E.S.S.I. - Option C.S.I

Projet de $3^{ième}$ année - Rapport de synthèse

Résolution de systèmes d'équations polynomiales et applications.

Hélène PRIETO

Encadreur(s): B. MOURRAIN

Organisme: Projet SAGA, INRIA

15 avril 1999

Résumé

Ce projet prend place dans l'étude des méthodes de résolutions d'équations polynomiales et des applications de ces méthodes à la robotique, la vision, le traitement du signal, la chimie... Pour cela, une bibliothèque C++, ALP (Algèbre Linéaire pour les Polynômes) est développée. Pour l'instant, celle-ci permet la manipulation de vecteurs, matrices (denses, structurées, creuses...) et de polynômes en une ou plusieurs indéterminées.
Un des soucis principal dans la conception de cette bibliothèque est l'efficacité, mais celle-ci ne doit pas nuire à la généricité et modularité des algorithmes. L'utilisateur dispose d'une grande liberté dans l'agencement des structures de données et dans le choix des algorithmes utilisés, mais la conception en est complexifiée.

Le premier objectif de ce projet a été de rechercher et mettre en place une structuration du code facilitant l'usage de la librairie par un utilisateur non-expert, en lui allégeant la mise en place de ses choix de structures, tout en gardant intacte la généricité et la liberté d'agencement. Cette recherche a fait l'objet de la première phase de ce projet. Pour ceci, diverses hiérarchies ont été expérimentées. Le principal souci étant de garder la notion de catégorie d'objets (par exemple, qu'une matrice, qu'elle soit pleine, creuse ou de Toeplitz, reste vue comme une matrice, et ceci sans perte d'efficacité ni de sécurité) permettant de fournir des implémentations génériques ou spécialisées.
L'objectif suivant de ce projet était de valider la phase précédente en utilisant et étendant certaines méthodes de résolution d'équations polynomiales de cette librairie, basées sur des techniques matricielles (valeurs propres, matrices de multiplication, résultants,...)

Table des matières

1 Contexte **5**
- 1.1 Environnement . . . 5
- 1.2 Cadre du projet . . . 5

2 Types de problème à résoudre **7**
- 2.1 Problème pratique industriel . . . 7
  - 2.1.1 Robot parallèle . . . 7
  - 2.1.2 Configurations de molécules . . . 9

3 Méthodes de résolution **13**
- 3.1 Différentes méthodes considérées . . . 13
  - 3.1.1 Réduction à un problème de recherche de valeurs propres . . . 13
  - 3.1.2 Méthode basée sur le résultant . . . 15
- 3.2 Choix de telles méthodes . . . 18

4 Implémentation **19**
- 4.1 Choix du C++ . . . 19
  - 4.1.1 Objectifs . . . 19
  - 4.1.2 Intérêts du C++ . . . 19
- 4.2 Structures de données . . . 20
  - 4.2.1 Approche . . . 20
  - 4.2.2 Architecture . . . 21

5 Les ingrédients des algorithmes **23**
- 5.1 Exemple de code générique . . . 23
- 5.2 Utilisation de librairie externe : GMP . . . 24
- 5.3 Généricité et efficacité . . . 25
- 5.4 Structure à deux étages : calcul de Bézoutien . . . 26
- 5.5 Mélange de calculs formels et numériques : les équations de Kruppa . . . 28
- 5.6 Connexion avec la librairie Lapack . . . 29
- 5.7 Calcul de valeurs propres . . . 30

6 Conclusion **33**

A Principales modifications effectuées lors de ce projet **34**
- A.1 Vecteurs . . . 34
- A.2 Matrices . . . 35
- A.3 Polynômes univariable . . . 36

B Vecteurs **39**

C Matrices **42**

D Polynômes univariables **57**

E Quelques algorithmes **63**
E.1 Méthode de Sebastio e sylva 63
E.2 Méthode des puissances inverses implicites 64

F Exemples-tests **66**
F.1 Matrices structurées - Triangularisation matrice dense 66
F.2 Racines d'un polynôme multivariable 67

Chapitre 1

Contexte

1.1 Environnement

L'**I.N.R.I.A.**, Institut National de Recherche en Informatique et Automatique, est un établissement public à caractère scientifique et technologique (EPST) opérant sous la double autorité du ministère de la recherche et du ministère de l'industrie. Le site de Sophia-Antipolis emploie environ 400 personnes. Ce site a été fondé en 1983, et a développé des liens étroits avec d'autres laboratoires, mais aussi des industries, à travers l'Europe ou la région.

Le centre de Sophia-Antipolis comprend 20 projets et avant-projets se regroupant en 4 thèmes. Ce projet de $3^{ième}$ année d'E.S.S.I. s'est déroulé au sein du projet **SAGA**, **S***ystèmes* **A***lgébriques,* **G***éométrie et* **A***pplications*, travaillant pour le thème *Génie logiciel et calcul symbolique*, et dont le responsable scientifique est Jean-Pierre Merlet. La géométrie algébrique effective joue un rôle important dans un éventail très large de domaines d'applications, allant de la chimie à l'infographie en passant par la théorie des mécanismes, le traitement du signal et la vision par ordinateur. En pratique, les phases où interviennent ces problèmes polynômiaux sont souvent des points bloquants. Les objectifs du projet **SAGA** sont de combler les lacunes existant dans ce domaine, en agissant à quatre niveaux :

- en recherchant de nouveaux algorithmes en Algèbre et en Géométrie, en développant de nouvelles méthodes formelles et/ou numériques, en relation avec les applications.
- en dégageant des classes de problèmes de généricité relativement large et en mettant en place des algorithmes dédiés à ces classes de problèmes.
- en fournissant à l'utilisateur des outils allant de la mise en équations en passant par des outils d'analyse de la géométrie des solutions, jusqu'aux implantation d'algorithmes de résolution, dans des bibliothèques dédiées, modulaires, d'accès et d'usage faciles. L'efficacité des codes produits et leur validation sur de gros problèmes est une préoccupation majeure.
- en utilisant ces bibliothèques dans le cadre d'applications particulières soit en connexion directe avec les utilisateurs finaux soit dans des domaines prometteurs.

1.2 Cadre du projet

Ce travail prend place dans le projet européen **FRISCO (LTR 21.024)**, **FR***amework for* **I***ntegrated* **S***ymbolic-numeric* **CO***mputation* dédié au calcul polynomial, sous la forme d'un mélange de calcul symbolique et numérique. Il fait suite aux précédents travaux proposés depuis 1996 sur des sujets similaires. Les outils de travail sont des stations Sun, le language C++, et la librairie **ALP**.

Chapitre 2

Types de problème à résoudre

De nombreux problèmes en robotique, imagerie, traitement du signal,..., demande une résolution de systèmes d'équations polynomiales.

2.1 Problème pratique industriel

Nous présentons ici deux problèmes industriels et la formulation algébrique qui en découle. Dans la dernière section nous présentons, de plus, un exemple de résolution d'un système surdéterminé fourni par le domaine de la reconstruction en vision par ordinateur.

2.1.1 Robot parallèle

Voici une brève présentation du problème direct d'un manipulateur parallèle (appelé aussi plate-forme de Stewart ou main-gauche) et des équations algébriques associées.

Problème

On considère, sur un solide fixe S_X, 6 points fixes $(X_i)_{1\leq i\leq 6}$, et 6 points Z_i d'un solide mobile S_Z. Les articulations entre les deux solides S_X et S_Z sont des barres extensibles $(X_i,Z_i)_{1\leq i\leq 6}$ attachées aux solides par des joints sphériques.

La plateforme subit des petits déplacements en modifiant la longueur des barres. Pour plus de détails sur les robots parallèles, voir []. Dans la pratique, il est facile de mesurer les longueurs des articulations, mais beaucoup moins de déterminer la position du solide S_Z en fonction de

ces longueurs. Des études portant sur les contraintes algébriques et plus particulièrement sur l'application qui associe à un déplacement du solide S_Z, le carré de la longueur $\|X_i,Z_i\|$, plus précisemment, ont permis de mieux comprendre la géométrie du problème. Il a ainsi été montré qu'il n'y a pas plus de 40 positions possibles lorsqu'on connait les distances $\|X_i,Z_i\|$. Des travaux (c.f. [6]) ont montrés que ces 40 positions peuvent être réelles. Pour une présentation plus profonde de ces études, on peut lire [10], [12], [11].

Formulation algébrique

Cette formulation est décrite en détail dans [11].
Le problème est donc d'obtenir le nombre de positions possibles du robot lorsque l'on connait les longueurs des bras.
On note $\mathbb{K}$ ($\mathbb{C}$ en théorie et plutôt $\mathbb{R}$ en pratique) un corps, et $\mathbb{A}^n = \mathbb{K}^n$ l'espace affine de dimension n sur $\mathbb{K}$. On considère $\mathbb{A}^n$ comme un ensemble ouvert de l'espace projectif[1] $\mathbb{P}(\mathbb{K}^{n+1})$. On note $\mathcal{D}$ l'ensemble des déplacements de $\mathbb{A}^3$. C'est un ensemble algébrique de dimension 6 et on a $\mathcal{D} = SO_3 \times \mathbb{K}^3$, $\mathbb{K}^3$ où $\mathbb{K}^3$ est l'ensemble des translations et SO_3 est la variété[2] des rotations, laissant fixe l'origine.

Si (Y_i) est la position initiale du solide S_Y, on s'intéresse à l'application :

$$\psi \ : \ \mathcal{D} \rightarrow \ \mathbb{K}^6$$

$$D \mapsto (\|D.Y_i - X_i\|^2)_{1\leq i\leq 6}$$

L'espace D est aussi appelé *espace des configurations* et $\mathbb{K}^6$ l'*espace des observations.*

La variété SO_3 peut-être paramétrisée (à l'aide des quaternions) par :

$$\omega \ : \ \mathbb{P}^3 - V(a^2+b^2+c^2+d^2=0) \rightarrow SO_3$$

$$(a,b,c,d) \mapsto R = \frac{1}{a^2+b^2+c^2+d^2} \begin{vmatrix} a^2+b^2-c^2-d^2 & 2(bc-da) & 2(bd-ca) \\ 2(bc+da) & a^2-b^2+c^2-d^2 & 2(cd-ba) \\ 2(bd+ca) & 2(cd+ba) & a^2-b^2-c^2+d^2 \end{vmatrix}$$

où $V(a^2+b^2+c^2+d^2=0)$ est la variété définie par l'équation $a^2+b^2+c^2+d^2$. De plus, ω est bijective.
On note $T \in \mathbb{K}^3$, le vecteur de translation, $T = [T_1,T_2,T_3]$.

Nous allons considérer la fibre[3] "générique" de l'application ψ. Elle possède le même nombre de points (comptés avec leurs multiplicités) que la fibre de $\psi(\omega \times Id)$. Ce nombre est appelé le *degré* de l'application $\psi(\omega \times Id)$ et correspond aussi à la dimension suivante ([11] ':

Définition 2.1.1 *Le degré d de l'application $\psi(\omega \times Id)$.est la dimension du corps $\mathbb{K}(\|\omega(\frac{a}{d},\frac{b}{d},\frac{c}{d}.Y_i + T - X_i\|^2_{a\leq i\leq 6})$ comme K-espace vectoriel, où $K = \mathbb{K}(\frac{a}{d},\frac{b}{d},\frac{c}{d},T_1,T_2,T3)$.*
Pour calculer explicitement ce degré, la proposition suivante est utilisée :

Proposition 1 *La fibre d'une valeur non critique est un ensemble algébrique de d points distincts sur lequel le jacobien de l'application considérée est de rang maximal.*

1. On appelle *espace projectif* de dimension n sur le corps $\mathbb{K}$, noté $\mathbb{P}^n(k)$, l'ensemble des classes d'équivalence de $\equiv$ sur $\mathbb{K}^{n+1} - \{0\}$, avec $(x'_0,\ldots,x'_n) \equiv (x_0,\ldots,x_n)$ si il existe $\alpha \in \mathbb{K}$ tel que $(x'_0,\ldots,x'_n) \equiv \alpha(x_0,\ldots,x_n)$.
2. Si $\mathbb{K}$ est un corps et si $f = (f_0,\ldots,f_s)$ sont des polynômes, on définit $V(f_1,\ldots,f_s) = \{(a_1,\ldots,a_n) \in \mathbb{A}^n(k) \ : \ f_i(a_0,\ldots,a_n) = 0$ pour tout $1 \leq i \leq s$ comme la *variété algébrique* définie par $f = (f_0,\ldots,f_s)$.
3.

Les valeurs non-crtiques sont dans le complémentaire de l'image de ... critique, qui est un ouvert dense, sous-ensemble de l'espace des observations. Donc en prenant une valeur au hasard dans ce dernier espace, il est assuré qu'elle ne sera pas une valeur critique. La multiplicité des points distincts des fibres aux valeurs non critique est donc de 1. Comme le nombre de points, compt'es avec multiplicité, est le degré d de l'application, il y a exactement d points distincts dans un fibre générique de ψ. Ceci permet de calculer le degré de ψ comme le degré de l'idéal d'efinissant une fibre de ψ.

Sont présentées ici les équations établies directement dans un espace de dimension 6, en utilisant une paramétrisation des rotations par des quaternions.
En se donnant 6 carrés des longueurs δ_i et les points (X_i), (Y_i), il faut déterminer combien de solutions $D = (R,T)$ satisfont le système :

$$\begin{aligned}[R.Y_i + T - X_i, R.Y_i + T - X_i] &= \delta_i \\ = \quad & [Y_i,Y_i] + [T,T] + [X_i,X_i] \\ & +2\,[R.Y_i,T] - 2\,[X_i,T] - 2\,[R.Y_i,X_i]\end{aligned}$$

pour $1 \leq i \leq 6$. Comme les points (Y_i) correspondent aux positions initiales du solide S_Z, nous pouvons supposer sans perte de généralité que $Y_6 = X_6 = 0$ De plus, on note $X_6 = X_0 = [0,0,0]$, $Y_6 = Y_0 = [0,0,0]$, $Z_0 = Z_6$ ... On a donc $[T,T] = \delta_6 = \alpha_0$, ce qui permet d'obtenir le système équivalent :

$$\left\{\begin{array}{l} [T,T] = \alpha_0 \\ [R.Y_1,T] - [X_1,T] - [R.Y_1,X_1] = \alpha_1 \\ [R.Y_2,T] - [X_2,T] - [R.Y_2,X_2] = \alpha_2 \\ [R.Y_3,T] - [X_3,T] - [R.Y_3,X_3] = \alpha_3 \\ [R.Y_4,T] - [X_4,T] - [R.Y_4,X_4] = \alpha_4 \\ [R.Y_5,T] - [X_5,T] - [R.Y_5,X_5] = \alpha_5 \end{array}\right. \tag{2.1}$$

Les valeurs α_i sont des combinaisons linéaires des carrés des longueurs initiales $(\delta_i)_{1\leq i\leq 6}$, par exemple $\alpha_1 = \delta_1 - \delta_6$.

2.1.2 Configurations de molécules

Une branche récente de la biologie et de la chimie algorithmique cherche à utiliser des algorithmes venant de la géométrie ou de la robotique en les appliquant à la recherche de configuration de molécules. Ici nous présentons le cas d'une molécule à 6 atomes, pour plus de détails on peut lire [8].

Problème

On considère des molécules cycliques à 6 atomes :

Ceci est une configuration possible d'une telle molécule. Le parallèle avec la robotique est évident si on considère les atomes comme des articulations et les liens entre eux comme des joints rigides. Les longueurs des liens et les angles entre deux liaisons consécutives sont considérés fixés. Seuls sont autorisés à varier les angles dihédraux, c'est-à-dire les angles solides formés par deux plans consécutifs, chacun étant défini par un atome et ses deux liens. Le seul mouvement permis est la rotation autour des axes des liaisons. En termes géométriques, la molécule est équivalente à un mécanisme série dans lequel tout couple d'axes consécutifs s'intersectent à un atome. Ceci nous permet de réduire le problème 6-dimensionnel à un système de 3 équations à trois inconnues.

