Rapport de Stage

Méthodes matricielles pour la résolution de systèmes d'équations polynomiales

Stage de fin d'étude
pour l'obtention du diplôme d'Ingénieur-Maître en
Génie Mathématique et Informatique
Université de Marne La Vallée
Institut National de Recherche en Informatique et en Automatique

Arona KANE

Avril - Septembre 97

INRIA
SOPHIA ANTIPOLIS

Table des matières

Remerciements

Cette enrichissante expérience que représentent ce stage à l'INRIA et ma formation à l'Université de Marne La Vallée n'aurait jamais été possible sans la confiance que m'ont accordée Mme Christianne Cocozza, MM Ioannis Emiris, Bernard Mourrain et Michel Roussignol. Qu'ils en soient sincérement remerciés.

Je voudrais aussi rémercier tous ceux qui ont contribué, de près ou de loin, à rendre ce travail possible, en particulier ma famille qui m'a soutenu et a suporté ma longue absence.

Chapitre 1

Cadre et objet du stage

1.1 L'I.N.R.I.A

L'**I**nstitut **N**ational de **R**echerche en **I**nformatique et en **A**utomatique est un établissement public à caractère scientifique et technologique placé sous la double tutelle du ministre chargé de la recherche et du ministre chargé de l'industrie. Il a pour mission : la recherhe, la réalisation de systèmes expérimentaux, la valorisation des résultats, la formation par la recherche et la diffusion des connaissances, le développement des échanges scientifiques internationaux, la mise en œuvre d'expertises scientifiques, la contribution à des programmes de coopération, une contribution à la normalisation, ...

L'**I.N.R.I.A**, dont le siège est à **Rocquencourt**, près de Versailles, comprend cinq unités de Recherche : Lorraine, Rennes, Rhônes-Alpes, Rocquencourt, **Sophia-Antipolis**.

L'**I.N.R.I.A** représente un potentiel de 1300 personnes, dont environ 1000 scientifiques, et un budget total de 505 MF, dont 20% proviennent de ressources propres. Mon stage s'est déroulé au sein de L'**I.N.R.I.A** de Sophia-Antipolis du 7 Avril 1996 au 6 Septembre 1996.

1.2 Sophia-Antipolis

L'Unité de Recherche de l'**I.N.R.I.A** Sophia-Antipolis a ouvert ses portes en 1983. Aujourd'hui, elle représente un potentiel de plus de 400 personnes, dont environ 120 sur poste budgétaire.

Ses grands domaines d'activité sont les suivants : Génie logiciel, langages et calcul symbolique, Robotique et vision par ordinateur, traitement d'images, Mécanique des fluides numérique, hypersonique, combustion, Automatique, Réseaux, systèmes à événements discrets, Intelligence artificielle et interaction homme-machine.

Au cœur de la plus grande technopole européenne, l'unité de recherche de Sophia-Antipolis entretient des relations étroites avec de nombreux laboratoires du site et s'investit largement dans la formation sur le site de Sophia-Antipolis. Ainsi, l'**E.S.S.I** (Ecole Supérieure des Sciences de l'Ingénieur) mais également l'**I.S.I.A** (Institut Supérieur d'Informatique et d'Automatique) et **EURECOM** sont les principaux bénéficiaires, mais l'unité de **Sophia Antpolis** accueuille aussi de nombreux stagiaires venus d'autres universités en France et dans le monde.

1.2.1 Le projet Safir

L'**I.N.R.I.A** travaille sur six grands programmes répartis sur 18 projets. Mon stage s'est déroulé au sein du projet **Safir** dont l'intitulé exact est : S*ystèmes* A*lgébriques* F*ormels pour l'*I*ndustrie et la* R*echerche* qui lui même fait partie du programme *Calcul Symbolique, Programmation et Génie logiciel.*

Les objectifs de ce projet sont de développer et de promouvoir des méthodes symboliques nouvelles, en particulier pour le **C**alcul **S**cientifique et de mettre au point des outils logiciels pour le mathématicien et l'ingénieur.

Safir est un projet commun à l'**I.N.R.I.A**, à l'Université de Nice-Sophia-Antipolis et au **CNRS**. Le principal domaine de recherche est le *Calcul Formel*. Deux thèmes principaux dominent :

- la recherche et l'amélioration d'algorithmes mathématiques liés aux systèmes d'équations polynômiales, la géométrie algébrique effective, les applications aux problèmes de robotique et de vision.
- les aspects informatiques du calcul formel et plus particulièrement la conception et architecture des sytèmes de calcul formel.

Le projet met également l'accent sur l'application des résultats de ces recherches dans l'industrie et plus particulièrement sur les liens entre calcul formel et calcul numérique.

Le responsable scientifique du projet, M^r **Stephen Watt**, est l'un des auteurs du célébre système de Calcul Formel *Maple*. Les membres du projet ont une part importante d'enseignement ou d'encadrement dans les différentes formations doctorales, classes préparatoires, écoles d'ingénieurs de la région.

1.2.2 Moyens matériels et logiciels

L'**I.N.R.I.A** étant un centre de recherche en informatique, il dispose d'un parc informatique très important. Un service spécial, le Service de Support Exploitation Maintenance Informatique et Réseaux, est chargé principalement de l'installation et la maintenance des matériels et de l'infrastructure de communication interne. L'**I.N.R.I.A** dispose d'un parc de matériels comprenant près de 300 postes de travail (**SUN** ou **DEC** principalement), des dizaine de serveurs, ..., un réseau très puissant permettant de communiquer avec plusieurs centaines de machines localisées dans le monde entier grâce à des liaisons spécialisées.

1.3 But du stage

Le projet **SAFIR** s'intéresse, dans l'une des ses activités, à la résolution de systèmes d'équations polynomiales et aux applications de ces méthodes à la robotique et la vision. Parmi ces méthodes, les approches basées sur des calculs matriciels commencent à emmerger. Ces méthodes consistent à réduirent le problème de résolution à un calcul de valeurs propres de matrices de multiplications. Historiquement, ces techniques partent de travaux de Bézout, Sylvester, Macaulay,... dans lesquels des algorithmes sont donnés explicitement pour construire des matrices de multiplications par les variables. Ceci ne s'applique cependant que pour des systèmes de polynômes, dits génériques. Hélas les systèmes que l'on rencontre en pratique sont rarement génériques (tous les monômes pour un degré donné n'apparaissent pas, il y a une grosse composante à l'infini,...). Ceci a donc conduit à des études ne prenant en compte que les monômes qui apparaissent dans les équations. d'autres méthodes sont actuellement à l'étude permettant de construire des matrices ayant comme taille, exactement le nombre de solutions du système, tenant compte de la géométrie des monômes et des relations qui existent entre les coefficients.

L'originalité de certaine de ces méthodes et donc l'absence d'implémentations dans les systèmes usuels (*Maple, Axiom, Mathematica,...*) ont justifié la réalisation d'outils logiciels permettant leur mise en oeuvre. C'est ainsi qu'une librairie de classes **C++**, **ALP** (*Algèbre Linéaire pour Polynômes*), a été developpé par le projet **SAFIR**.

Le stage de 5 mois que j'ai effectué au sein de ce projet avait pour but :

– dans un premier temps, de faire un rapide tour d'horizon des méthodes matricielles et de se familirariser avec la bibliothéque **ALP.**

– puis d'analyser et implémenter ces méthodes en langage **C++** en utilsant **ALP**

– et enfin d'etendre cette bibliothéque

Ce sujet s'inscrit dans le cadre du projet européen FRISCO (LTR 21.024) dédié à la recherche sur les calculs polynomiaux et les implémentations que j'ai réalisées ont donné conduit à une nouvelle version de la librairie **ALP** qui a été incluse dans la dernière distribution des logiciels FRISCO à parîıtre prochainement.

Chapitre 2

Aspects théoriques

2.1 Introduction

Le but de ce chapitre est de présenter les bases théoriques des méthodes dont l'implémentation a été l'objet de ce stage. Comme indiqué dans le chapitre précédent, ces méthodes sont axées sur deux principales approches. La première est celle des méthodes dites du *résultant homogène* ou *dense* initiées par certains mathématiciens de la fin du 19^e siècle et du début du 20^e (*Bézout, Sylvester, Cayley...*) et qui sont basées sur des constructions génériques, *i.e*, prenant en compte la totalité des monômes possibles pour un degré donné. Dans la seconde approche, celle du *résultant creux*, plus récente, la structure combinatoire des monômes est expolitées permettant de ne considérer que ceux des monômes effectivement présents dans chaque polynôme.

L'idée commune à ces deux voies est celle de passer d'un système non linéaire d'équations polynomiales à un système, éventuellement plus large, comportant autant de *termes* que d'equations, de sorte que chaque terme puisse être assimilé à une inconnue d'un système linéaire. En gros, il s'agit de construire, à partir du système de polynômes, des matrices dont la structure donne des conditions necessaires et suffisantes sur l'existence de racines commune à ces polynômes. Dans les sections suivantes sont présentées les différentes constructions de ces matrices.

Le travail que j'ai réalisé concerne les méthodes du résultant homogène que je présente ici. Une implémentation en langage **C** des méthodes creuses, réalisées par Ioannis Z. Emiris, existent déjà dans la distribution FRISCO.

2.2 Méthodes du résultant homogène

Dans toute la suite de ce rapport, $I\!K$ désigne un corps commutatif (en pratique les corps des nombres rationnels ou complexes).

Un polynôme en plusieurs variables $P(x_1, x_2, ..., x_n)$ peut être vu soit comme un élement de l'anneau $I\!K[x_1, x_2, ..., x_n]$ soit encore comme appartenant à $I\!K[x_1, .., x_{n-1}][x_n]$. Dans le premier cas il est vu comme une somme de produits dans lequel chaque produit ${x_1}^{d_1}{x_2}^{d_2}...{x_n}^{d_n}$ est appelé *terme* ; pris avec son coefficient il est appelé *monome*. Cette définition, donné dans certains ouvrages, est conforme au vocabulaire utilisé dans la librairie *ALP* mais dans d'autres ouvrages, un terme est appelé monôme et vice versa

Le *degré* d'un terme ${x_1}^{d_1}{x_2}^{d_2}...{x_n}^{d_n}$ est défini comme étant la somme des exposants $d_1 + d_2 + .. + d_n$. Dans la seconde écriture, $P(x_1, x_2, ..., x_n)$ est vu comme un polynôme en une seule variable dont les coefficients sont eux-mêmes des polynômes en les variables $x_1, .., x_{n-1}$. On verra l'utilité pratique d'une telle écriture plus loin.

Un $n-$*uplet* $(a_1, a_2, .., a_n) \in I\!K^n$ est dit *racine* ou *zéro* du polynôme $P(x_1, x_2, ..., x_n)$ si $P(a_1, a_2, .., a_n) = 0$. $(a_1, a_2, .., a_n)$ est dit racine commune au système $\{f_1(x_1, x_2, ..., x_n), f_2(x_1, x_2, ..., x_n), .., f_n(x_1, x_2, ..., x_n)\}$(également noté $\{f_i(x_1, .., x_n) = 0, i = 1, .., r\}$) si pour tout $i = 0, .., n$ $f_i(a_1, a_2, .., a_n) = 0$

2.2.1 Méthode de Sylvester

Considérons deux polynômes $f(x)$ et $g(x)$ en une variable de degré m et n

$$f(x) = f_n x^n + f_{n-1}x_{n-1} + ... + f_1 x + f_0$$

et

$$g(x) = g_m x^m + g_{m-1}x_{m-1} + ... + g_1 x + g_0$$

La question qu'on se pose est de savoir si ces deux polynômes admettent des racines communes. Cette question conduit naturellement à chercher des conditions sur leurs coefficients respectifs. Cette condition, dite de *nullité du Resultant de Sylvester de f et g (The vanishing of Sylvester Resultant)*, à été découverte par *Euler*.

Le résultant de Sylvester de f et g est le déterminant de la matrice :

$$R = \begin{pmatrix} f_0 & 0 & 0 & \cdots & 0 & g_0 & 0 & 0 & \cdots & 0 \\ f_1 & f_0 & 0 & \cdots & 0 & g_1 & g_0 & 0 & \cdots & 0 \\ f_2 & f_1 & f_0 & \cdots & 0 & g_2 & g_1 & g_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ f_n & f_{n-1} & f_{n-2} & \cdots & f_{n-m+1} & g_n & g_{n-2} & g_{n-1} & \cdots & g_0 \\ 0 & f_n & f_{n-1} & \cdots & f_{n-m+2} & g_{n+1} & g_n & g_{n-1} & \cdots & g_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & f_n & 0 & 0 & 0 & \cdots & g_m \end{pmatrix}.$$

Si au moins f_n ou g_m est non nul, alors la nullité du resultant de Sylvester est une condition nécessaire et suffisante pour que f et g ait des racines communes.

Les résultants ont surtout trouvé leur utilité dans le calcul de projections et d'éliminations successives des variables. Celui de Sylvester, en particulier, a fait l'objet de nombreuses études par le passé.

Pour un exposé plus complet, voir(*Knuth, 1980; Loos, 1983; Collins 1967; Brown and Traub, 1971; Collins, 1971)*. Des implémentations plus ou moins efficaces sont d'ailleurs disponibles sur certains systemes de *calcul Formel*.

2.2.2 La formulation Bézout-Dixon

En 1779, Bézout avait déjà developpé une méthode de calcul du resultant de deux polynômes en une variable. Plus tard, Dixon partant d'une reformulation de la méthode de Bézout due à Cayley, proposera une extension aux polynômes en deux variables.

Selon cette approche le résultant de deux polynômes f et g de degré m et $n \leq m$ peut être obtenu de la façon suivante : on considère le déterminant $\Delta(x, y)$ de la matrice suivante de la matrice

$$\begin{bmatrix} f(x) & g(x) \\ f(y) & g(y) \end{bmatrix}. \tag{2.1}$$

C'est un polynôme en x et y dont on peut remarquer qu'il vaut $zero$ pour $x = y$, ce qui signifie également que $(x - y)$ est un facteur de ce determinant.
Il est alors facile de vérifier que :

$$\delta(x, y) = \frac{\Delta(x, y)}{(x - y)}$$

est un polynôme de degré $(n-1)$ en y symétrique en x et y et qui s'annule en chaque point x_0 racine commune à f et g ; et ce, quelque soit y.

Pour construire la matrice de Bézout-Dixon, on calcule les coefficients des n premiers monômes en y de $\Delta(x, y) = \sum_{i=0}^{n-1} \Delta_i(x) y^i$ qui sont des polynomes en x de degré $m-1$. On ajoute à cette liste les multiples $x^i g(x)$ pour $0 \leq i \leq m - n - 1$. En décomposant ces m polynômes suivant les monômes en x, on obtient alors une matrice carrée (m, m).

$$m\left\{ \begin{array}{c} \overbrace{1 \cdot \quad \cdot \quad \cdot\, x^{m-1}}^{m} \\ \left[\begin{array}{ccc} & & \\ & & \\ & & \\ & & \\ & & \\ & & \\ & & \end{array} \right] \end{array} \right. \begin{array}{l} \\ \Delta_0 \\ \Delta_1 \\ \cdot \\ \Delta_{n-1} \\ g(x) \\ \cdot \\ x^{m+n-1} g(x) \end{array} \tag{2.2}$$

On obtient ainsi une matrice moins grande que celle de Sylvester et dont le déterminant est aussi égal au résultant de ces deux polynômes.

2.2.3 Construction de Macaulay

On considère $n+1$ polynômes $P_0, P_1, \ldots, P_n$ homogènes en les variables $x_0, x_1, .., x_n$ de degrés respectifs $d_0, d_1, .., d_n$. Soit $\mathcal{M}_k$ l'ensemble des monômes de degré k en

$x_0, \ldots, x_n$ et $\nu = \sum_{i=0} d_i^n - n$. Remarquons que tout monôme de degré ν est divisible par l'un des $x_i^{d_i}$. Ce qui nous donne la décomposition en parties disjointes :

$$\mathcal{M}_\nu = x_0^{d_0} N_0 \sqcup x_1^{d_1} N_1 \sqcup \cdots \sqcup x_n^{d_n} N_n$$

On prend tous les monômes divisibles par $x_0^{d_0}$, puis parmi ceux qui restent, ceux divisible par $x_1^{d_1}$, et ainsi de suite. On voit donc que $N_k = \{\prod_i x_0^{a_0} \cdots x_n^{a_n}; \sum_i a_i = \nu - d_k, a_0 < d_0, \ldots, a_{k-1} < d_{k-1}\}$ et $\#N_n = \prod_{i<n} d_i$.
Considérons l'application

$$\begin{array}{rrcl} M & : \langle N_0 \rangle \times \cdots \times \langle N_n \rangle & \to & \mathcal{M}_\nu \\ & (f_0, \ldots, f_n) & \mapsto & \sum_i P_i f_i. \end{array}$$

Notons $[M]$ sa matrice dans une base de monômes et D son déterminant. Rappellons que le nombre de monôme de $\mathcal{M}_\nu$ (c'est-à-dire la taille de la matrice $[M]$) est $C_{\nu+n}^n$
Le déterminant D est homogène par rapport aux coefficients de chaque polynôme. En particulier, le degré de D en P_n (par rapport aux coefficients de P_n) est $\prod_{i<n} d_i$. Remarquons que $D \in I\!\!K - \{0\}$ car en substituant les $a_{i,j}$ par 0 ou 1 de telle manière que P_i deviennent $x_i^{d_i}$ et en choisissant une bonne base, la matrice $[M]$ vaut l'identité. Notons $[m_\nu]$ le vecteur des monômes de degré ν. Chaque coefficient du vecteur $[m_\nu][M]$ est donc dans l'idéal $I = (P_0, \ldots, P_n)$. Notons un tel vecteur sous la forme $[I]$. En utilisant la matrice des cofacteurs de M on voit donc que

$$[m_\nu]\, D \in [I]$$

Ceci montre en particulier que $x_0^\nu D \in I$ et que D s'annule si les P_i ont une racine commune.

Exemple :

On veut construire la matrice de Macaulay pour $P_0 = x^2 + y^2 - 2, P_1 = x\,y - 1, P_2 = a\,x + b\,y + c$. On homogénéise en x_0 en posant $x = \frac{x_1}{x_0}$, $y = \frac{x_2}{x_0}$. Ici $\nu = 2+2+1-2 = 3$.

$$\mathcal{M}_3 = \{x_0{}^3, x_0{}^2 x_1, x_0{}^2 x_2, x_0 x_1{}^2, x_0 x_1 x_2, x_0 x_2{}^2, x_1{}^3, x_1{}^2 x_2, x_1 x_2{}^2, x_2{}^3\}$$

et

$$N_0 = N_1 = \{x_0, x_1, x_2\}, N_2 = \{x_0 x_1, x_0 x_2, x_1 x_2, x_2{}^2\}$$

La matrice M vaut

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & b \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & b & a \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & a & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & b & 0 & c \\ 0 & 0 & 0 & 1 & 0 & 0 & b & a & c & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & a & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & -1 & 0 & c & 0 & 0 \\ 0 & -2 & 0 & 0 & -1 & 0 & c & 0 & 0 & 0 \\ -2 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Son déterminant vaut

$$-(b+c+a)^2(b-c+a)^2.$$

2.2.4 Bézoutien en plusieurs variables

Considérons l'application polynômiale

$$\begin{array}{rcl} \mathcal{P} : I\!\!K^n & \to & I\!\!K^n \\ \mathbf{x} & \mapsto & \mathcal{P}(\mathbf{x}) = (P_1(\mathbf{x}), \ldots, P_n(\mathbf{x})) \end{array},$$

Soient $\mathbf{x} = (x_1, \ldots, x_n)$ et $\mathbf{y} = (y_1, \ldots, y_n)$, on note alors

$$X_{(0)} = (x_1, \ldots, x_n), X_{(1)} = (y_1, x_2, \ldots, x_n), \ldots, X_{(n)} = (y_1, \ldots, y_n).$$

Le Bézoutien de $n+1$ polynômes $P_0, P_1, \ldots, P_n$ de R est le polynôme $\Phi(P_0, P_1, \ldots, P_n)$ de $R \otimes R$ définit par,

$$\Phi(P_0, P_1, \ldots, P_n) = \frac{\begin{vmatrix} P_0(X_{(0)}) & \cdots & P_0(X_{(n)}) \\ P_1(X_{(0)}) & \cdots & P_1(X_{(n)}) \\ & \vdots & \\ P_n(X_{(0)}) & \cdots & P_n(X_{(n)}) \end{vmatrix}}{\prod_{i=1}^{n}(y_i - x_i)}.$$

C'est un polynôme en $\mathbf{x}$, $\mathbf{y}$, le numérateur étant le déterminant d'une matrice dont les coefficients sont des polynômes en plusieurs variables.

Matrice associée au Bézoutien

Comme le polynôme $\Phi(P_0, P_1, \ldots, P_n)$ est un polynôme en $\mathbf{x}$, $\mathbf{y}$, il se décompose naturellement sous la forme

$$\Phi(P_0, \ldots, P_n) = \sum_{i,j} c_{i,j} \mathbf{x}^{\mathbf{u}_i} \mathbf{y}^{\mathbf{v}_j}.$$

On note $[\Phi(P_0, \ldots, P_n)] = (c_{i,j})$, la matrice des coefficients des monômes.

Exemple

Soient $P_1 = x_1^3$, $P_2 = x_2^2$ et $R = I\!\!K[x_1, x_2]$. Avec les notations introduites pour le bézoutien, on a alors :

$P_0 = 1$, $P_1 = x_1^3$ et $P_2 = x_2^2$ et $X_{(0)} = (x_1, x_2)$, $X_{(1)} = (y_1, x_2)$ et $X_{(2)} = (y_1, y_2)$

Le bézoutien des 3 polynômes P_0, P_1 et P_2 est donc donné par :

$$\Phi(1,P_1,P_2)=\frac{\begin{vmatrix} P_0(X_{(0)}) & P_0(X_{(1)}) & P_0(X_{(2)}) \\ P_1(X_{(0)}) & P_1(X_{(1)}) & P_1(X_{(2)}) \\ P_2(X_{(0)}) & P_2(X_{(1)}) & P_2(X_{(2)}) \end{vmatrix}}{(y_1-x_1)(y_2-x_2)}=\frac{\begin{vmatrix} 1 & 1 & 1 \\ x_1^3 & y_1^3 & y_1^3 \\ x_2^2 & x_2^2 & y_2^2 \end{vmatrix}}{(y_1-x_1)(y_2-x_2)}$$

soit

$$\Phi(1,P_1,P_2)=\begin{vmatrix} (x_1^2+x_1y_1+y_1^2) & 0 \\ 0 & (x_2+y_2) \end{vmatrix}=(x_1^2+x_1y_1+y_1^2)(x_2+y_2)$$

La matrice associée est :

$$[\Phi(1,P_1,P_2)]=\begin{array}{c} \\ 1 \\ x_1 \\ \vdots \\ x_1^2x_2 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & \cdots & \cdots & y_1^2y_2 \end{array} \\ \begin{pmatrix} 0 & \dots & 0 & 1 \\ \vdots & \iddots & \iddots & 0 \\ 0 & \iddots & \iddots & \vdots \\ 1 & 0 & \cdots & 0 \end{pmatrix} \end{array}$$

On peut montrer que si la matrice associée à $\Phi(P_0,P_1,\ldots,P_n)$ est carrée, alors son déterminant est un multiple du résultant.

L'implémentation des objets liés à la construction de la matrice du Bézoutien en plusieurs variables a été, comme on le verra, la partie la plus imporatante du travail qui a été effectué lors de ce stage.

2.3 Application à la résolution de systèmes polynomiaux

Les résultants et les matrices précédentes peuvent être utilisés pour résoudre un système polynomial. Considérons par exemple le systèm de n polynômes en n variables suivant :

$$f_1(x_1,..,x_n)=...=f_n(x_1,..,x_n)=0$$

Pour le résoudre en utilisant par exemple la construction de $Macaulay$ on introduit une nouvelle variable x_0 afin d'homogénéiser le système.
Soit R_u le résultant de $Macaulay$ (le determinant de la matrice correspondante) des polynômes $f_1,f_2,...,f_n,f_u$ où f_u est la forme linéaire

$$f_u=u_0x_0+u_1x_1+...+u_nx_n$$

et $u_0,u_1,...,u_n$ sont de nouvelles inconnues. R_u est alors un polynôme de degré $d=\prod_{i=1}^{n}d_i$ (ceci résulte des propriétés de la matrice de Macualay). R_u est appelé le *u-resultant* des polynômes $f_1,..,f_n$ et on montre qu'il peut être factorisé sous la forme d'un produit de formes linéaires sur $I\!\!K$ (en génŕal le corps des nombres complexes), *i.e.*

$$R_u = \prod_{j=1}^{d} (u_0\alpha_{0,j} + u_1\alpha_{1,j} + ... + u_n\alpha_{n,j})$$

et que si $(u_0\alpha_{0,j} + u_1\alpha_{1,j} + ... + u_n\alpha_{n,j})$ est facteur de R_u, alors $(\alpha_{1,j}, ..., u_n\alpha_{n,j})$ est racine commune des polynômes $f_1, .., f_n$.

