Cours de Calcul Formel

Depuis les annés 70 on sait utiliser l'ordinateur pour non seulement le calcul numérique en variable flottante et en prcision finie mais également pour le calcul algébrique c'est à dire traitant des calculs avec paramtres, des polynômes, des fractions. Par rapport au calcul numérique traditionel, on chappe au calcul d'erreur traditionel, et au problème de propagation des erreurs, mais en contrepartie le calcul formel traite des nombres de plus en plus gros, sur lesquels les oprations sont de plus en plus longues à être exécutes.
Un exemple pour le calcul du déterminant d'une matrice $A = (a_{i,j})_{1 \leq i,j \leq 25}$, on peut vouloir utiliser la formule :

$$det(A) = \sum_{\sigma \in S_{25}} \epsilon(\sigma) a_{1,\sigma(1)} a_{2,\sigma(2)} \cdots a_{25,\sigma(25)}.$$

Or il y a dans cette somme $25! \approx 1,55 \cdot 10^{25}$ produit, en supposant qu'un ordinateurs effectue 10 milliards de produits par seconde, il faudrait compter 49.000.000 d'années.
Il devient donc indispensable d'valuer le temps que mettra un algorithme pour fournir un résultat avant même de le programmer. De même on voudrait pouvoir comparer les algoritmes entre eux.
On trouve toute une série de logiciels de calculs formels, les plus connues sont Derive, Mathematica et Maple, mais aussi Mupad qui a l'avantage d'être gratuit, nous choisissons de travailler sur Maple

Chapitre 1

Programmer sur Maple

S'agissant d'un langage de calcul formel deux utilisations sont possibles, l'une de boites noires, on utilise des fonctionalités déja enregistrés dans les bibliotèques du programme, Maple en l'occurence, en se servant de l'aide en ligne et des manuels. On explorera en Travaux Pratiques les différentes fonctionnalités de Maple, mais dans cet exposé on s'attachera essentiellement aux rudiments nécessaires à la programmation de procédures simples.

1.1 Structures itératives et conditionnelles

1.1.1 Les boucles

La structure générale pour définir une boucle itérative est :

for...from...by...to...while...do
ici est placé le corps de la boucle
od ;

La commande **for** introduit le nom de la variable qui sert de compteur ;
from fixe la valeur initiale de ce compteur, **by** la valeur de l'incrémentation réalisée à chaque tour de la boucle ;
to précise la valeur finale du compteur ;
quant à **while**, il permet d'introduire une condition supplémentaire qui si elle n'est pas réalisée, arrête la boucle.

Chacun des termes **for...from...by...to...while** peut être omis, pourvu que cela garde un sens et n'entrainne pas la création d'une boucle infernale : si c'est **from** qui est absent le compteur commencera à 1, si c'est **by** le compteur progressera de 1 à chaque tour de boucle.

Exemples :
Calculez la somme des carrés des nombres impairs plus petit que $\sqrt{1000}$:

```
> som :=0 : # initialisation
  for i by 2 while i^2 < 1000
  do som :=som+i^2
  od :
>som ; # évaluation
                              5456
> i ;
                               33
```

1.1.2 Les tests

Les test se font à l'aide de la structure :

if...then...else...fi ;

if introduit une quantité booléenne (qui aura pour valeur *true* ou *false*) ; à la suite de **then** et de **else**, qui peut être omis on écrit des séquences d'opération à réaliser suivant la valeur du booléen. Si l'on veut faire plus compliqué, on a aussi la structure :

if...then...elif...then...elif... ... ...then...else...fi ;

et ceci avec autant de **elif...then** que l'on voudra.

Ainsi on peut calculer la somme des carrés des premiers nombres premiers dont la somme ne dépasse pas $\sqrt{1000}$:

```
> som1 :=4 :
>for i from 3 by 2 while i^2 < 1000 do
  if isprime(i) then som1 :=som1+i^2 ;
  fi ;
  od ;
> som1 ;
                              3358
```

1.1.3 Procédures

Un programme s'écrit sous forme de procédure. La définition d'une procédure se fait de la manière suivante :

nom de la procédure := **proc**(séquence de paramètres)
local...# séquence de paramètres locales
option... ;
ici le corps de la procédure
end ;

Les deux lignes commençant par **option** et **local** peuvent être omises. La commande **local** introduit les variables internes à la définition de la procédure. Toute variable non déclarée locale est considérée comme globale et conserve sa valeur hors la procédure.

Exemples :

Voici une procédure qui retourne la somme des carrés des nombres premiers dont le carré ne dépasse pas x :

```
> somp :=proc(x)
 local i,s ;
 s :=4 ; i :=3 ;
 for i by 2 while i^2 < x do
 if isprime(i) then s :=s+i^2 ;
  fi ;
 od :
 s ;
 end ;
> # exemple d'application :
> somp(1000), somp(12865) ;
                3358,122705
> s ;
                      s
```

1.1.4 Récursivité

Une procédure est dite récursive si lors de son exécution elle fait appel à elle même. Sinon on dit que la procédure est itérative. Les répétitions sont alors gérés par des boucles définies à l'aide de **for** ou de **while**.

Un exemple de procédure récursive, la fonction puissance : $x \mapsto x^n$, et le

calcul pour une certaine valeur. La définition se fait par récurrence en posant

$$x^0 = 1 \text{ et pour } n \geq 1, \quad x^n = x.x^{n-1}.$$

Bien qu'étant immédiatement construite cette procédure atteint tout aussi immédiatement ses limites.
Essayons de calculer 2^{500} pour réponse un message indiquant qu'il y a trop de niveaux de récursion. Ayant à exécuter expr(2,500), Maple fait appel expr(2,499) et stocke en mémoire les données et les paramètres nécessaires au calcul de expr(2,500), puis exécute expr(2,499).
Après un certain nombre d'appel, la mémoire est saturée de résultats intermédiaires en attente.
La récursivité reste trés limitée et dans bien des cas il est préférable d'écrire une version itérative du programme.
Ainsi pour l'exponentiation il est préférable de faire appel à cette procédure itérative :

```
> expi :=proc(x,n : :nonnegint)
  local i,r ;
  for i from 2 to n do r :=x*r :
  od :
  r ;
  end :
```

1.1.5 Exponentiation dichotomique

On peut se demander si on peut réaliser le calcul de x^n en moins de $n-1$ opérations arithmétiques. La réponse est positive. On peut remarquer :

$$x^n = \begin{cases} x^{\frac{n}{2}} & \text{si } n \text{ est pair,} \\ x(x^{\frac{n-1}{2}}) & \text{si } n \text{ est impair.} \end{cases}$$

c'est l'exponentiation dichotomique.
L'exponentiation dichotomique se traduit immédiatement en une procédure récursive :

```
> expdr :=proc(x,n : :nonnegint)
  if n=0 then RETURN(1)
  elif (n mod 2)=0 then expdr(x, n/2)^2
  else x*expdr(x,(n-1)/2)^2
  fi
  end :
```

Calculons x^{27} en utilisant cette méthode

$$x^{27} = x.(x^{13})^2 = x.(x.(x^6)^2)^2 = x.(x.(x^3)^4)^2 = x.(x.(x.x^2)^4)^2.$$

On voit très bien que 7 multiplications au lieu de 26 ont été utilisées. On peut d'une manière générale borner le nombre de multiplications utilisées

Proposition 1.1.1 *La procédure expdr(x,n) nécessite au plus* $2([log_2(n)]+1)$ *multiplications.*

Démonstration : On note $N_a(n)$ le nombre d'appels récursifs de expdr que va effectuer la commande expdr(x,n). On va montrer : $N_a(n) = [log_2(n)]+1$; par récurrence.
On a $N_a(1) = 1$ et $N_a(2) = 2$. Supposons la formule exacte jusqu'au rang n. Si n est impair alors expdr(n+1) invoque expdr(x,$\frac{n+1}{2}$) ; expdr(x,n+1) appelle donc expdr une fois auxquelles s'ajoutent le nombres d'appels réalisés par expdr(x,$\frac{n+1}{2}$) lui même. De sorte qu'on a dans ce cas :

$$N_a(n+1) = 1 + \left(\left[\log_2\left(\frac{n+1}{2}\right)\right] + 1\right) = 1 + [\log_2(n+1)].$$

Si n est pair, expdr(x,n+1) va invoquer expdr(x,$\frac{n}{2}$), on a dans ce cas :

$$N_a(n+1) = 1 + \left(\left[\log_2\left(\frac{n}{2}\right)\right] + 1\right) = 1 + [\log_2(n)].$$

On termine en remarquant que pour n pair $[\log_2(n)] = [\log_2(n+1)]$. En effet sinon posons $A = [\log_2(n+1)]$, on a $\log_2(n) < A \leq \log_2(n+1)$ puis $n < 2^A \leq n+1$, puis puisque A est entier $2^A = n+1$, ce qui est absurde puisque $n+1$ est impair.
La procédure à chaque appel, fait au plus 2 multiplications (dans le cas où l'exposant est impair) ; d'où au pire $2N_a(n)$ multiplications.
On peut montrer que la méthode d'exponentiation dichotomique pour calculer les puissances n'est pas trop loin de l'optimale, au sens où si on peut calculer une puissance x^n en k multiplication alors $k \geq \log_2(n)$
En effet posant

$$\begin{aligned}
P_0 &= x \\
P_1 &= xP_0 \\
P_2 &= y_2z_2 \text{ avec } y_2, Z_2 \in \{P_0, P_1\} \\
P_3 &= y_3z_3 \text{ avec } y_3, Z_3 \in \{P_0, P_1, P_2\} \\
&\vdots \\
P_k &= y_kz_k = x^n \text{ avec } y_k, z_k \in \{P_0, P_1, \ldots, P_{k-1}\},
\end{aligned}$$

implique que tout $i \in [1,k]$, $P_i = x^{\alpha_i}$ avec $\alpha_1 = 2,\ \alpha_2 \leq 2^2,\ \alpha_3 \leq 2^3$ ainsi de suite $\alpha_k = n \leq 2^k$ et donc $k \geq \log_2(n)$.

Remarques : :

On s'aperçoit que l'exponentiation dichotomique n'est pas toujours optimale. En effet on peut écrire :

$$x^{27} = \left(\left(x^3\right)^3\right)^3.$$

Et chaque cube nécessitant 2 multiplications, x^{27} peut se calculer en 6 multiplications.

Autre exemple, le calcul de x^{15} par la méthode dichotomique nécessite 6 multiplications calculer x^2, x^4, x^8, puis multiplier x et ces trois quantités ensemble :$x^{15} = x(x^2)(x^4)(x^8)$. En revanche si on commence par calculer $x^5 = x(x^2)^2$ en 3 multiplications et poser $x^{15} = (x^5)^3$, il ne faut plus que 5 multiplications.

