Stage de fin de $2^{ième}$ année
ESSI
Ecole Supérieure en Sciences Informatiques
Option Calcul Scientifique pour l'Ingénieur

Linear algebra methods for solving polynomial systems of equations

Hélène Prieto

juin - septembre 1998

INRIA SOPHIA ANTIPOLIS

Projet SAGA

Sujet

Mise en place d'un environnement C++ permettant la manipulation de matrices structurées de type Toeplitz, Hankel, Quasi-Toeplitz, Quasi-Hankel, et de matrices creuses, en vue d'applications à la résolution d'équations polynomiales en une ou plusieurs variables.

Subject

Implementation of a C++ environment allowing the manipulation of structured matrices like Toeplitz matrices, Hankel matrices, Quasi-Toeplitz and Quasi-Hankel matrices, and the manipulation of sparse matrices. Applications of these tools for solving polynomial equations in one or several variables.

Mots Clés

Matrices structurées, matrices creuses, système d'équations polynomiales, idéal, algèbre quotient, opérateurs de multiplication, matrices de Sylvester, résultant de Macaulay, méthodes itératives controllées. C++, ALP, template, classe.

Key Words

Structured matrices, sparse matrices, system of polynomial equations, roots, ideal, quotient algebra, operators of multiplication, Sylvester matrices, Macaulay resultant, controlled iterative methods.
C++, ALP, template, class.

Remerciements

Ce stage ayant été très enrichissant pour moi, je tiens à remercier les personnes qui ont contribué à sa mise en place puis à son déroulement. Tout d'abord je voudrais remercier Ioannis Emiris, qui a été mon premier contact avec les membres du projet SAGA, Bernard Mourrain pour son accueil, la confiance qu'il m'a accordé et les aides qu'il m'a apporté tout au long de ce travail, Jean-Pierre Merlet pour son accueil et Didier Bondyfalat pour ses conseils. Je voudrais remercier aussi l'ESSI et son personnel pour son enseignement diversifié, son ouverture sur le monde extérieur et la possibilité qu'elle offre d'un cycle ingénieur accessible en formation continue.

Avertissement

La suite de ce rapport est en anglais. Le choix de cette langue plutôt que le francais est lié au contexte dans lequel ce stage s'est déroulé. En effet, les différents articles et livres concernant l'aspect théorique et qui m'ont largement aidé ainsi que les documentations sont en langue anglaise, de plus les commentaires et documentations du code devant être en anglais, il m'a paru naturel de poursuivre dans cette voie et de rédiger ce compte-rendu en anglais.

Contents

Chapter 1

Introduction

1.1 Context

1.1.1 Presentation of I.N.R.I.A.

I.N.R.I.A. is the **National Institute for Research in Computer Science and Control**. It is a scientific and technological institute operating under the dual authority of the ministry of research and the ministry of industry. **I.N.R.I.A.** is made up of five research units spread on the whole France, which are **INRIA Lorraine/Loria, INRIA Rennes/Irisa, INRIA Rhône-Alpes, INRIA Rocquencourt, INRIA Sophia-Antipolis**. **INRIA Rocquencourt** was the first research unit created and the headquarters of **I.N.R.I.A** is located on its site. The global number of people working for **I.N.R.I.A.** is about 2100, including 1700 scientists for a budget of 495 million french francs for 1997. The site of **Sophia-Antipolis** employs some 400 persons whom 140 are permanent staff. It has been founded in 1983, and has developed close ties with other laboratories in the region, as well as industry, both troughout Europe and inside Sophia-Antipolis itself.

1.1.2 The SAGA Project

For the unit of **Sophia-Antipolis** there are 20 projects and pre-projects for 4 themes. This practical period has taken place within the **SAGA** project, **S***oftware,* **A***lgebra,* **G***eometry and* **A***pplications* in the theme *Software Engineering and Symbolic Computing*, project whose scientific leader is **Jean-Pierre Merlet**. The aim of this project is to develop efficients methods concerning algebraic systems in particular some structured systems like those that appear frequently in some practical applications domains such as robotics, computer vision,... Its axis of researchs are:

1. Modelling : propose tools for the modelisation of geometric problems.
2. Analysis : study polynomial systems in order to get better a understanding of the geometry of solutions, in particular by using effective algebraic geometry.
3. Solving by approaches based on stable matrix computations and iteratives methods.
4. Applications : mecanisms theory, chemistry, computer vision, medical and signal processing.
5. Libraries : implementation of the studied algorithms in a library **ALP** (*Algèbre Linéaire pour les Polynômes*, see [18]).

Furthermore the project also works in some international and national projects, with some PME, and have contacts with some european and american universities.