Formulation algébrique

La molécule a une structure cyclique de 6 atomes, typiquement de carbone, ce qui détermine la structure primaire. Ce qui est en dehors de cette structure est ignorée. Les longueurs des liens et les angles qu'ils forment donnent les contraintes, les angles dihédraux pouvant varier.

Deux formulations peuvent être envisagées, l'une utilisant les angles, l'autre les distances. Seule la première est présentée ici.

La première donne le système polynomial suivant :

$$\alpha_{11} + \alpha_{12}cos\theta_2 + \alpha_{13}cos\theta_3 + \alpha_{14}cos\theta_2cos\theta_3 + \alpha_{15}sin\theta_2sin\theta_3 = 0 \quad (2.2)$$

$$\alpha_{21} + \alpha_{22}cos\theta_3 + \alpha_{23}cos\theta_1 + \alpha_{24}cos\theta_3cos\theta_1 + \alpha_{25}sin\theta_3sin\theta_1 = 0 \quad (2.3)$$

$$\alpha_{31} + \alpha_{32}cos\theta_1 + \alpha_{13}cos\theta_2 + \alpha_{14}cos\theta_1cos\theta_2 + \alpha_{35}sin\theta_1sin\theta_2 = 0 \quad (2.4)$$

$$cos^2\theta_1 + sin^2\theta_1 = 1 \quad (2.5)$$

$$cos^2\theta_2 + sin^2\theta_2 = 1 \quad (2.6)$$

$$cos^2\theta_3 + sin^2\theta_3 = 1 \quad (2.7)$$

où les α_{ij} sont les coefficients d'entrée.

Il peut être intéressant de travailler sur une formulation équivalente mais comportant moins d'équations. Ceci peut se faire en appliquant la transformation standard aux demi-angles, qui donne des équations rationnelles en de nouvelles inconnues t_i :

$$t_i = tan\frac{\theta_i}{2} \ : \ cos\theta_i = \frac{1 - t_i^2}{1 + t_i^2},\ sin\theta_i = \frac{2t_i}{1 + t_i^2},\ i = 1,2,3.$$

Ceci prend automatiquement en compte les trois dernières équations, et en multipliant les deux côtés de la i$^{\text{ème}}$ équation par $(t + t_j^2)(1 + t_k^2)$, où (i,j,k) est une permutation de 1,2,3, on obtient le système suivant :

$$f_1 = \beta_{11} + \beta_{12}t_2^2 + \beta_{13}t_3^2 + \beta_{14}t_2t_3 + \beta_{15}t_2^2t_3^2 = 0 \tag{2.8}$$
$$f_2 = \beta_{21} + \beta_{22}t_3^2 + \beta_{23}t_1^2 + \beta_{24}t_3t_1 + \beta_{25}t_3^2t_1^2 = 0 \tag{2.9}$$
$$f_3 = \beta_{31} + \beta_{32}t_1^2 + \beta_{33}t_2^2 + \beta_{34}t_1t_2 + \beta_{35}t_1^2t_2^2 = 0 \tag{2.10}$$

où les β_{ij} sont des coefficients d'entrée.

Chapitre 3

Méthodes de résolution

Certaines méthodes comme des méthodes de minimisation, des itérations de Newton sont souvent utilisées, mais le résultat n'est pas toujours garanti. Une autre méthode connue, la méthode des bases de **Groebner**, appliquée à l'arithmétique flottante ne donne pas de résultats corrects et a un coût de calcul élevé.
Nous présentons ici des méthodes rapides, stables numériquement et s'appliquant avec des coefficients approchés.

3.1 Différentes méthodes considérées

Nous allons exposer deux approches de résolutions de systémes polynomiaux ayant un nombre fini de solutions. La première méthode consiste à se ramener à un problème d'algèbre linéaire matricielle, en l'occurence une recherche de valeurs propres. La deuxième méthode est basée sur l'exploitation des propriétés du résultant d'un système polynomial.

3.1.1 Réduction à un problème de recherche de valeurs propres

Nous commencons par rappeler comment résoudre un système d'équations polynomiales peut se ramener à un problème de calcul de valeurs propres (c.f. [1], [13]). Pour ceci, nous commencons par rappeler certaines définitions et introduire quelques notations.

Notations-Définitions

Soit $R = \mathbb{K}[x_1, \ldots, x_n]$ l'algèbre des polynomes en n variables $\mathbf{x} = (x_1, \ldots, x_n)$ sur un corps $\mathbb{K}$ que l'on supposera algébriquement clos. Soient $f_1, \ldots, f_m$, m polynômes de R définissant le système polynomial $f_1(\mathbf{x}) = 0, \ldots, f_m(\mathbf{x}) = 0$, et I l'idéal de R engendré par ces m polynômes ; $I = (f_1, \ldots, f_m)$.

Cette approche est basée sur l'étude de l'algèbre quotient $\mathcal{A} = \mathbb{K}[\mathbf{x}]/I$, que l'on suppose de dimension finie, D sur $\mathbb{K}$.

On note $\mathcal{Z}(I) = \{\zeta \in \mathbb{K}^n ; f_1(\zeta) = \ldots = f_m(\zeta) = 0\}$, l'ensemble des racines du système. L'algèbre $\mathcal{A}$ étant supposée de dimension finie sur $\mathbb{K}$, $\mathcal{Z}(I)$ est lui aussi fini. On peut donc écrire $\mathcal{Z}(I) = \{\zeta_1, \ldots, \zeta_d\}$ avec $d \leq D$.

A cet ensemble de racines nous associons un ensemble d'idempotents orthogonaux (c.f. [2]), $\mathbf{e}_1, \ldots, \mathbf{e}_d$ satisfaisant les propriétés suivantes :

$$\mathbf{e}_1 + \cdots + \mathbf{e}_d \equiv 1, \text{ et } \mathbf{e}_i\,\mathbf{e}_j \equiv \begin{cases} 0 \text{ si } i \neq j, \\ \mathbf{e}_i \text{ si } i = j, \end{cases}$$

et tels que, si $I = Q_1 \cap \cdots \cap Q_d$ est la décomposition primaire minimale de I, on ait $\mathbf{e}_i\mathcal{A} \sim R/Q_i$, où $\mathcal{A}_i = \mathbf{e}_i\,\mathcal{A}$ est une algèbre locale d'idéal maximal $\mathbf{m}_{\zeta_i}$ définissant la racine ζ_i du système considéré. De plus, ceci implique que $\mathcal{A} = \mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_d$..

On note $\widehat{R}$ l'espace dual de R, et $\widehat{\mathcal{A}}$, le dual de $\mathcal{A}$ défini par $\widehat{\mathcal{A}} = \{\Lambda \in \widehat{R},\ \Lambda(I) = 0\}$ (parfois noté $I^{\perp}$).
Nous pouvons maintenant introduire l'opérateur de multiplication et la matrice associée dont les valeurs propres ont des propriétés intéressantes.

Pour tout $a \in \mathcal{A}$ on pose :

$$\begin{aligned} M_a : \mathcal{A} &\to \mathcal{A} \\ b &\mapsto a\,b \end{aligned}$$

qui défini l'opérateur de multiplication par a dans $\mathcal{A}$, et

$$\begin{aligned} M_a^{\mathtt{t}} : \widehat{\mathcal{A}} &\to \widehat{\mathcal{A}} \\ \Lambda &\mapsto a \cdot \Lambda \end{aligned}$$

l'application $\mathbb{K}$-linéaire induite par M_a.

Nous savons ([2], théorème 2.1) qu'il existe une base de $\mathcal{A}$ dans laquelle, pour tout $a \in \mathcal{A}$, la matrice $\mathtt{M}_a$ a la forme :

$$\mathtt{M}_a = \begin{pmatrix} \mathtt{M}_{a,1} & & 0 \\ & \ddots & \\ 0 & & \mathtt{M}_{a,d} \end{pmatrix}$$

où les $M_{a,i}$ sont les matrices de multiplication par a dans les algèbres locales $\mathcal{A}_i$ et ont la forme suivante :

$$\mathtt{M}_{a,i} = \begin{pmatrix} a(\zeta_i) & & * \\ & \ddots & \\ 0 & & a(\zeta_i) \end{pmatrix}$$

Dans le cas d'une racine simple, on a $\mathtt{M}_{a,i} = (a(\zeta_i))$.

Pour tout $\zeta \in \mathbb{K}^n$, on introduit la forme linéaire de $\widehat{\mathcal{A}}$

$$\begin{aligned} \mathbf{1}_\zeta : R &\to \mathbb{K} \\ p &\mapsto p(\zeta) \end{aligned}$$

qui est l'évaluation en ζ. Nous pouvons remarquer que $\mathbf{1}_\zeta \in \widehat{\mathcal{A}}$ si et seulement si $\zeta \in \mathcal{Z}(I)$. Dans le cas de racines multiples, il serait nécessaire d'introduire des formes linéaires supplémentaires invoquant des termes différentiels. Pour des raisons de simplicité, nous nous restreindrons dans cette présentation au cas de racines simples.
Voici maintenant quelques remarques et propositions :

Pour tout couple a,$b \in \mathcal{A}$,

$$M_a^{\mathtt{t}}(\mathbf{1}_{\zeta_i})(b) = \mathbf{1}_{\zeta_i}(a\,b) = a(\zeta_i)b(\zeta_i) = a(\zeta_i)\mathbf{1}_{\zeta_i}(b),$$

et $\mathbf{1}_{\zeta_i}$ est un vecteur propre de $M_a^{\mathtt{t}}$, associé à la valeur propre $a(\zeta_i)$.
De plus, pour tout couple $a,b \in \mathcal{A}$ les opérateurs de multiplication commutent entre eux, et ont des espaces propres communs. Ceci nous mène à la propriété suivante (c.f. [2]) : *les vecteurs propres communs aux M_a^t pour $a \in \mathcal{A}$ sont les multiples non nuls des $\mathbf{1}_{\zeta_i}$, for $i = 1,\ldots,d$.*

Pour une racine simple, ζ_i, le vecteur propre de $M_a^{\mathtt{t}}$ associé à la valeur propre $a(\zeta_i)$ est $\mathbf{1}_{\zeta_i}$ dont les coordonnées dans la base duale d'une base monomiale $(\mathbf{x}^{\alpha_1},\ldots,\mathbf{x}^{\alpha_D})$ de $\mathcal{A}$ sont $(\zeta_i^{\alpha_1},\ldots,\zeta_i^{\alpha_D})$, c'est à dire l'évaluation de cette base de $\mathcal{A}$ en la racine ζ_i.

Ceci nous donne l'algorithme suivant pour le calcul de racines simples d'un système polynomial.

3.1.1. **Algorithme de calcul des racines simples d'un système polynomial $f_1 = \cdots = f_m = 0$.**

1. Calculer la transposée de la matrice de multiplication $\mathtt{M}_a$ pour $a \in \mathcal{A}$.
2. Calculer ses valeurs propres $\mathbf{v}_i = (v_{i,1}, v_{i,x_1}, \ldots, v_{i,x_n}, \ldots)$ for $i = 1,\ldots,d$.
3. Pour $i = 1,\ldots,d$, calculer
$$\zeta_i = (\frac{v_{i,x_1}}{v_{i,1}}, \ldots, \frac{v_{i,x_n}}{v_{i,1}}).$$

Ce qui nous donne les racines simples du système, ce qui en pratique est souvant suffisant.

3.1.2 Méthode basée sur le résultant

Comme dans la section précédente, nous considérons $R = \mathbb{K}[x_1,\ldots,x_n]$ l'algèbre des polynomes en n variables $\mathbf{x} = (x_1,\ldots,x_n)$ sur un corps $\mathbb{K}$, $f_1,\ldots,f_m, m, m$ polynômes de R définissant le système polynomial $f_1(\mathbf{x}) = 0,\ldots,f_m(\mathbf{x}) = 0$ et $\mathcal{Z}(I) = \{\zeta \in \mathbb{K}^n; f_1(\zeta) = \ldots = f_m(\zeta) = 0\}$, l'ensemble des racines du système. Nous commencons par donner rapidement quelques résultats de la théorie des résultants ainsi que quelques propriétés du **Bézoutien** dans le cas général, où $\mathcal{Z}(I)$ n'est pas forcément fini. Puis nous présenterons une méthode permettant, à partir de ce cas général, de calculer les racines dans le cas fini.

Résultants

Tout d'abord, quelques définitions (pour plus de détails, voir [8] :

Définition 3.1.1 *Les variétés projectives définies par une seule équation homogène sont appelées* hypersurfaces.
Soient $n+1$ polynômes

$$\left\{ \begin{array}{rcl} f_0(\mathbf{x}) & = & \sum_{j=0}^{k_0} c_{0,j}\, \psi_{0,j}(\mathbf{x}) \\ & \vdots & \\ f_n(\mathbf{x}) & = & \sum_{j=0}^{k_n} c_{n,j}\, \psi_{n,j}(\mathbf{x}) \end{array} \right.$$

où

- $\mathbf{c} = (c_{i,j})$ sont des paramètres
- $\mathbf{x}$ est un point de la variété projective $X \subset \mathbb{P}^N$, de dimension n,
- Les fonctions $\psi_{i,j}(\mathbf{x})$ $(j = 1,\ldots,k_i)$ sont des polynômes homogènes, indépendant des paramètres $\mathbf{c}$, et sont des fonctions régulières sur X.
- Pour des valeurs génériques de c, le système n'a pas de solution sur X.

Le problème de la théorie de l'élimination est, dans le cas décrit ci-dessus, de trouver des conditions nécessaires et suffisantes sur les paramètres $\mathbf{c} = (c_{i,j})$ pour que les équations $f_1 = 0, \ldots, f_{n+1} = 0$ aient une racine commune dans X. Dans le cas classique, les $\psi_{i,j}(x)$ sont tous les monômes de degré d_i et $X{=}\mathbb{P}^n$. La condition nécessaire et suffisante sur les $\mathbf{c} = (c_{i,j})$ pour que les polynômes homogènes $f_0, \ldots, f_n$ aient une racine commune dans $X = \mathbb{P}^n$ est $\mathrm{Res}_{\mathbb{P}^n}(f_1, \ldots, f_{n+1}) = 0$ où $\mathrm{Res}_{\mathbb{P}^n}$ est le *résultant projectif classique.*
D'un point de vue géométrique, nous regardons l'ensemble des paramètres $\mathbf{c} = (c_{i,j})$ tels qu'il existe $x \in X$ avec $\sum_j c_{i,j}\psi_{i,j}(\mathbf{x}) = 0$ pour $i = 1, \ldots, n+1$. C'est à dire que $\mathbf{c}$ est la projection du point $(\mathbf{c},\mathbf{x})$ de la *variété incidente*

$$W_X = \{(\mathbf{c},\mathbf{x}) \in \mathbb{P}^{k_1} \times \cdots \times \mathbb{P}^{k_{n+1}} \times X \text{ s.t. } \sum_{j=0}^{k_i} c_{i,j}\psi_{i,j}(\mathbf{x}) = 0 \ ; \ i = 1, \ldots, n+1\}.$$

On note

$$\begin{aligned} \pi_1 : W_X &\rightarrow \mathbb{P}^{k_1} \times \cdots \times \mathbb{P}^{k_{n+1}}, \\ \pi_2 : W_X &\rightarrow X. \end{aligned}$$

les deux projections naturelles. L'image de W_X par π_1 est précisemment l'ensembles des paramètres pour lesquels le système a une racine. L'image par π_2 d'un point de W_X est une solution dans X du système associé. On a alors la définition suivante (voir [8]) :

Définition 3.1.2 *Soit $Z = \pi_1(W_X)$. Si Z est une hypersurface, alors son équation (unique à un facteur près) est appelée le* résultant *de $f_0, \ldots, f_n$ sur X. Il est noté* $\mathrm{Res}_X(f_0, \ldots, f_n)$.