De façon générale si $n = pq$ avec p plus petit facteur premier de n on peut commencer par calculer x^p et ensuite élever x^p à la puissance p, si n est premier on calcule x^{n-1} qu'on multiple ensuite par x. On connait le principe qui pour chaque exposant n particulier permet le calcul de x^n en un minimum de multiplications possibles. On le construit à partir du principe des chaines d'additions :

Une chaine d'addition pour n est une suite d'entiers

$$a_0 = 1, a_1, a_2, \ldots, a_r = n,$$

vérifiant :

$$\forall i \in [1,r]\ \exists j,k \quad 1 \leq k \leq j < i\ s.t.\ a_i = a_j + a_k.$$

On cherche alors, pour chaque n, la chaine d'addition la plus courte possible ; on applique à la description de la méthode la plus efficace permettant de calculer x^n à l'aide de multiplications.

Illustration de cette méthode :

Supposons que l'on cherche à calculer x^{54} Par la méthode dichotomique il faut calculer $x, x^2, x^4, x^8, x^{16}, x^{32}$, puis effectuer les produits nécessaires de ces nombres entre eux. Si on ne fait qu'une multiplication à la fois et qu'on range les termes obtenus par ordre de puissance croissante, on obtient la suite : $x, x^2, x^4, x^6, x^8, x^{16}, x^{22}x^{32}, x^{54}$ qui correspond a la chaine d'additions $(1,2,4,6,8,16,22,32,54)$. Cette chaine d'addition n'est pas minimale,

54 peut être obtenu par la chaine $(1, 2, 3, 6, 9, 18, 27, 54)$ ce qui signifie que x^{54} peut être obtenu en 7 multiplications :

$$x_0 = x, x_1 = x_0.x_0, x_2 = x_0.x_1, x_3 = x_2.x_2,$$
$$x_4 = x_3.x_2, x_5 = x_4.x_4, x_6 = x_5.x_4, x_7 = x_6.x_6$$

1.2 Preuve de Programme

On peut penser, conformément au principe énoncé plus haut, pouvoir améliorer les performances d'une procédure en l'écrivant sous forme itérative :

```
> expdi :=proc(x,n : :nonnegint)
  local p,r,k ;
  k :=n : p :=x : r :=1 :
  while k>0 do
  if irem(k,2)=1 then r :=p*r : k :=(k-1)/2 :
  else k :=k/2 :
  fi ;
  p :=p^2 :
  od :
  r ;
  end :
> # exemples d'application :
> expdi(2,5), expdi(2,10) ;
                              32,1024
```

On vérifie sur des exemples que cette procédure donne bien les résultats attendus ; mais il n'est pas évident qu'il en est toujours ainsi.
Pour prouver qu'un algorithme réalise bien le calcul pour lequel on l'a conçu, on doit faire : 1) prouver qu'il retourne bien un résultat en un nombre fini d'opérations et 2) que le résultat retourné est bien celui attendu.
Pour ce faire on fait apparaitre des invariants de la procédure. Sur notre exemple, pour montrer que l'algorithme ne possède qu'un nombre fini d'étapes, il suffit de remarquer qu'à chaque tour de boucle, la valeur contenue dans k est remplacée par $[\frac{k}{2}]$, de sorte que la valeur prise par k est est une suite d'entiers strictement décroissante. La valeur initiale étant $n > 0$, en $[\log_2(n)] + 1$ étapes, la valeur de k s'annulera et on sortira de la boucle while, de sorte que l'algorithme se termine.

Considérons maintenant les variables qui apparaissent dans la procédure. D'abord p initialisée à x, à chaque tour de boucle, p est remplacé par son carré, de sorte qu'au i-ième tour de boucle , p a pour valeur x^{2^i}.
Quelles sont les valeurs prises par r ? Notons c_i les restes de la division de k par 2 au i-ième tour de boucle ; il vaut 0 ou 1. A la fin du premier tour de boucle $r = x^{c_1}$. Au second tour de boucle r prend la valeur $x^{c_2 2 + c_1}$.
Supposons qu'au i-ième tour r contienne la quantité $x^{c_i 2^{i-1} + \cdots + c_i}$. Au tour suivant si c_{i+1} vaut 0, r conserve la même valeur, autrement r devient $r.p = x^{c_i 2^{i-1} + \cdots + c_i}.x^{2^i}$. Dans tous les cas, r est devenu $x^{c_{i+1} 2^i + \cdots + c_i}$.
À la fin de la procédure r contient x^n avec n écrit en base 2.

Chapitre 2

Codage et arithmétique élémentaire

2.1 Base de numération et codage informatique

Théorème 2.1.1 *Soit $b \in \mathbb{N}$ différent de 0 et 1. Soit $n \in \mathbb{N} \setminus \{0\}$, et soit k l'unique entier naturel tel que $b^{k-1} \leq n < b^k$; alors il existe une unique suite de nombres entiers $c_0, c_1, \ldots, c_{k-1}$ de l'intervalle [0,b-1] avec $c_{k-1} \neq 0$ tels que :*

$$n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}.$$

Démonstration : Par récurrence sur n. Si $n \in]0, b[$, alors on pose $c_0 = n$. Supposons alors que le résultat soit établi pour tout $n \in]0, b^k[$ avec $k > 0$, et considérons un $n \in [b^k, b^{k+1}[$. La division euclidienne de n par b permet d'écrire $n = n_1 b + c_0$ avec $c_0 \in [0, b-1]$. Il est clair que $n_1 \in]0, b^k[$.On peut donc écrire :

$$n_1 = c_k b^{k-1} + \cdots + c_2 b + c_1,$$

avec les $c_i \in [0, b-1]$ et $c_k \neq 0$.
Montrons l'unicité d'une telle écriture. Remarquons d'abord que si $n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}$ avec pour tout $i = 0, 1, \ldots, k-1,\ b > c_i \geq 0$ et $c_{k-1} \neq 0$. En effet, on a :

$$c_{k-1} b^{k-1} \leq n \leq (b-1) + (b-1)b + (b-1)b^2 + \cdots + (b-1)b^{k-1},$$

de sorte que $b^{k-1} \leq n \leq (b-1)\frac{b^k-1}{b-1}$. Il en résulte que k est unique. Supposons que n s'écrive également $n = c'_0 + c'_1 b + c'_2 b^2 + \cdots + c'_{k-1} b^{k-1}$. Soit j_0 le plus grand indice pour lequel $c_{j_0} \neq c'_{j_0}$. Si $j_0 = 0$, il est immédiat que $c_0 = c'_0$, sinon en faisant la différence des deux écritures de n, on obtient :

$$(c_{j_0} - c'_{j_0})b^{j_0} = \sum_{i=0}^{j_0-1}(c_i - c_i\prime)b^i.$$

Mais le terme de gauche est plus grand ou égal à b^{j_0}, alors que celui de droite est strictement inférieur à cette valeur. On dit alors que le nombre n est écrit en base b, les nombres $c_0, \ldots, c_{k-1}$ sont les chiffres de cette écriture, c_i chiffre de poids i. On note $n = (c_{k-1}, \ldots, c_1, c0)_b$.
En base décimale (b=10) on omet le b en indice, les parenthèses et les virgules.
Exemples : Pour $n = 10^6$, on a $n = 1000000 = (11110100001001000000)_2 = (11333311)_7 = (3, 5, 24, 8, 20)_{26}$. Cette méthode de codage d'un entier naturel s'appelle la numération de position.

Proposition 2.1.1 *L'entier naturel $n \neq 0$ s'écrit en base b avec $[Log_b(n)]+1$ chiffres.*

Démonstration : Le théorème précédent nous dit que si n est dans l'intervalle $[b^k, b^{k+1}[$ alors il s'écrit avec $k+1$ chiffres en base b avec $k\ln(b) \leq \ln(n) < (k+1)\ln(b)$ et donc $k = [\ln_b(n)]$.
Soit M la valeur maximale d'un nombres pouvant être contenu dans un registre de processeur. On peut prendre comme base de numération tout nombre entier b plus petit que M. Le codage de n se fait tout simplement en enregistrant la suite finie des chiffres. La base b choisie est b^{k-1} si $M = 2^k$, ou la plus grande puissance de 10 inférieure ou égale à 2^{k-1}.

Proposition 2.1.2 *Supposons qu'un entier n soit écrit sous la forme d'un polynôme b, où b est un entier plus grand que 1,*

$$n = \alpha_0 + \alpha_1 b + \cdots + \alpha_k b^k \quad \alpha_i \in \mathbb{N}.$$

Soit d'autre part l'écriture de n en base b : $n = (c_{k'}, c_{k'-1}, \ldots, c_0)_b$. Alors on a :

$$c_0 = \alpha_0 - q_0 b \text{ ou } q_0 = \left[\tfrac{\alpha_0}{b}\right]$$
$$c_i = \alpha_i + q_{i-1} - q_i b \text{ ou } q_i = \left[\tfrac{\alpha_i + q_{i-1}}{b}\right] \text{ si} i = 1, \ldots, k.$$

De plus $c_{k+j} = c'_{j-1}$ pour $j = 1, \ldots, k' - k$ si

$$q_k = c'_0 + c'_1 b + \cdots + c'_{k'-k-1} b^{k'-k-1} b,$$

désigne l'écriture de q_k en base b.