Réciproquement si $(\beta_{0,j}, \beta_{1,j}, ..., u_n\beta_{n,j})$ est un zéro commun au polynômes du système, alors $(u_0\alpha_{0,j} + u_1\alpha_{1,j} + ... + u_n\alpha_{n,j}$ divise R_u. On obtient ainsi un algorithme pour retrouver l'ensemble dessolutions du système.

Exemple

Soit à chercher l'intersection entre le cercle unité x^2+y^2-1 et le plan défini par $(x_1-x_2-1)(x_1-x_2+1)$. On calcule le $u-resultant$ des deux polynômes correspondants, soit le résultant de Macaulay des polynômes homogénéisés

$$\begin{aligned} f_1 &= x^2 + y^2 - x_0 \\ f_2 &= x_1^2 - 2x_1x_2 + x_2^2 - x_0^2 \\ f_u &= u_0x_0 + u_1x_1 + u_2x_2 \end{aligned}$$

qui vaut

$$u_1^2u_2^2 - u_1^2u_0^2 + u_0^4$$

et se factorise sous la forme

$$(0.u_1 - 1.u_2 + 1.u_0)(0.u_1 + 1.u_2 + 1.u_0)(-1.u_1 + 0.u_2 + 1.u_0)(1.u_1 + 0.u_2 + 1.u_0)$$

et on peut lire directement les solutions du système dans cette écriture ce qui donne 4 points d'intersection :

$$(0, -1), (0, 1), (-1, 0), (1, 0).$$

Dans le cas numérique (pas de paramètres) des techniques d'extraction de sous-matrices de multiplication permettant de résoudre efficacement ces systèmes ont été développés notamment par **Bernard Mourrain et Ionnis Z. Emiris** qui m'ont encadré lors de ce stage.

Chapitre 3

Outils informatiques : La librairie ALP

3.1 Introduction

Le but de cette section est de donner ube description de la librairie **ALP**sur que j'ai utilisé dans mes implémentations et étendu par de nouvelles fonctionnalités
Avant de commencer la description de chaque classe et de chaque fichier, je rappelle rapidement ce qu'est une classe en **C++**.

Vue comme un type abstrait, la classe en **C++** constitue une extension naturelle de la structure (`struct`) du langage **C**. Alors qu'une structure **C** ne possède que des champs de données (*membres de données*), et que ces champs sont manipulables sans contraintes, la classe ajoute trois nouveaux concepts :

- la possibilité de définir des *champs fonctionnels*, qui représentent les opérations licites sur les objets de la classe (on parle de *fonctions-membres*) ;
- l'*encapsulation* (*contrôle d'accès*) qui permet de déclarer que certains membres (données ou fonctions) ne sont visibles que dans le corps de la classe (*membres privés*), alors que d'autres le sont dans tout le programme (*membres publics*) ;
- enfin, la classe **C++** permet au programmeur de spécifier, grâce à des fonctions membres particulières (*les constructeurs*), la sémantique des opérations de création et d'initialisation d'objets (construction), des opérations de conversion de type, ou encore d'affectation.

Une classe **C++** définit donc un nouveau type de données en précisant :

- la *représentation* (en mémoire) d'un objet de ce type, grâce aux membres de données,
- les opérations licites sur les objets du type, grâce aux fonctions membres,
- la visibilité de cette représentation et de ces opérations pour le reste du programme.

Enfin, notons que si un objet est d'un type défini par une classe, on dit que c'est une *instance* de cette classe.

3.1.1 Contenu de la librairie ALP

Ainsi, comme nous l'avons déjà présenté, la librairie **ALP** a été conçue de manière à pouvoir construire et manipuler des polynômes, des matrices, et des matrices dont chaque ligne et/ou chaque colonne est indexée par des polynômes.

De plus, nous verrons que l'une des caractéristiques principales de cette librairie est la possibilité de construire des objets (*instances*) avec partage de la mémoire. L'utilisateur sera libre de concevoir ses objets avec ou sans mémoire partagée, ceci grâce au choix important de constructeurs exportés par la bibliothèque.

Ainsi, la librairie **ALP** exporte les classes ou fichiers suivants :

La classe Monome :
Elle permet de construire des monômes génériques en donnant la taille du monôme (Taille n : monôme s'exprimant en fonction des variables $(x_1, x_2, \ldots, x_n)$), la liste de ses exposants et son coefficient.

La classe Polynome :
Elle utilise la classe **Monome** et les fonctions amies et membres qu'elles exportent. Elle permet de construire des polynômes à partir d'une liste de monômes. C'est une représentation générique des polynômes en $(x_1, x_2, \ldots, x_n)$ à coefficients dans $I\!K$ que l'on notera $I\!K[x_1, x_2, \ldots, x_n]$.

La classe Polybase :
Cette classe exporte des objets permettant de créer des bases de polynômes à partir d'une liste de polynômes. On verra plus loin, lors de la description de son implémentation qu'il s'agit simplement de gérer un tableau de polynômes.

La classe Matrix :
Comme son nom l'indique cette classe permet la création de matrices de taille **mxn**, denses et dont le type des éléments est défini dans le fichier **utils.H** par un `#define Type ‘type‘`. De nombreux opérateurs ont été redéfinis pour cette classe. La diversité des constructeurs et la possibilité de déclarer des matrices ou sous-matrices partageant la mémoire (c'est-à-dire les mêmes éléments) avec celles déjà existantes en sont ses principales caractéristiques. Ceci a été réalisé dans un souci d'économie de stockage mémoire et de manière à pouvoir également manipuler de très grandes matrices.

La classe XMatrix :
C'est une classe dérivée de la classe précédente. Elle permet de construire des matrices dont les colonnes sont indexées par des polynômes.

La classe XYMatrix :

C'est une classe dérivée de la classe **XMatrix**. Elle permet de construire des matrices dont les lignes et les colonnes sont indexées par des polynômes. Enfin, afin d'éviter un stockage double ou triple de la matrice de base en mémoire, le partage d'éléments est également possible. On verra même plus loin qu'il est possible de partager aussi les polynômes qui indexent les lignes et/ou les colonnes des matrices. Ce principe de partage mémoire lors de la déclaration des objets est très simple à comprendre, on décrira son fonctionnement dans une section suivante.

Les classes SquareMatrix, XsquareMatrix et XYSquareMatrix :

Ce sont les homologues des classes `Matrix`, `XMatrix` et `XYMatrix` pour les matrices carrées.

Le Fichier **utils.H** :

Ce fichier exporte des fonctions et constantes de bases utiles. Il n'exporte aucun objet. Il s'agit seulement de programmation procédurale.
Toutes ces déclarations ont été utiles et nécessaires pour rendre plus convivial l'écriture des différentes classes exportées par la librairie **ALP**.

Le Fichier **outils.H** :

Ce fichier exporte des procédures ou fonctions qui permettent d'effectuer des opérations matricielles telles que la *triangularisation LU*, la *réduction* d'une matrice par une matrice triangulée, la *compression*, ...sur les matrices dont les lignes et les colonnes sont indexées par des polynômes.

Le Fichier **outils-lapack.H** :

Il offre une interface **C++** avec la bibliothèque **LAPACK**. Il permet d'utiliser certaines routines **LAPACK** pour effectuer des opérations matricielles telles que la triangularisation LU, la recherche de valeurs et vecteurs propres, la décomposition SVD, ...sur des objets `Matrix`, `XMatrix` et `XYMatrix`.

Le Fichier **ALP.H** :

C'est tout simplement un fichier contenant tous les fichiers d'entêtes afin d'utiliser la librairie **ALP**.

3.1.2 La classe Monome

Introduction

Cette première classe servira de base pour la classe `Polynome`. Elle permet de construire et de manipuler des monômes à plusieurs variables.

Implémentation

La question à se poser ici est : quelles sont les données nécessaires pour définir complètement un monôme ? Bien entendu, plusieurs choix sont ici possibles. Notre objectif ici est de pouvoir construire des monômes du type :

$$\boxed{C * x_1^{d_1} * x_2^{d_2} * x_3^{d_3} * \ldots * x_n^{d_n}}$$

Nous avons donc choisi de décrire un monôme par :

- son coefficient C
- sa taille n
- son degré total $d = \sum_{i=1}^{n} d_i$
- sa liste d'exposants $(d_1, d_2, d_3, \ldots, d_n)$

Le coefficient est de type `Type` défini dans le fichier **utils.H**, la taille et le degré seront de type `int`, et la liste d'exposants sera un tableau de `int`. Voici résumés sur ce tableau les membres de donnée de la classe `Monome` :

	type	*Identificateur*
Coefficient	`Type`	`Coef`
Degré	`int`	`Degre`
Taille	`int`	`SizeM`
Exposants	`int *`	`ListExp`
Rang	`int`	`Rank`

La donnée supplémentaire (`Rank` est ajouté pour fixer la position d'un monôme par rapport à d'autres dans des opérations de tri très utiles dans les applications.

On remarque donc que la construction et la destruction d'un monôme nécessitent l'allocation et la désallocation dynamique d'un vecteur d'entiers représentant la liste des exposants du monôme : c'est le champ `ListExp` qui est un pointeur sur un tableau d'entiers. Car en effet, **C++** nous impose de gérer nous même la mémoire. Ceci se fait grâce aux mécanismes de construction d'un objet (opérateur **new** pour allouer dynamiquement) et grâce à l'appel implicite d'un destructeur d'objet à la sortie d'un bloc dans lequel ce dernier est visible.

L'inconvénient d'une telle implémentation est le stockage inutile de zéros dans le cas où l'on veut par exemple représenter en mémoire le monôme : $5. * x_5^4$. En effet le stockage de la liste d'exposants serait alors le suivant : `ListExp` $\longrightarrow$ | 0 | 0 | 0 | 0 | 4 |.

Fonctions, opérateurs et opérations licites exportés par la classe `Monome`

Constructeurs :

C++ permet de maîtriser l'initialisation et la construction d'un objet à travers des fonctions-membres particulières : *les constructeurs*. Ce sont des fonctions membres dont le nom est le même que celui de la classe. La classe `Monome` exporte *un constructeur par défaut* (constructeur sans argument), *un constructeur de copie* (permet de régler tous les cas d'initialisation d'une instance avec une instance de la même classe), et trois *constructeurs de création*.

Supposons que nous voulons déclarer le monôme : $5 * x_1^2 * x_2^2 * x_3^3$. Il faut alors initialiser les champs de ce `Monome` avec :

- `Coef` à 5.0

- `Degre` à 2+2+3
- `SizeM` à 3
- `ListExp` par `ListExp` $\longrightarrow$ | 2 | 2 | 3 |

Accesseurs :

Ce sont des fonctions membres qui permettent d'accéder en écriture ou en lecture aux champs de la classe `Monome` qui ne sont pas visibles de l'extérieur. Trois de ces fonctions donnent un accès en lecture aux champs `Coef`, `Degre` et `SizeM` du monôme correspondant. Les autres permettent soit d'obtenir l'exposant d_i de la variable x_i, soit de positionner le coefficient du monôme, soit de positionner l'exposant de la variable x_i.

Affectation :

C++ permet de redéfinir l'affectation pour un nouveau type d'objet. La classe `Monome` redéfinit donc cette affectation.

Opérateurs de tests :

On souhaite pouvoir faire des tests et des opérations arithmétiques sur les `Monome`.**C++** fournit la possibilité de surcharger les opérateurs habituels `+, -, *, /, ==, !=, ...` ce qui allège et rend plus naturelle l'écriture des expressions. Ainsi on veut pouvoir comparer des monômes entre-eux et savoir si un monôme est nul ou pas. Les opérateurs redéfinis dans la classe `Monome` sont : `==`, `!=` et $<$. Ce dernier opérateur implique de définir un ordre sur les monômes. L'ordre choisi est le suivant : on compare d'abord le degré total des monômes, et si ils ont le même degré, on utilise alors l'ordre lexicographique.

Opérateurs arithmétiques :

Quelles sont les opérations licites sur les monômes ? On peut prendre l'opposé d'un monôme, multiplier deux monômes entre eux (le résultat donne bien un monôme), multiplier un monôme par une constante, diviser un monôme par une constante, ou élever un monôme à la puissance n (le résultat est bien un monôme). Par contre, on ne peut additionner ou soustraire deux monômes, car en effet le résultat ne sera pas un monôme si les deux monômes ne sont pas identiques.

Opérateurs d'entrée-sortie :

La bibliothèque standard **C++** fournit un mécanisme élégant pour réaliser l'affichage d'une classe, grâce à la prédéfinition de classes particulières, *"streams"*. Deux types de streams sont définis dans la bibliothèque (fichier `stream.h` : les *ostreams* correspondent aux flux de sortie, les *istreams* aux flux d'entrée. Les 2 opérateurs importants de **C++** pour les opérations d'Entrée-Sortie sont : $\ll$ et $\gg$. Ce sont ces opérateurs que nous avons surchargés.

Fonctions utiles :

La classe `Monome` exporte quelques fonctions utiles qui permettent d'éliminer complètement un `Monome` précédemment déclaré, de dériver un monôme, de sortir un `Monome` au format **Maple**, d'ajouter ou de swaper des variables dans un `Monome` , de fixer son rang dans une liste de monômesselon un ordre prédéfini et de le retrouver.

Chaque classe de la librairie **ALP** possède la même structure que la classe `Monome`. Chacune exporte des *constructeurs* (un constructeur par défaut, un constructeur de copie, et plusieurs constructeurs de création), des *accesseurs* et des *fonctions* propres à la manipulation de son instance. Chacune redéfinit également l'*affectation*, certains *opérateurs arithmétiques*, de *tests* ou *d'entrée-sortie*. Il serait trop lourd de détailler chacune de ces fonctions pour chaque classe de la librairie d'autant plus que le **Manuel de référence** est là pour ça justement.

Ainsi, je me limiterai à la description des points importants et des caractéristiques principales de chaque classe (implémentation, fonction particulière, lien avec les autres classes, ...). Enfin, je mettrai en valeur les concepts nouveaux que j'ai pu acquérir lors de leur implémentation.

3.1.3 La classe Polynome

Introduction

Elle utilise la **classe Monome**. Elle permet de construire des polynômes à partir d'une liste de monômes. C'est une représentation générique des polynômes en $(x_1, x_2, \ldots, x_n)$ à coefficients dans $I\!K[x_1, x_2, \ldots, x_n]$.

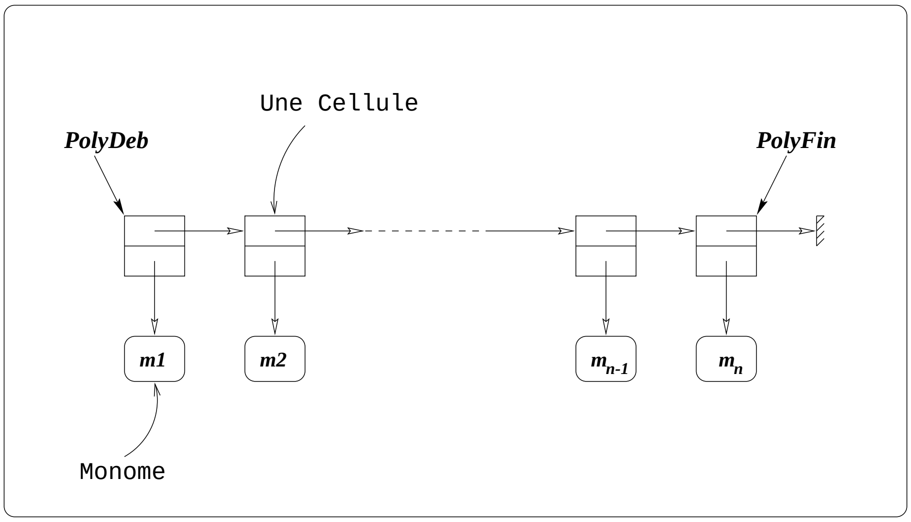
Implémentation

Un polynôme est un ensemble de monômes que l'on décide d'ordonner suivant un ordre préfixé (l'ordre sur la classe `Monome` a été décrit dans la section précédente). De plus il est nécessaire, dans un souci de stockage mémoire de simplifier le polynôme (éviter les redondances de `Monome`) lors de sa construction ou lors de l'insertion d'un `Monome` dans un `Polynome` (comme dans l'opérateur +, -, ...).
L'objectif est donc de pouvoir construire un ensemble de monômes ordonné et simplifié. Il y a plusieurs choix de représentation interne pour construire un `Polynome`. Notre choix s'est porté sur une **liste**.

Un `Polynome` est donc décrit par :

- Un pointeur sur la première **Cellule** de la liste
- Un pointeur sur la dernière **Cellule** de la liste
- Son degré définit comme le max des degrés de chacun des monômes constituant le polynôme.

Voici donc schématisée la représentation interne d'un polynôme :



Implémentation interne d'un `Polynome`

Les membres de données propres à la classe `Polynome` sont donc :

	type	*Identificateur*
Degré	`int`	`Degre`
liste	`Cellule *`	`PolyDeb`
liste	`Cellule *`	`PolyFin`

Notre choix de représenter un polynôme par une liste, nous impose de devoir effectuer des parcours de liste. Par exemple, l'insertion d'un `Monome` dans un `Polynome` nécessite un parcours de la liste jusqu'à la cellule où l'on veut insérer ce nouveau monôme. Pour cela, nous avons décidé d'utiliser classiquement un **itérateur** (pointeur courant sur la liste) pour parcourir la liste. C'est le but de la classe `PolyIter` définie dans le même fichier **"polynome.H"**. (Cf **Annexe B**)

Fonctions, opérateurs et opérations licites exportés par la classe `Polynome`

Comme chaque classe de la librairie **ALP**, la classe `Polynome` exporte un *constructeur par défaut*, *un constructeur de copie* et plusieurs *constructeurs de création* qui permettent de construire un `Polynome` à partir d'une liste de `Monome`. Elle exporte également un constructeur particulier. En effet, un `Monome` peut (doit) être vu comme un `Polynome`. Pour cela, j'ai implémenté un constructeur à un seul argument (un `Monome`) qui permet donc de définir une conversion implicite d'une instance de type `Monome` vers une instance de type `Polynome`. Ceci autorise le mélange d'objets de type `Monome` et de type `Polynome` dans les calculs notamment.

La classe `Polynome` définit des *accesseurs* qui permettent soit de donner le degré du polynôme, soit d'obtenir le degré maximum de la variable x_i dans le polynôme, soit d'obtenir le nombre de variables dans le polynôme.

Elle redéfinit également l'*affectation*, les opérateurs `==`, `!=`, $<$, `- unaire`, `+`, `-`, `*` ainsi que les opérateurs d'entrée-sortie $\ll$ et $\gg$.

La possibilité de sauvegarder les polynômes au format **Maple** a permis de vérifier, tester et comparer les résultats issus de mes programmes tests et ceux issus de **Maple**.
Le but de la classe `PolyIter` est de définir notre **itérateur** qui, je le rappelle, est simplement un pointeur courant sur la liste. Le but de cette classe est non seulement de rendre plus convivial l'écriture des *opérateurs de tests* qui nécessitent un parcours de liste mais également de donner la possibilité à l'utilisateur de travailler sur les monômes constituant le `Polynome` (grâce aux fonctions-membres exportées par la classe `PolyIter`).

3.1.4 La classe Polybase

Maintenant que nous sommes capables de créer et de manipuler des polynômes, nous pouvons construire une classe permettant de manipuler des "*bases polynô-*

miales" qui dans notre cas seront assimilées à un ensemble de polynômes.

Implémentation

Le but de cette classe est donc de travailler sur un ensemble de polynômes. La structure de données la plus simple pour manipuler cet ensemble est un simple tableau d'objets `Polynome`.
Soit $P_1, P_2, \ldots, P_n$, un ensemble de polynômes, on désire donc considérer cet ensemble comme un seul même objet. Notre objectif est donc de construire des "*bases polynômiales*" du type :

$$\boxed{B \;=\; (P_1, P_2, \ldots, P_n)}$$

Une "*base polynômiale*" est donc décrit par :

- la dimension du tableau n
- sa liste de polynômes $(P_1, P_2, \ldots, P_n)$

La spécificité et la caractéristique principale de la librairie **ALP** résident dans le fait que l'on puisse créer des objets avec "*mémoire partagée*". C'est le cas de cette classe `Polybase` qui permet de déclarer des "*sous-bases polynômiales*" à partir de bases déjà construites. Ceci a été conçu dans un souci d'optimisation de stockage en mémoire. On verra dans la présentation des classes qui suivent que ce mécanisme est largement repris.

Il est donc possible de construire une fois pour toute une grosse "*base polynômiale*" en mémoire, de déclarer des sous-bases de cette dernière partageant les mêmes éléments et de travailler sur ces sous-bases. Bien-entendu, du fait du partage de mémoire, toutes les opérations effectuées sur ces sous-bases seront répercutées sur la grosse base.

Le fait que **C++** nous impose de gérer nous-même la mémoire dynamiquement par l'intermédiaire des *constructeurs* et du *destructeur*, nous amène à nous poser la question : *Comment simuler le partage de mémoire lors de la construction, la destruction et la manipulation d'objets ?* En effet, si l'on décide qu'une "*base polynômiale*" partage la mémoire avec une autre, de grandes précautions doivent être prises lors de la destruction de l'objet qui partage la mémoire : il ne faut pas détruire le tableau de `Polynome` car il appartient physiquement à une autre base, la destruction de ce tableau impliquerait la destruction de toutes les bases partageant ce même tableau.

L'option que nous avons choisie pour simuler ce partage de mémoire est très simple. Il suffit d'ajouter un champ de données à chaque `Polybase` qui sera un `boolean` nommé `Allouer`.
Si `Allouer` est `TRUE`, l'objet `Polybase` ne partage pas la mémoire et s'il est à `FALSE`, il partage la mémoire.

Ainsi, par exemple, lors de la destruction d'un objet `Polybase`, il suffira de tester ce booléen pour savoir s'il faut détruire ou pas le tableau de `Polynome` attaché à cet objet.

Voici résumés sur ce tableau, les membres de données propres à `Polybase` :

	type	*Identificateur*
Taille	`int`	`Dim`
Tableau	`Polynome *`	`PolyBase`
Mémoire	`boolean`	`Allouer`

Fonctions, opérateurs et opérations licites exportés par la classe `Polynome`

En plus du constructeur par défaut et du constructeur de copie, la classe `Polybase` exporte 6 constructeurs de création qui permettent de construire des objets `Polybase` partageant la mémoire ou pas.

Supposons que nous ayons déclaré les `Polynome P1, P2, P3, P4` et `P5` et que nous voulons construire une "*base polynômiale*" `B` en mémoire. Il faut initialiser les champs de cette instance `B` avec :

- `Dim` à 5
- `PolyBase` par `PolyBase` $\longrightarrow$ | P1 | P2 | P3 | P4 | P5 |
- `Allouer` à `TRUE`

Supposons maintenant que nous voulons déclarer une sous-base `B`$'$ qui partage les trois premiers polynômes avec la base précédente `B`. Il faut alors initialiser les champs de cette instance `B`$'$ avec :

- `Dim` à 3
- `PolyBase` partage les polynômes `P1`, `P2` et `P3` avec la précédente
- `Allouer` à `FALSE`

La classe `Polybase` redéfinit l'affectation. Cette affectation n'est possible que si la partie gauche et la partie droite de l'affectation sont de même dimension. Cette classe redéfinit également les opérateurs `==`, `!=` et les opérateurs d'entrée-sortie $\ll$ et $\gg$.

Elle exporte également quatre fonctions membres pour effectuer des opérations élémentaires sur les polynômes constituant la base. Elles permettent soit :

- d'effectuer une opération élémentaire sur place sur le `Polybase` P_i qui est $P_i = P_i + \lambda * P_j$;
- de swaper deux polynômes de la base ;
- de supprimer les premiers ou les derniers polynômes de la base.

Remarques

La classe `Polybase` sera utilisée par les classes permettant de construire des matrices dont les lignes et/ou les colonnes sont indexées par des polynômes.

Le corps de la plupart des fonctions ou opérateurs exportés par la classe `Polybase` tiennent compte du fait que les instances de la classe partagent ou pas la mémoire (il suffit de tester le champ `Allouer`).

Les constructeurs qui permettent de déclarer des sous-bases à partir de bases déjà construites en mémoire prennent obligatoirement une instance de type `Polybase` en argument. C'est la façon de passer cet argument (par **pointeur** ou par **référence**) au constructeur qui va déterminer si la sous-base que nous voulons créer va partager la mémoire avec cet instance. Le principe que nous avons adopté est le suivant :

> Lorsque un objet déjà stocké en mémoire est passé en argument d'un constructeur par *référence* **&**, il y a allocation dynamique donc copie de mémoire. Lorsque celui-ci est passé par *pointeur* *****, il y a partage de mémoire donc pas d'allocation dynamique. Ce mécanisme de construction est le même dans toutes les déclarations d'objets avec mémoire partagée.

