ,
    0x788b0f4a ,    0x83407ab0 ,    0xe20dcf83 ,
    0xd7dfa3d9 ,    0xe7421c6f ,    0xd3c5e941 ,
    0xb854ce31 ,    0x1e368276 ,    0x4d22d637 ,
    0x2c401099 ,    0xbbbeca16 ,    0xc9573066 ,
    0x73cf5fae ,    0x62368ede ,    0x292caf78 ,
    0xde4fd39 ,     0x414338b3 };
/*****************************************************/
unsigned long WMNK3[] = {0x1 ,       0xe8000000 ,
    0xb84b2b02 ,    0xaee0b240 ,    0x90044efd ,
    0x63eeeac9 ,    0x7819764f ,    0xb32763ca ,
    0x15d10829 ,    0x54b6fd8b ,    0x53755752 ,
    0xe31fb90 ,     0xb837ae8f ,    0x5a67f3e6 ,
    0xd5a3fab9 ,    0x9f42d429 ,    0x703df597 ,
    0x1bd97be7 ,    0x3043f4dd ,    0x2b9833e5 ,
    0xde24b2fd ,    0x376432d1 ,    0xa9682fea ,
    0x2e3dd0e3 ,    0x5dc7ba19 ,    0x8a7d83ec ,
    0x93374b94 ,    0xd2849feb };
/*****************************************************/
```

```
void init1()
{PBASE=PBASE1; DEMI=DEMI1; WNK=WNK1; WMNK=WMNK1; SI=S1;}
/*****************************************************/
void init2()
{PBASE=PBASE2; DEMI=DEMI2; WNK=WNK2; WMNK=WMNK2; SI=S2;}
/*****************************************************/
void init3()
{PBASE=PBASE3; DEMI=DEMI3; WNK=WNK3; WMNK=WMNK3; SI=S3;}
/*****************************************************/
```

Arithmétique modulaire en C

```
/*****************************************************/
#define plus(a,b) ( a>=(PBASE-b) ?  a-(PBASE-b)   : a+b)
#define moins(a,b) ( a<b ?  a+(PBASE-b)   : a-b)
/*****************************************************/
unsigned fois(a,b)
register unsigned a,b;
{
register unsigned r,pb,aux;
  r=0;
  pb=PBASE;
  if(b>a) { aux=b; b=a;a=aux;}
  while(b)
  {
    aux=pb-a;
    if(1&b)
        if(r>=aux) r-=aux; else r+=a;
    if(a>=aux)  a-=aux; else a+=a;
    b>>=1;
  }
  return(r);
}
/*****************************************************/
unsigned unsur2(n)              /* 1/2**n modulo PBASE */
unsigned n;
{
unsigned x,d;
d=DEMI;
for(x=2;n;n>>=1)
   if(x&1) { x>>=1; x+=d;} else x>>=1;
return(x);
}
/*****************************************************/
/*Théorème des reste chinois : calcule B à partir de */
/* bi= B mod PBASEi                                   */
/* B=(*next)*2**64+(*current)*2**32+(*previous)       */
/*****************************************************/
trc(b1,b2,b3,next,current,previous)
unsigned b1,b2,b3,*next,*current,*previous;
{
register unsigned masque;
unsigned h0,h1,h2,h3,h4,h5;
unsigned n0,n1,n2,n3,s0,s1;
masque=0xffff;
h0=h1=h2=h3=h4=h5=0;

n1=N1L >>16; n0=N1L & masque;
n3=N1H >>16; n2=N1H & masque;
s1=b1 >>16; s0=b1 & masque;

h0+=s0*n0; h1+=(h0>>16); h0 &= masque;
h1+=s0*n1; h2+=(h1>>16); h1 &= masque;
h2+=s0*n2; h3+=(h2>>16); h2 &= masque;
h3+=s0*n3; h4+=(h3>>16); h3 &= masque;

h1+=s1*n0; h2+=(h1>>16); h1 &= masque;
```

```
h2+=s1*n1; h3+=(h2>>16); h2 &= masque;
h3+=s1*n2; h4+=(h3>>16); h3 &= masque;
h4+=s1*n3; h5+=(h4>>16); h4 &= masque;