Démonstration : La connaissance de n à l'aide des α_i implique que $n \leq b^k$ et donc que n en base b aura au moins $k+1$ chiffres de sorte que $k' \geq k$. On peut remarquer d'autre part que pour chaque $i = 0, \ldots, k$, par construction-même, les c_i sont positifs et inférieurs strictement à b car ce sont les restes de la division par b de $\alpha_i + q_{i-1}$. On a ensuite pour chaque $i = 1, \ldots, k,\ \alpha_i + q_{i-1} = bq_i + c_i$, d'où de proche en proche :

$$\begin{aligned}
n &= c_0 + q_0 b + \alpha_1 b + \alpha_2 b^2 + \cdots + \alpha_k b^k \\
&= c_0 + c_1 b + q_1 b^2 + \alpha^2 b^2 + \cdots + \alpha_k b^k \\
&= c_0 + c_1 b + (q_1 + \alpha_2) b^2 + \cdots + \alpha_k b^k \\
&= c_0 + c_1 b + (q_1 + \alpha_2) b^2 + \cdots + \alpha_k b^k \\
&= c_0 + c_1 b + (c_2 + q_3 b) b^2 + \cdots + \alpha_k b^k \\
&= \cdots \\
&= c_0 + c_1 b + c_2 b^2 + \cdots + (c_{k-1} + q_k b) b^{k-1} + \alpha_k b^k \\
&= c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1} + (q_k + \alpha_k) b^k \\
&= c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1} + (c_k + q_{k+1} b) b^k \\
&= c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1} + c_k b^k + q_{k+1} b^{k+1} \\
&= c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1} + c_k b^k + (c'_0 + c'_1 b + \\
&\quad c'_2 b^2 + \cdots + c'_{k'-1} b^{k'-1} + c_{k'} b^{k'}) b^{k+1} \\
&= c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1} + c_k b^k + c'_0 b^{k+1} + c'_1 b^{k+2} + \\
&\quad c'_2 b^{k+3} + \cdots + c'_{k'-1} b^{k+k'} + c_{k'} b^{k'+k+1}.
\end{aligned}$$

2.2 Addition et soustraction

Soient deux nombres n et n' chacun de k chiffres en base b.
$n = (c_{k-1}, \ldots, c_1, c_0)_b$
$n' = (c'_0 k - 1, \ldots, c'_1, c'_0)_b$
$n + n' = (c_0 + c'_0) + (c_1 + c'_1) b + \cdots + (c_{k-1} + c'_{k-1})_b^{k-1}$

Proposition 2.2.1 *Dans l'application du processus de propagation de retenues pour calculer l'écriture en base b de $n + n'$, les retenues valent toujours 0 ou 1. On n'a donc pas besoin d'effectuer de divisions.*

Démonstration : On a

$$0 \leq c_0 + c'_0 \leq 2(b-1) < 2b,$$

si bien que $0 \leq \frac{c_0+c'_0}{b} < 2$; ainsi q_0 vaut 0 si $c_0 + c'_0 < b$ et 1 sinon. Supposons que $q_{i-1} \in \{0, 1\}$ $i < k$. On a alors :

$$c_i + c'_i + q_{i-1} < 2b - 1$$

et donc $q_i \in \{0, 1\}$. La retenue vaut 0 si et seulement si $c_i + c'_i + q_{i-1} < b$

```
> som :=proc(N1,N2,b)
  local i,som,x ;
  som :=NULL, r :=0 ;
  for i to nops(N1) do
  x :=N1[i]+N2[i]+r ;
  if x<b then som :=som,x ; r :=0 ;
  else som :=som,x-b ;r :=1 ;
  fi ;
  od :
  if r=1 then som :=som,r ;
  fi ;
 [som] ;
  end :
> # exemples d'application :
> som([1, 8], [2, 3], 10) ;
                              [3, 1, 1]
> som([1, 0], [2, 3], 2) ;
                              [1, 2, 1]
> som([5, 0], [2, 3], 2) ;
                              [5, 2, 1]
```

Proposition 2.2.2 *La complexité binaire d'une addition de deux nombres entiers est en* $O(max(\log n, \log b))$

Démonstration : Il suffit de remarquer que si n et n' ont k chiffres chacun, leur somme en compte $k + 1$ au plus. En effet $n < 2^k$, $n' \leq 2^k$, d'où $n + n' < 2b^k < b^{k+1}$

2.3 Multiplication

Soient deux nombres n et n' chacun de k chiffres en base b.
$n = (c_{k-1}, \ldots, c_1, c_0)_b$

$n' = (c'_0 k - 1, \ldots, c'_1, c'_0)_b$
Pour obtenir nn', une méthode consiste à faire les calculs des k' produits $c'_j b^j n$ puis à faire la somme. Outre les additions les seules parties nouvelles consiste à évaluer les $c'_j n$, par décalage de j chiffres vers la gauche on déduit immédiatement l'écriture en base b de $c'_j n b^j$. Mais $c'_j n = \sum_0^k c'_j c_i b^i$ a peu de chance de représenter $c'_j n$ en base b, aussi doit -on appliquer la méthode de propagation des retenues.

Proposition 2.3.1 *L'application du processus de propagation des retenues pour calculer l'écriture en base b de $cn = \sum_0^k cc_i b^i$ où $n = (c_k, \ldots, c_0)$ et où $c \in [0, b[$, ne produit que des retenues qui s'écrivent avec un seul chiffre en base b, et ne nécessite que des multiplications et des divisions élémentaires de nombres qui, en base b, possède au plus deux chiffres par un nombre qui n'en posséde qu'un seul.*

Démonstration : Regardons d'abord q_0.
Comme $cc_0 \leq (b-1)^2$, on a , b étant un entier supérieur ou égal à 2 :

$$q_0 = \left[\frac{cc_0}{b}\right] \leq b - 2 + \frac{1}{b} \leq b - 1.$$

On voit que q_0 ne possède qu'un chiffre en base b et que le chiffre de poids le plus faible de cn est le reste de la division de cc_0 par b.
Supposons alors que $0 \leq q_{i-1} < b$. On a alors :

$$cc_i + q_{i-1} \leq (b-1)^2 + (b-1) \leq (b-1)b$$

aussi $cc_i + q_{i-1}$ possède au plus deux chiffres en base b, et donc q_i ne possède qu'un seul puisque :

$$q = \left[\frac{cc_1 + q_{i-1}}{b}\right] \leq b - 1.$$

Proposition 2.3.2 *Une multiplication de deux entiers naturels n e n' a une complexité au pire de $O(\log(n)\log(n'))$ opérations binaires.*

Démonstration : Le calcul d'un $c_j n$ coûte $O(k)$ opérations. Comme il y a k' produits à évaluer, cela nécessitera $0(kk')$ opérations. Nous devrons ensuite ajouter tous ces produits les uns aux autres. L'addition de $c'_0 n$ à $c'_1 nb$ fournit un résultat à $k+3$ chiffres au plus en $0(k+2)$ opérations ; qui ajouté à $c'_2 nb^2$ fournit un résultat à $k+4$ chiffres au plus en $0(k+3)$ opérations. Le coût de toutes ces additions sera

$$O((k+2)+(k+3)+(k+4)+\cdots+(k+k'+1)) = O(kk') = O(\log(n)\log(n')),$$

opérations élémentaires.

2.4 division

Commençons par la division de 14598 par 113 quotient 129 reste 21

Proposition 2.4.1 *Soient x et y deux entiers naturels non nuls et tels que $y < x$ et que $x = (c_n, \ldots, c_0)_b$ et $y = (c'_m, \ldots, c'_0)_b$.*
Soit x_0 le plus petit nombre $(c_n, \ldots, c_k)_b$ formé des chiffres les plus à gauche de x, qui soit supérieur ou égal à y.On définit alors les trois suites finies suivantes $(q_i), (r_i), (q_i)$ pour $1 \leq i \leq k$ par :

$$\begin{cases} q_i &= \left[\frac{x_{i-1}}{y}\right] \\ r_i &= x_{i-1} - q_i y \\ x_i &= r_i b + c_{k-i} \end{cases}$$

On pose aussi $q_{k+1} = \left[\frac{x_k}{y}\right]$ et $r_{k+1} = x_k - q_{k+1}y$. Alors, le quotient exact de x par y vaut $(q_1, \ldots, q_{k+1})_b$, et le reste de cette division vaut r_{k+1}.

Illustrons la démarche sur l'exemple : on a $x_0 = 145$, $q_1 = 1$, $r_1 = 32$, $x_1 = 329$, $q_2 = 2$, $r_2 = 103$, $x_2 = 1038$, $q_3 = 9$, $r_3 = 21 = r$.
Démonstration : On remarque qu'on a $x = x_0 b^k + c_{k-1} b^{k-1} + \ldots + c_0$.
Par récurrence. On a d'abord :

$$\begin{cases} x_1 &= r_1 b + c_{k-1} \\ &= (x_0 - q_1 y)b + c_{k-i} \\ &= (x_0 + c_{k-i}) - q_1 by \end{cases}$$

Supposons que l'on ait au rang $i < k$,

$$x_i = (x_0 b^i + c_{k-1} b^{i-1} + \cdots + c_{k-i}) - (q_1 b^i + \cdots + q_i b)y.$$

On a alors au rang $i + 1$:

$$\begin{cases} x_{i+1} &= r_{i+1} b + c_{k-(i+1)} \\ &= (x_i - q_{i+1} y)b + c_{k-(i+1)} \\ &= (x_0 b^{i+1} + \cdots + c_{k-i} b + c_{k-(i+1)}) - (q_1 b^{i+1} + \cdots + q_i b^2 + q_{i+1})y \end{cases}$$

A l'étape k on a donc

$$x_k = x - (q_1 b^k + \cdots + q_k b)y.$$

Mais $x_k = yq_{k+1} + r_{k+1}$, de sorte que :

$$x = (q_1 b^k + \cdots + q_k b + q_{k+1})y + r_{k+1}.$$

On peut remarquer que chaque quotient $q_i = \left[\frac{x_{i-1}}{y}\right]$ a bien un seul chiffre en base b. En effet on a par construction même $x_0 < by$ et ensuite pour tout $i \leq k$, $x_i < by$:

$$\left\{ \begin{array}{rcl} x_i & = & r_i b + c_{k-i} \\ & \leq & (y-1)b + c_{k-i} < (y-1)b + b = yb. \end{array} \right.$$

Il en résulte que l'écriture en base b du quotient $\left[\frac{x}{y}\right]$ est bien $(q_1, \ldots, q_{k+1})_b$ et que r_{k+1} est le reste de cette division.

Les divisions nécessaires pour obtenir les q_i successifs sont généralement irréalisables pour un processeur, dividende et diviseur pouvant avoir un nombre de chiffres quelconques, les seules divisions admissibles pour le processeur étant la division d'un nombre à deux chiffres par un nombre à un chiffre.

La méthode que l'on enseigne consiste à découvrir le chiffre qui convient en essayant le résultat de la division de la division du plus haut chiffre de x ou de ses deux plus haut chiffres par le plus haut chiffre de y. Mais cela ne marche pas à tous les coups, ainsi dans l'exemple traité, on a $q_1 = \left[\frac{145}{113}\right] = 1 = \left[\frac{1}{1}\right]$, mais $q_2 = \left[\frac{329}{113}\right] = 2 \neq \left[\frac{3}{1}\right]$, et pire $q_3 = \left[\frac{1038}{113}\right] = 9$ n'est égal ni à 1, ni à 10.