Le système $f_0(x) = \ldots = f_n = 0$ a donc une solution dans X si et seulement si $\mathrm{Res}_X(f_0, \ldots, f_n) = 0$, ceci généralise la définition du résultant classique sur $\mathbb{P}^n$ à d'autres variétés projectives. Ceci sous certaines conditions sur la dimension de Z (voir [8]).

Les méthodes de calcul effectif du résultant s'appuient sur la construction de matrices dont le déterminant va nous fournir le résultant ou un multiple non-trivial. Ces matrices de résultants peuvent se regrouper en deux grandes familles : les matrices de type *Sylvester* généralisant la construction de Sylvester (1835, voir [14]) au cas multivariable et les matrices de type *Bézout* généralisant la construction d'E. Bézout (1779, voir [3]) au cas multivariable. Cette dernière construction est présentée dans le paragraphe suivant.

Bézoutien et résultant

On considère toujours les polynômes $f_0, \ldots, f_n$ de $\mathbb{K}[x]$, où $\mathbf{x} = (x_1, \ldots, x_n)$.
Voici quelques définitions et propriétés de la théorie des Bézoutiens (pour plus de détails voir [5, 8]). Ceci généralise la construction de E. Bézout au cas univariable.
On ajoute à l'ensemble des variables $\mathbf{x}$ un nouvel ensemble de variables $\mathbf{y} = (y_1, \ldots, y_n)$ et on note $\mathbf{x}^{(0)} = \mathbf{x}$, $\mathbf{x}^{(1)} = (y_1, x_2, \ldots, x_n)$, $\ldots$, $\mathbf{x}^{(n)} = \mathbf{y}$.
Pour un polynôme $q \in R$, on définit $\theta_i(p) = \frac{p(\mathbf{x}^{(i)}) - p(\mathbf{x}^{(i-1)})}{y_i - x_i}$.

Pour $n+1$ polynômes $f = (f_0, f_2, \ldots, f_n) \in R$, le *Bézoutien* de $f_1, f_2, \ldots, f_{n+1}$ est un polynôme en $\mathbf{x}$ et $\mathbf{y}$ défini par:

$$\Theta_f = \det \begin{pmatrix} f_0(\mathbf{x}) & \theta_1(f_0) & \cdots & \theta_n(f_0) \\ \vdots & \vdots & & \vdots \\ f_n(\mathbf{x}) & \theta_1(f_n) & \cdots & \theta_n(f_n) \end{pmatrix} = \sum_{i,j} \theta_{i,j}^f \, \mathbf{x}^{u_i} \, \mathbf{y}^{v_j}, \tag{3.1}$$

C'est un polynôme de degré au plus $\sum_{i=0}^{n} deg(f_i) - n$. On note $\mathtt{B}(f_0,\ldots,f_n) = (\theta_{i,j})$, la matrice des coefficients une fois les monômes ordonnés suivant l'ordre monomial choisi. Cette matrice est la *matrice de Bézout*, ou *Bezoutienne*, des polynômes $f = (f_0,f_2,\ldots,f_n)$.

Une importante propriété de cet objet est que, sous les conditions citées dans le paragraphe précédent, *tout mineur maximal de la matrice* $\mathtt{B}(f_0,\ldots,f_n) = (\theta_{i,j})$ *est un multiple du résultant* (c.f. [8]). Ceci permet donc d'obtenir l'équation de l'hypersurface solution du système $f_0(x) = \ldots = f_n = 0$.

Application des résultants au calcul de racines dans le cas fini

Pour appliquer une méthode de résultant, il est nécessaire de transformer le système polynomial $f_1(\mathbf{x}) = 0,\ldots,f_n(\mathbf{x}) = 0$ sur $R = \mathbb{K}[x_1,\ldots,x_n]$ en système sur-contraint. Une des manières de faire est de "cacher" une des variables dans le corps des coefficients. On aura ainsi n polynômes pour $n-1$ variables. Un calcul de résultant nous donnera une équation en la variable cachée qu'il nous faudra résoudre pour obtenir la valeur de cette variable en les racines. En calculant le noyau de la matrice résultant pour les valeurs trouvées on obtiendra les évaluations des racines en les autres variables. Retrouver les valeurs aux racines demande juste de diviser un coefficient du vecteur par un autre pour chacune des variables. Ce qui nous donne l'algorithme suivant. Ici la matrice de résultant considérée est la Bezoutienne du système.

3.1.2. **Algorithme de calcul des racines simples en utilisant le résultant**

1. Transformer le système $f_1(\mathbf{x}) = 0,\ldots,f_n(\mathbf{x}) = 0$ sur $R = \mathbb{K}[x_1,\ldots,x_n]$ en un système sur $R = \mathbb{K}[x_n][x_1,\ldots,x_{n-1}]$
2. Calculer la Bézoutienne $\mathtt{B}$ de ce système.
3. Extraire un mineur maximal $\Delta(x_n)$ de $\mathtt{B}$, et une sous-matrice $\mathtt{SB}$.
4. Résoudre $\Delta(x_n) = 0$, ce qui donne $(\zeta_{i,n},i = 1,\ldots,d)$.
5. Pour $i = 1,\ldots,d$, résoudre $\mathtt{SB}(\zeta_{i,n})X = 0$, qui donne $v_i,i = 1,\ldots,d$.
6. Pour $j = 1,\ldots,n-1$, calculer $\zeta_{i,j} = \frac{v_{i,l}}{vi,k}$ pour un certain k, et un certain l.

Les coefficients de la matrice sont des polynômes.
Afin de réduire les coûts de calcul, le rang de la matrice $\mathtt{B}$ est calculé après avoir donné une valeur aléatoire à x_n. Ce calcul de rang se fait en effectuant une décomposition QR de $\mathtt{B}$, ce qui nous permet d'isoler un mineur maximal de $\mathtt{B}$.

Le calcul de $\Delta(x_n)$ se fait, lui, sur les coefficients polynômes. Les valeurs de x_n en chaque racine sont retrouvées par interpolation et transformée de Fourier inverse. Les résolutions des systèmes $\mathtt{SB}(\zeta_{i,n})X = 0$ pouvant être envisagées par une svd. Les solutions sont déduites de X par normalisation.

Cette méthode n'est pas encore complètement implantée, mais tous les outils nécessaires à son application, qui sont les différents algorithmes de calcul et les structures de données, sont mis en place. Il reste à faire l'assemblage.

Une présentation de chaque étape de cette méthode est donnée par des exemples utilisant les fonctions nécessaires dans la partie **4.2**. Ce sont :

- Calcul de la Bézoutienne d'un système polynomial :
- Calcul du rang d'une matrice :
- Calcul d'un déterminant :
- Résolution d'une équation polynomiale univariable :

– Résolution d'un système par svd :

3.2 Choix de telles méthodes

Ces méthodes ont de multiples avantages. Tout d'abord elles sont, si les problèmes sont raisonnables, rapides et stables numériquement, puisqu'on a une présentation matricielle du problème. Elles ont ainsi l'avantage d'être applicables en arithmétique flottante. De plus, les méthodes basées sur le Bézoutien permettent de calculer les points isolés même si le problème a des composantes de dimension non nulle.

Ce dernier point peut-être illustré en reprenant le système présenté pour le problème de la molécule à 6 atomes que l'on défini par la matrice des β_{ij} :

$$\left(\begin{array}{|ccccc|} -\frac{\sqrt{3}}{2} & \frac{1}{2} & \frac{1}{2} & 2 & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} & \frac{1}{2} & 2 & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} & \frac{1}{2} & 2 & \frac{\sqrt{3}}{2} \\ -\frac{\sqrt{3}}{2} & \frac{1}{2} & \frac{1}{2} & 2 & \frac{\sqrt{3}}{2} \end{array}\right)$$

Ce n'est donc pas un système zéro-dimensionnel. En effet, c'est l'union d'une courbe de degré 4 dont la projection sur le plan des coordonnées sont définies respectivement par l'équation $p_1 = 0$, $p_2 = 0$, $p_3 = 0$, et de 16 points. Parmi ces 16 points, 4 points sont sur la droite $t_1 = t_2 = t_3$ et sont isolés, tandis que les 12 autres points sont immergés dans la courbe. Une méthode basée sur le Bézoutien permet de retrouver les points qui sont immergés dans la courbe.

Chapitre 4

Implémentation

La librairie **ALP** est implémentée en **C++**. Elle est dédiée à la manipulation d'objets en algèbre linéaire et polynomiale. Dans ce chapitre, nous commencons par expliquer les raisons du choix d'un langage tel que le **C++** illustré par quelques exemples. Puis nous donnons un apercu des structures de données.

4.1 Choix du C++

Le langage **C++** commence à être utilisé pour l'écriture de librairies mathématiques, non pas pour son caractère orienté objet mais pour la généricité qu'il offre. Nous expliquons ici pourquoi, en fonction de nos objectifs, ce langage est le plus approprié actuellement. Cet exposé est illustré par quelques exemples significatifs.

4.1.1 Objectifs

La librairie **ALP** a pour objectif d'être le plus générique et efficace possible. Pour cela elle doit pouvoir utiliser des structures de données et des implémentations paramétrées, qui peuvent être spécialisées et optimisées, et ceci sans modification de la librairie. Nous souhaitons pouvoir, par exemple, changer facilement le type des coefficients des polynômes ou simplement effectuer une spécialisation d'un algorithme pour une structure de données particulière.
De plus cette librairie doit pouvoir avoir des connexions avec des librairies externes telles que **Lapack** pour l'algèbre linéaire numérique, **gmp** pour l'arithmétique en grande précision, ou **umfpack** pour l'accès à des solveurs de systèmes linéaires creux.

4.1.2 Intérêts du C++

La facilité de surcharge des opérateurs prédéfinis permet de traduire un algorithme en langage **C++** en utilisant des notations assez intuitives, comme nous le montrons dans l'exemple d'inversion d'un polynôme dans la section suivante. Ce qui a le grand avantage de rendre le code clair et de repousser la frontière entre les types et fonctions définis par l'utilisateur et ceux prédéfinis du langage.
De plus le **C++** en tant que langage orienté objet permet le *polymorphisme*, mais il a été choisi, pour des raisons d'efficacité de ne pas utiliser de classes abstraites ni de fonctions virtuelles qui définissent un polymorphisme dynamique. En effet, l'appel d'une fonction virtuelle est une décision prise pendant l'exécution, et même si un bon compilateur minimisera autant que possible le coût d'une telle fonction, celui-ci devient non-négligeable lors d'appels intensifs. De plus une fonction virtuelle ne peut pas être déclarée en ligne, puisque le type de l'objet n'est pas connu à la compilation. C'est pour cette raison que la librairie **ALP** utilisent de nouvelles techniques,

inspirées de la **STL**, bibliothèque appartenant maintenant à la distribution standard **C++**, à base de *template*, ce qui permet un polymorphisme "statique". Elles sont utilisées pour guider la compilation afin d'obtenir un code efficace. Ces nouvelles techniques, appelées *template expressions*, permettent de construire des programmes **C++** meilleurs en performance que leurs analogue **FORTRAN** (c.f. [15]). Pour une analyse poussée de ces techniques de programmation en calcul scientifique on peut lire [7].

De plus le **C++** offre une grande facilité de connexion avec d'autres librairies, ce qui permet de les intégrer parfaitement dans les structures de données et les algorithmes. Un exemple d'utilisation de la librairie **Lapack** est donnée dans **??**, un exemple avec la librairie **gmp** dans **??**

4.2 Structures de données

4.2.1 Approche

La particularité de notre approche est que le choix de la bonne implémentation est effectué à la compilation, ceci étant du à l'utilisation de *templates*. Un point intéressant des nouvelles techniques exposées ci-dessus, les *template expressions*, se situe dans la compilation de certaines opérations telles que multiplication (ou addition,...) d'une matrice par une matrice, ou d'une matrice par un vecteur. En effet, plutôt que de calculer un objet contenant le résultat de l'évaluation ces opérations retournent un moyen de le calculer. Le calcul proprement dit se faisant uniquement lorsqu'il est nécessaire, comme lors d'un assignement ou d'une impression.
Voici un exemple de l'utilisation des expressions template pour l'addition des vecteurs :

```
v = v1 + v2 + v3;
```

où `v, v1, v3` sont des vecteurs (de type `Vector`). Les implémentations habituelles définiraient l'opérateur

```
Vector & operator+(const Vector & v1, Vector & v2);
```

de telle facon que l'instruction précédente produirait un vecteur temporaire, `vtmp1`, pour l'addition de `v1` et `v2`, un autre, `vtmp2` pour l'addition de `vtmp1` et `v3` puis l'operateur

```
Vector & operator=(const Vector & v);
```

serait utilisé pour assigner à `v` la valeur de `vtmp2`, en utilisant une boucle

```
for(index_type i=0; i<v.size(); i++) v[i]=vtmp2[i];
```

Dans ce type d'implémentation, deux vecteurs temporaires sont alloués et détruits, ce qui est une consommation de temps et d'espace mémoire, spécialement dans le cas de gros vecteurs.

Dans **ALP**, en utilisant les expressions template, nous avons l'opérateur

```
VectHdl<Plus<Vector,Vector> > & operator+(const Vector & v1, Vector & v2);
```

où `VectHdl<Plus<Vector,Vector»` est le type d'une structure de données contenant des références sur les deux vecteurs `v1, v2` mais qui ne calcule pas la somme de ces vecteurs. L'instruction `v1+v2+v3` considérée produit alors une donnée du type

```
VectHdl<Plus<Vector,VectHdl<Plus<Vector,Vector> > > >;
```

Ainsi, l'assignement

```
v = v1 + v2 + v3;
```

invoquera l'opérateur

```
Vector & operator=(const Vector & v);
```

qui est implémenté avec

```
for(index_type i=0; i<v.size(); i++) (*this)[i]=_elem(v,i);
```

Le point important est que le compilateur expandera l'appel à la fonction en ligne `_elem(v,i)` dans `v1[i]+v2[i]+v3[i]`, puisque le ième coefficient d'un élément `v` de type `VectHdl<Plus<...,...>` est la somme des ième coefficients des ses deux composants.

Une grande attention a aussi été donnée à la gestion de l'espace mémoire. Et les copies inutiles de gros objets ont été limitées en ayant des fonctions retournant des pointeurs, qui peuvent être facilement détruits. Pour ce dernier point des classes spéciales ont été implémentées, contenant un pointeur sur le résultat. Elles sont utilisées uniquement comme type de retour des fonctions. Ainsi, la copie d'un tel objet correspond juste à une opération sur un pointeur.

4.2.2 Architecture

La librairie comprend trois niveaux de structures :

- Les **conteneurs**, des petites classes correspondant à la représentation interne des objets.
- Les **interfaces** qui sont des classes généralement paramétrées par le type de conteneur et qui spécifie comment voir celui-ci.
- Les **catégories** qui sont des familles d'interfaces ou de classes partageant des propriétés communes. Elles ne nécessitent pas forcément une implémentation propre, mais correspondent à des propriétés de classes.

A titre d'exemple `UPoly<toeplitz<double> >` est un polynôme univariable dense, représenté de manière interne par le conteneur `toeplitz<double> >`. `MatStr<toeplitz<double>` est une matrice structurée de type Toeplitz avec les implémentations spécialisées correspondantes (comme, par exemple, une multiplication matrice-vecteur par FFT) sur le même conteneur. Alors que la première classe appartient à la catégorie tt UPOLYNOMIAL, la deuxième fait partie de la catégorie
MATRIX.