n1=N2L >>16; n0=N2L & masque;
n3=N2H >>16; n2=N2H & masque;
s1=b2 >>16; s0=b2 & masque;
h0+=s0*n0; h1+=(h0>>16); h0 &= masque;
h1+=s0*n1; h2+=(h1>>16); h1 &= masque;
h2+=s0*n2; h3+=(h2>>16); h2 &= masque;
h3+=s0*n3; h4+=(h3>>16); h3 &= masque;
           h5+=(h4>>16); h4 &= masque;
h1+=s1*n0; h2+=(h1>>16); h1 &= masque;
h2+=s1*n1; h3+=(h2>>16); h2 &= masque;
h3+=s1*n2; h4+=(h3>>16); h3 &= masque;
h4+=s1*n3; h5+=(h4>>16); h4 &= masque;

n1=N3L >>16; n0=N3L & masque;
n3=N3H >>16; n2=N3H & masque;
s1=b3 >>16; s0=b3 & masque;
h0+=s0*n0; h1+=(h0>>16); h0 &= masque;
h1+=s0*n1; h2+=(h1>>16); h1 &= masque;
h2+=s0*n2; h3+=(h2>>16); h2 &= masque;
h3+=s0*n3; h4+=(h3>>16); h3 &= masque;
           h5+=(h4>>16); h4 &= masque;
h1+=s1*n0; h2+=(h1>>16); h1 &= masque;
h2+=s1*n1; h3+=(h2>>16); h2 &= masque;
h3+=s1*n2; h4+=(h3>>16); h3 &= masque;
h4+=s1*n3; h5+=(h4>>16); h4 &= masque;

n0=h0+(h1<<16);
n1=h2+(h3<<16);
n2=h4+( (h5 & masque) <<16);
n3=h5>>16;

while(n3)
  {  s1=(n0<NL ? 1 : 0);
     n0-=NL;s0=NM+s1;
     s1=(n1<s0 ? 1 : 0);
     n1-=s0;s0=NH+s1;
     s1=(n2<s0 ? 1 : 0);
     n2-=s0;n3-=s1;
  }
if (n2>=NH)
if (n2>NH || n1>=NM)
if (n2>NH || n1>NM || n0>=NL)
 {  s1=(n0<NL ? 1 : 0);
    n0-=NL;s0=NM+s1;
    s1=(n1<s0 ? 1 : 0);
    n1-=s0;s0=NH+s1;
    s1=(n2<s0 ? 1 : 0);
    n2-=s0;
  }
*previous=n0;
*current=n1;
*next=n2;
}
/*****************************************************/
```

Transformations arithmétiques en C

```
/*******************************************************/
pap(p,q,w)              /* papillon decimation en temps */
register unsigned *p,*q;
unsigned w;
{
register unsigned P,Q;
P= *p;
Q=fois(w,*q);
*p=plus(P,Q);
*q=moins(P,Q);
}

pop(p,q,w)          /* papillon decimation en frequence */
register unsigned *p,*q;
unsigned w;
{
register unsigned P,Q;
P= *p;Q= *q;
*p=plus(P,Q);
*q=fois(moins(P,Q),w);
}

/*******************************************************/
/*      transformee de fourier entiere                 */
/*      decimation en frequence                        */
/*      entrees normales sorties miroir                */
/*******************************************************/
fft(out,n)
unsigned *out;
int n;
{
register unsigned *p,*q,*r,*s,w;
register int i,j,d,l,nloop;
l=n; d=l>>1;
r=WNK;
while (d) { r++; d>>=1;}
d=l>>1;
nloop=1;
while(d)
  {
   p=out;w=1;
   for(j=d;j;j--,p++)
     {
        for(i=nloop,q=p;i;i--,q+=l) pop(q,q+d,w);
        w=fois(w,*r);
      }
    r--;
    nloop<<=1;
    l>>=1; d>>=1;
   }
}
```

```
/*******************************************************/
/*      transformee de fourier inverse                 */
/*      decimation en temps                            */
/*      entrees miroir sorties normales                */
/*******************************************************/
fftinv(out,n)
unsigned *out;
int n;
{
register unsigned *p,*q,*r,*s,w;
register int i,j,d,l,nloop;
l=2; d=1;
nloop=n>>1;
r=WMNK+1;
while(d<n)
  {
   p=out;w=1;
   for(j=d;j;j--,p++)
     {
        for(i=nloop,q=p;i;i--,q+=l) pap(q,q+d,w);
        w=fois(w,*r);
      }
    nloop>>=1;
    l<<=1; d<<=1;
    r++;
   }
j=unsur2(n);j=fois(SI,j);
for(i=n-1,p=out;i>=0;i--,p++) *p=fois(*p,j);
}
/*******************************************************/
```

Produit de grands entiers en C