Nous avons besoin d'un lemme technique

Lemme 2.4.1 *Soit y un entier naturel $y = (c_k, \ldots, c_0)_b$. Alors il existe un entier naturel ν tel que $\nu y = (c'_k, \ldots, c'_0)_b$ ait le même nombre de chiffre que y en base b et tel que de plus c'_k appartienne à $[[\frac{1}{2}b], b[$.*

Démonstration : Remarquons d'abord que si $c_k \in \left[\left[\frac{1}{2}b\right], b\right[$, il suffit de prendre $\nu = 1$. Dans le cas contraire on pose :

$$\nu = \left[\frac{b}{c_k + 1}\right].$$

On a $\nu \geq 1$ car $c_k < b$ et par suite

$$y \leq \nu y < \nu(c_k b^k + b^k) \leq \frac{b(c_k b^k + b^k)}{c_k + 1} = b^{k+1},$$

et donc νy a autant de chiffres que y. Montrons que c'_k appartient à $\left[\left[\frac{1}{2}b\right], b\right[$. Comme c'_k s'obtient en ajoutant à νc_k la retenue positive ou nulle, on a $c'_k \geq \nu c_k$, et on vérifie

$$c_k\left(\frac{b}{c_k+1}-1\right)-\left(\frac{b}{2}-1\right)=\frac{(c_k-1)(\frac{b}{2}-c_k-1)}{c_k+1}.$$

Cette quantité est positive puisque $c_k > 0$ et que l'on suppose $1 + c_k \leq \frac{b}{2}$. Il en résulte :

$$c_k\left(\frac{b}{c_k+1}-1\right) > \frac{b}{2}-1$$

et donc $\nu c_k > \frac{b}{2} - 1$ et donc $\nu c_k \geq \left[\frac{b}{2}\right]$.
Dans notre exemple on a $\nu = \left[\frac{10}{1+1}\right] = 5$, et $\nu y = 565$. On peut toujours se ramener lorsqu'on calcule un quotient $\left[\frac{x}{y}\right]$ au cas où le diviseur a un chiffre de plus fort poids compris dans $[[\frac{1}{2}b], b[$; il suffit pour cela de remplacer $\left[\frac{x}{y}\right]$ par $\left[\frac{\nu x}{\nu y}\right]$.
On suppose ce travail de normalisation déjà réalisé dans ce qui suit.
On a alors la proposition suivante qui nous permet de programmer à coup sûr, la recherche des chiffres successifs d'un quotient.

Proposition 2.4.2 *Soient $x = (x_{n+1}, \ldots, x_0)_b$ et $y = (y_n, \ldots, y_0)_b$. On suppose de plus que $y \leq x \leq by$ et que $y_n \in [[\frac{1}{2}b], b[$. Soit*

$$\tilde{q} = \min\left(b-1, \left[\frac{(x_{n+1}x_n)_b}{y_n}\right]\right).$$

Alors le quotient de x par y, $\left[\frac{x}{y}\right]$ vaut $\tilde{q}-2$, $\tilde{q}-1$, ou $\tilde{q}$.

Démonstration : On a d'une part $x = qy + r$ avec $0 \leq r < y$, d'où $qy \leq x$ puis :

$$qy_n b^n \leq qy \leq x < x_{n+1}b^{n+1} + x_n b^n + b^n,$$

et successivement

$$\begin{array}{rcl} qy_n & < & x_{n+1}b + x_n + 1, \\ qy_n & \leq & x_{n+1}b + x_n, \\ q & \leq & \frac{x_{n+1}b+x_n}{y_n}, \\ q & \leq & \tilde{q} \end{array}$$

On va montrer $(\tilde{q}-2)y \leq x$, ce qui prouvera $\tilde{q}-2 \leq q$.
On a

$$(\tilde{q}-2)y < (\tilde{q}-2)(y_n+1)b^n < \tilde{q}y_n b^n + (\tilde{q}-2y_n-2)b^n.$$

On a $\tilde{q}-2y_n-2<0$. En effet $\tilde{q}<b$ par définition et $2y_n+2 \geq b$ puisque $y_n \geq \left[\frac{b}{2}\right]$. D'où

$$(\tilde{q}-2)y < \tilde{q}y_n b^n.$$

Mais comme on a par construction, on a $y_n\tilde{q}b^n \leq (x_{n+1}b+x_n)b^n \leq x$; dès lors $(\tilde{q}-2)y \leq x$, et comme le quotient q est l'unique entier naturel vérifiant $qy \leq x < (q+1)y$, on a bien l'inégalité $\tilde{q}-2 \leq q$.

Une étude probabiliste fine montre que dans 67% des cas on a $q=\tilde{q}$ et que dans 32% $q=\tilde{q}-1$.

Dans l'exemple ci dessus, par normalisation on est amené à diviser 72990 par 565. on trouve $\tilde{q}_1 = \min(9, \left[\frac{7}{5}\right]) = 1$, $\tilde{q}_2 = \min(9, \left[\frac{16}{5}\right]) = 3$, $\tilde{q}_3 = \min(9, \left[\frac{51}{5}\right]) = 9$, un seul des chiffres, $\tilde{q}_2$ necessitera un essai.

Proposition 2.4.3 *Le calcul du quotient exact de deux entiers naturels n par $n' \neq 0$ a une complexité au pire de $O(\log(n)\log(n'))$ opérations binaires.*

Démonstration : Supposons $n > n'$ et la division normalisée (nécessite au plus une division deux multiplications par un nombre à un chiffre) et faisons le bilan des deux propositions précédentes : on a $\log(\frac{n}{n'})+1$ chiffres à trouver. Pour chacun deux, on calculera un chiffre $\tilde{q}$, obtenu comme un quotient élémentaire suivi d'une comparaison à $b-1$, puis on testera si : $0 \leq m-\tilde{q}n' < n'$ où m désigne l'un des dividendes partiels. Si c'est oui, $\tilde{q}$ est le chiffre cherché, sinon on teste $\tilde{q}-1$. Au pire chaque chiffre représente deux multiplications de n' par un nombre à un chiffre soit $O(\log(n'))$ opérations élémentaire, suivi deux soustractions à un nombre de même taille que n' ou ayant un chiffre de plus, soit encore $O(\log(n'))$ opérations élémentaires. C'est donc bien une complexité de l'ordre de $O(\log(n)\log(n'))$ que nous obtenons.

Chapitre 3

Arithmétique modulaire

Un autre codage possible pour le les nombres u de grandes tailles est de disposer d'une famille de nombres entiers $(m_1, m_2, \ldots, m_r)$ deux à deux premiers entre eux et d'effectuer les calculs sur les nombres u_i obtenus par divisions de u par les les nombres m_i, plutôt que directement sur u. Le nombre u étant de taille inférieure au produit $m_1 m_2 \cdots m_r$, il est alors possible de le reconstituer à partir de la connaissance des $(u_1, u_2, \ldots, u_r)$, ceci grâce au théorème des restes chinois.
On peut considérer la liste $(u_1, u_2, \ldots, u_r)$ comme une représentation interne de l'entier n.
Le grand problème de cette représentation réside dans la comparaison des entier. Comment décider si l'entier $(u_1, u_2, \ldots, u_r)$ est plus grand que l'entier $(v_1, v_2, \ldots, v_r)$, de même l'opération de division qui requière la connaissance de cœfficients de Bezout, d'où de fréquents recours, comme pour la repréntation des nombres rationnels à l'algorithme d'Euclide, mais plutôt sous forme étendue.

3.1 Théorème chinois

3.1.1 Cas de deux équations

Soit à résoudre un système :

$$\begin{cases} x & \equiv & a_1 & \mod & m_1 \\ x & \equiv & a_2 & \mod & m_2 \end{cases}$$

Proposition 3.1.1 *Supposons que m_1 et m_2 soient premiers entre eux, et soient u et v deux entiers tels que $um_1 + vm_2 = 1$. Alors $x = a_2um_1 + a_1vm_2$ est une solution du système. De plus y en est une autre solution si et seulement si $x \equiv y \bmod m_1m_2$.*

Démonstration : On a :

$$\begin{array}{rcll} x & \equiv & a_1vm_2 & \bmod \ m_1 \\ & \equiv & a_1(1-um_1) & \bmod \ m_1 \\ & \equiv & a_1 & \bmod \ m_1. \end{array}$$

De même on montre $x \equiv a_2 \bmod \ m_2$.
Soient y une autre solution du système, on a :

$$\left\{ \begin{array}{rcll} y & \equiv & x & \bmod \ m_1 \\ y & \equiv & x & \bmod \ m_2, \end{array} \right.$$

ce qui de façon équivalente s'ecrit :

$$m_1 \text{ divise } y-x, \text{ et } m_2 \text{ divise } y-x,$$

m_1 et m_2 étant premiers entre eux, ce qui équivaut à :

$$\text{le produit } m_1m_2 \text{ divise } y-x.$$

Une conséquence de la précédente proposition est l'important théorème qui suit

Théorème 3.1.1 *:* (Théorème des restes chinois)
Soient m_1 et m_2 deux entiers premiers entre eux. Alors l'application φ définie par :

$$\begin{array}{rccc} \varphi : & \frac{\mathbb{Z}}{m_1m_2\mathbb{Z}} & \rightarrow & \frac{\mathbb{Z}}{m_1\mathbb{Z}} \times \frac{\mathbb{Z}}{m_2\mathbb{Z}} \\ & x & \mapsto & (x \bmod m_1, x \bmod m_2), \end{array}$$

est un isomorphisme d'anneau. On a donc :

$$\frac{\mathbb{Z}}{m_1m_2\mathbb{Z}} \approx \frac{\mathbb{Z}}{m_1\mathbb{Z}} \times \frac{\mathbb{Z}}{m_2\mathbb{Z}}.$$

3.1.2 Cas général

On considère un système :

$$\mathcal{S}_n \left\{ \begin{array}{rcll} x & \equiv & a_1 & \bmod \ m_1, \\ x & \equiv & a_2 & \bmod \ m_2, \\ \vdots & & & \\ x & \equiv & a_n & \bmod \ m_n. \end{array} \right.$$

On suppose les entiers m_i premiers entre eux deux à deux. On procède par récurrence. Supposons que x_1 soit une solution particulière du sous-système :

$$\mathcal{S}_1 \begin{cases} x & \equiv & a_1 & \mod & m_1, \\ x & \equiv & a_2 & \mod & m_2. \end{cases}$$

Alors x est solution de $\mathcal{S}_n$ si et seulement s'il vérifie :

$$\mathcal{S}_{n-1} \begin{cases} x & \equiv & x_1 & \mod & m_1 m_2, \\ x & \equiv & a_2 & \mod & m_3, \\ \vdots & & & & \\ x & \equiv & a_n & \mod & m_n. \end{cases}$$

On peut ainsi énoncer une proposition qui généralise ce résultat :

Proposition 3.1.2 *Sous l'hypothèse que les m_i soient deux à deux premiers entre eux. Le système $(\mathcal{S}_n)$ possède des solutions.*
Si x est un solution, y en est une autre si et seulement si $x \equiv y \bmod m_1 m_2 \cdots m_n$.