On retrouvera ce principe lors des constructions de matrices. L'utilisateur a donc la possibilité de déclarer des objets avec mémoire partagée, ceci implique également une plus grande vigilance sur les opérations à effectuer sur de tels objets. Car toute modification d'un objet à mémoire partagée implique une modification sur l'objet "*père*" et réciproquement.

3.1.5 Schéma de dérivation de base

L'objectif principal de la librairie **ALP** est de manipuler des matrices dont les lignes et/ou les colonnes sont indexées par des polynômes. Il faudrait bien entendu pouvoir manipuler des matrices de n'importe quel type : *dense, carrée, triangulaire supérieure, inférieure, symétrique, ...*. La programmation orientée objets du langage **C++** permet de concevoir une hiérarchie naturelle sur ces matrices.

La classe de plus haut niveau de cette hiérarchie serait bien sûr celle des **matrices rectangulaires mxn**.

L'idée naturelle de représenter les différents types de matrices par une hiérarchie de classes **C++** doit bien être exploitée. En effet, même si les matrices seront de types différents, on veut pouvoir leur appliquer des propriétés, méthodes ou opérations élémentaires communes. Seul le corps de ces fonctions sera différent pour chacun des types de matrices (car d'implémentation différente).

Ainsi, **C++** offre la possibilité de concevoir une **classe abstraite** (une classe qui n'a pas d'instance) qui fédère les propriétés communes à toutes les matrices quel que soit leur type exact. Cette **classe abstraite** doit être la classe de plus haut niveau dans la hiérarchie.

Le premier schéma de dérivation que nous avons mis au point pour le type de base des matrices (**matrices rectangulaires mxn**) et des objets issus de ces matrices (**Xmatrices** et **XYmatrices**) est le suivant :

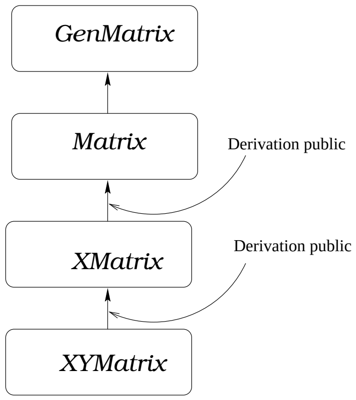


Schéma de dérivation de base

- La classe **GenMatrix** est la classe abstraite.
- La classe **Matrix** est la classe qui dérive de **GenMatrix** et qui représente les **matrices rectangulaires mxn**.
- La classe **XMatrix** dérive de la classe **Matrix**, elle représente les matrices rectangulaires dont les colonnes sont indexées par des polynômes.
- La classe **XYMatrix** dérive de la classe **XMatrix**, elle représente les matrices rectangulaires dont les lignes et les colonnes sont indexées par des polynômes.

Ce schéma de base permet la création et la manipulation de *matrices, Xmatrices et XYmatrices* du type le plus général (**matrices rectangulaires mxn**). Nous verrons plus loin, que lors de l'ajout d'un nouveau type de matrice, il sera nécessaire d'effectuer quelques modifications sur ce schéma au niveau de la déclaration des dérivations (à cause de l'héritage multiple qui sera introduit).
Les sous-sections qui suivent décrivent chacune de ces classes et expliquent le lien entre ces différentes classes.

3.1.6 La classe GenMatrix

Anisi, nous avons utilisé la possibilité offerte par **C++** de concevoir une **classe abstraite** (une classe qui n'a pas d'instance) qui fédère les propriétés communes à toutes les matrices quel que soit leur type exact. C'est donc la classe de plus haut niveau dans la hiérarchie. Toutes les classes hériteront de cette classe.

Cette classe abstraite ne contient que des *fonctions virtuelles pures* (mécanisme de liaison dynamique), c'est-à-dire qu'elles n'ont pas besoin de corps. Ce sont les classes concrètes dérivées de cette classe qui seront responsables de la définition de chacune d'elles. Le fait que ces fonctions soient déclarées *virtuelles* siginifie non seulement que la fonction doit être redéfinie dans la ou les classe(s) dérivée(s), mais surtout que c'est cette dernière version qu'il faudra utiliser à l'execution dès que l'on connaîtra le type exact de l'objet (sa classe concrète de plus bas niveau).
De plus, la définition de cette classe ne donne aucune information sur l'implémentation effective des matrices.

Les propriétés communes sont les suivantes :

- le destructeur ;
- l'opérateur `()` d'accès aux éléments des matrices ;
- les opérations élémentaires pour swaper les lignes ou les colonnes ;
- les opérations élémentaires `AddRow` et `AddCol` ;

- les méthodes permettant de supprimer les premières ou dernières colonnes ou lignes ;
- l'opération d'élimination d'une matrice.

Remarques

Le fait que toutes ces fonctions soient déclarées en fonctions virtuelles, implique que c'est le *type effectif de l'objet (pointé ou référencé) qui va déterminer la méthode à appeler.* Ce mécanisme est la **liaison dynamique** et elle est très intéressante pour les différents types de matrice. Car en effet, le typage est résolu à l'exécution afin d'invoquer la version correspondante au type exact de l'objet.
Notons que le mécanisme de typage dynamique n'est mis en œuvre que si les objets sont accédés à travers des pointeurs ou des références.

3.1.7 La classe Matrix

Introduction

La première classe *concrète* qui dérive de la classe abstraite `GenMatrix` est la classe permettant donc de manipuler des matrices rectangulaires **mxn** denses, la classe `Matrix`. C'est bien évidemment le type de matrices la plus générale. C'est pour cette raison qu'elle se positionne si haut dans la hiérarchie.

Implémentation

La question à se poser ici est "quelles sont les données membres nécessaires pour définir complètement une matrice rectangulaire dense et quelle est la meilleure implémentation interne pour stocker les éléments des matrices ? " A ces deux questions, plusieurs réponses sont possibles.

En ce qui concerne l'implémentation des matrices, notre objectif était double :

- offrir des matrices à mémoire partagée
- rendre l'implémentation proprement dite (le stockage des éléments de la matrice) totalement indépendante de la classe **Matrix**.

Il semble naturel de choisir deux membres pour définir le nombre de lignes et de colonnes de la matrice. Pour la mémoire partagée, nous avons appliqué le même mécanisme que pour la classe `Polybase`, à savoir un booléen. Cependant, pour certaines opérations, il ne suffit pas de savoir si la matrice est à mémoire partagée, il faut également savoir si les matrices à mémoire partagée sont extraites de la même matrice. Nous avons choisi pour cela un identificateur `who` qui sera un entier pour chaque matrice. Chaque fois qu'une matrice est allouée dynamiquement, un compteur est incrémenté de 1 et donne sa valeur à l'identificateur. Par contre, si on déclare une matrice à mémoire partagée, son identificateur sera le même que la matrice "*mère*".
Enfin, pour rendre indépendant la classe `Matrix` de son implémentation, nous avons choisi de prendre un membre comme un pointeur sur un type `_Matrix` qui est donc

une classe *qui définit la structure de données que nous avons choisie pour les matrices rectangulaires denses* ainsi que les fonctions ou méthodes propres à la structure de données.

Voici résumés sur ce tableau les membres de données propres à la classe `Matrix` :

	type	*Identificateur*
Nb	`int`	`numRows`
Nb	`int`	`numCols`
	`_Matrix *`	`matrix`
Mémoire	`boolean`	`Allouer`
Id	`int`	`who`

But de la classe `_Matrix`

Comme nous l'avons expliqué précédemment, un de nos objectifs était de rendre la classe `Matrix` indépendante de son implémentation proprement dite. C'est pourquoi nous avons défini cette classe `_Matrix`, qui représente donc la structure de données de la matrice.

Ce choix implique notamment **un second schéma de dérivation** pour la hiérarchie d'implémentation à cause des différents types de matrices et donc des différents types de structures de données. La classe `_Matrix` sera donc **la classe la plus haute de cette hiérarchie d'implémentation**.

Le second avantage, non négligeable, de dissocier l'implémentation de la matrice de son type est de pouvoir utiliser la même implémentation pour des types de matrices différents. Prenons un exemple pour mieux comprendre : les matrices symétriques, les matrices triangulaires supérieures et inférieures sont de type différents (elles auront une interface différente) mais par contre elles pourront avoir la même implémentation qui est celle d'une **demi-matrice** (elles utiliseront donc la même classe d'implémentation qu'on peut définir par `_Semi_Matrix`).

Implémentation de la classe `_Matrix`

C'est donc cette classe qui est responsable de la définition de la structure de données associée à la classe **Matrix** mais également de toutes les opérations intrinsèques à cette structure de données.

Il faut se demander quelle est la meilleure structure de données pour représenter en mémoire une matrice rectangulaire dense. Là encore, plusieurs possibilités. Certains préfèreront un tableau pour chaque ligne ou chaque colonne de la matrice (ceci implique plusieurs allocations ou désallocations de tableaux), d'autres, un unique tableau avec des pointeurs sur chaque début de ligne,.... Nous, nous avons privilégié le stockage en mémoire en choisissant d'allouer *un seul tableau* pour toute la matrice. Ceci implique donc un gain de mémoire (pas de tableau de pointeurs alloué) mais par contre exige un tout petit calcul élémentaire pour accéder à l'élément **(i, j)** de la matrice.

Reste à savoir si le stockage de la matrice se fera en lignes ou en colonnes. Anticipant sur le futur (en sachant que nous allons avoir besoin d'utiliser la librairie **LAPACK** écrite en **Fortran**), nous avons décidé de les stocker en colonnes (comme le fait **Fortran** pour ses tableaux bidimensionnels). Ceci afin d'éviter des copies de mémoire lors du passage par arguments des matrices de notre librairie aux routines **LAPACK**. On reviendra sur ce point plus en détail lors de la description de l'interface avec **LAPACK**.
Enfin, cette classe doit prévoir un membre pour prendre en compte le *partage de mémoire*.

Fonctions, opérateurs et opérations licites exportés par la classe `Matrix`

En plus du constructeur par défaut et du constructeur de copie, la classe `Matrix` exporte huit constructeurs de création. La possibilité de déclarer des matrices ou sous-matrices partageant la mémoire (c'est-à-dire les mêmes éléments) avec des matrices déjà stockées en mémoire en est sa principale caractéristique. Ceci a été réalisé dans un souci d'économie de stockage mémoire et de manière à pouvoir manipuler de très grandes matrices.

Les constructeurs qui permettent de construire des sous-matrices à partir de matrices déjà stockées en mémoire suivent le même principe que celui exposé pour la classe `Polybase`. Le passage par *référence* implique une copie de mémoire, le passage par *pointeur* implique un partage de mémoire.

La classe `Matrix` redéfinit l'affectation. Cette affectation n'est possible que si la matrice de la partie gauche et la matrice de la partie droite de l'affectation sont de même taille ou si la matrice de gauche est vide.

Cette classe redéfinit également les opérateurs de tests `==`, `!=`, les opérateurs arithmétiques `+`, `-`, `*`, `+=`, `-=`, `*=` et l'opérateur d'entrée-sortie $\ll$. Elle exporte également des fonctions booléennes permettant de savoir si tous les coefficients d'une ligne ou d'une colonne de la matrice sont nuls.

Bien-entendu, la classe `Matrix` surcharge les opérateurs `()` pour les `Matrix` de manière à pouvoir écrire l'accès aux éléments de la matrice très naturellement `M(i, j)`. C'est une redéfinition concrète des opérateurs `()` de la classe abstraite **GenMatrix** qui ont été déclarés virtuels.
Différentes fonctions membres donnent accès au nombre de lignes et de colonnes de la matrice, ou permettent d'affecter tous les éléments de la matrice à la valeur passée en argument.

Elle exporte également plusieurs fonctions membres pour effectuer des opérations élémentaires sur la matrice. Ce sont les opérations bien connues telles que `SwapRow, SwapCol, AddRow, AddCol, ...`. Ce sont la plupart des fonctions déclarées virtuelles dans la classe abstraite `GenMatrix`. Elles sont redéfinies dans cette première classe concrète de matrice rectangulaire dense. Elles seront redéfinies pour chacune des différentes implémentations de type de matrices (car structures de données différentes) mais également pour les **Xmatrices** et **XYmatrices** associées. Elles permettent soit :

– d'effectuer une opération élémentaire sur place sur les lignes ou les colonnes de la matrice qui est respectivement : $L_i = L_i + \lambda * L_j$ et $C_i = C_i + \lambda * C_j$;

– d'échanger deux lignes ou deux colonnes de la matrice ;

– de supprimer les premières ou les dernières lignes ou colonnes de la matrice.

Enfin, la classe `Matrix` exporte des fonctions utiles permettant d'éliminer une matrice, de swaper deux matrices, de redimensionner une matrice, de sortir une matrice au format **Maple** (très utile pour réaliser des tests avec **Maple**), de **transposer une matrice**, d'extraire la matrice inférieure ou la matrice supérieure.

3.1.8 La classe XMatrix

Introduction

Maintenant que nous sommes capables de créer des matrices et des bases polynômiales *avec ou sans mémoire partagée* on peut construire une classe permettant de gérer des matrices dont les colonnes sont indexées par des polynômes. C'est le but de la classe **XMatrix**.

Implémentation

Nous allons utiliser les avantages de la programmation orientée objets de **C++**. En effet, il semble tout à fait naturel de dériver la classe `Matrix` en une classe `XMatrix` et de lui ajouter un membre de données supplémentaire pour gérer les polynômes qui vont indexer les colonnes de la matrice. De ce fait, on utilise tous les avantages de la dérivation :

- il y a une conversion implicite de la classe dérivée vers la classe de base (ici de `XMatrix` vers `Matrix`) ;

- la classe dérivée **hérite** des propriétés de la classe de base : sa structure **C** sous-jacente comporte celle de `Matrix` plus le membre de données propre à `XMatrix`, **on peut appliquer à une `XMatrix` toutes les fonctions-membres publiques de `Matrix`** ;

- on peut ajouter de nouvelles propriétés (champs de données, fonctions-membres, fonctions amies) ;

- **on peut redéfinir des fonctions membres héritées** ;

- la profondeur de dérivation est illimitée.

Pour obtenir la classe `XMatrix`, il suffit de dériver la classe `Matrix` comme cela est fait sur le schéma de dérivation (*dérivation publique* : les membres **publics** de `Matrix` restent **publics** dans `XMatrix`, les membres **protégés** restent **protégés** dans `XMatrix`) et de lui ajouter un membre de donnée `BaseCol` qui sera de type

Polybase. C'est ce dernier qui sera responsable de la gestion des polynômes qui vont indexer les colonnes de la matrice.

La classe XMatrix hérite donc de tous les champs de données de la classe Matrix et ajoute le membre de données :

	type	*Identificateur*
	Polybase	BaseCol

L'intérêt de manipuler de tels objets (matrices dont les colonnes sont indexées par des polynômes) avec un tel schéma de dérivation est de pouvoir appliquer toutes les propriétés d'une Matrix à un tel objet mais aussi de pouvoir soit les redéfinir (par exemple : un SwapCol sur XMatrix va impliquer également un *Swap* sur les polynômes), soit lui en ajouter des nouvelles.

Fonctions, opérateurs et opérations licites exportés par la classe XMatrix

En **C++**, *une classe dérivée n'hérite jamais ses constructeurs de la classe de base* ; elle doit définir ses propres constructeurs. Seul le constructeur de copie peut échapper à cette règle.
Cependant, il suffit de définir des constructeurs de création dans la classe XMatrix qui invoquent les constructeurs de la classe Matrix et de la classe Polybase pour la création du champ BaseCol.
La classe XMatrix exporte pas moins de 16 constructeurs de création. La multitude et la diversité de ses constructeurs en sont ses principales caractéristiques. De plus, l'utilisateur sera libre de concevoir ses objets avec ou sans mémoire partagée (matrice et/ou base polynômiale avec mémoire partagée ou pas). Le mécanisme de construction d'objets avec mémoire partagée est bien entendu le même que celui de la classe Matrix et de la classe Polybase : supposons qu'un objet soit déjà déclaré (par ex une matrice et/ou une base), s'il est passé en argument par *référence* à un constructeur il y aura copie de mémoire, s'il est passé par *pointeur* il y aura partage de mémoire.

On a déjà expliqué que toutes les propriétés d'une Matrix s'appliquent à une XMatrix grâce à la dérivation publique. On peut donc appliquer tous les accesseurs de classe de base Matrix à une XMatrix.

La classe XMatrix hérite également des opérateurs de tests de la classe Matrix. Ainsi on peut tester l'égalité de deux XMatrix en faisant appel implicitement à l'opérateur == de classe Matrix, grâce à la conversion implicite du type dérivé vers le type de base. Cependant, le test ne s'effectuera que sur la matrice et non sur la base qui indexe les colonnes de la matrice. Aussi, à cause de la présence du champ BaseCol, il faut redéfinir les opérateurs == et != pour la classe XMatrix. Ces derniers se contentent d'appeler le même opérateur de la classe Matrix pour la partie héritée et le même opérateur de la classe Polybase pour tester le champ BaseCol. C'est le même cas pour l'opérateur d'entrée-sortie $\ll$ qui a été redéfini pour afficher à la fois la matrice et les polynômes qui indexent les colonnes.

On pourrait également se contenter d'utiliser les opérations élémentaires comme `AddRow, AddCol, SwapRow, SwapCol, ...` de la classe de base `Matrix`. Cependant, du fait de la présence d'une base polynômiale indexant les colonnes de la matrice, il est nécessaire de redéfinir toutes les opérations qui mettent en œuvre des modifications sur les colonnes. Par exemple, lorsqu'on fait un `SwapCol` sur une `XMatrix`, il faudrait également *swaper* les polynômes qui indexent les 2 colonnes à *swaper*. Par contre, les opérations sur les lignes d'une `XMatrix` n'ont aucune incidence sur les polynômes. C'est pourquoi, il n'est pas nécessaire de redéfinir toutes celles qui concernent les opérations sur les lignes (celles de la classe `Matrix` seront appelées implicitement).

De plus, il faut se rappeler que la classe `Matrix` dérive de la classe abstraite `GenMatrix` dans laquelle sont déclarées virtuelles toutes les opérations élémentaires. Ceci permettra lors de l'exécution du programme par le mécanisme du *typage dynamique* d'appliquer la bonne opération élémentaire en fonction du type effectif de l'objet (`Matrix`, `XMatrix`, `XYMatrix`, ...). Ce mécanisme sera très utile pour le passage d'une matrice en argument d'une fonction par *référence* ou par *pointeur*.

Quelles sont les opérations élémentaires de la classe `Matrix` sur les colonnes qu'il faut redéfinir ? Ce sont celles qui permettent :

- d'effectuer une opération élémentaire sur place sur les colonnes de la matrice. En effet, en plus de l'opération $C_i = C_i + \lambda * C_j$, il faut également effectuer l'opération "inverse" $P_j = P_j - \lambda * P_i$ sur les polynômes des colonnes i et j ;
- d'échanger deux colonnes de la matrice (il faut également swaper les deux polynômes) ;
- de supprimer les premières ou les dernières colonnes de la matrice (il faut également supprimer les polynômes correspondants).

3.1.9 La classe XYMatrix

Introduction

On désire maintenant construire des matrices dont les colonnes et les lignes sont indexées par des polynômes. C'est le but de la classe `XYMatrix`.

Implémentation

Pour obtenir la classe `XYMatrix`, il suffit de dériver la classe `XMatrix` (*dérivation publique*), et de lui ajouter un membre de données `BaseRow` de type `Polybase`. C'est ce dernier qui sera responsable de la gestion des polynômes qui vont indexer les lignes de la matrice. Ce nouveau champ aura exactement le même rôle que le champ `BaseCol` de classe `XMatrix` mais cette fois-ci pour les lignes.
La classe `XMatrix` hérite donc de tous les champs de données de la classe `Matrix`, du champ `BaseCol` de la classe `XMatrix` et ajoute le membre de données :

	type	*Identificateur*
	`Polybase`	`BaseRow`

Fonctions, opérateurs et opérations licites exportés par la classe `XYMatrix`

La classe `XYMatrix` hérite de toutes les propriétés de la classe `XMatrix` et de la classe `Matrix`. Nous ne détaillerons pas cette classe car en fait tous les constructeurs, les méthodes les opérateurs et les opérations élémentaires de la classe `XYMatrix` sont exactement les symétriques de ceux exportés par la classe `XMatrix` mais cette fois-ci pour gérer les polynômes indexant les lignes de la matrice.

Chapitre 4

Implémentations

4.1 Matrice de Sylvester

L'implémentation de la construction de Sylvester utilise des structures de données assez simples, structures déjà présentes dans la librairie que je n'ai pas eu à implémenter moi-même, contrairement aux autres constructions.

Les coefficients des deux polynômes sont récupérés dans de simples tableaux de types *Type*. Je rappelle que le terme *Type* désigne le type des coefficients utilisé dans le programme et qui est fixé à la compilation par l'instruction de préprocession `#define`. Il peut être du type *double* ou *float*; le type *int* pose problème à cause des fonctions **LAPACK** qui sont utilsées et qui attendent des arguments des deux types précédents. Cette contrainte pourrait toutefois disparaître avec l'utilisation des *Template*s dans une version révisée de la librairie.

Grâce à ce stockage en tableau des coefficients des deux polynômes, la matrice est construite dans un tableau unidimensionnel en mettant les colonnes les unes après les autres. Ensuite ce tableau est passé au contructeur de la classe `SquareMatrix`, qui nous donne la matrice de Sylvester. Le résultant est ensuite calculé simplement par la méthode `SquareMatrix::det()`

4.2 Matrice de Bézout-Dixon

Pour deux polynômes $P(x)$ et $Q(x)$ de degrés m et $n \leq m$, on calcule les coefficients des n premiers monômes en y de l'expression:

$$\Delta(x, y) = \frac{Q(x)P(y) - Q(y)P(x)}{x - y} = \sum_{i=0}^{m-1} \Delta_i(x) y^i \quad (1)$$

Chaque $\Delta_i(x)$ est alors un polynôme en x de degré $m-1$. On complète cette liste par les multiples de $x^i Q(x)$ puis on décompose ces m polynômes dans la base $\{1, x, x^2, x^3, .., x^{m-1}\}$. On obtient ainsi les m^2 entrées de la matrice de Bézout-Dixon.

4.2.1 Méthode pratique de calcul

Lorsque j'ai commencé ce travail, la librairie ALP, ne permettait pas d'effectuer certaines opérations nécessaires aux constructions qui nous intéressent. C'est notamment le cas de la division de deux polynômes qu'il a fallut implémenter plus tard à l'aide d'un algorithme semblable à l'algorithme classique d'Euclide mais qui basé sur la division des termes dominants des polynômes (*Leading Term*) que je décris dans la section suivante.

Un autre problème dans cette construction est le fait que nous devions considérer dans une même expression des polynômes dont les coefficients sont eux-mêmes de type `Polynome`; et donc de voir manipuler dans une même session des polynômes de types de coefficients différents. Bien que le C++ offre la possibilité de définir de tels objets (paramétrage par les *Templates*) la version actuelle de notre librairie n'offre pas cette possibilité. Le type des coefficients des polynômes(et celui des matrices) est fixé à la compilation grâce à la macro-définition *#define Type myType* où myType désigne le type de coefficients souhaité qui ne peut être d'ailleurs qu'un des types élémentaires(*doubles, float,...*).
Ces problèmes m'ont conduit à réfléchir sur la possibilité de construire la matrice de BD avec les moyens dont nous disposions et de repousser à plus tard les modifications nécessaires à ALP et que ma faible connaissance de la librairie ne permettait de faire à cette époque. La démarche proposée est donc la suivante.

Une définition de P(x) et Q(x) adaptée à ALP

On déclare les polynômes $P(x)$ et $Q(x)$ non pas comme des polynômes en une variable mais en deux variables, disons x_1 et x_2 (comme suggéré par la structure des monômes dans ALP) et dont une est affectée d'un exposant nul. Ainsi, le polynôme $P(x) = 2x^2 + x - 1$ sera représenté par l'objet

$$2x_1^2x_2^0 + x_1x_2^0 - x_1^0x_2^0.$$

De même $Q(y) = y^3 - 2y^2 + 4$ sera représenté par le polynôme à deux variables

$$x_1^0x_2^3 - 2x_1^0x_2^2 + 4x_1^0x_2^0.$$

Avec ce mode représentation, il sera tout à fait possible de calculer l'expression $Q(x)P(y) - Q(y)P(x) = Q(x_1)P(x_2) - Q(x_2)P(x_1)$ qui est un objet de type Polynome de *taille* 2 parfaitement défini. Le terme $P(x_1)Q(x_2)$ par exemple, sera calculé grâce à l'opérateur $'*'$ surchargé qui retournera le polynôme :

$$2x_1^2x_2^3 - 4x_1^2x_2^2 + 8x_1^2x_2^0 + x_1x_2^3 - 2x_1x_2^2 + 4x_1x_2^0 - x_1^0x_2^3 + 2x_1^0x_2^2 - 4x_1^0x_2^0$$

Cette dernière expression pouvant être vue comme un polynôme en x_2, soit

$$(2x_1^2 + x_1 - 1)x_2^3 + (-4x_1^2 - 2x_1 + 2)x_2^2 + (8x_1^2 + 4x_1 - 4)x_2^0$$

avec des coefficients polynômiaux que l'on pourra reconstituer grâce à l'itérateur PolyIter.