Chapitre 5

Les ingrédients des algorithmes

Nous présentons dans cette partie plusieurs exemples utilisant des fonctions qui sont utilisées par les algorithmes explicités precédemment. Nous en profitons pour souligner certaines particularités de l'implémentation.

5.1 Exemple de code générique

Nous proposons, ici, un exemple d'algorithme mettant en valeur la clarté et l'écriture intuitive de code.

Réciproque d'un polynôme
Cet algorithme est tiré de [4]. Il s'agit d'un algorithme de division, de complexité $nlog(n)$. Il permet, en particulier, de réduire la complexité de la résolution $T.X = V$, où T est une matrice de Toeplitz, triangulaire inférieure, d'un coût de $nlog^2(n)$ à $nlog(n)$. En effet, une fois les coefficients mis dans un vecteur t, l'inversion de T n'est plus que le calcul de l'inverse de t.

Nous supposons que $p[0] \neq 0$. Pour ce calcul, nous considérons l'équation $f(w) = P - \frac{1}{w} = 0$, où $P = p(z)$, $w = w(z) = \frac{1}{p(z)}$ sur l'anneau des séries formelles en z, et applique des itérations de Newton à cette équation. Nous obtenons ainsi : $w_0 = \frac{1}{p[0]}$, and $w_{j+1} = w_j(2 - Pw_j)$ où $j = 0,1,\ldots$
En réduisant les différents polynômes modulo $z^{2^{j+1}}$ à chaque pas, nous effectuons deux convolutions de deux couples de vecteurs de dimension au plus 2^{j+1} par pas.

Nous présentons ici une fonction générique, où le type générique R est vu comme un polynôme grâce au dernier argument passé.
Un des avantages de ce style de programmation réside dans le fait que tout type de données comportant les opérations élémentaires et des champs `index_type` et `coef_type` peut être passé, en particulier tout vecteur de la librairie **ALP**. L'utilisation de cette fonction se fait en voyant la structure de donnée comme un polynôme (UPOLYNOMIAL).

```
template <class R>
inline void reciprocal(R & w,const R & p, const UPOLYNOMIAL)
{
  typedef typename R::index_type index_type;
  typedef typename R::coef_type C;
```

```
  const index_type deg = p.degree()+1;
  R w0(deg),v(deg),xp(p);
  index_type K = (index_type) (log(p.degree()+1)/log(2)),j=1;
  C p0 = xp[deg-1];

  xp /= p0;
  w0 =xp/xp[0];
  v = w0;
  v *= C(-1.0);
  v[0]+=C(2);
  w=v/xp[0];
  w0=w;
  while (j <= K) {
    reduce(w0,pow(2,j+1));
    w=xp*w0;
    v =w;
    v*=C(-1.0);
    v[0]+=C(2);
    reduce(v,pow(2,j+1));
    w=w0*v;
    w0=w;
    j++;
  }
   reduce(w,deg);
   w/=p0;
}
```

Voici le programme principal appelant cette fonction, ainsi que la sortie :

```
#include "_UPoly.H"

int main(int argc, char** argv) {
  double u[]={-1,3,4,6,7,8};
  POL q(6,u), rec(9);

  cout << " q = " << q <<endl;
  reciprocal(rec,q);
  cout << " L'inverse de q : " << rec << endl;
}
```

La sortie donne :

```
     q = (-1)+(3)*x+(4)*x^2+(6)*x^3+(7)*x^4+(8)*x^5
     L'inverse de q : (-1)+(-3)*x+(-13)*x^2+(-57)*x^3+(-248)*x^4+(-1079)*x^5
```

5.2 Utilisation de librairie externe : GMP

Cet exemple se propose de résoudre un problème de calibration de robots parallèle en utilisant une arithmétique de grande précision. De plus, cette résolution se fait par un appel à l'implémentation générique de la fonction `Svd`, fonction qui sera utilisée dans l'implémentation de la méthode de résolution par le Bézoutien.

```
#include "_Matrix.H"
```

```
#include "_BFloat.H"
#include "linear/Svd.H"

#define Coef Scl<MP_FLOAT>
#define VECT VectStd<array1d<Coef> >
#define MATT  MatStd<array2d<Coef> >

int main (int argc, char **argv)
{
  MATT A("TestAlpSSerr");

int n= A.nbrow(),m=A.nbcol();
int mm=min(n,m);

MATT U(n,n), V(m,m);
VECT S(mm),Sol(m);

Svd(S,A,U,V);

Sol=V.col(m-1);
Coef c(Sol[0]);
Sol/=c;
cout << Sol << endl;
}
```

La solution obtenue est :

```
0.1e1, 0.9908069458630016923e0, 0.1010549694614195751e1, 0.1111111111109947332e1, 0.97244732
```

La solution théorique est :

```
 1.0000000000000000000e+00 9.9080694586312567651e-01 1.0105496946141032665e+00 1.11111111
```

5.3 Généricité et efficacité

Nous montrons ici un exemple de résolution d'une équation polynomiale en une variable. Il s'agit d'un polynôme de degré10^6, ce qui permet de souligner une certaine puissance dans la méthode utilisée. Celle-ci est la méthode de résolution de **Sebastio e Sylva** dérivée de la méthode des puissances. L'implémentation est une fonction générique qui sera utilisée dans le cadre de la méthode de résolution avec le Bézoutien pour le cas multivariable. inverses. Les différents calculs effectués par cette fonction utilisent une FFT pour les multiplications.

```
#include "util/basic.H"
#include "arithm/Arith.H"
#include "_UPoly.H"
#include "polynomial/UPolySpl.H"
#include "polynomial/Upar.H"
#include "algo/SolveUPol.H"
```

```
#include "CPUClock.h"

#define POL UPolySpl<upar<complex<double> > >
#define N 1000000


int main(int argc, char** argv) {
  complex<double> t[N];

  t[N-1]=complex<double>(1.0,0.0);
  for (unsigned int i = 0; i < N-1; i++) t[i] = complex<double> (rand()%100 *0.1,0.0);

  POL pol(N,t);
  complex<double> set, x0;

  x0 = complex<double> (0.4,0.6);
  CPUClock T;
  solve(set,pol,complex<double>(0.4,0.6),Sebastiao());
  T.stop();
  cout << "une racine : " << set << endl;
  cout << "Le temps mis est : " << T.time() << endl << endl;;
}
```

La sortie donne :

```
  une racine : (0.501551,0.772527)
  Le temps mis est : 2520
```

5.4 Structure à deux étages : calcul de Bézoutien

A partir d'un système d'équations polynomiales, nous construisons un ematrice dont le déterminant est le Bézoutien du système. Les coefficient de cette matrice sont des polynômes en les variables $\mathbf{x} = (x_1, \ldots, x_n)$ et $\mathbf{y} = (y_1, \ldots, y_n)$. Le déterminant est calculé à partir de la méthode de **Berhowitz** dont voici l'implémentation :

```
template<class R,class S>
typename R::coef_type deter(const R & M, const S & aux=S())
{
  typedef typename R::coef_type coef_type;
  typedef typename R::index_type index_type;
  coef_type d;
  R SM(M);

  d=Trace(SM,MATRIX());
  for (index_type i=2; i<=M.nbcol(); i++)
    {
      SM*=M;
      SM-=d*M;
      d=Trace(SM,MATRIX());
      d*=(S(typename S::coef_type(1.0/i)));
```

```
    }
  return d;
}
```

Voici maintenant l'exemple :

```
#include "_Matrix.H"
#include "_MPoly.H"
#include "algo/MBezout.H"

typedef Monom<double, dynamicexp<'x'> > Mon;
typedef MPoly<list<Mon>, Plex<Mon> >    Pol;
#define MAT MatStd<array2d<double> >

int main (int argc, char **argv) {
  list<Pol> l;
  int j=1,nb;
  MAT B;

  cin >> nb;

  while (j<=nb) {
    Pol Temp;
    cin >> Temp;
    l.push_back(Temp);
    j++;
    cout << Temp << endl;
  }
  B= MBezoutian(l,array2d<double>());
  cout << " La bezoutienne" << B << endl;
  return 0;
}
```

dont la sortie est :

```
(1)*x1*x3+(1)*x3+(1)*x2+(1)*x1+(1)
(1)*x3+(1)*x2+(1)*x1^2+(-1)
(1)*x3+(1)*x2^2+(1)*x1+(-1)
(1)*x3^2+(1)*x2+(1)*x1+(-1)

 Le bezoutien est :
((-1)*x3+(-1)*x2+(-1)*x1^2+(1))*y2*y3^2+((-1)*x2*x3+(1)*x3+(-1)*x1^2*x2+(1)*x2+(1)*x1+(-1))*

 La bezoutienne
        [ 0 0 0 0 0 0 0 0 0 -1 0 0 -1 -1 0 1 ]
        [ 0 0 0 0 0 0 0 -1 0 0 1 -1 0 1 0 1 -1 ]
        [ 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 1 ]
        [ 0 0 0 0 0 0 -1 0 1 0 0 1 0 0 2 0 ]
        [ 0 0 0 0 0 0 1 1 0 0 -1 0 1 1 0 -1 1 ]
        [ 0 0 0 0 0 0 1 0 0 -1 0 0 2 0 1 -1 -2 ]
        [ 0 0 0 0 -1 0 0 0 0 0 0 0 0 -1 0 -1 1 ]
        [ 0 0 0 0 0 0 1 0 1 2 0 -1 0 0 0 0 ]
        [ 0 0 1 0 0 1 -1 0 0 1 0 -1 1 0 1 -3 ]
        [ 0 -1 0 0 0 0 0 0 0 1 0 -1 1 0 1 -1 ]
```

```
[ 0 0 0 0 1 1 2 0 -1 -1 0 0 0 0 0 -2 ]
[ 1 0 0 -1 0 0 1 -1 3 -3 0 1 -3 0 -3 5 ]
```

5.5 Mélange de calculs formels et numériques : les équations de Kruppa

Nous nous intéressons maintenant à un système surdéterminé qui apparait dans le problème de reconstruction en vision, où tout couple de points, en correspondance avec les images, donnent une équation polynomiale. Ce qui fait que nous pouvons trouver autant d'équations que nous voulons, il n'existe qu'une solution. L'exemple donné ici, est un système de 6 quadriques dans un espace de dimension 5, venant du problème d'autocalibration de **Kruppa**. Pour plus de détails sur ces problèmes on peut lire [9].

```
#define RepMMon "RepDTab.H"
#define RepMPol "RepList.H"
#include "polynomial/ALP_include.H"
#include "linear/array1d.H"
#include "linear/Vector.H"
#include "polynomial/MPSolve.H"
#include "CPUClock.h"
#define DsVector(C) VectStd<array1d<C> > >
#define Mon MMon<double,DynamicArray>
#define Pol MPoly<list<Mon >, Dlex<Mon > >
#define YYSTYPE Pol

#include <string.h>
#include <math.h>

list<Pol> l;

#include "polynomial/parser.tab.h"
#include "polynomial/lex.yy.c"
#include "polynomial/parser.tab.c"
int yywrap(void);

int main (int argc, char **argv)
{
        yyparse();
        int nb =0;
        for (list<Pol>::const_iterator i = l.begin(); i != l.end(); ++i){
          nb++;
          cout << *i << endl; }

        Pol lp[nb];
        int j =0;
        for (list<Pol>::const_iterator i = l.begin(); i != l.end(); ++i)
          {if (j < nb) lp[j]=*i; j++;}
        lp[2]+=lp[1];
        DsVector(Pol) L(3,lp);
        CPUClock T;
```

```
        cout << MPSolve(L,compcol_mat_double()) << endl ;
        cout << "Le temps mis est : " << T.time() << endl;
        return 0;
}
```

La sortie est :

```
    Dimension of S : 792
    Number of non-zero entries : 16600
    Dimension of A : 1

    [-47.5396, -18.0428, 60.7231, 26.3451, 47.8788]

    Le temps mis est : 1.4
```

5.6 Connexion avec la librairie Lapack

Voici un exemple d'utilisation de la librairie **Lapack**. De plus, ce programme utilise une méthode `rank` de la librairie **ALP**, dont l'implémentation fait appel à une fonction de décomposition QR de la **Lapack**. Cette fonction `rank` sera aussi utilisée pour l'algorithme de résolution par le Bézoutien.

```
#include "_Lapack.H"

typedef VectStd<array1d<double> > VCT;
typedef MatSpl<lapack<double> >           MAT;

int main(int argc, char** argv) {
  double v[]={1,2,3,4,1,4,3,3,4,11,5,5,6,8,12,0,1,9,7,2};
  unsigned int r =0;

  MAT A(5,4,v);
```

```
  cout <<A<< endl;

  Lib<lapack<double> >::rank(A.rep,r);

  cout << endl << "Le rang de A est : " << r << endl;
};
```

En sortie :

```
        [ 1 4 5 0 ]
        [ 2 3 5 1 ]
        [ 3 3 6 9 ]
        [ 4 4 8 7 ]
        [ 1 11 12 2 ]

        Le rang de A est : 3
```

5.7 Calcul de valeurs propres

Ceci est un exemple de calcul de valeurs propres, qui peut être utilisé pour la méthode de résolution par la recherche de valeurs et vecteurs propres.

```
#define _Matrix_
#define _lapack_
#define _Maple_
#define _Latex_

#include "require.H"
#include "linear/Eigen.H"

#define VCT VectStd<array1d<double> >
#define MAT MatSpl<lapack<double>,Lib<lapack<double> > >
#define MATC MatSpl<lapack<complex<double> >,Lib<lapack<complex<double> > >

int main(int argc, char** argv) {

  lapack<double> r("mat33.in");
  MAT A(r);
  double v[]={1,2,0,2,1,0,0,0,1};
  MAT B(3,3,v);

  cout << A << endl<< B << endl;
  cout << Eigenval(A) <<endl <<endl;
  cout << Eigenval(B) <<endl <<endl;
};
```

En sortie :

```
 [ 58 9 2 ]
 [ 186 383 96 ]
 [ -912 -1551 -388 ]
```

```
[ 1 2 0 ]
[ 2 1 0 ]
[ 0 0 1 ]

[(50,0), (1,0), (2,0)]

[(3,0), (-1,0), (1,0)]
```

Chapitre 6

Conclusion

Ce projet fait suite au stage de deuxième année que j'ai effectué dans le projet **SAGA**. Une des remarques faites à la fin de ce stage et mentionnée dans la conclusion du rapport reconnaissait la complexité de l'usage des différentes structures par un utilisateur non-expert, ainsi que le problème soulevé par la volonté de ne pas perdre la notion de catégorie d'objets, l'efficacité acquise ni la grande modularité offerte, tout en palliant à cette difficulté de mise en place. Dans le cadre du projet, j'aurais contribué à la mise en place de cet allègement. De plus ce projet m'aura permis une continuation entre le stage de deuxième année et celui de fin de troisième année et DEA.

Annexe A

Principales modifications effectuées lors de ce projet

Une des principales tâche de ce projet a été de redéfinir la notion de catégorie d'objets, et le réagencement des différentes implémentations. Par exemple, pour dire qu'une matrice dense à coefficients génériques est de catégorie `MATRIX`, on donne l'instruction : `View(MatStd<C>,MATRIX)` équivalente à :

```
template<>struct View<MatStd<C» { typedef MATRIX val;};
```

A.1 Vecteurs

Cette première partie est consacrée à la catégorie `VECTOR`, catégorie regroupant les différentes classes de *vecteurs*. La catégorie est donc une classe vide, permettant uniquement la vue d'un objet d'un des différents types de vecteurs comme appartenant à la catégorie `VECTOR`.