1.1.3 Description and objectives of the work

The project is devoted for one part of its activity to solving polynomial systems of equations and for the other part to applications of these methods in robotics and computer vision. Some methods based on linear algebra computations begin to be known and used. They reduce the problem of solving a polynomial system to a matrix eigenproblem. These techniques exploit the properties of matrix constructions like those of **Bezout, Sylvester, Macaulay,**... or more recently toric resultant matrices. These matrices are highly structured. This motivates the study of their property to device efficient algorithms for solving polynomial systems.

The objectives of this work were

1. study some linear algebra methods and become familiar with the library **ALP**,
2. analyze and develop tools allowing the manipulation of structured and sparse matrices in the aim of testing some iterative methods for solving polynomial systems.

This work take place within the european project **FRISCO (LTR 21.024)** dedicated to the research on polynomial computations. It follows the previous works proposed since 1996 on similar subjects and so will extend the work yet done. The tools for this work are a Sun workstation, C++ language (with **g++** and **eg++**) and the library **ALP**.

1.2 Abstract

Many problems in robotics, computer vision, signal processing,..., involve the resolution of polynomial systems of equations. Some methods like minimization, Newton iterations are often used, but not always with guarentees on the result. Another known method is the Gröbner basis method, but it cannot be applied safely with floating point arithmetic, and has also a hight arithmetic computational cost. In order to study some controlled iterative methods for solving polynomial systems, we have to consider some linear algebraic objects, like structured or sparse matrices. We consider the quotient algebra of the polynomials ring over a field $\mathbb{K}$ by the ideal generated by one or several polynomials. First, some definitions of these objects and some notations are given, which allow to recall how solving polynomial equations reduces to a matrix eigenproblem. A well-known method for solving such a problem is the **power method** for which one we have considered two derivated methods, one for the univariate case, the **Sebastiao e Silva** method, the other one for the multivariate case, the **implicit inverse power method**. These methods have the great interest, in addition of their fast convergence, to give a way to compute a selected root, maximizing or minimizing the modulus of a fixed polynomial over all the roots of the system. This is often a need for many practical problems. The second part will be dedicated to the programming aspect of this work. So we will see how the algebraic structures and some operations on them have been defined and inserted in the yet existing library `ALP`. For this, we will present the library `ALP`, and next, the following structures which have been added during this work: classes of Toeplitz and Hankel matrices for which the product uses FFT, two classes of sparse matrices, class of elements of the quotient algebra. We also describe some specific operations like FFT in order to reduce the computational cost of some products. In last section, we have the implementation of the algorithms of the iterative methods mentionned above.

Part I

Theorical aspects

Chapter 2

Structured and sparse matrices.

In order to explain how some special kinds of matrices have correlations with operations on polynomials, we recall here some basic definitions of structured matrices (Toeplitz and Hankel matrices). We also take a look at sparse matrices and at their advantages from a computational point of view.

2.1 Structured matrices

2.1.1 Toeplitz matrices

By definition, a matrix $\mathtt{T} = (t_{i,j})$ is a **Toeplitz matrix** if for all i, j, the entry $t_{i,j}$ depends only on $i-j$, that is if $t_{i,j} = t_{i+1,j+1} = t_{i-j}$ for all pairs of (i, j) and (i+1,j+1) for which the entries $t_{i,j}$ and $t_{i+1,j+1}$ are defined. Associated with a **Toeplitz matrix**, we have the **Toeplitz operator** which is the projection of the multiplication by the fixed polynomial. That is, if $\mathbb{K}[x]$ is the ring of polynomials over the field $\mathbb{K}$, a map from $\mathbb{K}[x]$ to $\mathbb{K}[x]$. A great interest of this definition is to allow the computation of the product of a $n \times n$ Toeplitz matrix by a vector as a subvector of the coefficient vector of the product of two polynomials of $\mathbb{K}[\mathbf{x}]$. This can be done within $\mathcal{O}(nlog(n))$ operations using **FFT** (see [4]).

2.1.2 Hankel matrices

By definition, a matrix $\mathtt{H} = (h_{i,j})$ is a **Hankel matrix** if its entries $h_{i,j}$ depends only on $i+j$, that is, if $h_{i+1,j+1} = h_{i,j}$ for all pairs (i,j) for which the entries are defined. To this kind of matrix, we can associate a **Hankel operator** which is defined as the projection of the multiplication by a fixed Laurent polynomial (a Laurent polynomial is a polynomial of $\mathbb{K}[\mathbf{x}, \mathbf{x}^{-1}]$, that is a polynomial in both the variables $\mathbf{x}$ and $\mathbf{x}^{-1}$. So we can compute the product of a $(d+1) \times (d+1)$ Hankel matrix by a vector as a subvector of the product of a fixed polynomial $h(\mathbf{x})$ by a polynomial in $\mathbf{x}^{-1}$. This can be done within $\mathcal{O}(dlog(d))$ operations.