Un raccourci pour calculer les $\Delta_i(x)$

La représentation ci-dessus des polynômes P et Q nous permet également d'envisager une méthode de calcul des coefficients Δ_i sans butter contre le délicat problème de la division des deux objets de type Polynome, qu'il faudra de toute évidence traiter un jour. Cette méthode consiste à exprimer les coefficients du polynôme $(x-y)\Delta(x,y)$ comme polynôme en y et d'en déduire ceux de $\Delta(x,y)$ par un procédé admissible sous ALP comme nous le verrons plus loin.

En multipliant la relation (1) par $x-y$ on tire :

$$\begin{aligned}
Q(x)P(y)-Q(y)P(x) &= (x-y)\sum_{i=0}^{m-1}\Delta_i(x)y^i \\
&= \sum_{i=0}^{m-1}x\Delta_i(x)y^i - \sum_{i=1}^{m}\Delta_{i-1}(x)y^i \\
&= x\Delta_0(x)y^0 + \sum_{i=1}^{m-1}[x\Delta_i(x)-\Delta_{i-1}(x)]y^i - \Delta_{m-1}(x)y^m \\
&= \sum_{i=0}^{m}\Delta'_i(x)y^i
\end{aligned}$$

avec

$$\left\{\begin{array}{lcll}
\Delta'_0(x) &=& x\Delta_0(x) & \\
\vdots & & & \\
\Delta'_i(x) &=& x\Delta_i(x)-\Delta_{i-1}(x) & i=1..m-1 \\
\vdots & & & \\
\Delta'_m(x) &=& -\Delta_{m-1}(x) &
\end{array}\right. \qquad \text{et}$$

En partant de la dernière équation et en remontant on peut écrire :

$$\left\{\begin{array}{lcl}
\Delta_{m-1}(x) &=& -\Delta'_m(x) \\
\Delta_{m-2}(x) &=& -x\Delta'_m(x)-\Delta'_{m-1}(x) \\
\vdots & & \\
\Delta_0(x) &=& -x^{m-1}\Delta'_m(x)-x^{m-2}\Delta'_{m-1}(x)-\ldots-x\Delta'_2(x)-\Delta'_1(x)
\end{array}\right.$$

Ainsi on obtient une formule permettant de calculer les coefficients de $\Delta(x,y)$ sans effectuer la moindre division de polynômes, chose qui nous est pour le moment impossible comme nous l'avons noté plus haut.

Cette formule peut toutefois s'avérer coûteuse car après avoir calculé les $\Delta'_i(x)$, il nous faudra effectuer, pour le calcul de chaque $\Delta_i(x)$, plusieurs additions et multiplications. De telles opérations se font facilement mais se traduisent à chaque fois par un appel à une routine prévue à cet effet. Pour des polynômes P et Q de faibles degrés ceci n'est pas trop gênant, mais dès que m devient grand la multiplication des appels de procédures peut être source d'inefficacité. Et pire, le calcul de $\Delta_i(x)$ nécessite le stockage préalable de $m-i$ autres polynômes.

Pour éviter ce problème on utilise les mêmes relations que précédemment mais dans le sens inverse, i.e en partant de légalité $\Delta'_0(x)=x\Delta_0(x)$ et en substituant dans l'ordre croissant des relations. On obtient alors les formules suivantes :

$$\begin{cases} \Delta_0(x) = \dfrac{\Delta_0'(x)}{x} \\ \Delta_1(x) = \dfrac{\Delta_0(x) + \Delta_1'(x)}{x} \\ \vdots \\ \Delta_{m-1}(x) = \dfrac{\Delta_{m-2}(x) + \Delta_{m-1}'(x)}{x} \end{cases}$$

Exprimé de cette façon, le calcul de chaque $\Delta_i(x)$ se fait avec au plus deux appels de procédure (une addition et une division par x) avec l'avantage supplémentaire de n'exiger le stockage que du seul $\Delta_{i'}(x)$ qui pourra être désalloué avant le passage à l'étape $i+1$.

Enfin, la division d'un polynôme par une variable peut se faire simplement en altérant son exposant dans tous les monômes qui constituent ce polynôme. Ceci est réalisé sous ALP grâce à la méthode SetExpo() qui permet de changer l'exposant d'une variable dans un monôme.

Cette méthode a aussi l'avantage d'accélérer le calcul de façon appréciable dans certains cas.

4.3 Bézoutien en plusieurs variables

Cette partie a été la plus importante du travail que j'ai réalisé. Elle a été celle qui m'a posé le plus de difficultés et a impliqué de multiples interventions dans la structure interne de la librairie `ALP`, soit pour des extensions nécessaires aux calculs, soit pour des modifications ou autres corrections sur certaines parties comportant des *bugs* qui conduisaient à des résultats inexacts.

Dans la construction de *Macaulay* faite dans la section précédente, on a pu constaté la taille relativement importante de la matrice obtenue dans l'un des exemples que nous avons considérés par rapport aux données initiales. Avec trois polynômes en trois variables on tombe sur une matrice de taille 10. Ceci peut être inquiétant car, rappelons-le, le but du jeu est de se ramener à des problèmes linéaires que l'on cherchera ensuite à résoudre avec les outils classiques d'algèbre linéaire. Or ces outils, pour efficaces qu'ils soient, n'en demeurent pas moins limités et l'on peut se douter que pour des systèmes polynômiaux d'un certain ordre la construction de *Macaulay* peut s'avérer inexploitable. L'intérêt de la construction de *Bézout* est justement de donner des matrices de petites tailles. Sa mise en oeuvre exige, néanmoins, beaucoup plus d'effort, notamment dans l'organisation des données et des calculs, comme nous le verrons.

L'implémentation de la construction de *Bézout* pour les systèmes en plusieurs variables s'est faite en 3 principales étapes au cours desquelles j'ai travaillé directement sur la structure interne d'`ALP`. J'ai d'abord implémenté un algorithme efficace de division de deux polynômes, qui a ensuite été intégrée dans la classe `Polynome` pour surcharger l'opérateur de division "/" qui permet d'écrire naturellement, dans un programme utilisant la librairie, une expression du genre $R = P/Q$; où P, Q et R sont des polynômes en plusieurs variables, pour affecter à R le résultat de la division de P par Q.

La deuxième étape a consisté à construire la matrice à coefficients polynômiaux dont le déterminant entre dans le calcul du Bézoutien. Ensuite une implémentation de l'algorithme de calcul du déterminant a été effectué. Cet algorithme dont j'ignorais l'existence, est une variante de la méthode de Gauss qui a la particularité d'avoir la même complexité que cette dernière (en $O(n^3)$) et qui est effectif dans le cas de matrice quelconque, contrairement au procédé de Gauss qui exige, pour être cohérent, que les coefficients soient dans un corps où la division est une opération interne. Ce point sera précisé plus loin.

La troisième étape à consisté à construire la matrice du Résultant proprement dite, ainsi que le calcul du résultant dans le cas où la matrice obtenue est carrée.

Enfin, dans la dernière étape ces constructions ont été intégrées à la classe `Polybase` comme méthodes applicables à un objet de type *ensemble de polynômes*. Cette opération a encore une fois impliqué des modifications dans la structure de cette classe pour être réalisable.

4.3.1 Division de polynômes en plusieurs variables

Monômes ordonnés et *Leading term*

On cherche à effectuer la division de deux polynômes en plusieurs variables, soit $f(x_1, .., x_n)$ et $g(x_1, .., x_n)$. Autrement dit, on cherche deux polynômes a et $r \in I\!K[x_1, .., x_n]$ tels que $f = ag + r$ avec $degr(r) < degr(g)$. En examinant en détail l'algorithme classique de division des polynômes en une variable, on constate que l'ordre des termes est un facteur important dans la procédure. Pour illustrer cela, considérons l'exemple suivant:soit à diviser les polynômes $f(x) = x^5 - 3x^2 + 1$ et $g(x) = x^2 - 4x + 7$. La méthode standard consiste à :

- écrire les termes du polynôme dans l'ordre décroissant
- à la première étape, le terme du plus haut degré dans f (le *terme dominant*) étant $x^5 = x^3.x^2 = x^2$x(terme dominant dans g), on soustrait $x^3g(x)$ de f afin d'en éliminer le terme dominant garder $x^4 - 7x^3 - 3x^2 + 1$.
- Ce processus est ensuite appliqué à $f(x) - x^3g(x)$ et ainsi de suite jusqu'à obtenir un polynôme de degré inférieur à 2.

Ainsi pour diviser deux polynômes en une variable, on tient compte de l'ordre des degrés des monômes en une variable:

$$\ldots > x^{m+1} > x^m > \ldots > x^2 > x > 1$$

On obtiendrait le même résultat en choisissant l'ordre inverse des monômes. Le terme dominant de f serait donc le monôme unité. Dans les deux cas, la bonne conduite de cette opération est cependant conditionnée par le fait de ne considérer que les termes dominants dans f et g et d'éviter d'éliminer des termes de f à l'aide de termes quelconques de g.

On procède de façon analogue pour diviser des polynômes en plusieurs variables. Il nous faut cependant préciser la notion de *terme dominant* pour un polynôme en plusieurs variables car il est évident que le caractère dominant d'un monôme dans un polynôme en plusieurs variables ne saurait se résumer à son seul degré étant donné que plusieurs monômes différents peuvent avoir le même degré. Le terme dominant dépend donc d'un certain ordre qu'il est possible de définir arbitrairement à partir de règles qu'il faut alors rigoureusement observer dans la manipulation des polynôme (stockage, tri de monômes,etc.) afin d'en maintenir la cohérence. Il s'agit donc de définir une relation d'ordre sur les monômes

Notons qu'il est possible d'identifier le monôme $x^\alpha = x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$ au *n-uplet* $(\alpha_1, \alpha_2, ..., \alpha_n) \in N^n$. Ceci indique un isomorphisme de $I\!K[x_1, .., x_n]$ dans N^n. Ainsi d'un ordre quelconque défini sur l'ensemble des *n-uplets* d'entiers naturels on peut déduire un ordre sur les monômes de *taille n*. Si l'on note $>$ un tel ordre, nous dirons que $\alpha > \beta$ entraîne $x^\alpha > x^\beta$. Il existe plusieurs types d'ordre sur N^n mais ceux qui sont susceptibles de nous intéresser sont ceux qui maintiennent une certaine compatibilité avec les structures algébriques des polynômes. Le plus usuel d'entre eux est l'ordre *lexicographique "rafiné par le degré"* qui a été adopté dans l'implémentation de la classe `Monome` de la librairie *ALP*.

Un ordre sur N^n est dit lexicographique, noté $>_{lex}$, si pour $\alpha = (\alpha_1, \alpha_2, ..., \alpha_n)$ et $\beta = (\beta_1, \beta_2, ..., \beta_n) \in N^n$, $\alpha >_{lex} \beta$ si dans la coordonnée non nulle la plus à gauche du vecteur-difference $\alpha - \beta$ est positif. Et on dira que $x^\alpha >_{lex} x^\beta$ si $\alpha >_{lex} \beta$.

Plus clairement considérons les exemples suivants:

- $(1, 2, 0) >_{lex} (0, 3, 4)$ puisque $\alpha - \beta = (1, -1, -4)$.
- $(3, 2, 4) >_{lex} (3, 2, 1)$ puisque $\alpha - \beta = (0, 0, 3)$

En général lorsqu'on travail sur des polynômes en une, deux ou trois variables on se sert des lettre x, y et z plutôt que de x_1, x_2 et x_3. On admet ainsi que l'on considère l'ordre alphabétique.

L'ordre qui a été défini sur les monômes de la librairie est celui qui consiste, dans la comparaison de deux éléments, à comparer d'abord leurs degrés respectifs, dont l'inégalité entraîne immédiatement une supériorité du monôme ayant le plus haut degré ; ensuite, en cas dégalité, l'ordre lexicographique est considéré.

L'opérateur "$<$" a été surcharger dans la classe `Monome`, comme cela est possible en *C++* ; permet de comparer deux monômes selon ce principe et se comporte de la même façon que pour les types élémentaires. Il retourne, lorsqu'il est appliqué à deux monômes une valeur boolean renseignant sur leur position mutuelle. Ce mécanisme de comparaison des monômes est très important, en particulier dans les constructions de *Bézout* et *Macaulay* comme nous le verrons. Il est utilisé ici pour déterminant le *LeadingTerm* du polynôme qui entre dans l'algorithme de division ci-après.

L'ordre étant ainsi fixé, il nous reste à décider de l'attribution du caractère dominant. Je rappelle que ce choix est arbitraire même après avoir défini l'ordre de rangement, mais il doit néanmoins permettre une représentation cohérente. Dans l'utilisation qui en est faite dans l'algorithme de division qui a été implémenté, le choix, dicté naturellement par des considérations d'efficacité, s'est porté sur le terme le plus petit. En effet, un polynôme étant représenté par une liste chaînée de monômes contenus dans des cellules, rangées dans l'ordre croissant des monômes, le parcours nécessaire pour accéder à un élément de la liste étant séquentiel, choisir comme terme dominant le monôme le plus le plus grand (celui se trouvant à la fin de liste) suppose toujours un parcours complet de la liste. L'intérêt du choix que nous avons fait est alors évident puisque le terme dominant sera le premier élément rencontré.

Voyons maintenant comment, grâce à ces principes nous pouvons effectuer une division efficace de polynômes en plusieurs variables. Dans toute la suite le terme dominant d'un polynôme P sera noté $LT(P)$.

L'algorithme que j'ai implémenté est une adaptation de celui donné dans (*D. Cox, J. Little, D. OShea, An Introduction to Computational Algebraic Geometry and Commutative Algebra cf*) et qui y est exposé dans le cas général où l'on désire effectuer la "division" d'un polynôme avec plusieurs autres polynômes dans le théorème suivant :

Théorème 4.3.1 *(Division dans $I\!\!K[x_1, x_2, .., x_n]$)*
Soit fixé un ordre sur les monôme, et soit $f_1, f_2, .., f_s s$ polynômes ordonnés de $I\!\!K[x_1, x_2, .., x_n]$. Alors pour tout $f \in I\!\!K[x_1, x_2, .., x_n]$ il existe $a_1, a_2, .., a_s$ et $r \in I\!\!K[x_1, x_2, .., x_n]$ tels que

$$f = a_1 f_1 + a_2 f_2 + .. + a_s f_s + r \tag{4.1}$$

De plus r est soit nul, soit combinaison k-lineaire de monômes dont aucun n'est divisible par $LT(f_i)$, $i = 1, .., s$

Voici l'algorithme de division, dont il est facile de démontrer qu'il est effectif. Une preuve et une étude détaillée est donné dans l'ouvrage cité.

Algorithme 4.3.2 (Division de polynômes en plusieurs variables)
Entrée: f_1, f_2, ..., f_n, f
Sortie: a_1, a_2, ..., a_n, r

```
a = 0; ...; an = 0; r = 0;
p = f;
WHILE (p != 0)  DO
  i = 1;
  DivisionFaite = false;
```

```
  WHILE ((i <= n)  AND  (DivisionFaite == false)) DO
    IF (LT(fi)  divise  LT(p))   THEN
        ai = ai + LT(p);
        p = p - (LT(p)/LT(fi))fi;
        DivisionFaite = true;
     ELSE
       i = i + 1;
     FI
  OD
  IF (DivisionFaite == false) THEN
      r = r + LT(p);
      p = p - LT(p);
  FI
OD
```

A chaque entrée dans la boucle *While* il se passe une des deux choses suivantes:

- Si le terme dominant de p (la dividende intermédiaire), $LT(p)$ est divisible par le terme dominant de quelque f_i, alors l'algorithme se comporte exactement comme dans le cas d'une variable : on ajoute à a_i $LT(p)/LT(f_i)$ et on retranche de p la quantité $((LT(p)/LT(f_i))f_i)$
- Si, par contre, $LT(P)$ n'est divisible par aucun des $LT(f_i)$, alors $LT(p)$ est ajouté au reste.

L'adaptation de cet algorithme au cas qui nous intéresse, c'est-à-dire la division de seulement deux polynômes est immédiate. Il suffit de prendre $i = 1$ et d'opérer une légère modification dans ce processus afin d'éliminer l'incrémentation de i.
La variable *DivisionFaite* est un booléen qui indique quand le terme dominant d'un f_i divise le terme dominant de la dividende intermédiaire. Ce processus est répété jusqu'à ce la dividende intermédiaire soit nulle.

Le code suivant qui est une adaptation de cet algorithme au cas qui nous intéresse $(i = 1)$ est celui de la fonction `PolyDiv`, implémentée dans le fichier Polynome.C comme *fonction amie* de la classe `Polynome`.

Implémentation

```
Polynome  PolyDiv(const Polynome &P, const Polynome &Q){

  if(!Q.PolyDeb)
    Error("%s %s  PolyDiv \n",ERR_DIV_ZERO, ERR_FUNCTION);
  if(!P.PolyDeb)
    return(0);
    if(Q == (Polynome)1)
    return P;
  Polynome Tmp = P, Quot = 0, Rest = 0;
  Monome LT;
```

```
  while(Tmp.PolyDeb){
    Monome LTP = Monome(Tmp.LeadingTerm());
    Monome LTQ = Monome(Q.LeadingTerm());
     if(MonoDiv(LTP,LTQ, &LT)){
       Quot = Quot + LT;
       Tmp  = Tmp  - (LT*Q);
     }
     else{
       Rest = Rest + LTP;
       Tmp = Tmp - LTP;
     }
  }
  return(Quot);
}
```

La fonction `MonoDiv` qui apparaît dans la boucle est implémentée comme fonction *amie* de la classe `Monome`. Son rôle est de tester la divisibilité de deux monômes et, le cas échéant, les diviser. Sa valeur de retour est un booléen qui renseigne sur cette divisibilité, et le troisième argument est l'adresse d'une variable `Monome` qui recevra le résultat de cette éventuelle division. La fonction `LeadingTerm()`, qui calcule le terme dominant a, quant à elle, été implémentée comme méthode de la classe `Polynome`. Notons enfin que la fonction `PolyDiv` retourne la partie entière du quotient et que dans le cas d'une division non exacte, le reste n'est pas nul. La fonction `PolyMod` donne ce reste en utilisant le même algorithme. Notons cependant, que dans l'utilisation que nous ferons de cette division, ce reste sera toujours nul, comme nous le verrons plus loin, mais son examen a permis de corriger certaines erreurs dans les programmes.

Remarque: Des problèmes numériques, liés à la précision sur les machines où peuvent être exécutés nos programmes, peuvent survenir. Leur étude pourrait être intéressante mais tel n'est pas notre but ici. Notre implémentation s'est toutefois révélée suffisamment stable dans les exemples que nous avons utilisés dans nos tests.

4.3.2 Déterminant d'une matrice polynomiale

Jusque là les calculs de Déterminant que j'ai eu à effectuer avait toujours un cadre numérique, où les coefficients des matrices concernées étaient, dans la plupart des cas, localisés le corps des réels ou celui des nombres complexes. La méthode la plus efficace pour calculer le déterminant d'une matrice carrée dense non structurée, étant la méthode d'élimination de *Gauss* qui consiste à transformer la matrice concernée en une autre matrice triangulaire en réduisant successivement les colonnes de façon à y *faire apparaître des zéros*. Ceci est réalisé au moyen de la formule de réduction suivante:

Si A désigne la matrice initiale et $A^{(k-1)}$ la matrice réduite obtenue à la $k-1^{eme}$

étape de la transformation, on passe à la matrice A^k en posant :

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}, k+1 \leq i,j \leq n \qquad (4.2)$$

où l'élément $a_{kk}^{(k-1)}$ désigne le *Pivot* à l'étape k.
On montre alors que le déterminant de A est égal, au signe près, à celui de la matrice obtenue après la dernière réduction.

Exemple

$$A = A^{(0)} = \begin{bmatrix} 2 & 5 & 3 \\ 4 & 2 & 4 \\ 6 & 7 & 3 \end{bmatrix}$$

$$A^{(1)} = \begin{bmatrix} 2 & 5 & 3 \\ 0 & -8 & -2 \\ 0 & -8 & -6 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 2 & 5 & 3 \\ 0 & -8 & -2 \\ 0 & 0 & 4 \end{bmatrix}$$

ici on a $Det(A) = 64$

Cet algorithme suppose cependant que les divisions conduisent à des résultats qui restent dans l'anneau des coefficients de notre matrice. C'est le cas lorsque l'on a affaire à des matrices réelles où complexes. Ceci est moins vrai lorsque l'anneau des coefficients est quelconque, même si l'on sait, par définition, que le déterminant auquel on cherche à aboutir est un élément de cet anneau. Le problème qui peut se poser, précisément est celui de la division de deux éléments premiers entre eux, n'admettant pas de diviseurs communs non triviaux. Ceci entraîne l'apparition de coefficients qui ne sont pas définis dans l'anneau de la matrice et constitue un obstacle à la poursuite de la réduction. Notre but étant de calculer le déterminant d'une matrice dont les coefficients ne sont plus de simples scalaires mais des polynômes en plusieurs variables, nous sommes d'emblée confrontés à ce problème.

Une idée simple pour éviter l'apparition de fractions pendant la réduction serait de supprimer la division, et de considérer la formule de réduction suivante :

$${a_{ij}}^{(k)} = {a_{ij}}^{(k-1)} {a_{kk}}^{(k-1)} - {a_{ik}}^{(k-1)} {a_{kj}}^{(k-1)}, k+1 \leq i,j \leq n \qquad (4.3)$$

Cette formule a hélas le désavantage de faire exploser très rapidement les coefficients et de rendre son utilisation douteuse, car dans certains cas il faudrait beaucoup de mémoire pour supporter les données intermédiaires. On peut d'ailleurs montrer qu'avec cette formule, si $a_{ij}^{(0)}$ est stocké dans un entier de δ bits alors le stockage

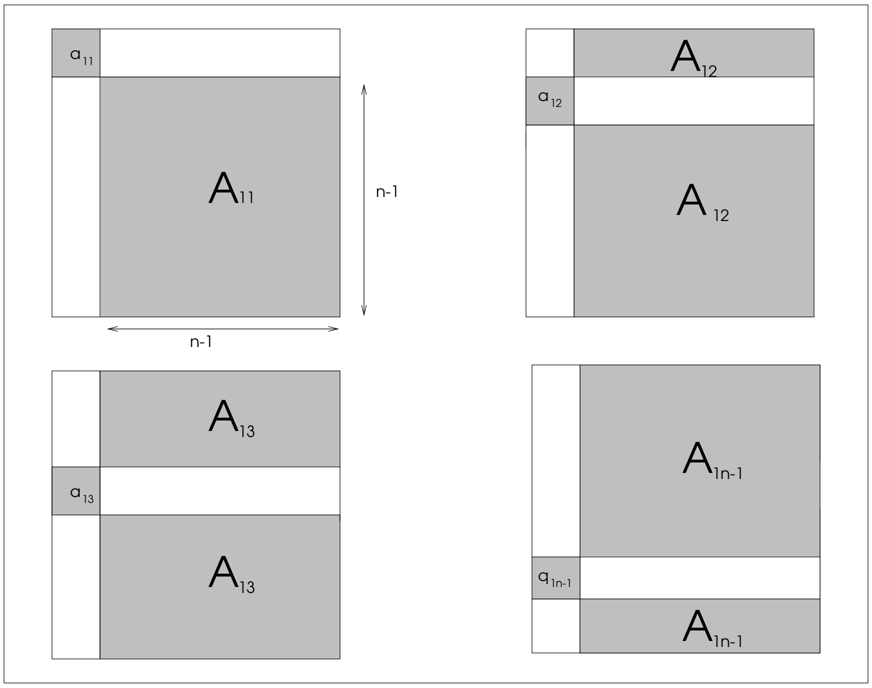


FIG. 4.1 – *Développement selon la première colonne et appel récursif*

de $a_{ij}^{(k)}$ peut nécessiter de l'ordre $2^k\delta$ bits(voir *Gedes, Czapor & Labahn 1992*). On peut également craindre pour le temps d'exécution d'un programme de calcul basé sur cette formule.

Une autre idée, pour calculer un déterminant quelconque sans diviser est de procéder en développant suivant une ligne ou une colonne comme on le fait à la main en utilisant la définition classique, et de faire des appels récursifs selon le schéma indiqué dans la figure 4.1.