Ce qui permet d'énoncer une version généralisée du théorème chinois

Théorème 3.1.2 *Soient $m_1, \ldots, m_n$ n entiers premiers entre eux deux à deux, alors on a un isomorphisme d'anneaux*

$$\frac{\mathbb{Z}}{m_1 m_2 \cdots, m_n \mathbb{Z}} \approx \frac{\mathbb{Z}}{m_1 \mathbb{Z}} \times \cdots \times \frac{\mathbb{Z}}{m_n \mathbb{Z}}.$$

L'application

$$\begin{array}{ccccccccc} \frac{\mathbb{Z}}{m_1 m_2 \cdots m_n \mathbb{Z}} & \to & \frac{\mathbb{Z}}{m_1 \mathbb{Z}} & \times & \frac{\mathbb{Z}}{m_2 \mathbb{Z}} & \times & \cdots & \times & \frac{\mathbb{Z}}{m_n \mathbb{Z}} \\ x \pmod{m_1 m_2 \cdots m_n} & \mapsto & (x \pmod{m_1} & , & x \pmod{m_2} & , & \ldots & , & x \pmod{m_n}), \end{array}$$

étant un isomorphisme d'anneau.

3.1.3 La relation de Bezout

La mise en œuvre du théorème (des restes) chinois, nécessite un procédé commode de calcul des coefficients u et v tels que $au + bv = pgcd(a, b)$.

Algorithme d'Euclide

Proposition 3.1.3 *Soit A un anneau euclidien (dans la pratique $A = \mathbb{Z}$ ou $A = K[X]$, K désignant un corps quelconque). Pour tout $(a, b) \in A^2$, il existe u et v dans A tels que $au + bv = pgcd(a, b)$.*

Démonstration : Une démonstration traditionelle, consiste à effectuer la division euclidienne de a par b : $a = bq + r$ avec $r < b$, de voir que nécessairement $pgcd(a, b) = pgcd(b, r)$. On construit une suite :

$$\begin{array}{l} r_0 = a, \\ r_1 = b \text{ ; et pour } i \geq 2 : \\ r_{i-1} = q_i r_i + r_{i+1}, \quad r_{i+1} < r_i, \end{array}$$

jusqu'à l'obtention d'un reste $r_{n+1} = 0$, le dernier reste r_n non nul est le $pgcd(a, b)$ cherché. En remontant l'algorithme, on construit deux constantes u et v,premières entre elles, tels que $au + bv = r_n$, soit (u', v') un autre couple vérifiant les mêmes propriétés, on a alors $a(u - u') = b(v' - v)$, ce qui implique :

$$\exists k \in \mathbb{Z},\ u' = u - kb/d, v' = v + ka/d.$$

Attacherons au calcul direct de ces cœficients u et v :

Algorithme d'Euclide étendu

On se place dans l'anneau-produit A^3 sur lequel on définit une suite de triplets par :

$$W_0 = (a, 1, 0),\ W_1 = (b, 0, 1), \text{ et } W_i = (r_i, u_i, v_i)$$

où pour $i \geq 2$, r_i est le reste de la division euclidienne de r_{i-2} par r_{i-1}, u_i et v_i étant défini par :

$$u_i = u_{i-2} - q_{i-1}u_{i-1},\ v_i = v_{i-2} - q_{i-1}v_{i-1},$$

on peut dès lors écrire :

$$W_i = (r_i, u_{i-2} - q_{i-1}u_{i-1}, v_{i-2} - q_{i-1}v_{i-1}) = W_{i-2} - q_{i-1}W_{i-1} \text{ pour } i \geq 2.$$

Montrons que pour tout $0 \leq i \leq n$, on a :

$$r_i = au_i + bv_i.$$

On a pour $i = 0$, $a = 1a + 0b$ et pour $n = 1$, $b = 0a + 1b$,
supposons que c'est vrai jusqu'à i ; au rang suivant on a :

$$\begin{array}{rcl} au_{i+1} + bv_{i+1} & = & a(u_{i-1} - q_i + u_i) + b(v_{i-1} - q_i v_i) \\ & = & au_{i-1} + bv_{i-1} - q_i(au_i) - bv_i) \\ & = & r_{i-1} - q_i r_i \\ & = & r_{i+1} \end{array}$$

Une procédure Maple de calcul des cœfficients de Bezout peut s'écrire sous la forme suivante :

```
> eucligen :=proc(a,b)
  local q,r,wp,wn,w ;
 wp := (a, 1, 0) :
  wn := (b, 0, 1) :
  r := irem(a, b,′ q′) :
  while r<>0 do
  w := (r, wp[2] − q ∗ wn[2], wp[3] − q ∗ wn[3]) :
 wp := wn : wn := w :
  r := irem(wp[1], wn[1],′ q′) :
  od :
  w ;
  end :
> # Application sur quelques exemples :
>eucligen(12,16) ;
                                    4, -1, 1
> eucligen(99099,43928) ;
                                 1, 547, -1234
> eucligen(1891,2499) ;
                                 1, -1007, 762
 > eucligen(1769,551) ;
                                   29, 5, -16
```

Proposition 3.1.4 *Dans le cas d'entiers positifs a et b vérifiant $a \neq b$ L'algorithme ci-dessus retourne un couple (u, v) de cœfficients de Bezout vérifiant :*

$$au + bv = d = pgcd(a, b) \text{ avec } |u| \leq \frac{b}{2d}, \quad \text{et } |v| \leq \frac{a}{2d}$$

Démonstration : :
On va supposer l'inégalité s $a > b$, car sinon l'algorithme fournit succesivement :

$$W_0 = (a, 1, 0), \quad W_1 = (b, 0, 1), \quad W_2 = (a, 1, 0),$$

le premier passage ne fait que renverser l'ordre des termes.

Dans le cas $a \geq 2,\ b = 1$, l'algorithme fournit $d = 1,\ u = 0,\ v = 1$ et les inégalités $|u| \leq \frac{b}{2d}$ et $|v| \leq \frac{a}{2d}$ sont évidemment vérifiées.

Dans le cas $a > 2, b = 2$, l'algorithme fournit la solution $d = 2,\ u = 0,\ v = 1$ si a est pair : $a = 2k,\ k \geq 2$ et les inégalités proposées sont bien vérifiées.

On va donc étudier le cas : $a > b > 2$
Les suite des quotients q_i et des restes r_i sont nécessairement positives :

$$r_i = q_{i+1}r_{i+1} + r_{i+2},\ 0 \leq r_{i+2} < r_{i+1} \text{ et } q_i \text{ vérifie } q_i \geq 1.$$

On a $u_0 = 1$ et $u_1 = 0$ et supposons qu'à l'ordre i on ait $u_{2i} \geq 0$ et $u_{2i+1} \geq 0$ à l'ordre suivant on aura $u_{2i+2} = u_{2i} - q_{2i+1}u_{2i+1} \geq 0$ et $u_{2i+3} = u_{2i+1} - q_{2i+2}u_{2i+2} \leq 0$, la suite des u_i est ainsi alternée et on a $u_2 i = u_{2i-2} - q_{2i-1}u_{2i-1} \geq -q_{2i-1}u_{2i-1} = [q_{2i-1}u_{2i-1}|$, comme $q_{2i-1} \geq 1$ on a $u_{2i} \geq |u_{2i-1}|$, de même $u_{2i+1} = u_{2i-1} - q_{2i}u_{2i}$ implique $|u_{2i+1}| \geq |q_{2i}u_{2i}| \geq |u_{2i}|$, la suite $|u_i|$ est croissante.

De plus le dernier quotient q_n vérifie nécessairement $q_n \geq 2$, en effet on a $r_{n-1} = q_n r_n + r_{n+1} = q_n r_n$, la condition $r_{n-1} > r_n$ entrainne dés lors $q_n \geq 2$.
On a à l'ordre n et $n+1$, $au_n + bv_n = d$ et $au_{n+1} + bv_{n+1} = r_{n+1} = 0$.

Posons $\frac{a}{d} = a'$, $\frac{b}{d} = b'$, il s'agit dès lors de montrer : $\quad d \leq \frac{a'}{2}$, et $d \leq \frac{b'}{2}$.
On aboutit à un système :

$$\left\{ \begin{array}{rcrcrcl} u_n & a' & + & v_n & b' & = & 1 \\ u_{n+1} & a' & + & v_{n+1} & b' & = & 0 \end{array} \right. ,$$

ce qui entrainne :

$$(u_{n+1}v_n - u_n v_{n+1})b' = u_{n+1}\ ,$$

et qui s'écrit encore :

$$-\left| \begin{array}{cc} u_n & v_n \\ u_{n+1} & v_{n+1} \end{array} \right| b' = u_{n+1}\ .$$

Mais nous avons pour tout $i = 1, \ldots, n$, :

$$\left(\begin{array}{cc} u_i & v_i \\ u_{i+1} & v_{i+1} \end{array} \right) = \left(\begin{array}{cc} 0 & 1 \\ 1 & -q_i \end{array} \right) \left(\begin{array}{cc} u_{i-1} & v_{i-1} \\ u_i & v_i \end{array} \right).$$

De sorte que :

$$\left| \begin{array}{cc} u_n & v_n \\ u_{n+1} & v_{n+1} \end{array} \right| = -\left| \begin{array}{cc} u_{n-1} & v_{n-1} \\ u_n & v_n \end{array} \right| = \cdots = (-1)^n \left| \begin{array}{cc} u_0 & v_0 \\ u_1 & v_1 \end{array} \right| = (-1)^n \left| \begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right|.$$

D'où :
$b' = |u_{n+1}| = |q_n u_n|$, ce qui vu l'inégalité $q_n \geq 2$ implique bien : $|u_n| \leq \frac{b'}{2}$.

3.1.4 Complexité de l'algorithme d'Euclide

Proposition 3.1.5 *Lors du calcul du pgcd de deux nombres positifs a et b, $a > b$, des restes successifs $r_1 = b$, $r_2, \ldots, r_n$ tels que $r_n = pgcd(a, b)$, on a :*

$$n \leq 2[\log_2(a)] + 1.$$

Par suite le nombre de divisions à réaliser pour calculer r_n est majoré par $2[\log_2(a)]$.