For more details on structured matrices and their correlations with univariate and multivariate polynomials see [4].

2.2 Sparse matrices

A sparse matrix is a matrix which number of non-zero terms is very small compared to its size. It presents some great advantages, like reducing the memory space required to store it and reducing the global number of arithmetic operations used for computations of the product matrix-vector for instance. Nowdays, many numerical and efficient algorithms are based on sparse linear algebra

(PDE, Graph theory,. . . , see [20]). In such applications it is not rare to treat (sparse) matrices of size $10^4 \times 10^4$ or even $10^5 \times 10^5$.

Chapter 3

Reduction of solving a polynomial system to a matrix eigenproblem

In this chapter we see how to reduce the resolution of a polynomial system to a matrix eigenproblem (see [1], [21]). But before some notations and definitions are recalled.

3.1 Notations - definitions

We consider $R = \mathbb{K}[x_1, \ldots, x_n]$ the polynomial ring over the field $\mathbb{K}$ which is assumed to be algebraically closed and $f_1, \ldots, f_m$, m polynomials of R, defining the polynomial system $f_1(\mathbf{x}) = 0, \ldots, f_m(\mathbf{x}) = 0$. We also consider I the ideal ($\mathbf{x} = (x_1, \ldots, x_n)$) generated by this set of polynomials. This particular approach is based on the study of the quotient algebra $\mathcal{A} = \mathbb{K}[\mathbf{x}]/I$. We assume here that $\mathcal{A}$ is of finite dimension D over $\mathbb{K}$. We denote $\mathcal{Z}(I) = \{\zeta \in \mathbb{C}^n; f_1(\zeta) = \ldots = f_m(\zeta) = 0\}$, the set of roots. Since $\mathcal{A}$ is of finite dimension : $\mathcal{Z}(I)$ is finite : $\mathcal{Z}(I) = \{\zeta_1, \ldots, \zeta_d\}$ with $d \leq D$. To this set of roots we associate a fundamental set of orthogonals idempotents (see [5]) $\mathbf{e}_1, \ldots, \mathbf{e}_d$, such that :

$$\mathbf{e}_1 + \cdots + \mathbf{e}_d \equiv 1, \text{ and } \mathbf{e}_i\,\mathbf{e}_j \equiv \left\{ \begin{array}{l} 0 \text{ if } i \neq j, \\ \mathbf{e}_i \text{ if } i = j, \end{array} \right.$$

We denote by $\widehat{R}$ the dual space of R and by $\widehat{\mathcal{A}}$ the dual space of $\mathcal{A}$, $\widehat{\mathcal{A}} = \{\Lambda \in \widehat{R},\ \Lambda(I) = 0\}$ (also denoted by $I^{\perp}$). Now we can introduce the operator of multiplication on which is based this study.

3.2 Operators of multiplication and their associated matrices

For each $a \in \mathcal{A}$, we denote by

$$\begin{array}{rcl} M_a : \mathcal{A} & \rightarrow & \mathcal{A} \\ b & \mapsto & a\,b \end{array}$$

the map of multiplication by a in $\mathcal{A}$, and by

$$\begin{array}{rcl} M_a^{\mathrm{t}} : \widehat{\mathcal{A}} & \rightarrow & \widehat{\mathcal{A}} \\ \Lambda & \mapsto & a \cdot \Lambda \end{array}$$

its transposed map.

By ([5], theorem 2.1) we know there exists a basis of $\mathcal{A}$ such that for all $a \in \mathcal{A}$ the matrix $\mathtt{M}_a$ of M_a in this basis is of the form

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

where $M_{a,i}$ of the form

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

is the matrix of multiplication by a in $\mathcal{A}$. In the case of a simple root ζ_i, we have $\mathtt{M}_{a,i} = (a(\zeta_i))$.

For each $\zeta \in \mathbb{K}^n$ we introduce,

$$\begin{array}{rccc} \mathbf{1}_\zeta : & R & \rightarrow & \mathbb{K} \\ & p & \mapsto & p(\zeta) \end{array}$$

We note that $\mathbf{1}_\zeta \in \widehat{\mathcal{A}}$ if and only if $\zeta \in \mathcal{Z}(I)$.

In the case of multiple roots it would be necessary to introduce some additional linear forms involving higher order differential forms. But hereafter for simplicity we only consider simple roots.