En clair, cette démarche consiste à appliquer récursivement la formule suivante :

$$Det(A) = \sum_{k=1}^{n} (-1)^{k+1} a_{1k} Det(A_{1k}) \tag{4.4}$$

où A_{1k} désigne la matrice définie par la partie hachurée qui est pour chaque $i = 1, ..n$ une matrice carrée de taille $n-1$ à laquelle on applique donc cette fonction recursive. Le point d'arrêt de cette récursion est donné par le fait, précisé au départ de l'algorithme, que si l'argument est une matrice $(1, 1)$ alors la fonction retourne l'unique coefficient.

Cette démarche est donc tout à fait applicable à une matrice polynomiale, pourvu de disposer d'un langage de programmation qui supporte la recursion, ce qui est le cas en particulier du C++ que nous utilisons. Malheureusement cette méthode (comme toutes les méthodes qui marchent!) a elle aussi son inconvénient. Elle a une complexité exponentielle ($O(n!)$) alors que la méthode *Gauss* est en $O(n^3)$. Elle n'est donc pas suffisamment intéressant dans notre contexte c'est-à-dire dans le cas de matrice polynomiale à plusieurs variables, denses.

Le contexte est, cependant, important dans le choix à faire, car une méthode aussi mauvaise qu'elle puisse être, peut s'avérer plus viable que les autres dans un contexte particulier. Ainsi, celle que nous venons de décrire, qui est aussi appelé *Minor expansion method(M.E.M)* est plus efficace que toutes les autres lorsque l'on a des matrices polynomiaux à plusieurs variables *complètement creuse*. Dans une analyse comparée des différentes méthode, *Gentelman et Johnson 1976* ont montré que lorsque les coefficients sont des polynomes en une variable comportant t termes alors la M.E.M est supérieure aux autres méthodes sauf si

- n = 6, t >1
- n = 7, t>3
- n >7

Ces cas sont, hélas, très particuliers, et dans l'application qui nous intéresse (calcul de Bézoutien en plusieurs variables) les entrées de la matrice comportent toutes plus d'une variable.

Quant à la fameuse méthode de la *Règle de Sarus*, son utilisation ne nous a, en aucun moment, effleuré l'esprit pour les raisons évidentes que l'on sait.

Le pivot de Bareiss

Nous avons heureusement trouvé une méthode qui résout ces problèmes. Cette méthode, due à *Bareiss*, est en fait une variante de la méthode de *Gauss*, qui part d'un principe très simple : on peut remarquer que lorsque l'on effectue l'élimination de *Gauss*, les entrées de la 3^{me} ligne (et des suivantes) de la matrice $A^{(2)}$ sont divisibles par le pivot $a = a_{11}^{(0)}$(i.e avec un résultat qui reste dans l'anneau des coefficients de la matrice). Il est donc possible de diviser les entrées de la matrice réduite $A^{(2)}$ par a avant de procéder à la réduction. Après une nouvelle itération, on peut également constater que les entrées de la matrice réduite $A^{(3)}$ sont divisibles par $a_{22}^{(1)}$. En fait de telles divisions sont possible tout au long du processus d'élimination. Il s'agit donc de diviser, non pas par le pivot trouvé à l'étape courante, mais par celui de l'étape précédente ; ce qui est fait à la première étape en posant $a_{00}^{-1} = 1$. On applique ainsi la formule de réduction suivante que nous avons implémentée et qui a l'avantage évident d'avoir la même complexité que l'élimination classique par le pivot de Gauss. Cet algorithme est aussi appelé *the Fraction-Free Gaussian Elimination.*

$$a_{00}^{(-1)} = 1, [a_{ij}^{(0)}] = A^{(0)}$$
$$a_{ij}{}^{(k)} = (a_{ij}^{(k-1)} a_{kk}^{(k-1)} - a_{ik}^{(k-1)} a_{kj}^{(k-1)}) / a_{k-1,k-1}^{(k-2)},$$

$$k+1 \leq i \leq n, k+1 \leq j \leq n+1$$

On montre que le déterminant de la matrice est donné par :

$$det(A) = a_{nn}^{(n-1)}$$

Exemple

L'exemple suivant a été testé sur notre implémentation, il concerne des polynômes relativement simples, constitués chacun d'un unique monôme. Le cas de polynômes plus compliqués avec plusieurs termes peut poser un problème sur lequel nous reviendrons plus loin.

$$A = A^{(0)} = \begin{bmatrix} 1 & 2 & -2x_1 \\ 2x_1 & 3x_2 & 0 \\ 4 & 12x_2 & 5x_1x_2 \end{bmatrix}$$

$$A^{(1)} = \begin{bmatrix} 1 & 2 & -2x_1 \\ 0 & 3x_2 - 4x_1 & 4x_1^2 \\ 0 & -8 + 12x_2 & 8x_1 + 5x_1x_2 \end{bmatrix}$$

$$A^{(2)} = \begin{bmatrix} 1 & 2 & -2x_1 \\ 0 & 3x_2 - 4x_1 & 4x_1^2 \\ 0 & 0 & 24x_1x_2 + 15x_1x_2^2 - 68x_1^2x_2 \end{bmatrix}$$

On obtient donc pour cet exemple :

$$\boxed{Det(A) = 24x_1x_2 + 15x_1x_2^2 - 68x_1^2x_2}$$

La simplicté de cet exemple ne cache cependant pas un problème qui pourrait s'avérer plus gênant dans le cas de matrices plus denses avec de plus larges coefficients. On constate en effet que plus on progresse dans la réduction, plus la taille des coefficients augmente. Cette croissance concerne aussi bien le degré des entrées des matrices obtenues que le nombre de termes que comporte chaque entrée. Ceci pose deux problème majeurs. D'une part, l'apparition de termes de plus en plus grands requiert de plus en plus de mémoire pour leur stockage, même si au bout du compte on se retrouve avec un déterminant taille plus raisonnable. L'exemple suivant en est une illustration significative. Il s'agit d'un des termes intermédiaires obtenus dans le calcul du déterminant de la matrice :

$$A = \begin{bmatrix} f_1 & f_2 & f_3 & f_4 \\ f_5 & f_6 & f_7 & f_8 \\ f_9 & f_{10} & f_{11} & f_{12} \\ f_{13} & f_{14} & f_{15} & f_{16} \end{bmatrix}$$

avec

$$
\begin{aligned}
f_1 &= x_3+x_2+x_1\\
f_2 &= x_3+x_2+x_4\\
f_3 &= x_3+x_5+x_4\\
f_4 &= x_6+x_5+x_4\\
f_5 &= x_2x_3+x_1x_3+x_1x_2\\
f_6 &= x_2x_3+x_3x_4+x_2x_4\\
f_7 &= x_3x_5+x_3x_4+x_4x_5\\
f_8 &= x_5x_6+x_4x_6+x_4x_5\\
f_9 &= -1+x_1x_2x_3\\
f_{10} &= -1+x_2x_3x_4\\
f_{11} &= -1+x_3x_4x_5\\
f_{12} &= -1+x_4x_5x_6\\
f_{13} &= 1\\
f_{14} &= 1\\
f_{15} &= 1\\
f_{16} &= 1
\end{aligned}
$$