Démonstration : On montre d'abord que pour tout $i = 1, \ldots, n$, on a $r_{i+2} < \frac{1}{2} r_i$.
En effet si on a $r_{i+1} \leq \frac{1}{2} r_i$, comme la suite des restes est strictement décroissante, on a bien :

$$r_{i+2} < r_{i+1} \leq \tfrac{1}{2} r_i \,;$$

mais si $\frac{1}{2} r_i < r_{i+1} < r_i$, le quotient de la division de r_i par r_{i+1} est alors égal à 1 et $r_{i+2} = r_i - r_{i+1} < \frac{1}{2} r_i$.
Dans tous les cas on a bien :

$$r_{i+2} < \tfrac{1}{2} r_i.$$

D'où si n pair $n = 2k$, $r_n < \frac{1}{2^2} a$; et si n impair $n = 2k + 1$, $r_n < \frac{1}{2^k} b < \frac{1}{2^k} a$.
Dans tous les cas on a $k < \log_2(a)$ et donc $n \leq 2[\log_2(a)] + 1$.

3.2 Numérotation mixte

On propose un autre système de représentation des entiers mieux adaptés à la résolution des systèmes (S_n).

Théorème 3.2.1 *Soient n entiers $m_1, m_2 \ldots, m_n$ distincts ou non, et ψ la fonction définie par :*

$$\begin{array}{rccc} \psi : & [0, m_1[\times \cdots [0, m_n[& \to & [0, m_1 m_2 \cdots m_n[\\ & (a_1, a_2, \ldots, a_n) & \mapsto & a_1 + a_2 m_1 + a_3 m_1 m_2 + \cdots + a_n m_1 \cdots m_{n-1}. \end{array}$$

L'application psi est bijective.

Démonstration :
Vérifions par récurrence sur n que ψ est bien défini sur $[0, m_1 m_2 \cdots m_n[$:
On a bien a_1 dans $[0, m_1[$, supposons qu'à l'ordre k on ait
$a_1 + a_2 m_2 + \cdots + a_k m_1 m_2 \cdots m_{k-1}$ dans l'intervalle $[0, m_1 \cdots m_k[$,

si $a_i \in [0, m_i[$ pour $i = 1, \ldots, k$. On a alors :

$$\begin{aligned} a_1 + a_2 m_2 + \cdots + a_{k+1} m_1 m_2 \cdots m_k &< m_1 \cdots m_k + a_{k+1} m_1 m_2 \cdots m_k \\ &< m_1 \cdots m_k (1 + a_{k+1}) \\ &< m_1 \cdots m_k m_{k+1} \end{aligned}$$

Montrons que ψ est surjective :
Soit a un élément de $[0, m_1 \cdots m_n[$, on réalise la suite des n divisions euclidiennes suivantes

$$\begin{aligned} a &= q_1 m_1 + a_1, \\ q_1 &= q_2 m_2 + a_2, \\ q_2 &= q_3 m_3 + a_3, \\ &\vdots \\ q_{n-1} &= q_n m_n + a_n. \end{aligned}$$

On a :
$a = a_1 + m_1(a_2 + m_2(a_3 + m_4(a_4 + \cdots + m_{n-1}(a_n + q_n m_n) \cdots))) = a_1 + a_2 m_1 + a_3 m_1 m_2 + a_4 m_1 m_2 m_3 + \cdots + a_n m_1 m_2 m_3 \cdots m_{n-1} + q_n m_1 m_2 m_3 \cdots m_{n-1} m_n$.
En particulier l'appartenance $a \in [0, m_1 \cdots m_n[$ implique la relation $q_n = 0$.
Il est clair que chaque a_i appartient à $[0, m_i[$ et il est immédiat de vérifier :

$$\psi(a_1, \ldots, a_n) = A.$$

Comme les cardinaux des ensembles d'arrivée et de départ de l'application ψ sont égaux, celle ci étant surjective est bijective.

L'écriture d'un nombre A de l'intervalle $[0, m_1 \cdots m_n[$ sous la forme $A = a_1 + \cdots + a_n m_1 \cdots m_{n-1}$ s'appelle écriture en base mixte de A.

Ainsi dans le cas où les nombres m_i sont premiers entre eux deux à deux on dispose de deux systèmes de numération des entiers de l'intervalle $[0, m_1 \cdots m_n[$: celle donnée par le théorème chinois, et celle donnée par l'écriture en base mixte.
Ces deux représentation sont distinctes et ont des propriétés différentes.

Un exemple :
$n = 3, m_1 = 2, m_2 = 5, m_3 = 7$ considérons $a = 51 \in [0, 70[$, on a $\phi(a) = (1, 1, 2)$ alors que $\psi^{-1}(a) = (1, 0, 5)$.

L'avantage de ce système de numération : Comparer entre eux deux entiers

écrits en base mixte ne pose aucun problème. Il suffit de comparer les chiffres de plus fort poids.

3.3 Algorithme de Garner

Le procédé trouvé dans les paragraphes précédents pour résoudre le système $(\mathcal{S}_n)$ à l'aide d'un algorithme d'Euclide étendu $x = a_1 v m_2 + a_2 u m_1$ ne donne pas toujours la meilleure solution (celle plus petite en valeur absolue).
Or on peut aboutir à des débordements de mémoire en obtenant des solutions plus grandes en valeur absolue, que $m_1 m_2$.

Un exemple :

$$\begin{cases} x & \equiv & 1000 & \mod & 1891 \\ x & \equiv & 3000 & \mod & 2499 \end{cases}$$

On obtient un couple de coefficients de Bezout (u, v) pour les deux moduli :

$$1891u + 2499v = 1,$$

à l'aide de l'algorithme d'Euclide étendu : $u = -1007$ et $v = 762$, ce qui fournit comme solution particulière $x = 2499000v + 5673000u = -3808473000$.
Cependant ce résultat comporte 10 chiffres, alors que $1891.2499 = 4725609$ n'en comporte que 7.
La solution proposée n'est pas optimale.
On sait qu'il existe une solution dans $[0, 4725609[$, qu'on sait facilement retrouver mais à partir de cette solution particulière en raisonnant modulo 4725609, et de fait celle ci vaut 367854.

Revenons au système $\mathcal{S}_n$: on cherche une solution x dans l'intervalle $[0, m_1 \cdots m_n[$, pour cela on code x dans la base mixte.
On sait qu'il existe $\nu_i \in [0, m_i[$ tels que :
centerline$x = \nu_1 + \nu_2 m_1 + \nu_3 m_1 m_2 + \nu_n m_1 \cdots m_{n-1}$.
On peut alors trouver les nombres ν_i les uns aprés les les autres. On a $\nu_1 \equiv a_1 \mod m_1$.
Supposons que l'on connaisse ν_i pour $i = 1, 2, \ldots, k-1$. On a puisque $x \equiv a_k \mod m_k$:

$$a_k \equiv \nu_1 + \nu_2 m_2 + \cdots + \nu_k m_1 \cdots m_{k-1} \mod m_k.$$

Mais ($m_1 \cdots m_{k-1}$ est premier avec m_k et donc inversible modulo m_k. On obtient alors

$$\nu_k = (m_1 \cdots m_{k-1})^{-1}(a_k - \nu_1 - \nu_2 m_1 - \ldots - \nu_{k-1} m_1 \cdots m_{k-2} \bmod m_k$$

Ceci donne lieu à une procédure qui suit :
On place en paramètre la liste M des moduli m_i et la liste A des valeurs a_i correspondants, ces deux listes ayant même longueur, et les moduli premiers entre eux deux à deux.

```
> garner :=proc(M,A)
  local Z,z,p,x,i,k ;
  Z := [A[1]] :
  x := 0;  p := 1 ;
 k := nops(M) ;
  for i from 2 to k do
  x := x + Z[i − 1] ∗ p ;
  p := p ∗ M[i − 1] ;
  z := p ∧ (−1) ∗ (A[i] − x) mod M[i] ;
  Z := [op(Z), z] ;
  od :
  Z ;
  end :
>## Application sur quelques exemples :
>garner([1891, 2499], [1000, 3000]) ;
                                  [1000, 194]
> garner([2,5,7,9],[1,3,5,7]) ;
                                    [1, 1, 3, 4]
 > garner([150, 49, 143], [8, 30, 100]) ;
                                    [8, 40, 82]
>## Vérification du dernier résultat :
 > 8 + 40 ⋆ 150 + 82 ⋆ 150 ⋆ 49 ;
                                      608708
 > irem(608708,150),irem(608708,49),irem(608708,143) ;
                                    8,30,100
```

Résolutions du système

$$\begin{cases} x & \equiv & 1000 \bmod 1891 \\ x & \equiv & 3000 \bmod 2499 \end{cases}$$

On cherche x sous la forme $x = \nu_1 + \nu_2 1891$ avec $0 \leq \nu_1 < 1891$ et $0 \leq \nu_2 < 2499$. On a $\nu_1 = 1000$, puis $\nu_2 = (1891)^{-1}(3000 - 1000) \equiv 194 \bmod 2499$, d'où $x = 367845$.
Un problème subsiste, toutefois, celui du signe du résultat obtenu, on le contourne comme suit :
Si on sait que $-M \leq x \leq M$, alors on choisit les m_i de sorte que leur produit soit strictement supérieur à $2M$ de sorte que dans l'intervalle $[-M, M]$, une seule solution mod $2M$ ne subsiste qui représentera bien x. Généralement on choisit pour moduli des nombres premiers p_i distincts, les anneaux $\mathbb{Z}/p_i\mathbb{Z}$ sont alors des corps, dans lesquels il n'y a pas de calcul de fractions.
Précisons une méthode générale avec ses diverses étapes : Soient $a_1, a_2, \ldots, a_n$ des entiers et $f(a_1, \ldots, a_n)$ une expression à calculer dans $\mathbb{Z}$ et de type rationnel. On a alors pour tout nombre premier p $f(a_1, \ldots, a_n) = f(a_1 \pmod p), \ldots, a_n \pmod p) \pmod p$. On suppose que l'on sait $|f(a_1, \ldots, a_n)| \leq M$, alors on accomplit les étapes suivantes :

1. Choisir k nombres premiers p_i distincts tels que $\prod_{i=1}^{k} p_i > 2M$. Il se peut suivant le problème traités certainnes valeurs de p_i soient à rejeter.
2. Calculer pour chaque $i = 1, \ldots, n \quad F_i \equiv f(a_1, \ldots, a_n) \bmod p_i$.
3. Résoudre par l'algorithme de Garner le système congruent :

$$\begin{cases} F \equiv F_1 \bmod p_1 \\ F \equiv F_2 \bmod p_2 \\ \\ F \equiv F_k \bmod p_k \end{cases}$$

$f(a_1, \ldots, a_n)$ est l'unique nombre vérifiant cette congruence dans l'intervalle $[-M, M]$ que l'on trouve au besoin par une soustraction supplémentaire s'il se trouve que F appartient à l'intervalle $]M, 2M]$.

exercice
calculer le déterminant

$$D = \begin{vmatrix} 6 & -4 & 5 & 1 \\ -5 & 0 & 5 & 2 \\ 3 & 1 & 6 & -5 \\ 0 & 1 & 2 & 2 \end{vmatrix}$$

sachant $|D| \leq 4!2.5.6^2 = 8640$. On montre que les nombres premiers $11, 13, 17, 19$ conviennent : $11 \times 13 \times 17 \times 19 = 46189 > 2 \times 8640 = 17280$.