```
/*****************************************************/
unsigned *FastMult(P,pl,M,ml,N,nl)
unsigned *P,pl,*M,ml,*N,nl;
{
unsigned i,aux,l,*p,*q,*r;
unsigned SIZE;
l=ml+nl;
for(SIZE=2;SIZE<l;SIZE<<=1);

init1();
for(p=M,q=T1,i=ml;i;i--,p++,q++)
   if((aux= *p)<PBASE) *q=aux; else *q=aux-PBASE;
for(i=ml;i<SIZE;i++,q++) *q=0;
for(p=N,q=T2,i=nl;i;i--,p++,q++)
   if((aux= *p)<PBASE) *q=aux; else *q=aux-PBASE;
for(i=nl;i<SIZE;i++,q++) *q=0;
fft(T1,SIZE);
fft(T2,SIZE);
for(i=SIZE,p=T4,q=T1,r=T2;i;i--) *(p++)=fois(*(q++),*(r++));
fftinv(T4,SIZE);

init2();
for(p=M,q=T1,i=ml;i;i--,p++,q++)
   if((aux= *p)<PBASE) *q=aux; else *q=aux-PBASE;
for(i=ml;i<SIZE;i++,q++) *q=0;
for(p=N,q=T2,i=nl;i;i--,p++,q++)
   if((aux= *p)<PBASE) *q=aux; else *q=aux-PBASE;
for(i=nl;i<SIZE;i++,q++) *q=0;
fft(T1,SIZE);
fft(T2,SIZE);
for(i=SIZE,p=T3,q=T1,r=T2;i;i--) *(p++)=fois(*(q++),*(r++));
fftinv(T3,SIZE);

init3();
for(p=M,q=T1,i=ml;i;i--,p++,q++)
   if((aux= *p)<PBASE) *q=aux; else *q=aux-PBASE;
for(i=ml;i<SIZE;i++,q++) *q=0;
for(p=N,q=T2,i=nl;i;i--,p++,q++)
   if((aux= *p)<PBASE) *q=aux; else *q=aux-PBASE;
for(i=nl;i<SIZE;i++,q++) *q=0;
fft(T1,SIZE);
fft(T2,SIZE);
for(i=SIZE,p=T2,q=T1,r=T2;i;i--) *(p++)=fois(*(q++),*(r++));
fftinv(T2,SIZE);

for(i=SIZE,p=T1;i;i--,p++) *p=0;
Propage_retenue(T4,T3,T2,T1,( l>1 ? l : SIZE));
for(i=l,p=P,q=T1;i;i--,p++,q++) *p= *q;
for(i= l-1;i<pl;i++,p++) *p=0;
}
```

```
/*******************************************************/
Propage_retenue(T1,T2,T3,T,SIZE)
register unsigned *T1,*T2,*T3,*T;
unsigned SIZE;
{
register unsigned i;
unsigned next,cur,prev;

for(i=SIZE-2;i;i--)
  {
   trc(*(T1++),*(T2++),*(T3++),&next,&cur,&prev);
   if( ( *(T++) +=prev) < prev) (*T)++;
   if( ( (*T)+=cur) < cur) next++;
   T[1]=next;
  }
trc(*(T1++),*(T2++),*(T3++),&next,&cur,&prev);
if( ( *(T++) +=prev) < prev) (*T)++;
(*T)+=cur;
trc(*(T1++),*(T2++),*(T3++),&next,&cur,&prev);
T[0]+=prev;
}
/*******************************************************/
```

Arithmétique modulo p en assembleur 68020

```
|*****************************************************|
    FACTEUR = 0xfff
    PBASE = 0xfff00001
    PRIM = 17
    PUISS2 = 20
|*****************************************************|
    .text
    .globl  _fois
_fois:
    movl    a7@(0x8),d1
    mulul   a7@(0x4),d0:d1
    divul   #PBASE,d0:d1
    rts

|*****************************************************|
    .globl  _plus
_plus:
    movl    #PBASE,d1
    movl    a7@(0x8),d0
    addl    a7@(0x4),d0
    bcss    L2
    cmpl    d1,d0
    bcss    L1
L2:
    subl    d1,d0
L1:
    rts

|*****************************************************|
    .globl  _moins
_moins:
    movl    a7@(4),d0
    subl    a7@(8),d0
    bccs    L35
    addl    #PBASE,d0
L35:
    rts
```

```
|*****************************************************|

     .data
     .text

     .globl  _fft
_fft:

     link    a6,#0
     moveml  #0x3f3c,sp@-

     movl    _PBASE,a1
     movl    a6@(0xc),d3      |l=n
     movl    d3,d2
     subql   #1,d2
     beq L25
     movl    _WNK,a3      |r=wnk
     movl    d3,d4            |d=l>>1
     asrl    #0x1,d4