Category of vectors. It is an empty class, only provided to view classes of vectors being in the same category, VECTOR. For this, these classes must be declared as VECTOR, with the instruction : `VIEW(class_name,VECTOR);`

struct VECTOR { };

This is the category for vectors

Et voici quelques fonctions spécifiques pour la catégorie `VECTOR` :

template <class R>
inline void Print(ostream & os, const R & v, const VECTOR view)

Output for general vectors. It is called by the output operator of the classes of vectors, and callable by `Print(os, p, VECTOR());` .

template <class R>
typename R::coef_type Norm(const R & v, int p =2,const VECTOR)

Norm of a vector. The default is the L_2 norm.

template <class R, class S>
typename R::coef_type Prod(const S & v, const R & w, const VECTOR)

Innerproduct of two vectors.

Ces fonctions peuvent être appelées directement comme spécifié, mais aussi par des fonctions entièrement génériques, comme par exemple pour les fonctions `Norm` et `Prod` :

```
template<class R>
inline typename Real<typename R::coef_type>::TYPE Norm(const R & v, int p=2)
```

{ return Norm(v,p,typename View<R>::val()); }

```
template <class R,class S>
inline typename R::coef_type Prod(const S & v, const R & w)
```

{ Prod(v,w,View<R>::val());}

Ainsi si V est, par exemple, un vecteur standard, de type `VectStd<...>`, on peut avoir sa norme par `n = Norm(V)`, la norme étant par défaut la norme euclidienne.

Les différentes classes de vecteurs implémentées et décrites en détail en annexe, sont :

- `VectStd<...>`, pour les vecteurs standards.
- `VectRef<...>` , pour les vecteurs référencés, il s'agit d'une classe permettant à certaines fonctions de retourner un pointeur et ainsi d'éviter les copies inutiles de gros objets.
- `VectHdl<...>`, classe de vecteurs "non-évalués", les opérations arithmétiques sur les vecteurs standards ne sont pas globalement évaluées, cette évaluation se faisant au moment oportun lors de l'accès à un élément.

Ces classes de vecteurs restent utilisables avec les différents conteneurs disponibles, tels que `array1d<C>`, où `C` peut-être n'importe quel type (double, int, grands nombres,...) ou classes d'objets (polynômes par exemple).

Un exemple d'utilisation de vecteurs est donné dans différents exemples de la section suivante **Exemples - Tests**.

A.2 Matrices

Comme pour les vecteurs la catégorie `MATRIX` est une classe vide :

Category of matrices. It is an empty class, only provided to view classes of matrices as the same category, MATRIX. For this, these classes must be declared as MATRIX, with the instruction :

```
VIEW(class_name,MATRIX);
```

```
struct MATRIX { };
```

This is the category for matrices

Comme pour la catégorie des vecteurs, une fonction `Print(...)` est implémentée, appelée par les opérateurs de sortie des différentes classes de matrices, mais aussi appelable directement pourvu que le dernier paramètre passé en argument soit un objet de type `MATRIX`, ce qui peut être : `MATRIX()`, ou `View<MatStd<R>>::val` par exemple.

```
template <class R>
inline void Print(ostream & os,const R & m, const MATRIX)
```

Output for general matrices. It is called by the output operator of the classes of matrices, and callable by `Print(os, p, MATRIX())` .

Les différentes classes de matrices implémentées et décrites en annexe sont :

- `MatStd<...>`, classe de matrices denses, ou standards.
- `MatStr<...>`, classe de matrices structurées, permettant de manipuler des matrices de Toeplitz ou de Hankel.
- `MatSps<...>`, classe de matrices creuses.
- `MatSpl<...>`, classe de matrices spécialisées, permettant, en particulier, la connection avec la librairie **lapack**.
- `MatRef<...>`, classe de matrices référencées, permettant d'éviter les copies d'objets dans les retours de fonctions.
- `MatHdl<...>`, classes de matrices "non-évaluées", de même fonctionnement que la classe `VectHdl<...>` (cf paragraphe précédent).

Pour un exemple de manipulation de matrices de Toeplitz et de matrices denses, voir le premier paragraphe de la section consacrée aux exemples, **Matrices structurées**, pour les matrices spécialisées permettant une connection avec **lapack**, voir le deuxième paragraphe de cette section d'exemples. Un exemple d'utilisation de matrice creuse est donné dans le dernier exemple, pour la recherche de racines de polynômes multivariables : **Racines d'un polynôme multivariables**.

A.3 Polynômes univariable

La catégorie correspondant aux polynômes univariable, `UPOLYNOMIAL`, est définie comme suit :

Category of univariate polynomials. It is an empty class, only provided to view classes of univariate polynomials as the same category, UPOLYNOMIAL. For this, these classes must be declared as UPOLYNOMIAL, with the instruction:

`VIEW(class_name,UPOLYNOMIAL);` **struct UPOLYNOMIAL { };**

This is the category for univariate polynomials

Ici aussi plusieurs fonctions sont implémentées pour la catégorie `UPOLYNOMIAL`. Comme pour les catégories `VECTOR` et `MATRIX`, il existe une fonction `Print(...)` appelée par les opérateurs de sortie, ou directement par l'utilisateur :

```
template <class R>
inline void Print(ostream & os, const R & p, const UPOLYNOMIAL)
```

Output for general univariate polynomials. It is called by the output operator of the classes of polynomials, and callable by `Print(os, p, UPOLYNOMIAL())`.

De plus, certaines fonctions particulières sont disponibles :

Calcul du réciproque, dans un anneau de séries formelles, d'un polynôme :

```
template <class R>
inline void reciprocal(R & w,const R & p, const UPOLYNOMIAL)
```

Computing the reciprocal of the univariate polynomial p. The result is given in w. We assume p[0] != 0. For this, we consider the equation $f(w) = P - \frac{1}{w} = 0$, *where* $P = p(z)$, $w = w(z) = \frac{1}{p(z)}$ *over the ring of formal power series in* z, *and apply Newton's iteration to this equation. So we have :* $w_0 = \frac{1}{p(0)}$, *and* $w_{j+1} = w_j(2 - Pw_j)$ *where* $j = 0,1,\ldots$ *By reducing the different polynomials modulo* $z^{2^{j+1}}$ *at each step, we have two convolutions of two pairs of vectors of dimensions at most* 2^{j+1} *by step.*

For a call to this function, the list of arguments must specifies, in last, an instance of UPOLYNOMIAL, the generic type `R` *is assumed to be a valid class of univariate polynomial. Example of such a call :*

```
UPolyDense<...>w, p(...);
reciprocal(w,p,UPOLYNOMIAL());
```

Troncature d'un polynôme à un certain degré passé en paramètre :

```
template <class T>
inline void reduce(T & p, typename T::index_type & e, const UPOLYNOMIAL)
```

Truncate an univariate polynomial p which is modified in output, with a new degree of $e - 1$. *For a call to this function, the list of arguments must specifies, in last, an instance of UPOLYNOMIAL, the generic type* `R` *is assumed to be a valid class of univariate polynomial. Example of such a call :*

```
UPolyDense<...>p(...);
UPolyDense<...>::index_type i=...;
reduce(p,i,UPOLYNOMIAL());
```

Inversion des coefficients d'un polynôme :

```
template <class T>
inline void reverse(T & p, const UPOLYNOMIAL)
```

Reverse the coefficients of an univariate polynomial, it is performed in place.
For a call to this function, the list of arguments must specifies, in last, an instance of UPOLYNOMIAL, the generic type `T` *is assumed to be a valid class of univariate polynomial. Example of such a call :*

```
UPolyDense<...>p(...);
reverse(p,i,UPOLYNOMIAL());
```

Evaluation de la dérivée d'un polynôme en un point :

template <class R>
inline typename R::coef_type derive(const R p, typename R::coef_type x, const UPOLYNOMIAL)

Returns an evaluation of the derivation of a univariate polynomial for the value x.
For a call to this function, the list of arguments must specifies, in last, an instance of UPOLYNOMIAL, the generic type `R` *is assumed to be a valid class of univariate polynomial. Example of such a call :*

```
UPolyDense<...>p(...);
UPolyDense<...>::coef_type x,d;
d=derive(p,x,UPOLYNOMIAL());
```

Toutes ces fonctions peuvent être appelées directement par l'utilisateur de la facon décrite pour la catégorie `VECTOR` ou par des fonctions entièrement génériques :

All these functions are callable by the user directly as described for the category `VECTOR` or via the generic functions:

template <class R>
inline void reciprocal(R & w,const R & p)

{ reciprocal(w, p, View<R>::val()); }

template <class T>
inline void reduce(T & p, const typename T::index_type & e)

{ reduce(p,e,View<T>::val()); }

template <class T>inline void reverse(T & p)

{ reverse(p,View<T>::val()); }

template <class R>
inline typename R::coef_type derive(const R p, typename R::coef_type x)

{ derive(p,x,View<R>::val()); }

Les différentes classes de polynômes en une variables implémentées et décrites en annexe sont :

- `UPolyDense<...>`, classe de polynômes denses.
- `UPolySpl<...>`, classe de polynômes spécialisés.
- `UPolyQuot<...>`, classe de polynômes appartenant à une algèbre quotient, i.e. les classes d'équivalences de polynômes modulo un polynôme particulier.
- `UPolyRef<...>`, classe de polynômes référencés, afin d'éviter les copies d'objets lors des retours de certaines fonctions.
- `UPolyHdl<...>`, classe de polynômes "non-évalués", de même utilisation que les classes équivalentes décrites dans les sections précédentes.

Annexe B

Vecteurs

template <class R>struct VectStd<R>

This is the standard class of vectors

```
{
  R rep;

  typedef typename R::coef_type   coef_type;

  VectStd(): rep() {}
  VectStd(const R & r):rep(r){}
  VectStd(unsigned int s): rep(s){}
  VectStd(unsigned int s, coef_type* t): rep(s,t){}
  VectStd(const char * nm): rep(nm) {}
  VectStd(const VectStd$<$R$>$ & r): rep(r.rep) {}
  VectStd(const VectRef$<$R$>$ & V) {rep.scopy(*(V.rep));}

  VectStd$<$R$>$ & operator=(const VectStd$<$R$>$ & V);
  VectStd$<$R$>$ & operator=(const VectRef$<$R$>$ & V);
  template$<$class S$>$
  VectStd$<$R$>$ & operator=(const VectHdl$<$S$>$ & V);
  template$<$class S$>$
  VectStd$<$R$>$ & operator+=(const VectStd$<$S$>$ & a);
  template$<$class S$>$
  VectStd$<$R$>$ & operator-=(const VectStd$<$S$>$ & V);
  template$<$class S$>$
  VectStd$<$R$>$ & operator+=(const VectHdl$<$S$>$ & a);
  template$<$class S$>$
  VectStd$<$R$>$ & operator-=(const VectHdl$<$S$>$ & V);
  VectStd$<$R$>$ & operator*=(const coef_type & c);
  VectStd$<$R$>$ & operator/=(const coef_type & c);
  template$<$class T$>$
  VectStd$<$R$>$ & add(const T & a);
  template$<$class T$>$
  VectStd$<$R$>$ & sub(const T & a);

  coef_type & operator[] (unsigned int i){return rep[i];}
  coef_type   operator[] (unsigned int i) const {return rep[i];}
```

```
  unsigned int size() const {return rep.size();}
};
```

template <class R, class S>
inline VectHdl<Plus<VectStd<R>,VectStd<S> > > operator+(const VectStd<R>& a, const VectStd<S>& b)

Addition operator. Return a referenced "sum".

template <class R, class S>
inline VectHdl<Minus<VectStd<R>,VectStd<S> > > operator-(const VectStd<R>& a, const VectStd<S>& b)

Substraction operator.

template <class R>
inline VectHdl<UMinus<VectStd<R> > > operator-(const VectStd<R>& a)

Unitary minus operator.

template <class R>
inline VectHdl<Mult<VectStd<R>, typename VectStd<R>::coef_type> >
operator*(const VectStd<R>& a, const typename VectStd<R>::coef_type & c)

Multiplication by a scalar.

template<class R>
inline VectHdl<Divd<VectStd<R>, typename VectStd<R>::coef_type> >
operator/(const VectStd<R>& a, const typename VectStd<R>::coef_type & c)

Division by a scalar.

template <class R>struct VectHdl

This class of vector is a referenced class. In other words, it can be used for accesing to the elements of the vector but no modification are allowed.

```
{
  R rep;

  typedef typename Coef<R>::coef_type coef_type;

  VectHdl() {}
  VectHdl(const R & r, unsigned int s): rep(r), _size(s){}
  VectHdl(const VectHdl<R> & v): rep(v.rep), _size(v.size()) {}

  coef_type operator[](unsigned int i) const {return _elem(*this,i);}
  unsigned int size () const {return _size;}
protected:
  unsigned int _size;
};
```

template <class R>
inline VectHdl<UMinus<VectHdl<R» >operator-(const VectHdl<R>& a)

Unitary minus operator.

template <class R>
inline VectHdl<Mult<VectHdl<R>, typename VectHdl<R>::coef_type»
operator*(const VectHdl<R>& a, const typename VectHdl<R>::coef_type & c)

Multiplication by a scalar.

template<class R>
inline VectHdl<Divd<VectHdl<R>, typename VectHdl<R>::coef_type»
operator/(const VectHdl<R>& a, const typename VectHdl<R>::coef_type & c)

Division by a scalar.

template <class R>struct VectRef

This is the referenced class of vectors

```
{
  R* rep;
  typedef typename R::coef_type  coef_type;

  VectRef() {}
  VectRef(R * r): rep(r),_size(r->size()){}
  VectRef(R * r, unsigned int s): rep(r),_size(s){}
  VectRef(const VectRef<R> & v): rep(v.rep), _size(v.size()) {}
  VectRef(const VectStd<R> & v): rep(&(v.rep)), _size(v.size()) {}

  typename R::coef_type operator[](unsigned int i) const;
  unsigned int size () const {return _size;}
protected:
  unsigned int _size;
};
```

Annexe C

Matrices

Interface for dense matrices.. The class `R` is the container corresponding to the internal representation. The data are owned by the interface. The inplace arithmetic operations are members of this class. The external arithmetic operators $+, -, *, /$ are defined by the generic implementation.