On obtient :

$$\begin{cases} D \equiv 7 \mod 11 \\ D \equiv 12 \mod 13 \\ D \equiv 2 \mod 17 \\ D \equiv 0 \mod 19 \end{cases}$$

La résolution de ce système donne $D \equiv 45239 \mod 46189$. Le nombre cherché est alors $D = 42239 - 46189 = -950$.

Chapitre 4

Transformation de Fourier discrète

4.1 Racines principales (primitives) de l'unité

Soient A un anneau non nécessairement intègre, par exemple $\mathbb{Z}/N\mathbb{Z}$, n un entier > 0, et ω une racine n-ième de l'unité dans A, c'est à dire une racine du polynôme $P = X^n - 1$ de $A[X]$. L'ensemble des racines de P est un groupe multiplicatif.

Les conditions suivantes sont équivalentes :
(i) Pour tout entier i, tel que $0 < i < n$, l'élément $1 - \omega^i$ n'est pas diviseur de zéro dans A, c'est à dire que : $a \in A$ et $(1 - \omega^i)a = 0$ implique $a = 0$.
(ii) Pour tout couple d'entiers i et j non congrus modulo n, l'élément $\omega^i - \omega^j$ n'est pas diviseur de zéro dans A.

En effet ω est inversible et on a $\omega^i - \omega^j = \omega^i(1 - \omega^{j-i})$.
On dira qu'un tel ω est une *racine principale n-ième de l'unité*.

En particulier, le polynôme $X^n - 1$ possède n racines distinctes, l'ensemble de ces racines possède une structure de sous-groupe cyclique d'ordre n du groupe multiplicatif $\mathcal{U}(A)$, de générateur ω.

Exemples :
Dans $\mathbb{C}$, le nombre $e^{i\frac{2\pi}{n}}$ est une racine principale n-ième de l'unité.

Proposition 4.1.1 *Soit ω une racine principale n-ième de l'unité dans A.*

1. *On a dans $A[X]$ la relation :*
$$X^n - 1 = \prod_{i=0}^{n-1}(X - \omega^i).$$

2. L'élément $n.1_A$ n'est pas un diviseur de zéro dans A et on a :
$$n.1_A = \prod_{i=1}^{n-1}(1-\omega^i)$$

Démonstration :
i) Soit $P = X^n - 1$, on a $P(1) = 0$ donc $P = (X-1)Q$ où Q est un polynôme unitaire.
On a $P(\omega) = 0$ donc $(\omega - 1)Q(\omega) = 0$, mais comme $\omega - 1$ n'est pas diviseur de 0 dans A, on a :
$Q(\omega) = 0$ et Q s'écrit $Q = (X - \omega)R$, avec R polynôme unitaire.
Ainsi P s'écrit :
$P = (X-1)(X-\omega)R$ et $P(\omega^2) = (\omega - 1)(\omega^2 - 1)R(\omega^2) = 0$, etc.
On trouve ainsi : $P = \prod_{i=0}^{n-1}(X - \omega^i)$.

ii) Le polynôme P est divisible par $X - 1$, et Q s'écrit :
$$Q = \frac{X^n-1}{X-1} = X^{n-1} + X^{n-2} + \cdots + X + 1 = \prod_{i=1}^{n-1}(X - \omega^i),$$
en particulier :
$$Q(1) = n1_A = \prod_{i=1}^{n-1}(1 - \omega^i) \neq 0.$$

L'évaluation d'un polynôme de $A[X]$ en ces n racines n-ièmes de l'unité se fait beaucoup plus vite qu'en n points quelconques de A. De plus l'opération inverse qui consiste à retrouver un polynôme dont on connait les valeurs en les n racines n-ièmes de l'unité, problème d'interpolation, se fait aussi rapidement que l'évaluation en ces points.
Pour multiplier deux polynômes de degré convenable, il suffira alors d évaluer chacun d'eux en les n racines de l'unité, de multiplier ensuite les valeurs obtenues en ces points, puis de reconstruire le polynôme produit par interpolation.

4.2 Évaluation d'un polynôme en les racines n-ièmes de l'unité

Soit dans A, ω une racine n-ième de l'unité où $n = 2^k$.
Considérons un polynôme $P = \sum_{i=0}^{n-1} a_i X^i$.
On veut calculer P en les n éléments $1, \omega, \omega^2, \ldots, \omega^{n-1}$.
À priori chacune de ces opération nécessite $O(n)$ opérations dans A par utilisation de Hörner. On peut donc espérer ces n évaluations en $O(n^2)$ opérations

arithmétiques. Cependant les évaluations peuvent s'effectuer simultanément en beaucoup moins de n^2 opérations.

Théorème 4.2.1 *Soient A un anneau commutatif, ω une racine n-ième de l'unité avec $n = 2^k$, et P un polynôme de degré strictement plus petit que n.*

Alors l'évaluation de $P(\omega^i)$ pour $i = 0, 1, \ldots, n-1$ nécessite $O(n\log(n))$ opérations dans A.

Démonstration : Notons O_n le nombre maximum d'opérations arithmétiques nécessaires pour le calcul des $P(\omega^i),\ i = 0, 1, \ldots, n-1$, lorsque $deg(P) < n$. On peut écrire P sous la forme :

$$P(X) = P_0(X^2) + XP_1(X^2),$$

où P_0 et P_1 sont des polynômes de $A[X]$ de degrés strictement inférieurs à $\frac{n}{2}$. On est donc amené à calculer les deux polynômes P_0 et P_1 en les $\frac{n}{2}$ points $\omega^{2i},\ i = 0, 1, \ldots, \frac{n}{2} - 1$. Or ω^2 est une racine $\frac{n}{2}$-ième de l'unité. On obtient ainsi la relation de récurrence :

$$O_n = 2O_{\frac{n}{2}} + 2n \quad .$$

Le terme $2n$ provenant des n multiplications et des n additions. De plus $O_1 = 0$, $O_2 = 4$, puis $O_{2^k} = 2^{k+1}k = 2n\log_2(n)$.

Exemples : $A = \frac{\mathbb{Z}}{5\mathbb{Z}}$, $\omega = 2$ est alors une racine quatrième principale de l'unité, les puissances successives étant 4, 3, 1.
Soit $P := 1 + 3X + 4X^2 + X^3$, la méthode de Hörner va nécessiter 8 multiplications à cause du 1, et 12 additions, au lieu de 12 multiplications et 12 additions.
Cependant on peut décomposer P sous la forme :
$P = (1 + 4X^2) + X(3 + X^2) = P_0(X^2) + XP_1(X^2), P_0 = 1 + 4X, P_1 = 3 + X.$
Et on a en 1 et 4, les racines 2-ième de l'unité :

$$P_0(1) = 0, P_0(4) = 2, P_1(1) = 4, P_1(4) = 2.$$

Et alors :

$$\begin{array}{lll} P(1) & = P_0(1) + P_1(1) & = 4, \\ P(2) & = P_0(4) + 2P_1(4) & = 1, \\ P(3) & = P_0(4) + 3P_1(4) & = 3, \\ P(4) & = P_0(1) + 4P_1(1) & = 1, \end{array}$$

en 8 multiplications et 8 addition au total.

On peut encore abaisser le nombre de multiplication en utilisant l'identité :

$$\omega^n - 1 = 0 = (\omega^{\frac{n}{2}} - 1)(\omega^{\frac{n}{2}} + 1),$$

on voit que si ω est une racine principale n-ième de l'unité on a, dès lors, nécessairement : $\omega^{\frac{n}{2}} = -1$, et donc $\omega^{i+\frac{n}{2}} = -\omega^i$.

4.3 Le problème réciproque

Théorème 4.3.1 *Soit ω une racine principale n-ième de l'unité dans un anneau A, telle que $n.1_A$ soit inversible dans A, d'inverse noté $\frac{1}{n}$.*
Soit un n-uplet $(\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ appartenant à A^n et soit le polynôme :

$$\tilde{P} = \sum_{i=0}^{n-1} \alpha_i X^i.$$

Alors il existe dans $A[X]$ un unique polynôme P

$$P = \sum_{i=0}^{n-1} a_i X^i,$$

dont la valeur en ω^i est α_i pour tout $i = 0, 1, \ldots, n-1$.
De plus pour chaque i, on a :

$$a_i = \frac{\tilde{P}(\omega^{-i})}{n}.$$

Démonstration : On est amené à chercher les n cœfficients :

$$a_i \ i = 0, 1, \ldots, n-1$$

d'un polynôme $P = \sum_{i=0}^{n-1} a_i X^i$ tels que :

$$\left\{\begin{array}{lllllll} a_0 & +a_1 & +a_2 & +\cdots & +a_{n-1} & = & \alpha_0 \\ a_0 & +a_1\omega & + + a_2\omega^2 & +\cdots & +a_{n-1}\omega^{n-1} & = & \alpha_1 \\ a_0 & +a_1\omega^2 & + + a_2\omega^4 & +\cdots & +a_{n-1}\omega^{2(n-1)} & = & \alpha_2 \\ & & & & & \vdots & \\ a_0 & +a_1\omega^{n-1} & + + a_2\omega^{2(n-1)} & +\cdots & +a_{n-1}\omega^{(n-1)^2} & = & \alpha_{n-1}, \end{array}\right.$$

et donc à résoudre un système linéaire de n équations à n inconnues.
On considére la matrice de ce système linéaire, elle est de Vandermonde et de la forme :

$$V_\omega = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & & & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \ldots & \omega^{(n-1)^2} \end{pmatrix} = (\omega^{ij}) \begin{array}{l} 0 \le i \le n-1 \\ 0 \le j \le n-1 \end{array}.$$

Soit c_{ij} le coefficient d'ordre (i, j) du produit $V_\omega.V_{\omega^{-1}}$. On a :

$$c_{ij} = \sum_{l=0}^{n-1} \omega^{il}\omega^{-lj} = \sum_{l=0}^{n-1} \omega^{(i-j)l}.$$

De l'identité :

$$1 - X^n = (1 - X)(1 + X + X^2 + \cdots + X^{n-1}),$$

on peut déduire :

$$c_{ij}(1 - \omega^{i-j}) = (1 - (\omega^{(i-j)})^n) = 0.$$

Cependant ω étant une racine principale n-ieme de l'unité, $1 - \omega^{i-j}$ ne divise pas 0 pour $i \neq j$ dans l'intervalle considéré. On a donc :

$$V_\omega . V_{\omega^{-1}} = n.Id$$

En particulier si $n.1_A$ est inversible dans A, alors :
V_ω est inversible et $(V_\omega)^{-1} = \frac{1}{n} V_{\omega^{-1}}$. Ce qui est la traduction matricielle de l'énoncé.