A l'étape 3 de la réduction et avant division par le pivot, on obtient pour $A_{44}^{(3)}$ le terme suivant :

```
(1x4^2.x5^4.x6^2.x7^1.) + (-1x4^3.x5^3.x6^2.x7^1.) + (1x4^3.x5^4.x6^1.x7^1.) +
(-1x4^4.x5^2.x6^2.x7^1.) + (-1x4^4.x5^3.x6^1.x7^1.) +
(-1x3^1.x4^2.x5^4.x6^2.) + (1x3^1.x4^3.x5^2.x6^2.x7^1.) +
(1x3^1.x4^3.x5^3.x6^2.) + (-1x3^1.x4^3.x5^4.x6^1.) +
(1x3^1.x4^4.x5^2.x6^1.x7^1.) + (1x3^1.x4^4.x5^2.x6^2.) +
(1x3^1.x4^4.x5^3.x6^1.) + (-1x3^2.x4^3.x5^2.x6^1.x7^1.) +
(-1x3^2.x4^3.x5^2.x6^2.) + (-1x3^2.x4^4.x5^2.x6^1.) +
(1x3^3.x4^3.x5^2.x6^1.) + (2x2^1.x4^1.x5^4.x6^2.x7^1.) +
(-3x2^1.x4^2.x5^3.x6^2.x7^1.) + (1x2^1.x4^2.x5^4.x6^1.x7^1.) +
(-2x2^1.x4^3.x5^2.x6^2.x7^1.) + (-2x2^1.x4^3.x5^3.x6^1.x7^1.) +
(-1x2^1.x4^3.x5^4.x7^1.) + (2x2^1.x4^4.x5^2.x6^1.x7^1.) +
(1x2^1.x4^4.x5^3.x7^1.) + (-2x2^1.x3^1.x4^1.x5^4.x6^2.) +
(3x2^1.x3^1.x4^2.x5^2.x6^2.x7^1.) + (3x2^1.x3^1.x4^2.x5^3.x6^2.) +
(-1x2^1.x3^1.x4^2.x5^4.x6^1.) + (2x2^1.x3^1.x4^3.x5^2.x6^1.x7^1.) +
(2x2^1.x3^1.x4^3.x5^2.x6^2.) + (2x2^1.x3^1.x4^3.x5^3.x6^1.) +
(1x2^1.x3^1.x4^3.x5^4.) + (-1x2^1.x3^1.x4^4.x5^2.x7^1.) +
(-2x2^1.x3^1.x4^4.x5^2.x6^1.) + (-1x2^1.x3^1.x4^4.x5^3.) +
(-3x2^1.x3^2.x4^2.x5^2.x6^1.x7^1.) + (-3x2^1.x3^2.x4^2.x5^2.x6^2.) +
(1x2^1.x3^2.x4^3.x5^2.x7^1.) + (-2x2^1.x3^2.x4^3.x5^2.x6^1.) +
(1x2^1.x3^2.x4^4.x5^2.) + (3x2^1.x3^3.x4^2.x5^2.x6^1.) +
(-1x2^1.x3^3.x4^3.x5^2.) + (1x2^2.x5^4.x6^2.x7^1.) +
(-3x2^2.x4^1.x5^3.x6^2.x7^1.) + (-1x2^2.x4^1.x5^4.x6^1.x7^1.) +
```

```
(-1x2^2.x4^2.x5^2.x6^2.x7^1.) + (-2x2^2.x4^2.x5^4.x7^1.) +
(4x2^2.x4^3.x5^2.x6^1.x7^1.) + (3x2^2.x4^3.x5^3.x7^1.) +
(-1x2^2.x4^4.x5^2.x7^1.) + (-1x2^2.x3^1.x5^4.x6^2.) +
(3x2^2.x3^1.x4^1.x5^2.x6^2.x7^1.) + (3x2^2.x3^1.x4^1.x5^3.x6^2.) +
(1x2^2.x3^1.x4^1.x5^4.x6^1.) + (1x2^2.x3^1.x4^2.x5^2.x6^2.) +
(2x2^2.x3^1.x4^2.x5^4.) + (-3x2^2.x3^1.x4^3.x5^2.x7^1.) +
(-4x2^2.x3^1.x4^3.x5^2.x6^1.) + (-3x2^2.x3^1.x4^3.x5^3.) +
(1x2^2.x3^1.x4^4.x5^2.) + (-3x2^2.x3^2.x4^1.x5^2.x6^1.x7^1.) +
(-3x2^2.x3^2.x4^1.x5^2.x6^2.) + (3x2^2.x3^2.x4^2.x5^2.x7^1.) +
(3x2^2.x3^2.x4^3.x5^2.) + (3x2^2.x3^3.x4^1.x5^2.x6^1.) +
(-3x2^2.x3^3.x4^2.x5^2.) + (-1x2^3.x5^3.x6^2.x7^1.) +
(-1x2^3.x5^4.x6^1.x7^1.) + (2x2^3.x4^1.x5^3.x6^1.x7^1.) +
(-1x2^3.x4^1.x5^4.x7^1.) + (2x2^3.x4^2.x5^2.x6^1.x7^1.) +
(3x2^3.x4^2.x5^3.x7^1.) + (-2x2^3.x4^3.x5^2.x7^1.) +
(1x2^3.x3^1.x5^2.x6^2.x7^1.) + (1x2^3.x3^1.x5^3.x6^2.) +
(1x2^3.x3^1.x5^4.x6^1.) + (-2x2^3.x3^1.x4^1.x5^2.x6^1.x7^1.) +
(-2x2^3.x3^1.x4^1.x5^3.x6^1.) + (1x2^3.x3^1.x4^1.x5^4.) +
(-3x2^3.x3^1.x4^2.x5^2.x7^1.) + (-2x2^3.x3^1.x4^2.x5^2.x6^1.) +
(-3x2^3.x3^1.x4^2.x5^3.) + (2x2^3.x3^1.x4^3.x5^2.) +
(-1x2^3.x3^2.x5^2.x6^1.x7^1.) + (-1x2^3.x3^2.x5^2.x6^2.) +
(3x2^3.x3^2.x4^1.x5^2.x7^1.) + (2x2^3.x3^2.x4^1.x5^2.x6^1.) +
(3x2^3.x3^2.x4^2.x5^2.) + (1x2^3.x3^3.x5^2.x6^1.) +
(-3x2^3.x3^3.x4^1.x5^2.) + (1x2^4.x5^3.x6^1.x7^1.) +
(1x2^4.x4^1.x5^3.x7^1.) + (-1x2^4.x4^2.x5^2.x7^1.) +
(-1x2^4.x3^1.x5^2.x6^1.x7^1.) + (-1x2^4.x3^1.x5^3.x6^1.) +
(-1x2^4.x3^1.x4^1.x5^2.x7^1.) + (-1x2^4.x3^1.x4^1.x5^3.) +
(1x2^4.x3^1.x4^2.x5^2.) + (1x2^4.x3^2.x5^2.x7^1.) +
(1x2^4.x3^2.x5^2.x6^1.) + (1x2^4.x3^2.x4^1.x5^2.) +
(-1x2^4.x3^3.x5^2.) + (-2x1^1.x4^2.x5^3.x6^2.x7^1.) +
(2x1^1.x4^3.x5^2.x6^2.x7^1.) + (-2x1^1.x4^3.x5^3.x6^1.x7^1.) +
(2x1^1.x4^4.x5^1.x6^2.x7^1.) + (2x1^1.x4^4.x5^2.x6^1.x7^1.) +
(2x1^1.x3^1.x4^2.x5^3.x6^2.) + (-2x1^1.x3^1.x4^3.x5^1.x6^2.x7^1.) +
(-2x1^1.x3^1.x4^3.x5^2.x6^2.) + (2x1^1.x3^1.x4^3.x5^3.x6^1.) +
(-2x1^1.x3^1.x4^4.x5^1.x6^1.x7^1.) + (-2x1^1.x3^1.x4^4.x5^1.x6^2.) +
(-2x1^1.x3^1.x4^4.x5^2.x6^1.) + (2x1^1.x3^2.x4^3.x5^1.x6^1.x7^1.) +
(2x1^1.x3^2.x4^3.x5^1.x6^2.) + (2x1^1.x3^2.x4^4.x5^1.x6^1.) +
(-2x1^1.x3^3.x4^3.x5^1.x6^1.) + (-4x1^1.x2^1.x4^1.x5^3.x6^2.x7^1.) +
(6x1^1.x2^1.x4^2.x5^2.x6^2.x7^1.) + (-2x1^1.x2^1.x4^2.x5^3.x6^1.x7^1.) +
(4x1^1.x2^1.x4^3.x5^1.x6^2.x7^1.) + (4x1^1.x2^1.x4^3.x5^2.x6^1.x7^1.) +
(2x1^1.x2^1.x4^3.x5^3.x7^1.) + (-4x1^1.x2^1.x4^4.x5^1.x6^1.x7^1.) +
(-2x1^1.x2^1.x4^4.x5^2.x7^1.) + (4x1^1.x2^1.x3^1.x4^1.x5^3.x6^2.) +
(-6x1^1.x2^1.x3^1.x4^2.x5^1.x6^2.x7^1.) + (-6x1^1.x2^1.x3^1.x4^2.x5^2.x6^2.)+
(2x1^1.x2^1.x3^1.x4^2.x5^3.x6^1.) + (-4x1^1.x2^1.x3^1.x4^3.x5^1.x6^1.x7^1.)+

(-4x1^1.x2^1.x3^1.x4^3.x5^1.x6^2.) + (-4x1^1.x2^1.x3^1.x4^3.x5^2.x6^1.) +
(-2x1^1.x2^1.x3^1.x4^3.x5^3.) + (2x1^1.x2^1.x3^1.x4^4.x5^1.x7^1.) +
(4x1^1.x2^1.x3^1.x4^4.x5^1.x6^1.) + (2x1^1.x2^1.x3^1.x4^4.x5^2.) +
```

```
(6x1^1.x2^1.x3^2.x4^2.x5^1.x6^1.x7^1.) + (6x1^1.x2^1.x3^2.x4^2.x5^1.x6^2.) +
(-2x1^1.x2^1.x3^2.x4^3.x5^1.x7^1.) + (4x1^1.x2^1.x3^2.x4^3.x5^1.x6^1.) +
(-2x1^1.x2^1.x3^2.x4^4.x5^1.) + (-6x1^1.x2^1.x3^3.x4^2.x5^1.x6^1.) +
(2x1^1.x2^1.x3^3.x4^3.x5^1.) + (-2x1^1.x2^2.x5^3.x6^2.x7^1.) +
(6x1^1.x2^2.x4^1.x5^2.x6^2.x7^1.) + (2x1^1.x2^2.x4^1.x5^3.x6^1.x7^1.) +
(2x1^1.x2^2.x4^2.x5^1.x6^2.x7^1.) + (4x1^1.x2^2.x4^2.x5^3.x7^1.) +
(-8x1^1.x2^2.x4^3.x5^1.x6^1.x7^1.) + (-6x1^1.x2^2.x4^3.x5^2.x7^1.) +
(2x1^1.x2^2.x4^4.x5^1.x7^1.) + (2x1^1.x2^2.x3^1.x5^3.x6^2.) +
(-6x1^1.x2^2.x3^1.x4^1.x5^1.x6^2.x7^1.) +(-6x1^1.x2^2.x3^1.x4^1.x5^2.x6^2.) +
(-2x1^1.x2^2.x3^1.x4^1.x5^3.x6^1.) + (-2x1^1.x2^2.x3^1.x4^2.x5^1.x6^2.) +
(-4x1^1.x2^2.x3^1.x4^2.x5^3.) + (6x1^1.x2^2.x3^1.x4^3.x5^1.x7^1.) +
(8x1^1.x2^2.x3^1.x4^3.x5^1.x6^1.) + (6x1^1.x2^2.x3^1.x4^3.x5^2.) +
(-2x1^1.x2^2.x3^1.x4^4.x5^1.) + (6x1^1.x2^2.x3^2.x4^1.x5^1.x6^1.x7^1.) +
(6x1^1.x2^2.x3^2.x4^1.x5^1.x6^2.) + (-6x1^1.x2^2.x3^2.x4^2.x5^1.x7^1.) +
(-6x1^1.x2^2.x3^2.x4^3.x5^1.) + (-6x1^1.x2^2.x3^3.x4^1.x5^1.x6^1.) +
(6x1^1.x2^2.x3^3.x4^2.x5^1.) + (2x1^1.x2^3.x5^2.x6^2.x7^1.) +
(2x1^1.x2^3.x5^3.x6^1.x7^1.) + (-4x1^1.x2^3.x4^1.x5^2.x6^1.x7^1.) +
(2x1^1.x2^3.x4^1.x5^3.x7^1.) + (-4x1^1.x2^3.x4^2.x5^1.x6^1.x7^1.) +
(-6x1^1.x2^3.x4^2.x5^2.x7^1.) + (4x1^1.x2^3.x4^3.x5^1.x7^1.) +
(-2x1^1.x2^3.x3^1.x5^1.x6^2.x7^1.) + (-2x1^1.x2^3.x3^1.x5^2.x6^2.) +
(-2x1^1.x2^3.x3^1.x5^3.x6^1.) + (4x1^1.x2^3.x3^1.x4^1.x5^1.x6^1.x7^1.) +
(4x1^1.x2^3.x3^1.x4^1.x5^2.x6^1.) + (-2x1^1.x2^3.x3^1.x4^1.x5^3.) +
(6x1^1.x2^3.x3^1.x4^2.x5^1.x7^1.) + (4x1^1.x2^3.x3^1.x4^2.x5^1.x6^1.) +
(6x1^1.x2^3.x3^1.x4^2.x5^2.) + (-4x1^1.x2^3.x3^1.x4^3.x5^1.) +
(2x1^1.x2^3.x3^2.x5^1.x6^1.x7^1.) + (2x1^1.x2^3.x3^2.x5^1.x6^2.) +
(-6x1^1.x2^3.x3^2.x4^1.x5^1.x7^1.) + (-4x1^1.x2^3.x3^2.x4^1.x5^1.x6^1.) +
(-6x1^1.x2^3.x3^2.x4^2.x5^1.) + (-2x1^1.x2^3.x3^3.x5^1.x6^1.) +
(6x1^1.x2^3.x3^3.x4^1.x5^1.) + (-2x1^1.x2^4.x5^2.x6^1.x7^1.) +
(-2x1^1.x2^4.x4^1.x5^2.x7^1.) + (2x1^1.x2^4.x4^2.x5^1.x7^1.) +
(2x1^1.x2^4.x3^1.x5^1.x6^1.x7^1.) + (2x1^1.x2^4.x3^1.x5^2.x6^1.) +
(2x1^1.x2^4.x3^1.x4^1.x5^1.x7^1.) + (2x1^1.x2^4.x3^1.x4^1.x5^2.) +
(-2x1^1.x2^4.x3^1.x4^2.x5^1.) + (-2x1^1.x2^4.x3^2.x5^1.x7^1.) +
(-2x1^1.x2^4.x3^2.x5^1.x6^1.) + (-2x1^1.x2^4.x3^2.x4^1.x5^1.) +
(2x1^1.x2^4.x3^3.x5^1.) + (1x1^2.x4^2.x5^2.x6^2.x7^1.) +
(-1x1^2.x4^3.x5^1.x6^2.x7^1.) + (1x1^2.x4^3.x5^2.x6^1.x7^1.) +
(-1x1^2.x4^4.x6^2.x7^1.) + (-1x1^2.x4^4.x5^1.x6^1.x7^1.) +
(-1x1^2.x3^1.x4^2.x5^2.x6^2.) + (1x1^2.x3^1.x4^3.x6^2.x7^1.) +
(1x1^2.x3^1.x4^3.x5^1.x6^2.) + (-1x1^2.x3^1.x4^3.x5^2.x6^1.) +
(1x1^2.x3^1.x4^4.x6^1.x7^1.) + (1x1^2.x3^1.x4^4.x6^2.) +
(1x1^2.x3^1.x4^4.x5^1.x6^1.) + (-1x1^2.x3^2.x4^3.x6^1.x7^1.) +
(-1x1^2.x3^2.x4^3.x6^2.) + (-1x1^2.x3^2.x4^4.x6^1.) +
(1x1^2.x3^3.x4^3.x6^1.) + (2x1^2.x2^1.x4^1.x5^2.x6^2.x7^1.) +
(-3x1^2.x2^1.x4^2.x5^1.x6^2.x7^1.) + (1x1^2.x2^1.x4^2.x5^2.x6^1.x7^1.) +
(-2x1^2.x2^1.x4^3.x6^2.x7^1.) + (-2x1^2.x2^1.x4^3.x5^1.x6^1.x7^1.) +
(-1x1^2.x2^1.x4^3.x5^2.x7^1.) + (2x1^2.x2^1.x4^4.x6^1.x7^1.) +
(1x1^2.x2^1.x4^4.x5^1.x7^1.) + (-2x1^2.x2^1.x3^1.x4^1.x5^2.x6^2.) +
(3x1^2.x2^1.x3^1.x4^2.x6^2.x7^1.) + (3x1^2.x2^1.x3^1.x4^2.x5^1.x6^2.) +
```

```
(-1x1^2.x2^1.x3^1.x4^2.x5^2.x6^1.) + (2x1^2.x2^1.x3^1.x4^3.x6^1.x7^1.) +
(2x1^2.x2^1.x3^1.x4^3.x6^2.) + (2x1^2.x2^1.x3^1.x4^3.x5^1.x6^1.) +
(1x1^2.x2^1.x3^1.x4^3.x5^2.) + (-1x1^2.x2^1.x3^1.x4^4.x7^1.) +
(-2x1^2.x2^1.x3^1.x4^4.x6^1.) + (-1x1^2.x2^1.x3^1.x4^4.x5^1.) +
(-3x1^2.x2^1.x3^2.x4^2.x6^1.x7^1.) + (-3x1^2.x2^1.x3^2.x4^2.x6^2.) +
(1x1^2.x2^1.x3^2.x4^3.x7^1.) + (-2x1^2.x2^1.x3^2.x4^3.x6^1.) +
(1x1^2.x2^1.x3^2.x4^4.) + (3x1^2.x2^1.x3^3.x4^2.x6^1.) +
(-1x1^2.x2^1.x3^3.x4^3.) + (1x1^2.x2^2.x5^2.x6^2.x7^1.) +
(-3x1^2.x2^2.x4^1.x5^1.x6^2.x7^1.) + (-1x1^2.x2^2.x4^1.x5^2.x6^1.x7^1.) +
(-1x1^2.x2^2.x4^2.x6^2.x7^1.) + (-2x1^2.x2^2.x4^2.x5^2.x7^1.) +
(4x1^2.x2^2.x4^3.x6^1.x7^1.) + (3x1^2.x2^2.x4^3.x5^1.x7^1.) +
(-1x1^2.x2^2.x4^4.x7^1.) + (-1x1^2.x2^2.x3^1.x5^2.x6^2.) +
(3x1^2.x2^2.x3^1.x4^1.x6^2.x7^1.) + (3x1^2.x2^2.x3^1.x4^1.x5^1.x6^2.) +
(1x1^2.x2^2.x3^1.x4^1.x5^2.x6^1.) + (1x1^2.x2^2.x3^1.x4^2.x6^2.) +
(2x1^2.x2^2.x3^1.x4^2.x5^2.) + (-3x1^2.x2^2.x3^1.x4^3.x7^1.) +
(-4x1^2.x2^2.x3^1.x4^3.x6^1.) + (-3x1^2.x2^2.x3^1.x4^3.x5^1.) +
(1x1^2.x2^2.x3^1.x4^4.) + (-3x1^2.x2^2.x3^2.x4^1.x6^1.x7^1.) +
(-3x1^2.x2^2.x3^2.x4^1.x6^2.) + (3x1^2.x2^2.x3^2.x4^2.x7^1.) +
(3x1^2.x2^2.x3^2.x4^3.) + (3x1^2.x2^2.x3^3.x4^1.x6^1.) +
(-3x1^2.x2^2.x3^3.x4^2.) + (-1x1^2.x2^3.x5^1.x6^2.x7^1.) +
(-1x1^2.x2^3.x5^2.x6^1.x7^1.) + (2x1^2.x2^3.x4^1.x5^1.x6^1.x7^1.) +
(-1x1^2.x2^3.x4^1.x5^2.x7^1.) + (2x1^2.x2^3.x4^2.x6^1.x7^1.) +
(3x1^2.x2^3.x4^2.x5^1.x7^1.) + (-2x1^2.x2^3.x4^3.x7^1.) +
(1x1^2.x2^3.x3^1.x6^2.x7^1.) + (1x1^2.x2^3.x3^1.x5^1.x6^2.) +
(1x1^2.x2^3.x3^1.x5^2.x6^1.) + (-2x1^2.x2^3.x3^1.x4^1.x6^1.x7^1.) +
(-2x1^2.x2^3.x3^1.x4^1.x5^1.x6^1.) + (1x1^2.x2^3.x3^1.x4^1.x5^2.) +
(-3x1^2.x2^3.x3^1.x4^2.x7^1.) + (-2x1^2.x2^3.x3^1.x4^2.x6^1.) +
(-3x1^2.x2^3.x3^1.x4^2.x5^1.) + (2x1^2.x2^3.x3^1.x4^3.) +
(-1x1^2.x2^3.x3^2.x6^1.x7^1.) + (-1x1^2.x2^3.x3^2.x6^2.) +
(3x1^2.x2^3.x3^2.x4^1.x7^1.) + (2x1^2.x2^3.x3^2.x4^1.x6^1.) +
(3x1^2.x2^3.x3^2.x4^2.) + (1x1^2.x2^3.x3^3.x6^1.) +
(-3x1^2.x2^3.x3^3.x4^1.) + (1x1^2.x2^4.x5^1.x6^1.x7^1.) +
(1x1^2.x2^4.x4^1.x5^1.x7^1.) + (-1x1^2.x2^4.x4^2.x7^1.) +
(-1x1^2.x2^4.x3^1.x6^1.x7^1.) + (-1x1^2.x2^4.x3^1.x5^1.x6^1.) +
(-1x1^2.x2^4.x3^1.x4^1.x7^1.) + (-1x1^2.x2^4.x3^1.x4^1.x5^1.) +
(1x1^2.x2^4.x3^1.x4^2.) + (1x1^2.x2^4.x3^2.x7^1.) +
(1x1^2.x2^4.x3^2.x6^1.) + (1x1^2.x2^4.x3^2.x4^1.) + (-1x1^2.x2^4.x3^3.) +
(1x4^6.x5^4.x6^2.x7^1.) + (-1x3^1.x4^5.x5^4.x6^2.x7^1.) +
(-1x3^1.x4^6.x5^4.x6^1.x7^1.) + (-1x3^1.x4^6.x5^4.x6^2.) +
(-1x3^2.x4^4.x5^4.x6^2.x7^1.) + (1x3^2.x4^5.x5^3.x6^2.x7^1.) +
(1x3^2.x4^5.x5^4.x6^2.) + (1x3^2.x4^6.x5^3.x6^1.x7^1.) +
(1x3^2.x4^6.x5^4.x6^1.) + (1x3^3.x4^4.x5^4.x6^2.) + (-1x3^3.x4^5.x5^3.x6^2.)+
(-1x3^3.x4^6.x5^3.x6^1.) + (3x2^1.x4^5.x5^4.x6^2.x7^1.) +
(-1x2^1.x4^6.x5^4.x6^1.x7^1.) + (-3x2^1.x3^1.x4^4.x5^4.x6^2.x7^1.) +
(-2x2^1.x3^1.x4^5.x5^4.x6^1.x7^1.) + (-3x2^1.x3^1.x4^5.x5^4.x6^2.) +
(1x2^1.x3^1.x4^6.x5^4.x7^1.) + (1x2^1.x3^1.x4^6.x5^4.x6^1.) +
(-2x2^1.x3^2.x4^3.x5^4.x6^2.x7^1.) + (3x2^1.x3^2.x4^4.x5^3.x6^2.x7^1.) +
```

```
(2x2^1.x3^2.x4^4.x5^4.x6^1.x7^1.) + (3x2^1.x3^2.x4^4.x5^4.x6^2.) +
(-1x2^1.x3^2.x4^5.x5^2.x6^2.x7^1.) + (2x2^1.x3^2.x4^5.x5^3.x6^1.x7^1.) +
(2x2^1.x3^2.x4^5.x5^4.x6^1.) + (-1x2^1.x3^2.x4^6.x5^2.x6^1.x7^1.) +
(-1x2^1.x3^2.x4^6.x5^3.x7^1.) + (-1x2^1.x3^2.x4^6.x5^4.) +
(2x2^1.x3^3.x4^3.x5^4.x6^2.) + (-3x2^1.x3^3.x4^4.x5^3.x6^2.) +
(-2x2^1.x3^3.x4^4.x5^4.x6^1.) + (1x2^1.x3^3.x4^5.x5^2.x6^2.) +
(-2x2^1.x3^3.x4^5.x5^3.x6^1.) + (1x2^1.x3^3.x4^6.x5^2.x6^1.) +
(1x2^1.x3^3.x4^6.x5^3.) + (3x2^2.x4^4.x5^4.x6^2.x7^1.) +
(-3x2^2.x4^5.x5^4.x6^1.x7^1.) + (-3x2^2.x3^1.x4^3.x5^4.x6^2.x7^1.) +
(-3x2^2.x3^1.x4^4.x5^4.x6^2.) + (3x2^2.x3^1.x4^5.x5^4.x7^1.) +
(3x2^2.x3^1.x4^5.x5^4.x6^1.) + (-1x2^2.x3^2.x4^2.x5^4.x6^2.x7^1.) +
(3x2^2.x3^2.x4^3.x5^3.x6^2.x7^1.) + (4x2^2.x3^2.x4^3.x5^4.x6^1.x7^1.) +
(3x2^2.x3^2.x4^3.x5^4.x6^2.) + (-2x2^2.x3^2.x4^4.x5^2.x6^2.x7^1.) +
(-1x2^2.x3^2.x4^4.x5^4.x7^1.) + (-1x2^2.x3^2.x4^5.x5^2.x6^1.x7^1.) +
(-3x2^2.x3^2.x4^5.x5^3.x7^1.) + (-3x2^2.x3^2.x4^5.x5^4.) +
(1x2^2.x3^2.x4^6.x5^2.x7^1.) + (1x2^2.x3^3.x4^2.x5^4.x6^2.) +
(-3x2^2.x3^3.x4^3.x5^3.x6^2.) + (-4x2^2.x3^3.x4^3.x5^4.x6^1.) +
(2x2^2.x3^3.x4^4.x5^2.x6^2.) + (1x2^2.x3^3.x4^4.x5^4.) +
(1x2^2.x3^3.x4^5.x5^2.x6^1.) + (3x2^2.x3^3.x4^5.x5^3.) +
(-1x2^2.x3^3.x4^6.x5^2.) + (1x2^3.x4^3.x5^4.x6^2.x7^1.) +
(-3x2^3.x4^4.x5^4.x6^1.x7^1.) + (-1x2^3.x3^1.x4^2.x5^4.x6^2.x7^1.) +
(2x2^3.x3^1.x4^3.x5^4.x6^1.x7^1.) + (-1x2^3.x3^1.x4^3.x5^4.x6^2.) +
(3x2^3.x3^1.x4^4.x5^4.x7^1.) + (3x2^3.x3^1.x4^4.x5^4.x6^1.) +
(1x2^3.x3^2.x4^2.x5^3.x6^2.x7^1.) + (2x2^3.x3^2.x4^2.x5^4.x6^1.x7^1.) +
(1x2^3.x3^2.x4^2.x5^4.x6^2.) + (-1x2^3.x3^2.x4^3.x5^2.x6^2.x7^1.) +
(-2x2^3.x3^2.x4^3.x5^3.x6^1.x7^1.) + (-2x2^3.x3^2.x4^3.x5^4.x7^1.) +
(-2x2^3.x3^2.x4^3.x5^4.x6^1.) + (1x2^3.x3^2.x4^4.x5^2.x6^1.x7^1.) +
(-3x2^3.x3^2.x4^4.x5^3.x7^1.) + (-3x2^3.x3^2.x4^4.x5^4.) +
(2x2^3.x3^2.x4^5.x5^2.x7^1.) + (-1x2^3.x3^3.x4^2.x5^3.x6^2.) +
(-2x2^3.x3^3.x4^2.x5^4.x6^1.) + (1x2^3.x3^3.x4^3.x5^2.x6^2.) +
(2x2^3.x3^3.x4^3.x5^3.x6^1.) + (2x2^3.x3^3.x4^3.x5^4.) +
(-1x2^3.x3^3.x4^4.x5^2.x6^1.) + (3x2^3.x3^3.x4^4.x5^3.) +
(-2x2^3.x3^3.x4^5.x5^2.) + (-1x2^4.x4^3.x5^4.x6^1.x7^1.) +
(1x2^4.x3^1.x4^2.x5^4.x6^1.x7^1.) + (1x2^4.x3^1.x4^3.x5^4.x7^1.) +
(1x2^4.x3^1.x4^3.x5^4.x6^1.) + (-1x2^4.x3^2.x4^2.x5^3.x6^1.x7^1.) +
(-1x2^4.x3^2.x4^2.x5^4.x7^1.) + (-1x2^4.x3^2.x4^2.x5^4.x6^1.) +
(1x2^4.x3^2.x4^3.x5^2.x6^1.x7^1.) + (-1x2^4.x3^2.x4^3.x5^3.x7^1.) +
(-1x2^4.x3^2.x4^3.x5^4.) + (1x2^4.x3^2.x4^4.x5^2.x7^1.) +
(1x2^4.x3^3.x4^2.x5^3.x6^1.) + (1x2^4.x3^3.x4^2.x5^4.) +
(-1x2^4.x3^3.x4^3.x5^2.x6^1.) + (1x2^4.x3^3.x4^3.x5^3.) +
(-1x2^4.x3^3.x4^4.x5^2.) + (-2x1^1.x4^6.x5^3.x6^2.x7^1.) +
(2x1^1.x3^1.x4^5.x5^3.x6^2.x7^1.) + (2x1^1.x3^1.x4^6.x5^3.x6^1.x7^1.) +
(2x1^1.x3^1.x4^6.x5^3.x6^2.) + (2x1^1.x3^2.x4^4.x5^3.x6^2.x7^1.) +
(-2x1^1.x3^2.x4^5.x5^2.x6^2.x7^1.) + (-2x1^1.x3^2.x4^5.x5^3.x6^2.) +
(-2x1^1.x3^2.x4^6.x5^2.x6^1.x7^1.) + (-2x1^1.x3^2.x4^6.x5^3.x6^1.) +
(-2x1^1.x3^3.x4^4.x5^3.x6^2.) + (2x1^1.x3^3.x4^5.x5^2.x6^2.) +
(2x1^1.x3^3.x4^6.x5^2.x6^1.) + (-6x1^1.x2^1.x4^5.x5^3.x6^2.x7^1.) +
```

```
(2x1^1.x2^1.x4^6.x5^3.x6^1.x7^1.) + (6x1^1.x2^1.x3^1.x4^4.x5^3.x6^2.x7^1.) +
(4x1^1.x2^1.x3^1.x4^5.x5^3.x6^1.x7^1.) + (6x1^1.x2^1.x3^1.x4^5.x5^3.x6^2.)+
(-2x1^1.x2^1.x3^1.x4^6.x5^3.x7^1.) + (-2x1^1.x2^1.x3^1.x4^6.x5^3.x6^1.) +
(4x1^1.x2^1.x3^2.x4^3.x5^3.x6^2.x7^1.)+(-6x1^1.x2^1.x3^2.x4^4.x5^2.x6^2.x7^1.) +
(-4x1^1.x2^1.x3^2.x4^4.x5^3.x6^1.x7^1.) + (-6x1^1.x2^1.x3^2.x4^4.x5^3.x6^2.) +
(2x1^1.x2^1.x3^2.x4^5.x5^1.x6^2.x7^1.) + (-4x1^1.x2^1.x3^2.x4^5.x5^2.x6^1.x7^1.)
(-4x1^1.x2^1.x3^2.x4^5.x5^3.x6^1.) + (2x1^1.x2^1.x3^2.x4^6.x5^1.x6^1.x7^1.) +
(2x1^1.x2^1.x3^2.x4^6.x5^2.x7^1.) + (2x1^1.x2^1.x3^2.x4^6.x5^3.) +
(-4x1^1.x2^1.x3^3.x4^3.x5^3.x6^2.) + (6x1^1.x2^1.x3^3.x4^4.x5^2.x6^2.) +
(4x1^1.x2^1.x3^3.x4^4.x5^3.x6^1.) + (-2x1^1.x2^1.x3^3.x4^5.x5^1.x6^2.) +
(4x1^1.x2^1.x3^3.x4^5.x5^2.x6^1.) + (-2x1^1.x2^1.x3^3.x4^6.x5^1.x6^1.) +
(-2x1^1.x2^1.x3^3.x4^6.x5^2.) + (-6x1^1.x2^2.x4^4.x5^3.x6^2.x7^1.) +
(6x1^1.x2^2.x4^5.x5^3.x6^1.x7^1.) + (6x1^1.x2^2.x3^1.x4^3.x5^3.x6^2.x7^1.) +
(6x1^1.x2^2.x3^1.x4^4.x5^3.x6^2.) + (-6x1^1.x2^2.x3^1.x4^5.x5^3.x7^1.) +
(-6x1^1.x2^2.x3^1.x4^5.x5^3.x6^1.) + (2x1^1.x2^2.x3^2.x4^2.x5^3.x6^2.x7^1.) +
(-6x1^1.x2^2.x3^2.x4^3.x5^2.x6^2.x7^1.) + (-8x1^1.x2^2.x3^2.x4^3.x5^3.x6^1.x7^1.)
(-6x1^1.x2^2.x3^2.x4^3.x5^3.x6^2.) + (4x1^1.x2^2.x3^2.x4^4.x5^1.x6^2.x7^1.) +
(2x1^1.x2^2.x3^2.x4^4.x5^3.x7^1.) + (2x1^1.x2^2.x3^2.x4^5.x5^1.x6^1.x7^1.) +
(6x1^1.x2^2.x3^2.x4^5.x5^2.x7^1.) + (6x1^1.x2^2.x3^2.x4^5.x5^3.) +
(-2x1^1.x2^2.x3^2.x4^6.x5^1.x7^1.) + (-2x1^1.x2^2.x3^3.x4^2.x5^3.x6^2.) +
(6x1^1.x2^2.x3^3.x4^3.x5^2.x6^2.) + (8x1^1.x2^2.x3^3.x4^3.x5^3.x6^1.) +
(-4x1^1.x2^2.x3^3.x4^4.x5^1.x6^2.) + (-2x1^1.x2^2.x3^3.x4^4.x5^3.) +
(-2x1^1.x2^2.x3^3.x4^5.x5^1.x6^1.) + (-6x1^1.x2^2.x3^3.x4^5.x5^2.) +
(2x1^1.x2^2.x3^3.x4^6.x5^1.) + (-2x1^1.x2^3.x4^3.x5^3.x6^2.x7^1.) +
(6x1^1.x2^3.x4^4.x5^3.x6^1.x7^1.) + (2x1^1.x2^3.x3^1.x4^2.x5^3.x6^2.x7^1.) +
(-4x1^1.x2^3.x3^1.x4^3.x5^3.x6^1.x7^1.) + (2x1^1.x2^3.x3^1.x4^3.x5^3.x6^2.) +
(-6x1^1.x2^3.x3^1.x4^4.x5^3.x7^1.) + (-6x1^1.x2^3.x3^1.x4^4.x5^3.x6^1.) +
(-2x1^1.x2^3.x3^2.x4^2.x5^2.x6^2.x7^1.) + (-4x1^1.x2^3.x3^2.x4^2.x5^3.x6^1.x7^1.)
(-2x1^1.x2^3.x3^2.x4^2.x5^3.x6^2.) + (2x1^1.x2^3.x3^2.x4^3.x5^1.x6^2.x7^1.) +
(4x1^1.x2^3.x3^2.x4^3.x5^2.x6^1.x7^1.) + (4x1^1.x2^3.x3^2.x4^3.x5^3.x7^1.) +
(4x1^1.x2^3.x3^2.x4^3.x5^3.x6^1.) + (-2x1^1.x2^3.x3^2.x4^4.x5^1.x6^1.x7^1.) +
(6x1^1.x2^3.x3^2.x4^4.x5^2.x7^1.) + (6x1^1.x2^3.x3^2.x4^4.x5^3.) +
(-4x1^1.x2^3.x3^2.x4^5.x5^1.x7^1.) + (2x1^1.x2^3.x3^3.x4^2.x5^2.x6^2.) +
(4x1^1.x2^3.x3^3.x4^2.x5^3.x6^1.) + (-2x1^1.x2^3.x3^3.x4^3.x5^1.x6^2.) +
(-4x1^1.x2^3.x3^3.x4^3.x5^2.x6^1.) + (-4x1^1.x2^3.x3^3.x4^3.x5^3.) +
(2x1^1.x2^3.x3^3.x4^4.x5^1.x6^1.) + (-6x1^1.x2^3.x3^3.x4^4.x5^2.) +
(4x1^1.x2^3.x3^3.x4^5.x5^1.) + (2x1^1.x2^4.x4^3.x5^3.x6^1.x7^1.) +
(-2x1^1.x2^4.x3^1.x4^2.x5^3.x6^1.x7^1.) + (-2x1^1.x2^4.x3^1.x4^3.x5^3.x7^1.) +
(-2x1^1.x2^4.x3^1.x4^3.x5^3.x6^1.) + (2x1^1.x2^4.x3^2.x4^2.x5^2.x6^1.x7^1.) +
(2x1^1.x2^4.x3^2.x4^2.x5^3.x7^1.) + (2x1^1.x2^4.x3^2.x4^2.x5^3.x6^1.) +
(-2x1^1.x2^4.x3^2.x4^3.x5^1.x6^1.x7^1.) + (2x1^1.x2^4.x3^2.x4^3.x5^2.x7^1.) +
(2x1^1.x2^4.x3^2.x4^3.x5^3.) + (-2x1^1.x2^4.x3^2.x4^4.x5^1.x7^1.) +
(-2x1^1.x2^4.x3^3.x4^2.x5^2.x6^1.) + (-2x1^1.x2^4.x3^3.x4^2.x5^3.) +
(2x1^1.x2^4.x3^3.x4^3.x5^1.x6^1.) + (-2x1^1.x2^4.x3^3.x4^3.x5^2.) +
(2x1^1.x2^4.x3^3.x4^4.x5^1.) + (1x1^2.x4^6.x5^2.x6^2.x7^1.) +
(-1x1^2.x3^1.x4^5.x5^2.x6^2.x7^1.) + (-1x1^2.x3^1.x4^6.x5^2.x6^1.x7^1.) +
(-1x1^2.x3^1.x4^6.x5^2.x6^2.) + (-1x1^2.x3^2.x4^4.x5^2.x6^2.x7^1.) +
```

```
(1x1^2.x3^2.x4^5.x5^1.x6^2.x7^1.) + (1x1^2.x3^2.x4^5.x5^2.x6^2.) +
(1x1^2.x3^2.x4^6.x5^1.x6^1.x7^1.) + (1x1^2.x3^2.x4^6.x5^2.x6^1.) +
(1x1^2.x3^3.x4^4.x5^2.x6^2.) + (-1x1^2.x3^3.x4^5.x5^1.x6^2.) +
(-1x1^2.x3^3.x4^6.x5^1.x6^1.) + (3x1^2.x2^1.x4^5.x5^2.x6^2.x7^1.) +
(-1x1^2.x2^1.x4^6.x5^2.x6^1.x7^1.) + (-3x1^2.x2^1.x3^1.x4^4.x5^2.x6^2.x7^1.) +
(-2x1^2.x2^1.x3^1.x4^5.x5^2.x6^1.x7^1.) + (-3x1^2.x2^1.x3^1.x4^5.x5^2.x6^2.) +
(1x1^2.x2^1.x3^1.x4^6.x5^2.x7^1.) + (1x1^2.x2^1.x3^1.x4^6.x5^2.x6^1.) +
(-2x1^2.x2^1.x3^2.x4^3.x5^2.x6^2.x7^1.) + (3x1^2.x2^1.x3^2.x4^4.x5^1.x6^2.x7^1.)
(2x1^2.x2^1.x3^2.x4^4.x5^2.x6^1.x7^1.) + (3x1^2.x2^1.x3^2.x4^4.x5^2.x6^2.) +
(-1x1^2.x2^1.x3^2.x4^5.x6^2.x7^1.) + (2x1^2.x2^1.x3^2.x4^5.x5^1.x6^1.x7^1.) +
(2x1^2.x2^1.x3^2.x4^5.x5^2.x6^1.) + (-1x1^2.x2^1.x3^2.x4^6.x6^1.x7^1.) +
(-1x1^2.x2^1.x3^2.x4^6.x5^1.x7^1.) + (-1x1^2.x2^1.x3^2.x4^6.x5^2.) +
(2x1^2.x2^1.x3^3.x4^3.x5^2.x6^2.) + (-3x1^2.x2^1.x3^3.x4^4.x5^1.x6^2.) +
(-2x1^2.x2^1.x3^3.x4^4.x5^2.x6^1.) + (1x1^2.x2^1.x3^3.x4^5.x6^2.) +
(-2x1^2.x2^1.x3^3.x4^5.x5^1.x6^1.) + (1x1^2.x2^1.x3^3.x4^6.x6^1.) +
(1x1^2.x2^1.x3^3.x4^6.x5^1.) + (3x1^2.x2^2.x4^4.x5^2.x6^2.x7^1.) +
(-3x1^2.x2^2.x4^5.x5^2.x6^1.x7^1.) + (-3x1^2.x2^2.x3^1.x4^3.x5^2.x6^2.x7^1.) +
(-3x1^2.x2^2.x3^1.x4^4.x5^2.x6^2.) + (3x1^2.x2^2.x3^1.x4^5.x5^2.x7^1.) +
(3x1^2.x2^2.x3^1.x4^5.x5^2.x6^1.) + (-1x1^2.x2^2.x3^2.x4^2.x5^2.x6^2.x7^1.) +
(3x1^2.x2^2.x3^2.x4^3.x5^1.x6^2.x7^1.) + (4x1^2.x2^2.x3^2.x4^3.x5^2.x6^1.x7^1.) +
(3x1^2.x2^2.x3^2.x4^3.x5^2.x6^2.) + (-2x1^2.x2^2.x3^2.x4^4.x6^2.x7^1.) +
(-1x1^2.x2^2.x3^2.x4^4.x5^2.x7^1.) + (-1x1^2.x2^2.x3^2.x4^5.x6^1.x7^1.) +
(-3x1^2.x2^2.x3^2.x4^5.x5^1.x7^1.) + (-3x1^2.x2^2.x3^2.x4^5.x5^2.) +
(1x1^2.x2^2.x3^2.x4^6.x7^1.) + (1x1^2.x2^2.x3^3.x4^2.x5^2.x6^2.) +
(-3x1^2.x2^2.x3^3.x4^3.x5^1.x6^2.) + (-4x1^2.x2^2.x3^3.x4^3.x5^2.x6^1.) +
(2x1^2.x2^2.x3^3.x4^4.x6^2.) + (1x1^2.x2^2.x3^3.x4^4.x5^2.) +
(1x1^2.x2^2.x3^3.x4^5.x6^1.) + (3x1^2.x2^2.x3^3.x4^5.x5^1.) +
(-1x1^2.x2^2.x3^3.x4^6.) + (1x1^2.x2^3.x4^3.x5^2.x6^2.x7^1.) +
(-3x1^2.x2^3.x4^4.x5^2.x6^1.x7^1.) + (-1x1^2.x2^3.x3^1.x4^2.x5^2.x6^2.x7^1.) +
(2x1^2.x2^3.x3^1.x4^3.x5^2.x6^1.x7^1.) + (-1x1^2.x2^3.x3^1.x4^3.x5^2.x6^2.) +
(3x1^2.x2^3.x3^1.x4^4.x5^2.x7^1.) + (3x1^2.x2^3.x3^1.x4^4.x5^2.x6^1.) +
(1x1^2.x2^3.x3^2.x4^2.x5^1.x6^2.x7^1.) + (2x1^2.x2^3.x3^2.x4^2.x5^2.x6^1.x7^1.) +
(1x1^2.x2^3.x3^2.x4^2.x5^2.x6^2.) + (-1x1^2.x2^3.x3^2.x4^3.x6^2.x7^1.) +
(-2x1^2.x2^3.x3^2.x4^3.x5^1.x6^1.x7^1.) + (-2x1^2.x2^3.x3^2.x4^3.x5^2.x7^1.) +
(-2x1^2.x2^3.x3^2.x4^3.x5^2.x6^1.) + (1x1^2.x2^3.x3^2.x4^4.x6^1.x7^1.) +
(-3x1^2.x2^3.x3^2.x4^4.x5^1.x7^1.) + (-3x1^2.x2^3.x3^2.x4^4.x5^2.) +
(2x1^2.x2^3.x3^2.x4^5.x7^1.) + (-1x1^2.x2^3.x3^3.x4^2.x5^1.x6^2.) +
(-2x1^2.x2^3.x3^3.x4^2.x5^2.x6^1.) + (1x1^2.x2^3.x3^3.x4^3.x6^2.) +
(2x1^2.x2^3.x3^3.x4^3.x5^1.x6^1.) + (2x1^2.x2^3.x3^3.x4^3.x5^2.) +
(-1x1^2.x2^3.x3^3.x4^4.x6^1.) + (3x1^2.x2^3.x3^3.x4^4.x5^1.) +
(-2x1^2.x2^3.x3^3.x4^5.) + (-1x1^2.x2^4.x4^3.x5^2.x6^1.x7^1.) +
(1x1^2.x2^4.x3^1.x4^2.x5^2.x6^1.x7^1.) + (1x1^2.x2^4.x3^1.x4^3.x5^2.x7^1.) +
(1x1^2.x2^4.x3^1.x4^3.x5^2.x6^1.) + (-1x1^2.x2^4.x3^2.x4^2.x5^1.x6^1.x7^1.) +
(-1x1^2.x2^4.x3^2.x4^2.x5^2.x7^1.) + (-1x1^2.x2^4.x3^2.x4^2.x5^2.x6^1.) +
(1x1^2.x2^4.x3^2.x4^3.x6^1.x7^1.) + (-1x1^2.x2^4.x3^2.x4^3.x5^1.x7^1.) +
(-1x1^2.x2^4.x3^2.x4^3.x5^2.) + (1x1^2.x2^4.x3^2.x4^4.x7^1.) +
(1x1^2.x2^4.x3^3.x4^2.x5^1.x6^1.) + (1x1^2.x2^4.x3^3.x4^2.x5^2.) +
```

```
(-1x1^2.x2^4.x3^3.x4^3.x6^1.) + (1x1^2.x2^4.x3^3.x4^3.x5^1.) +
(-1x1^2.x2^4.x3^3.x4^4.)
```