Notice: M = M*N; M= N*M; are correct. Warning: the second argument of setCol, setRow should not involve (*this)

template<class R>struct MatStd<R>

```
{
  R rep;

  typedef typename R::coef_type coef_type;
  typedef typename R::row_rep  row_rep;
  typedef typename R::col_rep  col_rep;

  MatStd():rep() {}
  MatStd(const R & r): rep(r) {}
  MatStd(unsigned int n, unsigned int m): rep(n,m){}
  MatStd(unsigned int n, unsigned int m, coef_type * t): rep(n,m,t){}
  MatStd(const char * fl): rep(fl) {}
  MatStd(const MatStd<R> & M): rep(M.rep) {}
  MatStd(const MatRef<R> & M) {rep.scopy(*(M.rep));}
  template <class T>
  MatStd(const T & M): rep(M.nbrow(),M.nbcol()) ;

  MatStd<R> & operator=(const MatStd<R> & M) ;
  MatStd<R> & operator=(const MatRef<R> & M) ;
  template<class T>
  MatStd<R> & operator=(const MatRef<T> & M);
  template<class T>
  MatStd<R> & operator=(const MatHdl<T> & M);

  unsigned int nbrow() const {return rep.nbrow();}
  unsigned int nbcol() const {return rep.nbcol();}

  coef_type & operator() (unsigned int i, unsigned int j);
  coef_type operator() (unsigned int i, unsigned int j) const;
```

```
MatStd<R> & operator() (Range I1, Range I2);

template<class S>
MatStd<R> & operator+=(const MatStd<S> & a);
template<class S>
MatStd<R> & operator+=(const MatHdl<S> & a);
template<class S>
MatStd<R> & operator-=(const MatStd<S> & a);
template<class S>
MatStd<R> & operator-=(const MatHdl<S> & a);
MatStd<R> & operator*=(const coef_type & c);
MatStd<R> & operator/=(const coef_type & c);
template<class S>
MatStd<R> & operator*=(const MatStd<S>& B);
template<class S>
MatStd<R> & operator*=(const MatHdl<S>& B);


MatRef<R> operator^(const int n);

VectRef<row_rep> row(const unsigned int i);
VectRef<col_rep> col(const unsigned int i) const;

template <class T>
void setCol(const unsigned int i, const T & v);
template<class T>
void set_col(const unsigned int i, const T &, const VECTOR);
template <class T>
void setRow(const unsigned int i, const T & v);
template<class T>
void set_row(const unsigned int i, const T & v, const VECTOR);

// Operations elementaires
void swapRow(unsigned int i, unsigned int j);
void swapCol(unsigned int, unsigned int);
void permRow(unsigned int *pivot);
void permCol(unsigned int *pivot);
void addRow(unsigned int i,unsigned int j, const coef_type & c = 1);
void addCol(unsigned int i,unsigned int j, const coef_type & c = 1);
void transpose();

protected:
template <class T>
MatStd<R> & rightAddMat(const T & a);
template <class T>
MatStd<R> & rightSubMat(const T & a);
template <class T>
MatStd<R> & rightMultMat(const T & B);
template <class T>
```

```
  MatStd<R> & leftMultMat(const T & B);
};
```

template <class R>VIEW(MatStd<R>,MATRIX)

```
template <class T>
inline MatStd<T>operator/(const MatStd<T>& a, const typename T::coef_type & c)
```

Divide the entries of a matrix by a scalar.

```
template<class R>
ostream & operator«(ostream & os, const MatStd<R>& p)
```

Output operator for standard matrices.

```
template <class R, class S>
inline MatHdl<Mult<MatStd<R>, MatStd<S» >
operator*(const MatStd<R>& A, const MatStd<S>& B)
```

Multiplication Matrix-Matrix. Return an unevaluated Matrix-Matrix product of Matrix type.

```
template <class R, class S>
inline VectHdl<Mult<MatStd<R>, VectStd<S» >
operator*(const MatStd<R>& A, const VectStd<S>& V)
```

Multiplication Matrix-Vector. Return an unevaluated Matrix-Vector product of Vector type.

```
template <class R, class S>
inline MatHdl<Mult<MatStd<R>, MatHdl<S» >
operator*(const MatStd<R>& A, const MatHdl<S>& B)
```

Multiplication Matrix-Matrix. Return an unevaluated Matrix-Matrix product of Matrix type.

```
template <class R, class S>
inline MatHdl<Mult<MatHdl<R>, MatStd<S» >
operator*(const MatHdl<R>& A, const MatStd<S>& B)
```

Multiplication Matrix-Matrix. Return an unevaluated Matrix-Matrix product of Matrix type.

template <class R>struct MatHdl

Handled matrices. It gives the number of rows and columns and the access of the element (i,j)*. No modification of the entries is allowed. This class is used in order to avoid temporary matrices, during multiple arithmetic operations and assignment.*

```
{
  R rep;

  typedef typename Coef<R>::coef_type coef_type;

  MatHdl() {}
  MatHdl(const R & r, unsigned int n, unsigned int m):
    rep(r), _nbrow(n), _nbcol(m) {}
```

```
  MatHdl(const MatHdl<R> & r): rep(r.rep),
      _nbrow(r.nbrow()), _nbcol(r.nbcol()) {}

  coef_type operator()(unsigned int i, unsigned int j) const
         {return _elem(*this,i,j);}

  inline unsigned int nbrow() const {return _nbrow;}
  inline unsigned int nbcol() const {return _nbcol;}

private:
  unsigned int _nbrow, _nbcol;
};
```

Interface for structured matrices. It is a subclass of standard matrices class.

template<class R>struct MatStr<R>: public MatStd<R>

Structured matrices.

```
{

  typedef typename R::coef_type coef_type;

  MatStr() {}
  MatStr(const R & r): MatStd<R>(r) {}
  MatStr(unsigned int n, unsigned int m, coef_type * t):
    MatStd<R>(n,m,t){}
  MatStr(const char * fl): MatStd<R>(fl) {}
  MatStr(const MatStr<R> & M): MatStd<R>(M.rep){}

  MatStr<R> &  operator=(const MatStr<R> & M);
  template<class S>
  MatStr<R> &  operator=(const S & M);
  MatStr<R> & operator*=(const MatStr<R>& B);

  MatStr<R> & rightMult(const MatStr<R>& M);
  MatStr<R> & leftMult (const MatStr<R>& M);

  template<class C>
  static void mult(array1d<C> & r, const R & a, const array1d<C>& v);
  }
};
```

template <class R>VIEW(MatStr<R>,MATRIX)

template<class R, class S>
VectRef<S>operator*(const MatStr<R>& M, const VectStd<S>& V)

template<class S, class C>
inline void assign(toeplitz<C>& r, const S & M)

Assignment of a matrix to a container `toeplitz`

template<class S, class C>
inline void assign(hankel<C>& r, const S & M)

Assignment of a matrix to a container `hankel`

Interface for sparse matrices. It is a subclass of standard matrices class.

template<class C>struct MatSps<umfpack<C» : public MatStd<umfpack<C»

Sparse matrices. Specialization for `umfpack<...>`
container.

Specific constructors have been added. They take as parameters :
`n` *which specifies the number of rows,*
`m` *which specifies the number of columns,*
`nz` *for the number of non-zero entries,*
`val` *an array of values for the non-zero entries of the matrix,*
`r` *and* `c` *are, respectively, arrays for rows index and columns index for specific representation like* `compcol_mat_double`*, in this case, base is the base for access index (i.e* $A(0,0)$ *is the first element).*
`ind` *is an array for both rows and columns indexes for specific representation like* `umfpack<C>`
.
In addition to standard matrices methods, we have an access method for the number of non-zero entries of the matrix.

```
{

  typedef typename umfpack<C>::coef_type coef_type;
  typedef typename umfpack<C>::solve_type solve_type;

  MatSps() {}
  MatSps(const umfpack<C>  & r): MatStd<umfpack<C> >(r) {}
  MatSps(int n, int m, int nz, coef_type *val, int *ind) :
    MatStd<umfpack<C> >(umfpack<C> (n,m,nz,val,ind)) {}
  MatSps(unsigned int n, unsigned int m, coef_type * t):
    MatStd<umfpack<C> >(n,m,t){}
  MatSps(const char * fl): MatStd<umfpack<C> >(fl) {}
  MatSps(const MatSps<umfpack<C> > & M): MatStd<umfpack<C> >(M.rep){}

  unsigned int nzero() { return rep.nzero();}
  MatSps<umfpack<C> > &  operator=(const MatSps<umfpack<C> > & M);
  MatSps<umfpack<C> > & operator=(const MatRef<umfpack<C> > & M);
  MatSps<umfpack<C> > & operator*=(const MatSps<umfpack<C> >& B);
  MatSps<umfpack<C> > & rightMult(const MatSps<umfpack<C> >& M);
  MatSps<umfpack<C> > & leftMult (const MatSps<umfpack<C> >& M);
  MatSps<umfpack<C> > & transpose ();
  VectRef<col_rep>  col (const unsigned int i) const;
  static void assign(umfpack<C>  & r, const umfpack<C> & M);
  static void transpose(umfpack<C>  & A);
  template<class C>
  static void mult(array1d<C> & r, const umfpack<C>  & a, const array1d<C>& v);
  static void submatrix(umfpack<C>  & a, const unsigned int & f,
                        const unsigned int & l, ByCol from);
```

```
  static void submatrix(umfpack<C>  & a, const unsigned int & ff,
                        const unsigned int & ll, ByRow from);
  static void col(typename umfpack<C>::col_rep & c,
                        const umfpack<C>  & a, const unsigned int i);
  static void writetoHSL(const umfpack<C>  & a, const char *filename);
  static void readfromHSL(umfpack<C>  & a, const char *filename);
};
```

template<>struct MatSps<compcol_mat_double>: public MatStd<compcol_mat_double>

Sparse matrices. Specialization for `compcol_mat_double` *container.*
Specific constructors have been added. They take as parameters :
`n` *which specifies the number of rows,*
`m` *which specifies the number of columns,*
`nz` *for the number of non-zero entries,*
`val` *an array of values for the non-zero entries of the matrix,*
`r` *and* `c` *are, respectively, arrays for rows index and columns index for specific representation like* `compcol_mat_double`*, in this case, base is the base for access index (i.e* $A(0,0)$ *is the first element).*
In addition to standard matrices methods, we have an access method for the number of non-zero entries of the matrix.

```
{

  typedef typename compcol_mat_double::coef_type coef_type;
  typedef typename compcol_mat_double::solve_type solve_type;

  MatSps() {}
  MatSps(const compcol_mat_double & r);
  MatSps(int n, int m, int nz, double *val, int *r, int *c, int base=0);
  MatSps(const MatSps<compcol_mat_double> & M);

  unsigned int nzero();
  MatSps<compcol_mat_double> &  operator=(const MatSps<compcol_mat_double> & M);
  MatSps<compcol_mat_double> & operator=(const MatRef<compcol_mat_double> & M);
  MatSps<compcol_mat_double> & transpose ();
  VectRef<typename MatStd<compcol_mat_double>::col_rep>
          col (const unsigned int i) const;
  template<class S>
  static void assign(compcol_mat_double & r, const MatSps<S> & M);
  static void transpose(compcol_mat_double & A);
  template<class C>
  static void mult(array1d<C> & r, const compcol_mat_double & a, const array1d<C>& v);
  static void submatrix(compcol_mat_double & a, const unsigned int & f,
                  const unsigned int & l, ByCol from);
  static void submatrix(compcol_mat_double & a, const unsigned int & f,
                  const unsigned int & l, ByRow from);
  static void col(typename compcol_mat_double::col_rep & c,
                  const compcol_mat_double & a, const unsigned int i);
  static void writetoHSL(const compcol_mat_double & a, const char *filename);
```

```
  static void readfromHSL(compcol_mat_double & a, const char *filename);
};
```

template <class R>VIEW(MatSps<R>,MATRIX)

template<class R>
inline void WriteToHSL(const MatSps<R>& M, const char *file_to_write)

Writes a Sparse Matrix M in Harwell-Boeing Format in the file `file_to_write`.

template<class R>
inline void ReadFromHSL(MatSps<R>& M, const char *file_to_write)

Reads a Sparse Matrix M in Harwell-Boeing Format from the file `file_to_write`.

template<class R>
MatRef<R>operator +(const MatSps<R>& M1, const MatSps<R>& M2)

template<class R, class S>
VectRef<S>operator*(const MatSps<R>& M, const VectStd<S>& V)

template<class R, class S>
inline VectRef<S>transMult (const MatSps<R>& M, const VectStd<S>& V)

template<class C>
void MatSps<umfpack<C»::assign(umfpack<C>& r, const umfpack<C>& M)

Sparse matrices specialization for `umfpack<...>`
representation.

template <class C>
void MatSps<umfpack<C»::submatrix(umfpack<C>& a, const unsigned int & f, const unsigned int & l, ByRow from)

In place extraction of a submatrix of the matrix a, it is returned with its previous rows f to l removed.

template <class C>
void MatSps<umfpack<C»::col(typename umfpack<C>::col_rep & c, const umfpack<C>& a, const unsigned int i)

Extraction of a column of index i from the matrix a.
Result is given in c, a is unchanged.

template <class C>
void MatSps<umfpack<C»::writetoHSL(const umfpack<C>& a, const char *filename)

Writes a Sparse Matrix a in Harwell-Boeing Format in the file `filename`.

template <class C>
void MatSps<umfpack<C»::readfromHSL(umfpack<C>& a, const char *filename)

Reads a Sparse Matrix a in Harwell-Boeing Format from the file `filename`.

template<class S>
void MatSps<compcol_mat_double>::assign(compcol_mat_double & r, const MatSps<S>& M)

Sparse matrices specialization for `compcol_mat_double` *representation.*

void MatSps<compcol_mat_double>::transpose(compcol_mat_double &a)

Transpose of a sparse matrix with `compcol_mat_double` *representation*

void MatSps<compcol_mat_double>::mult(array1d<double>& r, const compcol_mat_double & a, const array1d<double>& v)

Product of a sparse matrix by a vector.
Puts the result in `r`.
Uses the SparseLib++ *matrix class* `operator *`.

void mult_trans(array1d<double>& r, const compcol_mat_double & a, const array1d<double>& v)

Product of the transposed of the product of a sparse matrix by a vector.
Puts the result in `r`.
Uses the SparseLib++ *matrix class method* `trans_mult`.

void MatSps<compcol_mat_double >::submatrix(compcol_mat_double & a, const unsigned int & f, const unsigned int & l, ByCol from)

In place extraction of a submatrix of the matrix a, it is returned with its previous columns f to l removed.

void MatSps<compcol_mat_double >::submatrix(compcol_mat_double & a, const unsigned int & ff, const unsigned int & ll, ByRow from)

In place extraction of a submatrix of the matrix a, it is returned with its previous rows f to l removed.

void MatSps<compcol_mat_double>::col(typename compcol_mat_double::col_rep & c, const compcol_mat_double & a, const unsigned int i)

Extraction of a column of index i from the matrix a.
Result is given in c, a is unchanged.

void MatSps<compcol_mat_double>::writetoHSL(const compcol_mat_double & a, const char *filename)

Writes a Sparse Matrix a in Harwell-Boeing Format in the file `filename`.

void MatSps<compcol_mat_double>::readfromHSL(compcol_mat_double & a, const char *filename)

Reads a Sparse Matrix a in Harwell-Boeing Format from the file `filename`.

void solve_gmres(array1d<double>& u, const compcol_mat_double & a, const array1d<double>& b, double tol = 1.e-6,int restart = 32,int maxit = 150)

Solves the unsymmetric linear system $a\,u = b$ by an iterative method using the preconditionned Generalized Minimum Residual method.
Puts the result in `u`.
Uses the SparseLib++ *procedure* `CompCol_ILUPreconditionner_double` *to perform an incomplete LU preconditionning of the matrix, and the* IML++ *procedure* `GMRES` *for solving the system.*
tol specifies the convergence tolerance for the method, its default value is 1.e-6,
maxit specifies the maximum number of iterations that the method will take, its default value is 150,
restart specifies the number of iterations for each restart, its default value is 32.

Interface for specialized matrices.

template<class R, class L=Lib<R» struct MatSpl : public MatStd<R>

Specialized matrices.

```
{

  typedef typename R::coef_type coef_type;

  MatSpl() {}
  MatSpl(const R & r): MatStd<R>(r) {}
  MatSpl(unsigned int n, unsigned int m): MatStd<R>(n,m){}
  MatSpl(unsigned int n, unsigned int m, coef_type * t): MatStd<R>(n,m,t){}
  MatSpl(const char * fl): MatStd<R>(fl) {}

  MatSpl<R,L> &  operator=(const MatSpl<R,L> & M);
  MatSpl<R,L> &  operator=(const MatRef<R> & M);
  template<class T>
  MatSpl<R,L> & operator=(const MatHdl<T> & M);
  MatSpl<R,L> & operator*=(const MatSpl<R,L>& B);
  MatSpl<R,L> & rightMult(const MatSpl<R,L>& M);
  MatSpl<R,L> & leftMult (const MatSpl<R,L>& M);
  MatSpl<R,L> & transpose ();
  static inline MatSpl<R,L>::solve_ls(array1d<typename R::coef_type> & x, R & A, array1d<typ
  static inline MatSpl<R,L>::solve_lu(array1d<typename R::coef_type> & x, R & A,const array1
};
```

template<class R,class L>
inline MatRef<R>operator *(const MatSpl<R,L>& M1, const MatSpl<R,L>& M2)

Implementation of Toeplitz matrices.
In this particular case, the product of a $(n+1)\mathbf{x}(n+1)$ Toeplitz matrix by a vector is performed as a subvector of the product of two polynomials, by FFT computations.

template<class C>struct toeplitz : public array1d<C>

Representation for Toeplitz matrices as a subclass of `array1d<C>`

.