Si l'anneau A est un corps, alors la condition $n.1_A$ inversible est satisfaite dans le cas où ω est une racine principale n-ième de l'unité.
Exemples : Dans l'exemple traité plus haut $P = 1 + 3X + 4X^2 + X^3$, on a trouvé $P(1) = 4,\ P(2) = 1,\ P(4) = 1,\ P(3) = 3$, le polynôme réciproque $\tilde{P} = 4 + X + X^2 + 3X^3$ que l'on évalue en $1, 3, 4, 2$ qui sont les puissances successives de 2^{-1} mod (5). On obtient $\tilde{P}(1) = 4,\ \tilde{P}(3) = 2,\ \tilde{P}(4) = 1,\ \tilde{P}(2) = 4$, et en multipliant par $4^{-1} = 4$ mod (5), on retrouve bien les coefficients de P.

4.4 Transformée de Fourier discrète (DFT)

Soit ω une racine primitive n-ième de l'unité dans un anneau A, n étant inversible dans A, c'est à dire premier avec l'ordre de A, si A est fini.
On considère l'homorphisme d'anneau

$$\begin{array}{rccc} \mathcal{F}_\omega : & A[X] & \to & A^n \\ & P & \mapsto & (P(\omega^0), P(\omega), P(\omega^2), \ldots, P(\omega^{n-1})), \end{array}$$

de noyau l'idéal de $A[X]$ engendré par $X^n - 1$.

Théorème 4.4.1 *Soit A un anneau muni d'une racine n-ième principale de l'unité. Soient deux polynômes P et Q de $A[X]$ tels que $deg(P) + deg(Q) < n$. Alors en utilisant la DFT leur multiplication se réalise en $O(n \log(n))$ opérations dans A.*

Démonstration : On calcule les DTF associées à ω de P et Q, soit $O(n\log(n))$ opérations dans A. On obtient ainsi les n-uplets :

$$(P(1), P(\omega), \ldots, P(\omega^{n-1})), \text{ et } (Q(1), Q(\omega), \ldots, Q(\omega^{n-1})).$$

On calcule alors dans A^n les produits :

$$(P(1)Q(1), P(\omega)Q(\omega), \ldots, P(\omega^{n-1})Q(\omega^{n-1}))$$

en n opérations arithmétiques dans A, puis on utilise le théorème précédent pour retrouver l'antécédent PQ de ce n-uplet, ce qui nécessite un calcul de DTF associé au point ω^{-1}, soit encore $O(n\log(n))$ opérations dans A.

Si K est un corps fini d'ordre q impair, son groupe multiplicatif K^* est d'ordre $q-1 = 2^k m$ (m impair), et est cyclique. Connaissant un élément primitif de ce corps, on en déduira facilement un élément d'ordre 2^k.
Cependant on aura besoin de trouver un corps de cardinal $q = 2^k m + 1$ en partant de k fixé à priori. Un théorème dû à Dirichlet nous permet d'affirmer l'existence d'une infinité de nombres premiers de la forme $2^k m + 1$.

Dans un corps K dire que ω est une racine 2^k-ième principale de l'unité revient à dire que $\omega^{2^{k-1}}$ est une racine carré de l'unité différente de 1, et comme il n'y en a que deux, cela revient à dire que : $\omega^{2^{k-1}} = -1$,
et qu'on a : $-1_A \neq 1_A$, c'est à dire : $21_A \neq 0$.

Ce critère reste valable pour les racines principales :

Lemme 4.4.1 *Soit A un anneau tel que 2.1_A ne soit pas un diviseur de 0, k un entier > 0 et ω un élément de A tel que $\omega^{2^{k-1}} = -1_A$. Alors ω est une racine principale de l'unité.*

Démonstration : On a $\omega^{2^k} = (-1)^2 = 1$. Soit i un entier tel que $0 < i < 2^k$. Montrons que la relation $(\omega^i - 1)a = 0$ implique $a = 0$. Or il existe r et s avec $i = 2^r s$, $0 \leq r < k$ et $s = 1 \pmod 2$.
Comme on a $a = \omega^i a = (\omega^i)^2 a = (\omega^i)^{2^2} a$, on a aussi $a = (\omega^i)^{2^{k-r-1}} a$.
Mais comme $i2^{k-r-1} = 2^{k-1}s$, on aboutit à $a = \omega^{2^{k-1}s} a = (-1)^s a = -a$.
En définitive $2a = 0$, et, comme par hypothèse 2 ne divise pas 0 ; donc $a = 0$.

Cela s'applique au cas où A est de la forme $\mathbb{Z}/N\mathbb{Z}$ avec N entier impair de telle sorte que 2 soit inversible dans A, et ω classe d'un entier a tel que $a^{2^{k-1}} = -1 \pmod N$.
Exemples :
Soit $N = a^{2^{k-1}} + 1$ alors a est racine 2^k-ième principale dans $\mathbb{Z}/N\mathbb{Z}$.
En particulier on peut prendre le nombre de Fermat $N = 2^{2^{m+k}} + 1 = F_{m+k}$ et $\omega = 2^{2^{m+1}}$.

4.5 Multiplication rapide d'entiers à l'aide de la DFT

Soient deux nombres entiers u et v écrits en base b, à multiplier et que nous utilisions un ordinateur dont le mot machine est M. La base b est choisie de telle sorte que le produit de deux chiffres tienne encore dans un mot machine, soit $b^2 < M$. Les nombres se présentent alors sous la forme, U et V étant des éléments de $\mathbb{Z}[X]$:

$$\begin{array}{rcccl} u & = & u_0 + u_1 b + \cdots + u_{n-1} b^{n-1} & = & U(b), \\ v & = & v_0 + v_1 b + \cdots + v_{n-1} b^{n-1} & = & V(b), \end{array}$$

où les u_i et les v_i $(i = 0, 1, \ldots, n-1)$ sont dans $[0, b-1]$ et on prend $n = 2^k$. On applique alors la DFT pour multiplier les deux polynômes U et V. Pour retrouver l'écriture de uv on fait une propagation des retenues, de l'ordre de $O(n)$ opérations dans $\mathbb{Z}$.

Pour réaliser le calcul de UV en appliquant la DFT, il nous faut choisir un entier p premier tel que $\mathbb{Z}_p$ possède une racine primitive d'ordre 2^n de l'unité qui satisfait : $b^2 < p < M$.
Il est facile de vérifier que les coefficients de UV sont positifs et majorés par $n(b-1)^2$.
Si on veut que les calculs faits modulo p nous donnent la valeur exacte de UV dans $\mathbb{Z}[X]$, il faudra que $n(b-1)^2 = 2^k(b-1)^2 < p$.
Sous ces conditions nous pourrons réaliser le produit des deux polynômes en $O(n \log(n))$ opérations dans $\mathbb{Z}_p$, et ainsi obtenir uv avec la même complexité.

Les conditions d'application de la méthode sont assez restrictives. L'inéquation $n(b-1)^2 < p$ nous montre que pour un p fixé plus on cherche à travailler sur des nombres de grande taille, plus on est obligé de réduire la valeur de la base b.
Par exemple, si on veut travailler sur une machine 32 bits ($M = 2^{32}$), un des plus gros nombres premiers inférieur à 2^{32} est $p = 2013265921 = 15.2^{27} + 1$, on peut alors prendre $n = 2^{26}$ et l'inégalité $2^{26}(b-1)^2 \leq 15.2^{27}$ impose de prendre $b = 6$. Les nombres u que l'on pourra multiplier vérifieront $u < 6^{2^{26}}$ ainsi u et v seront-ils des nombres qui posséderont de l'ordre de 52 millions de chiffres en base $b = 10$.

L'idée de Pollard pour repousser cette limite est de travailler non pas sur un seul corps, mais sur un produit de corps finis $\mathbb{Z}_p$, et d'utiliser le théorème

chinois.
On choisit des entiers premiers distincts plus petit que M, $p_1, \ldots, p_l$ sur chacun desquels on puisse calculer une racine n-ième primitive de l'unité avec $n = 2^k$ et on réalise le calcul de UV modulo $p_1 p_2 \cdots p_l$. Afin que le résultat soit exact dans $\mathbb{Z}$ on doit choisir b tel que $n(b-1)^2 < p_1 \cdots p_l$. Soient U_i et V_i $i = 1, \ldots, l$ les projections de U et V dans $\mathbb{Z}_{p_i}$.

Supposons que l'on travaille avec une machine 32 bits ; on utilisera les trois moduli suivants qui tiennent sur un mot machine : $p_1 = 15.2^{27} + 1$, $p_2 = 7.2^{26} + 1$, $p_3 = 27.2^{26} + 1$. On peut alors sur les trois corps ainsi définis pratiquer une DFT à $n = 2^{26}$ points qui permettra de multiplier des nombres à 2^{25} chiffres dans une base b qui devra vérifier $2^{26}(b-1)^2 < p_1 p_2 p_3$. On choisit alors $b = 2^{16}$; les chiffres seront mémorisés sur trois octets.
La borne des nombres que l'on pourra multiplier entre eux est alors $6^{2^{26}} = 2^{805306368}$ soit des nombres qui peuvent posséder environ 240 millions de chiffres en base 10.

Un algorithme dû à Schönhage prolonge cette méthode en utilisant non pas un choix fixe de moduli, mais en procédant à un choix dynamique des p_i en cours d'algorithme ce qui permet de mieux utiliser les ressources de la machine.

Bibliographie

[1] **SAUX PICCART P.,** *Cours de Calcul Formel. Algorithmes fondamentaux* Ellipses , 1999.

[2] **NAUDIN, P. QUITTÉ,C.** *Algorithmique algébrique.* Masson, 1992.

[3] **KNUTH D.E.,** *The Art of computer Programming : seminumerical algorithms.* Addison Wesley, 1992.