Il s'agit d'un polynôme de degré 13 ccomportant 612 monômes!

On voit donc à travers cet exemple qu'une des difficultés majeures dans le calcul du Bézoutien impliquant la manipulation de grosses matrice, est la quantité de mémoire qu'il faut mobiliser (dont il faut disposer) pour mener les calculs jusqu'à leur terme.

Un autre problème auquel cette impressionnante croissance des coefficients peut nous conduire, est celui des performances à l'exécution. On peut imaginer d'avance, en effet, ce que peut représenter le temps de calcul nécessaire pour appliquer l'algorithme de division sur d'aussi gros termes.

Ces deux problèmes, qui sont au coeur de l'activité informatique, nous ont préocupé tout au long de ce stage et nous ont obligé à insister particulièrement sur la recherche d'algorithmes efficaces et, comme nous le verrons dans la section suivante, de structures de données adéquates (*tas, table de hachage, etc.*) afin d'obtenir de bonnes implémentations.

Afin d'éviter l'interruption des calculs dans certains cas, il est possible d'optimiser la mémoire en choisissant une option de calcul qui consiste à positionner le deuxième argument (`Optimize`) de la fonction `Polybase::Determinant(int , boolean)` à la valeur **TRUE**. Ceci aura pour effet de procéder à la délétion dynamique des éléments de la matrice qui ne servent plus dans le calcul (fig 4.2). C'est un argument optionnel, comme l'autorise le C++, qui vaut **FALSE** par défaut. S'il n'est pas passé à la fonction, le calcul se fait sans optimisation. Dans ce cas tous les éléments de la matrice sont maintenus jusqu'à la fin du calcul. Ceci peut être inutilement coûteux, mais noter tout de même que dans certains cas désespérés, cette astuce ne sert strictement à rien, car les besoins des derniers éléments qui apparaissent sont tels que la mémoire explose malgré tout.

Remarque : concernant la croissance des coefficients dans le calcul du déterminant, il est possible d'obtenir des termes intermédiaires de taille moins importantes, grâce à une légère modification de la méthode de *Bareiss* due à *Sasaki* et *Murao* (1982). Cette méthode que nous avons découvert un peu tard pour pouvoir l'implémenter consiste, dans le développement des éléments des matrices réduites, à ne prendre en compte que ceux des termes effectivement nécessaires à la suite du calcul. Cet algorithme sera sans doute substitué à l'actuel dans une version future de la librairie.

Exemple de calcul du Bézoutien par notre implémentation

Maintenant que nous sommes en mesure de calculer un déterminant (disons quelconque) et que nous sommes capables de diviser des polynômes en plusieurs variables, le calcul du Bézoutien est dès lors assez simple.

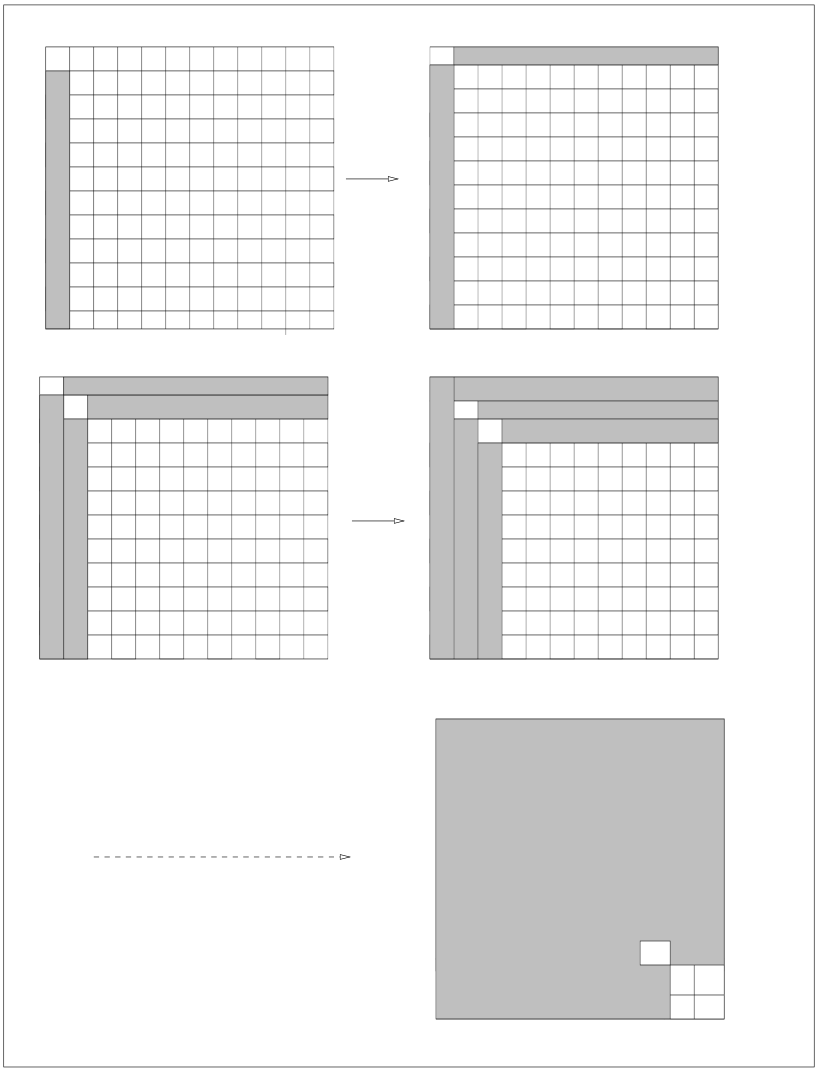


FIG. 4.2 – *Optimsation en mémoire du calcul du Détermiinant*

Le construction du dénominateur $\prod_{i=1}^{n}(y_i - x_i)$ demande une petite manipulation qui se fait d'ailleurs sans peine ; la matrice du numérateur auquel s'applique la fonction de calcul du déterminant est quant à elle obtenu par une série de permutations sur les variables des polynômes initiaux pour obtenir les autres coefficients de la dite matrice. Ceci a été implémenté dans la méthode `Polybase::PreBezout()` de la classe `Polybase`.

C'est pour des raisons de commodité et pour garder une certaine uniformité par rapport au langage C++ que les fonctions de calcul du Bézoutien ainsi que toutes les autres fonctions de construction des matrices implémentées ont été intégrées comme méthode à la classe *Polybase* qui, rappelons-le, est une classe d'*ensemble de polynômes* ; ceci permet de programmer simplement avec la librairie en créant des objets de type `Polybase`, les initialiser pour qu'il contiennent une liste de polynômes, et leur appliquer ensuite les differentes méthodes pour en obtenir le Bézoutien, les matrices associées, un déterminant etc, comme dans l'exemple suivant :

```
#include <iostream.H>
#include ''ALP.H''

main()
{
   int n = 3;
   Polybase PB(''PolyFile.alp'', n);
   Polynome P = PB.Bezoutian();
   cout << P << endl;

}
```

Dans ce petit programme, on a crée un objet `Polybase`, *PB*, initialisé avec les trois premiers éléments d'une liste de polynômes contenus dans un fichier au *format ALP* que nous avons défini dans le module `Outils-IO`. Le Bézoutien des éléments de la polybase est calculé en appliquant simplement la méthode `Polybase::Bezoutian()` à *PB* le résultat, qui est de type `Polynome`, est affecté à *P* puis affiché sur la sortie standard.

Dans cet exemple où *PolyFile.alp* contient les polynômes

$$\begin{cases} f_1 &= x_3 + x_2 + x_1 \\ f_2 &= x_2x_3 + x_1x_3 + x_1x_2 \\ f_3 &= -1 + x_1x_2x_3 \end{cases}$$

nous avons obtenu le resultat suivant sur une machine Sun sous Solaris :

$$\Phi(f_1, f_2, f_3) = -x_4^2x_5 + x_3x_4x_5 + x_3x_4^2 - x_3^2x_4 + x_2x_4x_5 - x_2x_3x_5 - x_2x_3x_4 + x_2x_3^2$$

4.3.3 Une méthode de tri et de hachage dynamiques pour construire la Matrice du Résultant

A présent que nous avons une fonction de calcul du *Bézoutien* nous passsons à la construction de la matrice du résultant proprement dite. Comme je l'ai déjà indiqué, cette construction a nécessité beaucoup plus d'effort et d'heures de travail que toutes les autres. Elle a nécessité, entre autre, la mise en place de structures de données particulières, pour plus d'efficacité.

Je rappelle que la matrice du *Bézoutien* de $n+1$ polynômes $f_0, f_1, .., f_n$, notée $[\Phi(f_0, \ldots, f_n)]$ est la matrice $[c_{ij}]$ tel que

$$\Phi(f_0, \ldots, f_n) = \sum_{ij} c_{i,j} \mathbf{x}^{\mathbf{u}_i} \mathbf{y}^{\mathbf{v}_j}.$$

où $\mathbf{x}^{\mathbf{u}_i} = x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$ et $\mathbf{y}^{\mathbf{u}_j} = x_{n+1}^{j_{n+1}} x_{n+2}^{j_{n+2}} \ldots x_{2n}^{j_{2n}}$ étant entendu que le Bézoutien est en $2n$ variables. Par abus de notation on écrira

$$\Phi(f_0, \ldots, f_n) = \sum_{ij} c_{i,j} \mathbf{X}^i \mathbf{Y}^j.$$

Pour construire la matrice, on regroupe les termes en $\mathbf{X}$ de façon à écrire le Bézoutien comme polynôme en $\mathbf{Y}$, soit

$$\Phi(f_0, \ldots, f_n) = \sum_j \left(\sum_i c_{i,j} \mathbf{X}^i \right) \mathbf{Y}^j$$

ou encore

$$\Phi(f_0, \ldots, f_n) = \Phi(\mathbf{X}, \mathbf{Y}) = \sum_{\mathbf{j}} \mathbf{P}^{\mathbf{j}} \mathbf{Y}^{\mathbf{j}}$$

Si n_x et n_y désignent respectivement les nombres de monômes en $\mathbf{X}$ et en $\mathbf{Y}$ distincts contenus dans cette expression, alors la matrice du *Bézoutien* est la matrice (n_x, n_y) dont les lignes sont indexées par les indexées par les $\mathbf{P}^j$ et les colonnes par les $\mathbf{X}^{\mathbf{i}}$ ordonnés selon l'ordre défini sur les monômes. Autrement dit, l'élément (i, j) de la matrice est égal au coefficient c_{ij} de $\mathbf{X}^{\mathbf{i}}$ dans le polynôme $\mathbf{P}^{\mathbf{j}}$. Et comme tous les monômes n_x monômes en $\mathbf{X}$ ne sont pas forcément présent dans chaque $\mathbf{P}^{\mathbf{j}}$ les autres entrées de la matrice sont nuls.

Pour achever la construction, il suffit donc de déclarer une matrice de la bonne taille, initialement remplie de *zéros* et d'y mettre les coefficients non nuls issus du *Bézoutien* de façon analogue à ce qui a été fait dans la construction de *Bézout-Dixon*.

Mais dans la construction de *Bézout-Dixon* on avait d'abord commencé par déclarer la matrice, avant de récupéré puis insérer les coefficients non nuls. Cela a été simplifié par le fait que l'on connaissait d'avance la taille de la matrice, donnée par le nombre de monômes. La difficulté majeure dans le cas du Bézoutien en plusieurs variables c'est que l'on ignore le nombre des monômes en $\mathbf{X}$ et celui des polynômes qui seront issus du regroupement des termes ; donc la taille de la matrice finale est inconnue avant le regroupement complet des termes. L'utilisation de tableaux pour regrouper les monômes en **X** tel qu'on l'avait fait dans le cas uni-variable, suppose la déclaration d'un tableau de taille au moins égale au nombre total de termes effectivement présent dans le Bézoutien, ce, afin d'éviter de possibles débordements de mémoire (fatals aux programmes C). Il s'en suivrait alors un gaspillage dans le cas (fréquent) où le Bézoutien ne comporterait pas autant de monômes en **X**.

D'autre part, le Bézoutien en plusieurs variables pouvant être un polynôme très gros, le regroupement des termes peut être coûteux en temps de calcul ; d'où la nécessité d'adopter une démarche qui puisse permettre à la fois de gérer de très gros polynômes et dont y accéder rapidement.

Afin de prendre en compte ces obstacles, nous avons mis en oeuvre une organisation efficace des données en combinant, dans la construction, deux types de structure

- Une structure *liste de monômes*, implémentée dans la classe `MonoList`, qui permet de regrouper un nombre variable de monômes tout en les maintenant ordonnés.
- Une structure *Table de Hachage dynamique*, implémentée dans la classe `PolyHash`, qui permet d'effectuer le regroupement rapide de polynômes.

Listes triées de monômes

La classe `MonoList` a été crée spécialement pour la construction de la matrice du Bézoutien. Elle a ensuite été utiliseée pour dans la construction de Macaulay qui a été fait en dernier lieu. Il s'agit d'une classe simplifiée supportant seulement certaines des opérations habituellement disponibles sur les implémentations de *Listes.* D'autre part on aurait dû utiliser la classe `Polynome` étant donnée qu'un polynôme est implémenté sous la forme d'une liste de monôme ; nous n'en avons pas fait le choix car les éléments chaînés de la classe polynôme ne sont pas seulement de simple monôme mais des cellules contenant ces monômes. De plus le parcours de cette liste nécessite la déclaration d'une instance d'une autre classe (la classe `PolyIter`). Donc les opérations d'insertion et de consultation des élément de la liste peuvent manquer d'efficacité si l'on se contente d'utiliser la classe `Polynome.`

Je ne vais pas revenir sur les spécifications de la classe `MonoList` que j'ai insérées dans le *Manuel de Référence* de la librairie, je rappelle simplement les composantes de cette classes qui ont aidé à la construction de la matrice de Bézout. Il s'agit principalement d'un constructeur de création par défaut (l'unique) et des méthodes suivantes :

- `Length()` : donne le nombre d'éléments de la liste
- `Search(const Monome&)` : permet de vérifier si un monôme est présent dans la liste. Si tel est le cas, un entier correspondant à sa position est retournée, sinon 0 est retourné.
- `Mark()` : cette méthode est invoquée pour marquer les éléments de la liste. Ce marquage est réalisé en affectant au champ `Rank` de chaque monôme l'entier qui

correspond à sa position sur la liste. Ce champ, ainsi que les méthodes qui permettent sa manipulation(`Monome::SetRank(int)` et `Monome::GetRank()`) ont d'ailleurs été ajoutés à la classe `Monome` pour la circonstance.

– `Insert(const Monome&)` : permet d'insérer un monôme dans la liste. Chaque monôme est inséré une seule fois à l'endroit qui maintient la liste tríee. Une valeur booléenne est ensuite retournée pour indiquer si le monôme a déjà été inséré ou pas.

Ces quatre méthodes donnent une idée assez précise du rôle qui sera celui de la classe `MonoList` qui comprend, en outre, un destructeur ainsi que des opérateurs d'entrée sortie et d'autres méthodes utilisées dans la construction de la matrice de Macaulay.

Tables de hachage

Les tables de hachage sont, sans doute, la structure de donnée la plus intéressante pour gérer notamment des données en grande quantité. Elles sont largement utilisées notamment en *Bases de données* pour le stockage et l'accès rapide à de telles données via des *transformations de clés*.

L'idée des tables de hachage part du principe simple selon lequel si l'on dispose d'un moyen d'indexer les données, il est possible de les ranger dans des tableaux adressés, à des endroits qui correspondent aux entrées spécifiées par les différents indices des éléments à ranger. Le langage C/C++ offre notamment des tableaux indexés par des entiers commençant à *zéro* et où l'on peut placer des éléments et y accéder en un temps constant. Le problème est donc de trouver un moyen d'associer à chacune des données à ranger un indice, c'est-à-dire un nombre entier, qui correspond à une adresse dans le tableau. Ceci est fait au moyen de fonctions particulières dites de hachage dont le but est de fournir ces entiers.

Un exemple simple de hachage est celui qui consiste, dans la gestion d'un dictionnaire orthographique, d'associer à chaque mot du vocabulaire la somme des positions (dans l'alphabet) des lettres qui le composent. Une telle fonction de hachage associerait, par exemple, au mot *table* l'indice $20+1+2+12+5=40$. On pourra donc placer (le texte de) la définition de ce mot à l'adresse d'indice 40 d'un grand tableau de *chaînes de caractères*, de sorte que pour retrouver cette définition il suffit de lui appliquer la fonction de hachage pour avoir l'adresse correspondante et y accéder directement. On dit alors que le mot est la *clé* d'accès à l'*information* que constitue le texte de la définition.

En fait c'est légèrement plus compliqué que cela, puisque des *collisions* peuvent survenir, c'est à dire des valeurs de hachage identiques pour des mots différents. Pour résoudre ces collisions deux démarches peuvent être envisagées qui détermineront la structure interne de la table. On peut utiliser un tableau de listes chaînées comme l'indique la figure 4.2. Dans ce cas toutes les informations (et les clés associées) qui correspondent à des valeurs de hachage identiques seront placées dans une liste chaînée débutant à l'adresse du tableau indiquée par cette valeur commune. Pour retrouver la bonne définition d'un mot, il suffit alors de le comparer aux éléments de la liste débutant à l'adresse donnée par sa valeur de hachage. La deuxième solution, que nous avons implémentée, consiste à maintenir un tableau simple dont les éléments sont constitués d'une structure regroupant la clé et l'information associée ; et, lorsqu'une entrée est déjà occupée par un élément, un prochain élément ayant la même valeur sera rangé à la première adresse libre du tableau à partir de celle initialement donnée par la fonction de hachage. Si on arrive à la fin du tableau et qu'aucune place libre n'ait été trouvée, on recommence à partir du début du tableau. Si la table est de taille fixe, il est possible qu'elle soit pleine à un certain moment auquel cas on retombe sur le premier élément de la collision. Il est toutefois possible de modifier dynamiquement la taille du tableau pour que tout nouveau élément trouve une place dans la table, dans la limite de la mémoire disponible, bien entendu. C'est ce que nous avons fait dans notre implémentation.

L'avantage, commun aux deux approches, qui fait l'originalité des tables de hachage (leur efficacité) est de permettre de stocker de grandes quantités d'informations et d'y accéder en un temps en $O(1)$ pour une table de bonne taille et une bonne fonction de hachage. Le cas pire conduisant à des accès en $O(n)$.

Un élément essentiel dans cette technique concerne, cependant, la fonction de hachage utilisée. En effet, il est clair que le hachage sera d'autant plus performant que les collisions seront limitées. L'importance des tables de hachage a d'ailleurs justifier de nombreuses études sur les fonctions de hachage et il existe même des programmes de génération de fonctions de hachage parfaites telles que `gperf` de GNU.

Dans notre contexte, les informations à stocker sont les polynômes $\mathtt{P}^{\mathtt{j}}$ tandis que les clés associées sont les sous-monômes $\mathbf{Y}^{\mathtt{j}}$. Enfin la fonction de hachage est une fonction, implémentée comme *amie* de la classe monôme et qui retourne la somme des exposants de son argument modulo la taille de la table de hachage, fixée par défaut à 100 mais qui, comme je l'ai déjà souligné, est automatiquement agrandie en cas de besoin.

4.3.4 Construction de la matrice de Bézout

Avec les deux structures que nous venons de décrire, il est désormais très simple de construire la matrice du Bézoutien. La démarche, schématisée dans la figure 4.3, est la suivante :

- On parcourt le Bézoutien à l'aide d'un itérateur `PolyIter` et chacun des termes rencontrés est passé à la fonction `SplitMonome()` dont le rôle est de scinder son argument, donnant d'un côté le sous-monôme en $\mathbf{X}$ muni du coefficient et de l'autre le sous-monôme en $\mathbf{Y}$.

- Puis les deux sous-monômes sont insérés respectivement dans deux `MonoLists` (une seule fois pour chaque sous-monôme) afin de les ordonner

- Le sous-monôme en $\mathbf{Y}$ est ensuite passé la fonction de hachage pour obtenir l'indice ky d'entrée dans la table.