Defined by its number of row nbrow_, its number of columns nbcol_.

```
{

  unsigned int nbrow_, nbcol_;

  typedef C           coef_type;
  typedef array1d<C>  row_rep;
  typedef array1d<C>  col_rep;

  toeplitz(): array1d(){}
  toeplitz(const toeplitz<C> & v);
  toeplitz(unsigned int i, unsigned int j);
  toeplitz(unsigned int i, unsigned int j, C* t):;

  rebuild(unsigned int i, unsigned int j);

  C & operator()(unsigned int i, unsigned int j);
  C   operator()(unsigned int i, unsigned int j) const;

  int nbrow() const;
  int nbcol() const;
};
```

template <class C>
void mult_fft(array1d<C>& r, const toeplitz<C>& a,
const array1d<C>& b)

Multiplication Toeplitz Matrix - Vector.

Result, given by the first parameter r, is a Vector representation type

Implementation of Hankel matrices.

In this particular case, the product of a $(d+1)\mathbf{x}(d+1)$ Hankel matrix by a vector is a subvector of the coefficients vector of the product of two polynomials.

template<class C>struct hankel : public array1d<C>

Representation of Hankel matrix as a subclass of `array1d`.

Defined by its number of row nbrow_, its number of columns nbcol_.

```
{

  unsigned int nbrow_, nbcol_;

  typedef C           coef_type;
  typedef array1d<C>  row_rep;
  typedef array1d<C>  col_rep;

  hankel();
  hankel(const hankel<C> & v);
  hankel(unsigned int i, unsigned int j):;
  hankel(unsigned int i, unsigned int j, C* t):;
```

```
  rebuild(unsigned int i, unsigned int j);{

  C & operator()(unsigned int i, unsigned int j);
  C   operator()(unsigned int i, unsigned int j) const;

  int nbrow() const;
  int nbcol() const;
};
```

template <class C>
void mult_fft(array1d<C>& r, const hankel<C>& a,
const array1d<C>& b)

Multiplication Hankel Matrix - Vector.
Result, given by the first parameter r, is a Vector representation type.

Connection with the fortran library LAPACK. The container used is `array1a<C>`

.

template <class C>struct lapack: public array2d<C>

Class of array derived from `array2d<C>`

, with `lda_` *field. This value indicates the leading dimension array, that is the number of rows for the allocated array. The actual array2d is a submatrix of it. It is defined by its number of rows* `nbrow_` *(≤* `lda_`*) and its number of columns* `nbcol_`*.*

```
{
  unsigned int lda_;
  typedef C            coef_type;
  typedef unsigned int index_type;
  typedef array1d<C>   row_rep;
  typedef array1d<C>   col_rep;
  typedef array1d<C>   vect_rep;

  lapack();
  lapack(const lapack<C> & v);
  lapack(unsigned int i, unsigned int j=1);
  lapack(unsigned int i, unsigned int j, C* t);
  lapack(unsigned int i, unsigned int j, unsigned int k, C* t);
  lapack(const char * nm);

  lapack<C> & operator=(const lapack<C> & v);

  void rebuild(unsigned int i,unsigned int j);
  void scopy(const lapack<C> & v);

  void get_Q(lapack<double> & Res, const lapack<double> & A, double * tau);

  C & operator()(unsigned int i, unsigned int j)       {return tab_[i+lda_*j];}
  C   operator()(unsigned int i, unsigned int j) const {return tab_[i+lda_*j];}
};
```

template<>
void Lib<lapack<double»::transpose(lapack<double>& A)

Transposition of the matrix.
If it is a square matrix, the operation is performed inplace.
Otherwise, a new memory space is allocated and restored.

```
template <>
integer Lib<lapack<double»::dcmp_qr(lapack<double>& A)
```

Compute the QR decomposition of the matrix A.
The operation is done inplace. The output matrix is upper-triangular.

```
template <>
integer Lib<lapack<double»::dcmp_qr(lapack<double>& A,
unsigned int * pivot, double * tau)
```

Compute the QR decomposition of the matrix `A`*. The operation is done inplace. The pivots are stored in* `pivot` *and the matrix Q is represented by the lower part of* `A` *and the coefficients in* `tau`*.*

```
template <>
integer Lib<lapack<double»::dcmp_lu(lapack<double>& A, integer *pivot)
```

Compute the LU decomposition of the matrix `A`*. The pivots on the columns are stored in* `pivot`*.*

```
template <>
void Lib<lapack<double»::solve_lu(array1d<double>& x,
const lapack<double>& A, const array1d<double>& b)
```

Solve the system $A\,x = b$ by LU decomposition of the matrix `A`*. Use the* LAPACK *routine* `dgesv`*.*

```
template <>
void Lib<lapack<double»::solve_ls(lapack<double>& A, array1d<double>& b)
```

Solve the system $A\,x = b$ by a Least Square method.
Put the result in `b`*.*
Use the LAPACK *routine* `dgelsx`*.*

```
template <>
void Lib<lapack<double»::solve_ls(array1d<double>& x,
const lapack<double>& A, const array1d<double>& b)
```

Solve the system $A\,x = b$ by a Least Square method.
A copy of `A` *is made and a copy of* `b` *is put in* `x`*.*
Then the previous function
`solve_ls(lapack<double>& A, array1d<double>& b)` *is used.*

```
template<>
void Lib<lapack<double»::svd(array1d<double>& x,const lapack<double>& A)
```

Compute the Singular value of the matrix, calling Lapack subroutine dgesvd. The result, put in `x`*, is the vector of singular values in decreasing order. The matrix* `A` *is copied.*

template<>
void Lib<lapack<double»::svd(array1d<double>& x,const lapack<double>& A,
lapack<double>& U, lapack<double>& V)

Compute the Singular value of the matrix, calling Lapack subroutine dgesvd. The result, put in `x`*, is the vector of singular values in decreasing order. The matrix* `A` *is copied in a temporary matrix.*
The output matrix U and V are orthogonal matrices, which satifie $\mathtt{U}^t A V = S$.
The correct space for the matrix `U` *and* `V` *should be allocated before using this procedure.*

template <>
void Lib<lapack<double»::leigvct(lapack<complex<double» & E, lapack<double>& A)

Compute the left eigenvectors of `A` *and put it in the complex matrix* `E`*. Use the* LAPACK *routine* `dgeev`.

template <>
void Lib<lapack<double»::leigvct(lapack<complex<double» & E,
array1d<complex<double»& V, lapack<double>& A)

Compute the left eigenvectors and eigenvalues of `A`.
The matrix of eigenvector is put in the complex matrix `E` *and the eigenvalues are stored in* `V`.

void Lib<lapack<double»::reigvct(lapack<complex<double» & E,
lapack<double>& A)

Compute the right eigenvectors of the matrix `A`.
The complex matrix of eigenvector is put in `E`.
Use the LAPACK *routine* `dgeev`.

template <>
void Lib<lapack<double»::reigvct(lapack<complex<double» & E,
array1d<complex<double» & V, lapack<double>& A)

Compute the left eigenvectors and eigenvalues of `A`.
The matrix of eigenvector is put in the complex matrix `E` *and the eigenvalues are stored in* `V`.

Connection with the fortran library UMFPACK. Allow to solve sparse linear systems $A\,x = b$.

template <class C>struct umfpack : public array1d<C>

Class of array derived from `array1d<C>`

, with `nbnzero_` *field, which indicates the number of non-zeros coefficients of the matrix created and so the real size of the array,* `index_` *field which contains rows index for all coefficients, plus all the columns index such* $A(row,col) = array1d < C >:: tab_[k]$, *with* $row = index_[k]$, $col = index_[k]$, *for a sparse matrix A. This matrix is defined by its number of rows* `nbrow_`*, its number of columns* `nbcol_` *and its number of non-zero coefficients* `nbnzero_`.

```
{
  unsigned int nbrow_, nbcol_, nbnzero_;

  int * index_; // of size 2*nbnzero_

  typedef C           coef_type;
  typedef array1d<C>  row_rep;
  typedef array1d<C>  col_rep;
```

```
  typedef UMFso solve_type;

  umfpack() : array1d<C>(){}
  umfpack(const umfpack & v);
  umfpack(int M, int N, int nz);
  umfpack(int M, int N, int nz, C *val, int * ind);

  void rebuild(unsigned int i,unsigned int j, unsigned int nz = i*j);
  void scopy(const umfpack<C> & v);

  C   operator()(unsigned int i, unsigned int j) const ;

  unsigned int nbrow() const {return nbrow_;}
  unsigned int nbcol() const {return nbcol_;}
  unsigned int nzero() const {return nbnzero_;}

  int index(unsigned int i, unsigned int j) const;
};
```

void solve_umf(array1d<double>& u, const umfpack<double>& a, const array1d<double>& b)

Real implementation. Solves the system $A\,x = b$ *by a combined unifrontal/multifrontal method for a sparse matrix* A *which can be unsymmetric. Puts the result in* `u`. *Uses the fortran library* UMFPACK 2.2.1 *routines* `umd21i, umd2fa, umd2so`.

void solve_umf(array1d<complex<double» & u, const umfpack<complex<double» & a, const array1d<complex<double»& b)

Complex implementation. Solves the system $A\,x = b$ *by a combined unifrontal/multifrontal method for a sparse matrix* A *which can be unsymmetric. Puts the result in* `u`. *Uses the fortran library* UMFPACK 2.2.1 *routines* `umz21i, umz2fa, umz2so`.

Connection with the C++ library SparseLib++ which allows to use the C++ iterative methods library of IML++ to solve unsymmetric linear systems applied to sparse matrices.

struct compcol_mat_double : public CompCol_Mat_double

Class of matrix representation derived from `CompCol_Mat_double` *which is a sparse matrix representation in* SparseLib++ *library with real coefficients. It is a compressed column storage. Defined by its number of rows* `nbrow_`, *its number of columns* `nbcol_`,*and by some private fields of* `CompCol_Mat_double` *class which are : its number of non-zero coefficients* `nz_`, *the index base* `base`, *vector of data values* `val_`, *vector of rows index* `rowind_`, *vector of pointers to the first element in each column* `colptr_`.

```
{
  unsigned int nbrow_, nbcol_;

  typedef double      coef_type;
  typedef array1d<double>  row_rep;
  typedef array1d<double>  col_rep;
  typedef GMres solve_type;

  compcol_mat_double() : CompCol_Mat_double(){}
  compcol_mat_double(const compcol_mat_double & v);
```

```
  compcol_mat_double(int M, int N, int nz, double *val, int *r, int *c, int base=0);

  double operator() (int i, int j) const;

  void rebuild(unsigned int i, unsigned int j, int nz);

  int nbrow() const;
  int nbcol() const;
  int nzero() const;
};
```

Annexe D

Polynômes univariables

Univariate polynomials. Standard (dense) representation. B. Mourrain.

template<class R>struct UPolyDense

Interface for standard univariate polynomials.

```
{

  R rep;

  typedef typename R::coef_type  coef_type;
  typedef typename R::index_type index_type;

  UPolyDense() : rep() {}
  UPolyDense(const R & r) : rep(r){}
  UPolyDense(index_type n) : rep(n) {}
  UPolyDense(index_type n, coef_type * t) : rep(n,t) {}
  UPolyDense(const char * fl) : rep(fl) {}
  UPolyDense(const UPolyDense<R> & P) :rep(P.rep) {}

  UPolyDense<R> & operator=(const UPolyRef<R> & P);
  UPolyDense<R> & operator=(const UPolyDense<R> & P);
  template <class S>
  UPolyDense<R> & operator=(const S & P);

  index_type & degree();
  index_type  degree() const;

  coef_type & operator[] (index_type i);
  coef_type   operator[] (index_type i) const;

  template<class S>
  UPolyDense<R> & operator+=(const UPolyDense<S> & a);
  template<class S>
  UPolyDense<R> & operator+=(const UPolyHdl<S> & a);
  template<class S>
  UPolyDense<R> & operator-=(const UPolyDense<S> & a);
  template<class S>
  UPolyDense<R> & operator-=(const UPolyHdl<S> & a);
```

```
 template<class S>
 UPolyDense<R> & operator*=(const UPolyDense<S>& P);
 template<class S>
 UPolyDense<R> & operator*=(const UPolyHdl<S>& P);
 UPolyDense<R> & operator*=(const coef_type & c);
};
```

template <class R>
VIEW(UPolyDense<R>,UPOLYNOMIAL)

template<class R>
UPolyRef<R>operator /(const UPolyDense<R>& P1, const typename UPolyDense<R>::coef_type & c)

template <class R>
ostream & operator«(ostream & os, const UPolyDense<R>& p)

template <class R,class S>
inline UPolyHdl<Plus<UPolyDense<R>,UPolyDense<S» >
operator+(const UPolyDense<R>& a, const UPolyDense<S>& b)

template <class R,class S>
inline UPolyHdl<Minus<UPolyDense<R>,UPolyDense<S» >
operator-(const UPolyDense<R>& a, const UPolyDense<S>& b)

template <class R>
inline UPolyHdl<UMinus<UPolyDense<R» >
operator-(const UPolyDense<R>& a)

template <class R,class S>
inline UPolyHdl<Plus<UPolyDense<R>,UPolyHdl<S» >
operator+(const UPolyDense<R>& a, const UPolyHdl<S>& b)

template <class R,class S>
inline UPolyHdl<Minus<UPolyDense<R>,UPolyHdl<S» >
operator-(const UPolyDense<R>& a,const UPolyHdl<S>& b)

template <class R,class S>
inline UPolyHdl<Plus<UPolyHdl<S>,UPolyDense<R» >
operator+(const UPolyHdl<S>& b,const UPolyDense<R>& a)

template <class R,class S>
inline UPolyHdl<Minus<UPolyHdl<S>,UPolyDense<R» >
operator-(const UPolyHdl<S>& b,const UPolyDense<R>& a)

```
template <class R>
inline UPolyHdl<Mult_S<UPolyDense<R>,typename UPolyDense<R>::coef_type»
operator*(const UPolyDense<R>& a, const typename UPolyDense<R>::coef_type & c)
```

```
template <class R>
inline UPolyHdl<S_Mult<typename UPolyDense<R>::coef_type,UPolyDense<R» >
operator*(const typename UPolyDense<R>::coef_type & c, const UPolyDense<R>& a)
```

```
template <class R, class S>
inline UPolyHdl<Mult<UPolyDense<R>, UPolyDense<S» >
operator*(const UPolyDense<R>& A, const UPolyDense<S>& B)
```

```
template <class R, class S>
inline UPolyHdl<Mult<UPolyDense<R>, UPolyHdl<S» >
operator*(const UPolyDense<R>& A, const UPolyHdl<S>& B)
```

```
template <class R, class S>
inline UPolyHdl<Mult<UPolyHdl<R>, UPolyDense<S» >
operator*(const UPolyHdl<R>& A, const UPolyDense<S>& B)
```

Handle polynomials. It gives the degree, and the access to the element [i]. No modification of the entries is allowed. This class is used in order to avoid temporary polynomials during multiple arithmetic operations and assignment.

```
template <class R>VIEW(UPolyHdl<R>,UPOLYNOMIAL)
```

```
template <class R>inline UPolyHdl<UMinus<UPolyHdl<R» >operator-(const UPolyHdl<R>& a)
```