L22:                          |while(d)
     addql   #0x4,a3          |r++
     asrl    #0x1,d4          |d>>=1
     bnes    L22

     moveq   #0x1,d2          |nloop=1
     movl    d3,d4            |d=l>>1
     asrl    #0x1,d4
     asll    #0x2,d3          |l entiers longs

L24:                          |while(d)
     movl    a6@(0x8),a5      |p=out
     moveq   #0x1,d7          |w=1
     movl    d4,d5            |j=d
     asll    #2,d4            |d entiers longs
     exg d4,a1

     bras    L27              |while(j)
L28:
         movl    d7,d0
         mulul   a3@,d7:d0
         divul   d4,d7:d0          |w=fois(*r,w)
     addql   #0x4,a5          |p++
L27:
     movl    d2,d6            |i=nloop
     movl    a5,a4            |q=p
     movl    a5,a2            |s=q+d
     addl    a1,a2
     bras    L30
L31:                          |while(i)
     addl    d3,a4            |q+=l
     addl    d3,a2            |s+=l
L30:
     |************************
     movl    a4@,d1       |q
     movl    d4,d0        |base
     subl    d1,d0        |base-q
```

```
    movl    a2@,a0          |s
    subl    a0,d1           |q-s
    bccs    Lpop1
    addl    d4,d1
Lpop1:
    subl    a0,d0           |base-q-s
    bcss    Lpop2
    subl    d4,d0
Lpop2:
    negl    d0
    movl    d0,a4@          |q=q+s
    mulul   d7,d0:d1
    divul   d4,d0:d1
        movl    d0,a2@              |s=w(q-s)
    |************************
    subql   #0x1,d6             |i--
    bnes    L31             |while(i)

    subql   #0x1,d5             |j--
    bnes    L28             |while(j)

    exg d4,a1
    subql   #0x4,a3             |r--
    asll    #0x1,d2             |nloop<<1
    asrl    #0x1,d3             |l>>1
    asrl    #0x3,d4             |d>>1   (1+2)
    bnes    L24             |while(d)
L25:
LE20:
    moveml  sp@+,#0x3cfc
    unlk    a6
    rts
```

```
|************************************************|
    .globl  _fftinv
_fftinv:

    link    a6,#0
    moveml  #0x3f3c,sp@-

    movl    _PBASE,a1
    moveq   #0x8,d3      |l=2 longs entiers
    moveq   #0x1,d4      |d=1
    movl    a6@(0xc),d2
    movl    d2,d5
    subql   #1,d5
    beq L46
    asrl    #0x1,d2      |nloop=n>>1
    movl    _WMNK,a3     |r=wmnk+1
    addql   #0x4,a3

L36:                |while(d>n)
    movl    a6@(0x8),a5  |p=out
    moveq   #0x1,d7      |w=1
    movl    d4,d5        |j=d
    asll    #2,d4        |d entiers longs
    exg d4,a1

    bras    L39      |while(j)
L40:
    movl    d7,d0
    mulul   a3@,d7:d0       |w=fois(w,*r)
    divul   d4,d7:d0
    addql   #0x4,a5         |p++
L39:
    movl    d2,d6           |i=nloop
    movl    a5,a4           |q=p
    movl    a4,a2           |s=q+d
    addl    a1,a2
    bras    L42
L43:
    addl    d3,a4             |q+=l
    addl    d3,a2             |s+=l
L42:
    |************************
    movl    a2@,d0      |s
    mulul   d7,d1:d0    |sw
    divul   d4,d1:d0
Lpap0:
    movl    a4@,a0      |q
    movl    d4,d0       |base
    subl    d1,d0       |base-sw
    subl    a0,d0       |base-sw-q
    bcss    Lpap1
    subl    d4,d0
Lpap1:
    negl    d0
    movl    d0,a4@
    movl    a0,d0
    subl    d1,d0       |q-sw
    bccs    Lpap2
```