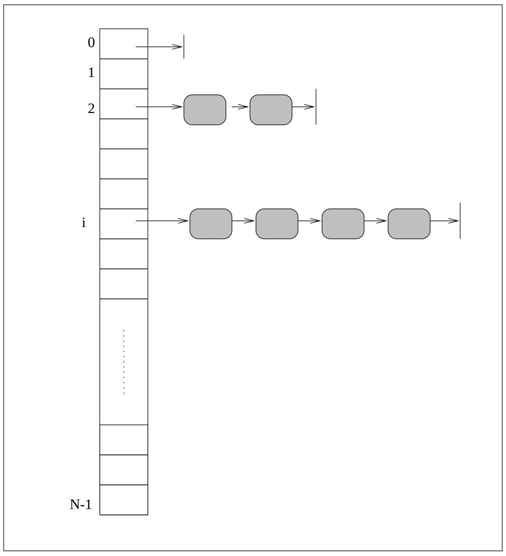


FIG. 4.3 – *Table de hachage implémentée en tableau de listes chaînées*

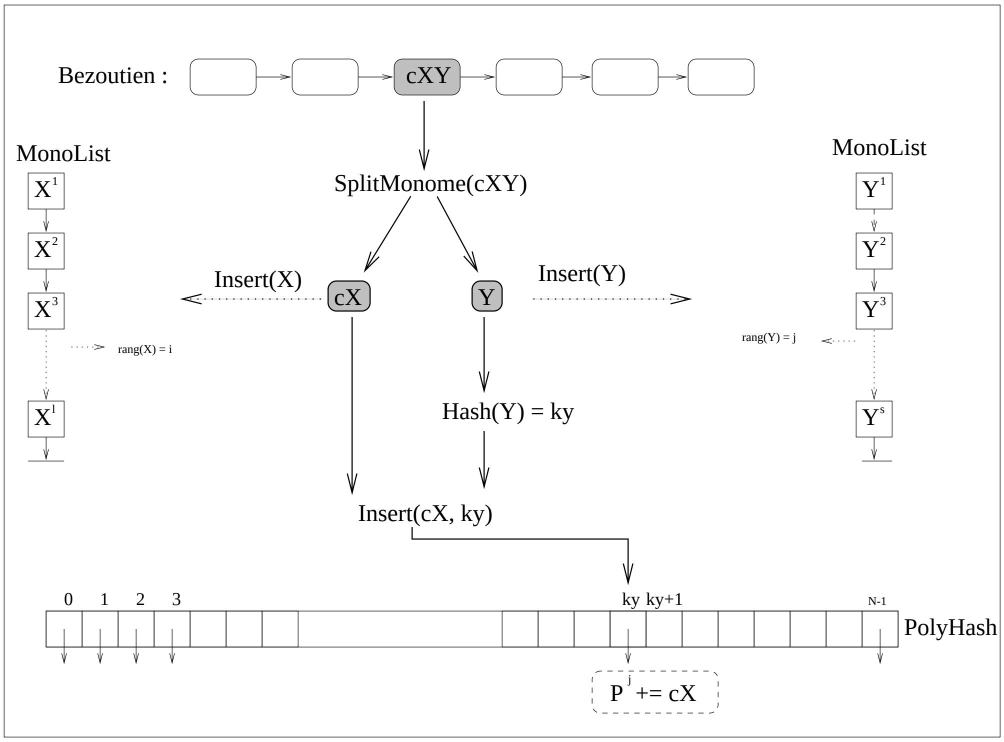


FIG. 4.4 – *Schéma de Construction de la matrice du Bézoutien*

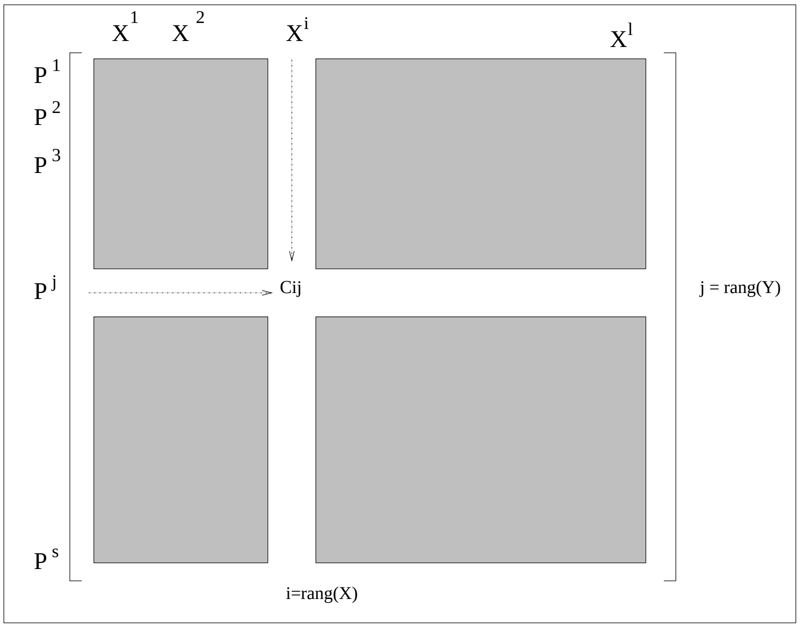


FIG. 4.5 – *Construction de la matrice du Bézoutien*

- Le sous-monôme $c\mathbf{X}$ est alors ajouté au polynôme situé à l'adresse ky de la table.
- A la fin du parcours, la méthode `MonoList::Mark()` est invoquée pour positionner les rangs des différents sous-monômes des deux listes. Ce marquage permet de fixer sur les sous-monômes $\mathbf{X^i}$ et $\mathbf{Y}^j$ les numéros des lignes et des colonnes de la matrice dont l'entrée correspondante est le coefficient c_{ij}.
- Puis on récupère les dimensions de la matrice en appliquant la méthode `MonoList::Length()` aux deux listes. On peut alors déclarer la matrice en invoquant le constructeur par défaut de la classe `Matrix` qui fournit une matrice initialisée avec des coefficients nuls.
- Enfin la `MonoList` des $\mathbf{Y}$ (ordonnée par construction) est parcourue et la fonction de hachage appliquée à chacun de ses éléments , la valeur de hachage obtenue permet d'accéder directement au polynôme $\mathbf{P^j}$ correspondant dont chaque coefficient c_{ij} est ensuite rangé dans la matrice (fig4.4) à la place indiquée là encore par la position du monôme $\mathbf{X^i}$ sur la `MonoList` des $\mathbf{X}$.

Remarque importante: les objets étant passés *par référence* aussi bien dans les `MonoLists` que dans la table de hachage, il ne sont pas dupliqués contrairement l'apparence. On pourrait, eneffet, croire que les différentes insertions conduisent à des copies, mais en fait tous les sous-monômes demeurent physiquement dans le polynôme de départ (le Bézoutien) ce qui permet un stockage optimal.

Cas d'une matrice carrée

On constate dans cette construction que la matrice du Bézoutien en plusieurs variables ne conduit pas nécessairement à des matrices carrées, mais on peut tout aussi en extraire des sous-matrices donnant des informations utiles à la résolution du système. Une méthode `Polybase::Determinant(Type&)` est toutefois fournie qui retourne un booléen renseignant sur la nature (carrée ou pas) de la matrice et calcule, le cas échéant, la valeur du résultant qui est ensuite mis dans une variable dont l'adresse est passé en argument.

Une modification utile a été faite dans la classe `SquareMatrix` pour rendre possible le calcul du Résultant dans le cas où la matrice obtenue est carrée. Il n'était pas possible, en effet, d'appliquer la fonction de calcul du déterminant à un objet de type `Matrix` (matrice nxp, n et p quelconques). En effet cette méthode ne s'applique qu'à des matrices de type `SquareMatrix`, et il n'y a pas de conversion implicite entre ces deux type; or dans la construction du Bézoutien, c'est une matrice de dimensions quelconque qui est déclarée. Donc il nous fallait trouver un moyen de passer d'une matrice quelconque à une matrice carrée dans le cas où la matrice du Bézoutien est de ce type. Pour ce faire nous avons ajouté un nouveau constructeur à la classe `Squarematrix` avec comme argument un objet `Matrix` dont le nombre de ligne est égal au nombre de colonnes. On peut ainsi calculer le déterminant de la matrice de Bézout (le Résultant) dans le cas carré.

Exemple

Voici un exemple de construction réalisée avec notre implémentation.

Polynômes :

$$\begin{aligned} f_0 &= x_3 + x_2 + x_1 \\ f_1 &= x_2x_3 + x_1x_3 + x_1x_2 \\ f_2 &= -1 + x_1x_2x_3 \\ f_3 &= 1 \end{aligned}$$

Bezoutien :

$$\Phi(f_0, f_1, f_2, f_3) = -x_4^2x_5 + x_3x_4x_5 + x_3x_4^2 - x_3^2x_4 + x_2x_4x_5 - x_2x_3x_5 - x_2x_3x_4 + x_2x_3^2$$

Matrice de Bézout

$$[\Phi(f_0, f_1, f_2, f_3)] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

4.3.5 Matrice de Macaulay

Bien que la classe `MonoList` ait été conçue pour les besoins de la construction de Bézout, elle s'est avérée très utile dans celle de Macaulay. Quelques extensions ont toutefois été ajoutées pour réaliser cette construction. Elles concernent, entre autres, une méthode de transformation de la liste en un tableau de monômes, afin de les récupérer facilement "de l'extérieur". Nous l'avons préféré ainsi pour les raisons de simplicité, déjà évoquées, que nous avons souhaitées sur cette classe. En effet, une autre façon de récupérer les éléments de la liste sans parcours interne serait de la doter de méthode d'accès, ce qui nécessite la définition d'un classe d'itérateur que nous avons voulu éviter.

Une méthode `Macaulay::MacaulayBasis(MonoList&)`, dont nous verrons l'utilité, a également été ajoutée à la classe.

Construction

La construction de Macaulay, elle aussi, a fait l'objet d'une méthode dans la classe `Polybase`, la méthode `Polybase::Macaulay` et d'une méthode de calcul du Résultant qui se contente simplement d'invoquer la fonctiom **LAPACK** de calcul du déterminant par la classe `SquareMatrix`.

Les $n+1$ polynômes $f_0, f_1, .., f_n$ de degrés respectifs, $d_0, d_1, .., d_n$ étant mis dans une `Polybase`, on commence par générer l'ensemble des monômes de degré $\nu = \sum(d_i) - n$ et de taille $n+1$ (voir définition de la matrice de Macaulay). Cette opération est faite grâce à la méthode `Choose(int, int, MonoList&)`. Ces monômes, donnés dans une `MonoList` sont ensuite passés à la méthode `MonoList::MacaulayBasis(...)` qui donne un tableau de $n+1$ autres listes de monômes, soit $N_0, N_1, .., , N_n$. Ensuite pour chaque $i = 0, .., n$ les éléments de la liste N_i sont tour à tour multipliés par le polynôme f_i, ce qui donne les coefficients du bloc i de la matrice.

Exemple

Voici un exemple de construction avec la méthode `Polybase::Macaulay()`

Polynômes :

$$\begin{aligned} f_0 &= x_3^2 + x_2^2 + x_1 x_2 \\ f_1 &= 2x_2 x_3 + 2x_1 x_2 \\ f_2 &= 3x_3 + 3x_2 + 3x_1 \end{aligned}$$

$$\nu = 2 + 2 + 1 - 2 = 3$$

Liste des monômes de taille 3, de degré 3 :

$$\mathcal{M} = \{x_3^3, x_2 x_3^2, x_2^2 x_3, x_2^3, x_1 x_3^2, x_1 x_2 x_3, x_1 x_2^2, x_1^2 x_3, x_1^2 x_2, x_1^3\}$$

Listes auxiliaires :

$$\mathcal{N}_{\prime} = \{x_3, x_2, x_1\}$$

$$\mathcal{N}_\infty = \{x_3, x_2, x_1\}$$
$$\mathcal{N}_\in = \{x_2^3, x_2x_3, x_1x_3, x_1x_2\}$$

Matrice de Macaulay :

$$M(f_0, f_1, f_2) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 & 3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 & 2 & 0 & 0 & 3 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 3 & 0 & 3 & 0 \\ 1 & 0 & 0 & 2 & 0 & 2 & 0 & 3 & 3 & 3 \\ 0 & 1 & 1 & 0 & 2 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

4.3.6 Conclusion

La construction des differentes matrices, bien que partant d'un formalisme assez complexe, devient simple à implémenter dès que l'on dispose objets et de structures de données adéquates. Cette modèlisation a été la partie la plus éxigente aussi bien en terme d'interprétation des données théoriques qu'en terme de créativité. Il a fallut comme on l'a vu adopté une méthodologie qui fasse appel à l'immagination. Cet aspect a été particulièrement intéressant et servira cerатainement dans d'autres contextes.

Chapitre 5

Problèmes et Perspectives

Ce stage, qui portait sur l'implémentation d'outils logiciels utilisant un langage de programmation *haut niveau* plutôt qu'un système éprouvé et validé tel que *Maple* ou *Mathematica*, a été l'occasion pour moi d'appréhender certains des problèmes inhérents à la tâche ardue de développement.

Je peux citer parmi ces problèmes, dont certains ont constitué pour moi une exprience enrichissante (mais aussi un véritable calvaire!), 3 principaux :

- La poursuite d'un travail qui a été entammé par un autre programmeur.
- La necéssité de construire, soi-même, de nombreux objets dans un cadre global.
- Une nécessaire maîtrise, à la fois, de l'outil de programmation et des fondements théoriques des objets à implémenter.

Ce sont là quelques uns des inombrables problèmes auxquels j'ai été confrontés au cours de ces cinq mois de stage. Ce fut également l'occasion de faire un riche apprentissage du dur métier programmeur.

5.1 Problèmes liés à la librairie

Le développement de la librairie **ALP** avait débuté en 1996 avec le stage de F. Livigni qui, je tiens à le souligner, avait effectué un excellent travail. Le manuel de Référence qu'il avait alors produit a été pour moi une aide très précieuse dans la poursuite de la tâche qu'il avait commencée. La conception de la librairie s'est faite d'une façon très cohérente et claire, exploitant à fond les possibilités du langage **C++**, ce qui m'a permis de travailler sur des objets faciles à manier et à inserer dans de nouveaux contextes.

Il n'est malheureusement pas rare, dans l'utilisation d'un logiciel, ou d'une simple librairie de classes, de rencontrer des problèmes liés à la conception même des objets qu'ils proposent. C'est notamment le cas en ce qui concerne la librairie **ALP**. Certains constructeurs de `Monome`de la première version présentaient en effet un *bug* qui relève, en fait, plus de la portabilité que de véritables erreurs de programmation. En effet, les monômes étant implémentés typiquement dans des tableaux d'entiers qui en représentent les exposants,lorsqu'un des constructeurs est invoqué, ces tableaux sont alloués à l'aide de l'opérateur `new` qui se contente de fournir un espace mémoire suffisamment large pour contenir les données spécifiées. Mais, cet espace alloué n'etant pas systématiquent occupé l'abcence d'intialisation (c'esr le cas dans tous les constructeurs de `Monome`) conduit à un comportement indéfini complètement à la merci du type de machine utilisé. Un exemple typique est celui de l'allocation d'un tableau de n entiers pour stocker le monôme x_n^k. Pour $n \geq 1$ les $n-1$ cases du tableau ne contiennent "rien" à priori. Donc si aucune initialisation n'a suivi l'appel à l'opérateur `new`, il se peut, dans son esprit, que le programmeur pense que les cases sont remplies de 0 par défaut. Ceci est vrai sur certaines machines, les SUN OS4 par exemple (pour ce qui nous concerne, *kama.inria.fr* et l'ancienne version de *korrigan.inria.fr*). Ceci n'est déjà plus vrai sur SUN SOLARIS (*psyche.inria.fr* et nouvelle version de *korrigan*). Elle l'est encore moins sur un PC tournant sous *LINUX*(*medee.inria.fr* et les machines de chez les *gnus!*) si bien qu'un monôme du type cité peut se comporter comme un véritable monôme de taille n dont les exposants, à l'éxécution, sont non seulement tous non nuls mais peuvent se retrouver avec dénormes valeurs dont une conséquence entre autre peut être de compromettre les calculs par des défaut de mémoire.

Le problème que je viens de décrire serait sans doute encore invisible si *korrigan* n'était pas passée sous SOLARIS, ou si je m'étais contenté de travailler exclusivement sur *kama*. J'ai en effet, eu l'idée de compiler la librairie sous linux moyennant quelques aménagements dans les *Makefile*s pour inclure les bonnes `libxxx.a`. Ceci avait le avantage de tourner plus vite que sur les autres machines ; et donc m'a séduit. Il s'en est suivi de nombreux problèmes d'execution dans nos programmes (`Segmentation fault, Memory fault,...`) que j'attribuais plusieurs semaines durant à mes propres codes, à tort. Ceci a eu pour conséquence une grande perte de temps. Cela avait particulièrement perturbé le calcul du Bézoutien et avait quelque peu bouleversé le programme de travail que je m'etais fixé.

Il a donc fallut revoir la structure interne des monômes sur lesquelles nous avons pu identifier et corriger ce *bug*. D'autres suivront, plus ou moins sérieux, mais ceci est le propre de tout logiciel.

D'autre part, les raisons ayant motivé la conception de cette librairie, à laquelle j'ai eu l'honneur de participer, ayant été principalement de se doter d'un outil permettant de traiter des problèmes par des approches nouvelles, non encore implémentées dans les systèmes usuels ; et ce, de façon indépendante de ces derniers, il s'imposait alors à nous de construire tous les objets, structures et opérations élémentaires dont on peut disposer de façon naturelle dans un système *prêt-à-utiliser*. Il en fut ainsi des structures complexes mais indispensables, telles que les listes chaînées qui décrivent les polynômes ; mais aussi des opérations aussi élémentaires que la division ou la multiplication de polynômes en plusieurs variables. J'ai été confronté à ce problème dès le début, notamment lors de l'implémentation de la construction de Dixon où j'ai dû adopter une solution temporaire pour éviter une division polynômiale que j'ai ensuite implémenté (lorsqu'elle était devenue une fatalité!). Là aussi, ce fut l'occasion d'apprendre à concevoir et à réaliser "ses propres outils".

5.2 Problèmes liés aux calculs polynômiaux

Le problème principal, dans la manipulation des systèmes polynomiaux, est la gourmandise des calculs, notamment intermédiaires, en mémoire. La recherche d'algorithmes, qui minimisent ces besoins est sans doute d'un intérêt incontestable quand on pense que nombre de calculs sont souvent compromis, même lorsque les sorties attendus sont éventuellement petites.

Le second problème pour l'étudiant en IUP (plutôt spécialisé *Proba-Stat*) que je suis, a été l'effort nécessaire à la maitrise de notions jusque là inconnues et qui relèvent de la recherche approfondie; une maitrise nécessaire à la bonne conduite de leur implémentation. Le plaisir d'acquérir des notions nouvelles en complément à ma formation initiale, et de participer à un travail de recherche tout en étant entouré de spécialistes dont la compétence est reconnue, ont été, à cet égard, un vrai stimulant pour moi.

5.3 Des idées pour améliorer la librairie

5.3.1 Un meilleur stockage des monômes

Le stockage des monômes tel qu'il est implémenté dans **ALP**, est sans doute perfectible. J'ai évoqué dans le chapitre 3 le gaspillage de mémoire qui consiste à stocker les exposants du monôme x_5^4 dans un tableau des cinq entiers: $\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{4}$.

Un idéee pour éliminer ce problème serait d'exploiter les propriétés de la représentation des entiers en multiple précision. Il est en effet possible de représenter les exposants d'un monôme dans un tel entier écrit dans une base suffisamment grande (selon les possibilités du matériel) pour permettre de traiter tous les monômes renc ontré dans la pratique. Ainsi pour une écriture en base 10 les exposants du monôme $x_1^2. * x_2^5. * x_3^3. * x_4^6. * x_6^7. * x_7^9.$ ne seront plus stockés dans le tableau d'entiers: $\longrightarrow \boxed{2}\boxed{5}\boxed{3}\boxed{6}\boxed{0}\boxed{7}\boxed{9}$ mais simplement dans le nombre 2536079. Il est clair qu'avec ce mode de stockage et suivant la qualité de la représentation des entiers dans la machine utilisée, le gain peut être considérable.

5.3.2 Un meilleur stockage des polynômes

Les polynômes sont stockés dans des listes chaînées. Ce choix implique des parcours séquentiels lors des itérations et des complexités de calcul au moins linéaires, et parfois quadratiques pour des opération élémentaires. Cela me semble améliorable, notamment en explorant de nouvelles structures de données; je pense prcisément aux arbres de type *AVL* qui permettent de gérer des données triées, et des accès logarithmiques aux éléments au moyen de transformations de clées semblables à celles des tables de hachage que nous avons utilisées dans le chapitre précédent.

On peut en effet envisager que les `Cellules` de `Polynome`, plutot que d'être contenues dans des listes chaînées, soient les noeuds d'un *AVL* dont on peut accéder en

$Log(n)$ en utilisant le monôme comme clé de la même façon que nous l'avions fait pour la table de hachage.

On pourrait aussi bénéficier d'un accès en temps constant en utilsant une table de hachage pour stocker un polynôme, au sacrifice de l'ordre des monômes car les tables de hachage ont la particulrité (le seul défaut?) de ne pas supporter le tri des éléments qu'il contiennent.De même peut-on imaginer une implémentataion combinant toutes ces structures, afin d'en tirer les avantages et d'en ecarter les inconvénients en fonction de besoins spécifiques. Notons par exemple que dans *Maple* les (monômes des) polynômes ne sont pas triés par défaut.

5.3.3 Limiter les copies de mémoire

Lors des *débogages* que nous avions faits pour corriger nos programmes, nous avions constaté que dans les constructeurs et les surcharges d'opérateurs de base, il se faisait énormément de copies de mémoire. Il s'agit de duplication inutile d'objets ou de copies temporaires à la sortie d'une fonction avant l'utilisation du résultat dans une autre variable. Ce phénomène est particulièrement visible avec l'utilisation du programme `gprof` d'analyse des *graphes d'appels* qui donne des indications précises sur le temps que le prgramme passe dans chaque fonction, blocs d'apels, sous-procédures, etc.

On gagnerait sans aucun doute en efficacité et à réduire le nombre de copie de mémoire et des appels système correspondants. Cette opération, suppose certainement une refonte au moins partielle de la structure des monômes.

5.3.4 Utilisation de *Template*s et de la librairie standard

La librairie **ALP** étant implémentée en **C++**, il serait naturel de la doter de l'un des fonctionnalités qui fait l'originalité de ce langage : le paramétrage par les *Template*s. Ceci nous permettrait par exemple de se doter de sturctures de matrices génériques aussi bien pour les scalaires réels, entiers, rationnels, complexes que pour les polynômes. Cela serait encore plus homogène par rapport à la notion de classe que nous avons voulu maintenir en intégrant le calcul du déterminant polynômial à la classe `Polybase`.

Enfin, la librairie standard du **C++** est d'une richesse qu'en général nombre de programmeurs ne soupçonnent. Il n'est pas rare, en effet, que l'on passe une bonne partie de son temps à concevoir et à programmer des objets (des classes) qu'une simple instruction du préprocesseur peut fournir en plus fiable et plus riche, directement importés de fichiers dont le chemin d'accès est constamment emprunté, par défaut, par la plupart des compilateurs.

Bibliographie

[1] D.N. Bernstein. The number of roots of a system of equations. *Funct. Anal. and Appl.*, 9(2):183–185, 1975.

[2] B. Buchberger. Applications of Gröbner bases in non-linear computational geometry. In D. Kapur and J. Mundy, editors, *Geometric Reasoning*, pages 415–447. M.I.T. Press, 1989.

[3] J.P. Cardinal and B. Mourrain. Algebraic approach of residues and applications. In J. Renegar, M. Shub, and S. Smale, editors, *Proc. AMS-SIAM Summer Seminar on Math. of Numerical Analysis, (Park City, Utah'95)*, volume 32 of *Lectures in Applied Math.*, pages 189–210. Am. Math. Soc. Press, 1996.

[4] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. Intern. Symp. Applied Algebra, Algebraic Algor. and Error-Corr. Codes, Lect. Notes in Comp. Science 263*, pages 89–104, Puerto Rico, 1993. Springer Verlag.

[5] A. Cayley. On the theory of elimination. *Dublin Math. J.*, II:116–120, 1848.

[6] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Undergraduate Texts in Mathematics. Springer, 1992.

[7] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra.* Academic Press, London, 1988.

[8] A.L. Dixon. The eliminant of three quantics in two independent variables. *Proc. of Lond. Math. Society*, 6:49–69,473–492, 1908.

[9] M. Elkadi and B. Mourrain. Approche Effective des Résidus Algébriques. Rapport de Recherche 2884, INRIA, 1996.

[10] F.S. Macaulay. Some formulae in elimination. *Proc. London Math. Soc.*, 1(33):3–27, 1902.

[11] F.S. Macaulay. *The Algebraic Theory of Modular Systems.* Cambridge Univ. Press, 1916.

[12] B. Mourrain and V. Y. Pan. Multidimensional structured matrices and polynomial systems. *Calcolo, Special Issue, workshop on Toeplitz matrices: Structure, Algorithms and Applications*, 1997.

[13] B.W. Kernigan and D.M. Ritchie Le langage C *Masson*, 1997.