Unary minus of a handled polynomial

```
template <class R>inline UPolyHdl<Mult_S<UPolyHdl<R>,typename UPolyHdl<R>::coef_type» operator*(const UPolyHdl<R>& a, const typename UPolyHdl<R>::coef_type & c)
```

Product of a handled polynomial by a scalar

```
template <class R>inline UPolyHdl<Mult_S<UPolyHdl<R>,typename UPolyHdl<R>::coef_type» operator*(const typename UPolyHdl<R>::coef_type & c, const UPolyHdl<R>& a)
```

Product of a scalar by a handled polynomial

```
template <class R>ostream & operator«(ostream & os, const UPolyHdl<R>& p)
```

Output for a handled polynomial

template <class R>struct UPolyRef

Class of referenced polynomials

```
template <class R>VIEW(UPolyRef<R>,UPOLYNOMIAL)
```

```
template <class R>ostream & operator«(ostream & os, const UPolyRef<R>& p)
```

Output for referenced polynomials

template<class R>struct UPolySpl : public UPolyDense<R>

Interface for specialized univariate polynomials.

```
{

  typedef typename R::coef_type  coef_type;
  typedef typename R::index_type index_type;

  UPolySpl(): UPolyDense<R>() {}
  UPolySpl(const R & r) : UPolyDense<R>(r) {}
  UPolySpl(index_type n) : UPolyDense<R>(n) {}
  UPolySpl(index_type n, coef_type * t) : UPolyDense<R>(n,t) {}
  UPolySpl(const char * fl) : UPolyDense<R>(fl) {}
  UPolySpl(const UPolySpl<R> & P) :  UPolyDense<R>(P.rep){}

  UPolySpl<R> & operator=(const UPolyRef<R> & P);
  template <class S>
  UPolySpl<R> & operator=(const S & P);
  UPolySpl<R> & operator*=(const UPolySpl<R>& P);
  UPolySpl<R> & operator*=(const coef_type & c);
  UPolySpl<R> & operator/=(const coef_type & c);

  static void mult(R & x, const R & a, const R & b);
};
```

template <class R>VIEW(UPolySpl<R>,UPOLYNOMIAL)

template<class R>
UPolyRef<R>operator *(const UPolySpl<R>& P1, const UPolySpl<R>& P2)

Product of two specialized polynomials

template <class R>
void mult_fft(R & r, const R & a, const R & b)

Multiplication of a specialized polynomial by an other specialized polynomial. Result, given by the first parameter r, is a polynomial of the same type as the two polynomials. Method of multiplication is fft. We assume that R is a valid class of container of polynomial.

Univariate polynomials. Dense representation.
Interface for elements of a quotient algebra associated to a polynomial.
Elements are translated in the Horner basis by constructors. All the operations are in this basis, only the display of a polynomial needs a change to the monomial basis.

template<class R>struct UPolyQuot : public UPolyDense<R>

Class of polynomial of a quotient algebra.

```
{

  static R poly;
```

```
typedef typename R::coef_type  coef_type;
typedef typename R::index_type index_type;

UPolyQuot() {}
UPolyQuot(const R & r): UPolyDense<R>(r) {}
UPolyQuot(index_type n): UPolyDense<R>(n){}
UPolyQuot(index_type n, coef_type * t):  UPolyDense<R>(n,t){
  MontoHorn(rep,true);}
UPolyQuot(const char * fl): UPolyDense<R>(fl) {}
UPolyQuot(const UPolyQuot<R> & P):  UPolyDense<R>(P.rep) {}

// Returns the quotient dimension
static index_type dim();

UPolyQuot<R> & operator=(const UPolyQuot<R> & P);
UPolyQuot<R> & operator=(const UPolyRef<R> & P);
template <class S>
UPolyQuot<R> & operator=(const UPolyHdl<S> & P);
template <class S>
UPolyQuot<R> & operator=(const S & P);
UPolyQuot<R> & operator*=(const coef_type & c);
UPolyQuot<R> & operator/=(const coef_type & c);
};
```

static void square(R & x,const R & f)

Computes squares in $A = \frac{C[x]}{(poly)}$
Allows to compute products in A by the formula $fg = \frac{1}{4}((f+g)^2 - (f-g)^2)$

static void mult(R & x, const R & a, const R& b)

*Product a*b, using only squares, by the formula* $a * b = frac14((a+b)^2 - (a-b)^2)$

static void MontoHorn(R &x, boolean MtoH)

Expresses the polynomial x from a basis (Monomial or Horner) in the other basis.
For this, computes B1 for the passage of the Horner basis to the monomial basis. Computes its inverse, H1, in the other case.

template <class R>VIEW(UPolyQuot<R>,UPOLYNOMIAL)

template<class R>
UPolyRef<R>operator *(const UPolyQuot<R>& P1,const UPolyQuot<R>& P2)

Product of two polynomials in the quotient algebra

template<class R>
UPolyRef<R>operator *(const UPolyQuot<R>& P1,const typename UPoly-Quot<R>::coef_type & c)

Product of a polynomial by a scalar in the quotient algebra

```
template<class R>
UPolyRef<R>operator *(const typename UPolyQuot<R>::coef_type & c,const UPoly-
Quot<R>& P1)
```

Product of a scalar by a polynomial in the quotient algebra

```
template<class R>
UPolyRef<R>operator ^(const UPolyQuot<R>& P1,int n)
```

Power of a polynomial in the quotient algebra

```
template <class R>
ostream & operator«(ostream & os, const UPolyQuot<R>& p)
```

Output operator for a polynomial of the quotient algebra

Annexe E

Quelques algorithmes

E.1 Méthode de Sebastio e sylva

Univariate polynomials
Iterative Process for numerically approximating roots of a complex polynomial P.
Sebastiao e Silva method.
Based of the power method, but, instead of computing the succesive powers f^n, one rather computes the sequence of powers f^{2^n}.

template <class R,class P>
inline void solve(typename R::coef_type & x, P & p, const typename R::coef_type & x0, Sebastiao mth)

Computes the root of the polynomial p closest in modulus to $x0$, using Sebastiao e Silva method.
Result is given in x.

```
{
  typedef typename R::coef_type C;
  UPolyQuot<R>::assignpoly(p.rep);
  UPolyQuot<R>  u,un;
  typename R::index_type k=0, degree=UPolyQuot<R>::dim()-1;

  u = StartingValue(p, x0);
  un = u^2;
  un/=Norm(un);
  u-=un;
  Real<C>::TYPE bound(Tolerance);

  while ((Norm(u) > bound) && (k++ <= Max_Iter))
    {
      u=un;
      un=u^2;
      un/=un[degree];
      u-=un;
    }
  x =  un[degree-1];
```

```
  cout << " nb d'iterations : "<< k << endl;
}
```

E.2 Méthode des puissances inverses implicites

Finding a root of a polynomial system in n variables by the `Implicit Inverse Power method`. Computes the `Sylvester-like` matrix of the system. Polynomial system is passed as a standard vector of polynomials, for which we assume that the first element is one of the quotient by the set of the others polynomials, and the `shift` is, eventually included in this first element. Result is returned by a vector of size n. The second argument specifies the type of the sparse `Sylvester-like` matrix. For instance, `MPSolve(V, umfpack<double>());` will construct a sparse matrix with a representation of type UMFPACK<DOUBLE>

, and `ImplicitInverse(..)` will use specific library for this representation.

```
template <class POL, class RepS>
VectRef<array1d<typename RepS::coef_type> >  MPSolve(const VectStd<array1d<POL> > & f, RepS
  typedef typename RepS::coef_type C;
  MatSps<RepS> Syl;

  Syl =  Macaulay_T(f,RepS());

  cout << " Dim de S : " << Syl.nbrow() << endl;
  cout << " Non-zeros : " << Syl.nzero() << endl;
  int dim =1;

  //we assume dim(A) = product(di)
  for (typename array1d<POL>::index_type i = 1;i<f.size();i++) dim*=Degree(f[i]);
  cout << " Dim de A : " << dim << endl;

  VectStd<array1d<C> > Res(dim);

  unsigned int  nbiter = ImplicitInverse(Res,Syl);

  cout << " Nombre d'iterations : " << nbiter << endl << endl;

  C * x = new C[f.size()-1];
  for (typename array1d<POL>::index_type i = 0; i < f.size()-1; i++) x[i] = Res[i+1];

  return VectRef<array1d<C> > (new array1d<C>(f.size()-1,x));
}
```

Implicit inverse power method for computing a simple root of a polynomial system. Result is returned in x and computing from the `Sylvester-like` matrix of the system. For a multiple root, it also converges but more slowly.

```
template <class R, class M>
unsigned int ImplicitInverse(R & x, const MatSps<M> & s)
{
  unsigned int dim = s.nbcol(),j=1, e0 = x.size();
```

```
  R vn(dim), un(dim),u(dim);
  typedef typename MatSps<M>::solve_type Meth;

  for (unsigned int i=0; i < e0; i++) {
    un[i]=rand()%10;
    u[i]=0.0;
  }

  vn = Solve(s,un,Meth());

  if (vn[0]!= 0.0)
    for (unsigned int i=0; i < e0; i++) un[i] = vn[i]/vn[0];

  while ((Norm(un-u) > hold) && (j++<max_iter)){
    u=un;
    vn = Solve(s,un,Meth());
    if (vn[0]!= 0.0)
      for (unsigned int i=0; i < e0; i++) un[i] = vn[i]/vn[0];
  }

  for (unsigned int i=0; i < e0; i++) x[i] = un[i];

  return j;
}
```

Annexe F

Exemples-tests

F.1 Matrices structurées - Triangularisation matrice dense

```
#include "linear/array2d.H"
#include "Complex.H"
#define COMPLEX complex<double>

#include "linear/MATRIX.H"

#include "linear/Toeplitz.H"
#include "linear/array1d.H"
#include "algo/Triang.H"

#define VCT VectStd<array1d<double> >
#define MTR MatStr<toeplitz<double> >
#define MTD MatStd<array2d<double> >

int main(int argc, char** argv) {

  double u[]={1,2,3};
  VCT v(3,u), w(3);

  cout << v <<endl;
  cout << v*3 <<endl;
  cout << 3*v << endl;
  cout << v/2 <<endl<<endl;

  double t[]={1,2,3,4,5,6,7,8,9};

  MTR m(3,3,t);
  MTD n(3,3);

  n=m;

  cout << m <<endl;

  cout << " Produit matrice - vecteur : ";
  w = m * v;
```

```
    cout << w  << endl << endl;

    cout << " Triangularisation de la matrice n : \n ";
    Triang(n);
    cout << n << endl;
}
```

Ce programme donne en sortie :

```
 Opérations sur les vecteurs :
   [1, 2, 3]

   [3, 6, 9]

   [3, 6, 9]

   [0.5, 1, 1.5]

 Matrice de Toeplitz :
   [ 3 2 1 ]
   [ 4 3 2 ]
   [ 5 4 3 ]

 Produit matrice - vecteur : [10, 16, 22]

 Triangularisation de la matrice n :
    [ 1 4 7 ]
    [ 0 -3 -6 ]
    [ 0 0 0 ]
```

F.2 Racines d'un polynôme multivariable

```
 #define RepMMon "RepDTab.H"
 #define RepMPol "RepList.H"
 #include "polynomial/ALP_include.H"
 #include "arithm/Complex.H"
 #include "linear/array1d.H"
 #include "linear/MATRIX.H"
 #include "algo/MPSolve.H"
 #include "CPUClock.h"
 #define DsVector(C) VectStd<array1d<C> >
 #define Mon MMon<double,DynamicArray>
 #define Pol MPol<list<Mon >, Dlex<Mon > >
 #define YYSTYPE Pol
 #include <string.h>
 #include <math.h>

 list<Pol> l;

 #include "polynomial/parser.tab.h"
```

```
  #include "polynomial/lex.yy.c"
  #include "polynomial/parser.tab.c"
  #define MAT MatSps<compcol_mat_double>
  int yywrap(void);

  int main (int argc, char **argv)
  {
yyparse(l);
int nb =0;
for (list<Pol>::const_iterator i = l.begin(); i != l.end(); ++i){
  nb++;
  cout << *i << endl; }
cout << endl;

  Pol lp[nb];
int j =0;
     for (list<Pol>::const_iterator i = l.begin(); i != l.end(); ++i)
  {if (j < nb) lp[j]=*i; j++;}
DsVector(Pol) L(nb,lp);

MAT S;

S = Macaulay_T(L, compcol_mat_double());
cout << S << endl;
cout << MPSolve(L,compcol_mat_double()) << endl ;
cout << "Le temps mis est : " << T.time() << endl;
  return 0;
}
```

Sortie :

```
(-0.1)+(1)*x_1+(1)*x_2
(1)*x_1^4+(1)*x_1^3*x_2+(2)*x_1^2*x_2^2
(1)*x_1^2*x_2^2+(1)*x_1*x_2^3+(1)*x_2^4

 Dim de S : 36
 Non-zeros : 108
 Dim de A : 16
 Nombre d'iterations : 4

Solution obtenue : [0.0672811, -0.000781372]

Matrice de Macaulay :

[ -0.1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 -0.1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 -0.1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 -0.1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 -0.1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 -0.1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
```

```
[ 0 0 0 0 0 0 -0.1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 -0.1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 -0.1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 -0.1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 -0.1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 -0.1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 -0.1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 -0.1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 2 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 2 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 ]
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 ]

Le temps mis est : 0.03
```

Bibliographie

[1] W. Auzinger and H.J. Stetter. An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. In *Proc. Intern. Conf. on Numerical Math.*, volume 86 of *Int. Series of Numerical Math*, pages 12–30. Birkhäuser Verlag, 1988.

[2] Victor Y. Pan Bernard Mourrain, Didier Bondyfalat. Controlled iterative methods for solving polynomial systems. *Proceedings for Issac'98, ACM Press*, 1998.

[3] E. Bézout. *Théorie Générale des Équations Algébriques.* Paris, 1779.

[4] D. Bini and V.Y. Pan. *Polynomial and Matrix Computations*, volume 2: Selected Topics. Birkhäuser, Boston, 1998. To appear.

[5] 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, 1995)*, volume 32 of *Lectures in Applied Math.*, pages 189–210. Am. Math. Soc. Press, 1996.

[6] P. Dietmaier. The Stewart-Gough platform of general geometry can have 40 real postures. In *ARK*, pages 7–16, Strobl, 294JuinJuillet1998.

[7] G. Dos Reis. Vers une nouvelle approche du calcul scientifique en c++. Rapport de Recherche 3362, INRIA, 1998.

[8] M. Elkadi and B. Mourrain. Some applications of bezoutians in effective algebraic geometry. Rapport de Recherche 3572, INRIA, 1998.

[9] O. Faugeras. *Three-Dimensional Computer Vision: a Geometric Viewpoint.* mit-press, 1993.

[10] D. Lazard. Generalized Stewart Platform: How to compute with rigid motions? In *IMACS Symp. on Symbolic Computation*, pages 85–88, Lille, 14-17Juin1993.

[11] B. Mourrain. The 40 generic positions of a parallel robot. In M. Bronstein, editor, *ISSAC'93*, ACM press, pages 173–182, Kiev (Ukraine), Juillet1993.

[12] F. Ronga and T. Vust. Stewart platforms without computer?, 1992. Preprint.

[13] H. J. Stetter. Eigenproblems are at the heart of polynomial system solving. *SIGSAM Bulletin*, 30(4):22–25, 1996.

[14] J.J. Sylvester. On a theory of syzygetic relations of two rational integral functions, comprising an application to the theory of Sturm's functions, and that of the greatest algebraic common measure. *Philosophical Trans.*, 143:407–548, 1853.

[15] T. Veldhuizen. Blitz++, 1999. `http://monet.uwaterloo.ca/blitz/`.