```
    addl    d4,d0
Lpap2:
    movl    d0,a2@
    |************************
    subql   #0x1,d6          |i--
    bnes    L43         |while(i)

    subql   #0x1,d5         |j--
    bnes    L40       |while(j)

    exg d4,a1
    asrl    #0x1,d2       |nloop>>=1
    asll    #0x1,d3       |l<<=1
    asrl    #0x1,d4       |d<<=1 (1-2)
    addql   #0x4,a3       |r++
    cmpl    a6@(0xc),d4 |while(d<n)
    blts    L36
L37:
    |*******
    movl    a6@(0xc),d5
    moveq   #1,d0
    lsrl    #1,d5
    bcss    L50
    movl    _DEMI,d4
L52:
    lsrl    #1,d0
    bccs    L51
    addl    d4,d0
L51:
    lsrl    #1,d5
    bccs    L52
L50:
    |*******
    movl    _PBASE,d4
    mulul   _SI,d1:d0    |j=j*SI
    divul   d4,d1:d0
    movl    d1,d5

    movl    a6@(0xc),d6 |i=n
    movl    a6@(0x8),a5 |p=out
L47:
    movl    d5,d0
    mulul   a5@,d1:d0
    divul   d4,d1:d0
    movl    d1,a5@+
    subql   #0x1,d6         |i--
    bnes    L47      |while(i)
L46:
    moveml  sp@+,#0x3cfc
    unlk    a6
    rts
```

```
|*************************************************|
    .globl  _trc
_trc:

    lea a7@(4),a0
    moveml  #0x3f04,sp@-

    clrl    d7
    movl    a0,a5
    movl    a5@+,d6
    movl    d6,d0
    mulul   _N1L,d5:d6
    mulul   _N1H,d4:d0
    addl    d0,d5
    addxl   d7,d4

    movl    a5@+,d2
    movl    d2,d0
    mulul   _N2L,d1:d0
    mulul   _N2H,d3:d2
    addl    d2,d1
    addxl   d7,d3
    addl    d0,d6
    addxl   d1,d5
    addxl   d3,d4
    bcc L1
    subl    _NL,d6
    movl    _NM,d0
    subxl   d0,d5
    movl    _NH,d0
    subxl   d0,d4
L1:
    movl    a5@+,d2
    movl    d2,d0
    mulul   _N3L,d1:d0
    mulul   _N3H,d3:d2
    addl    d2,d1
    addxl   d7,d3
    addl    d0,d6
    addxl   d1,d5
    addxl   d3,d4
    bcc L2
    subl    _NL,d6
    movl    _NM,d0
    subxl   d0,d5
    movl    _NH,d0
    subxl   d0,d4
L2:
    movl    d6,d3
    movl    d5,d2
    movl    d4,d1
    subl    _NL,d6
    movl    _NM,d0
    subxl   d0,d5
    movl    _NH,d0
    subxl   d0,d4
    bcc L3
    movl    d3,d6
```

```
    movl    d2,d5
    movl    d1,d4
L3:
    movl    a5@+,a0
    movl    d4,a0@
    movl    a5@+,a0
    movl    d5,a0@
    movl    a5@+,a0
    movl    d6,a0@

    moveml  sp@+,#0x20fc

    rts
|*****************************************************|
```

Arithmétique modulo p en assembleur T414

```
/*************************************************************/
#define pbase 4293918721
#define facteur 0xfff
#define puissde2 20
#define primroot 17
/*************************************************************/
unsigned fois(a,b)
unsigned a,b;
{
asm("   ldc 0");
asm("   ldl %a");
asm("   ldl %b");
asm("   lmul");
asm("   ldc 4293918721");
asm("   ldiv");
asm("   rev");
}
/*************************************************************/
unsigned plus(a,b)
unsigned a,b;
{
asm("   ldc 0");
asm("   ldc 4293918721");
asm("   ldl %b");
asm("   diff");
asm("   ldl %a");
asm("   rev");
asm("   ldiff");
asm("   rev");
asm("   cj  Lp1");
asm("   ldc 4293918721");
asm("Lp1:   sum");
}
/*************************************************************/
unsigned moins(a,b)
unsigned a,b;
{
asm("   ldc 0");
asm("   ldl %a");
asm("   ldl %b");
asm("   ldiff");
asm("   rev");
asm("   cj  Lm1");
asm("   ldc 4293918721");
asm("Lm1:   sum");
}
/*************************************************************/
```

INPG

Grenoble, le 2 octobre 1991

DÉPARTEMENT DES ÉTUDES DOCTORALES

Affaire suivie par Michèle SIMEON
Tél : 76.57. 4525

N/Réf. : JPV/MS

Objet :

AUTORISATION de SOUTENANCE

Vu les dispositions de l'arrêté du 23 Novembre 1988 relatif aux Etudes Doctorales

Vu les rapports de présentation de :

Monsieur COSNARD Michel Docteur d'Etat Professeur ENS LYON
Monsieur VUILLEMIN Jean Docteur d'Etat DEC PARIS

Monsieur HERREROS Yvan

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de l'INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE, spécialité : "Mathématiques appliquées"

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
M. GARNIER