We express now some propositions and remarks:

For any pair $a, b \in \mathcal{A}$,

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

so that $\mathbf{1}_{\zeta_i}$ is an eigenvector of $M_a^{\mathtt{t}}$, for the eigenvalue $a(\zeta_i)$. Moreover for any pair $a, b \in \mathcal{A}$, the multiplication maps M_a, M_b commute with each other. It follows that they share common eigenvector spaces. And so, the common eigenvectors of M_a^t for all $a \in \mathcal{A}$ are the non-zero multiples of $\mathbf{1}_{\zeta_i}$, for $i = 1, \ldots, d$ ([5], proposition 2.3).

If a root ζ_i is simple, the eigenvector of $M_a^{\mathtt{t}}$ associated to the eigenvalue $a(\zeta_i)$ is $\mathbf{1}_{\zeta_i}$. The coordinates of $\mathbf{1}_{\zeta_i}$ in the dual basis of $(\mathbf{x}^{\alpha_1}, \ldots, \mathbf{x}^{\alpha_D})$ (any monomial basis of $\mathcal{A}$) are $(\zeta_i^{\alpha_1}, \ldots, \zeta_i^{\alpha_D})$, that is the evaluation of this basis of $\mathcal{A}$ in the root ζ_i.

This yields the following algorithm:

3.3 Algorithm of computing the simple roots of a polynomial system $f_1 = \cdots = f_m = 0$.

Algorithm 3.3.1

1. *Compute the transpose of the matrix of multiplication* $\mathtt{M}_a$ *for* $a \in \mathcal{A}$.
2. *Compute its eigenvectors* $\mathbf{v}_i = (v_{i,1}, v_{i,x_1}, \ldots, v_{i,x_n}, \ldots)$ *for* $i = 1, \ldots, d$.
3. *For* $i = 1, \ldots, d$, *compute and output*

$$\zeta_i = (\frac{v_{i,x_1}}{v_{i,1}}, \ldots, \frac{v_{i,x_n}}{v_{i,1}}).$$

Let us start with an example of application of this algorithm.

Example 3.3.2 (see [17])
Let $n = 2$,

$$p1 = x^2 + 2xy - x - 1,$$
$$p2 = x^2 + y^2 - 8x.$$

A basis of $\mathcal{A}$ *is* $\mathcal{B} = (1, x, y, xy)$.
We apply the following program to this previous system:

```
#include <complex.h>
#include "basic.H"
#include "linear/Lapack.H"

#define VCT Vector<Std<array1d<double> > >
#define MAT Matrix<Std<lapack<double> > >
#define MATC Matrix<Std<lapack<complex<double> > > >

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

  double v[]={0,1,0,-14.0/5.0,1,1,0,-12/5,0,0,0,1.0/5.0,0,-2.,1.,29.0/5.0};

  // Construction of the transposed of the matrix of multiplication.
  MAT A(4,4,v);

  cout <<A<< endl;

  // Construction of the matrix for the result
  MATC B(4,4);

  // Computation of the eigenvectors of A.
  // The matrix of eigenvector is put in B.
  Std<lapack<double> >::reigvct(B.rep,A.rep);

  cout << B << endl;
};
```

and we obtain,
the transposed of the matrix of multiplication by x_1,

```
[ 0      1  0   0  ]
[ 1      1  0  -2  ]
[ 0      0  0   1  ]
[ -2.8  -2 0.2 5.8 ]
```

and the normalized matrix of generalized eigenvectors (in columns),

```
[ (1,0)         (1,0)                (1,0)                   (1,0)        ]
[ (6.700098,0)  (-0.193954,0.205206)  (-0.19395427,-0.205206) (0.420481,0) ]
[ (-2.775830,0) (-0.618371,-1.389510) (-0.61837124,1.389519)  (1.478870,0) ]
[ (-18.60050,0) (0.405268,-1.389519)  (0.40526841,-1.389519)  (0.611838,0) ]
```

where the columns of this matrix are the evaluation of $\mathcal{B}$ *in the roots of* $p1(x, y) = p2(x, y) = 0$.
Thus, we can deduce that these roots are given by

ζ_1	ζ_1	ζ_1	ζ_1
6.700098	*-0.193954+0.205206i*	*-0.193954i-0.205206i*	*0.420481*
-2.775830	*-0.618371-1.38951i*	*-0.618371+1.38951i*	*1.478870*

Chapter 4

The classical power method and inverse power method

The previous algorithm reduces the solution of a polynomial system to eigenvector computations. Before studying the derived methods applied during this work, let us have a look to the classical (inverse) power method for matrices (for more details see [5]).

Let $\mathtt{M}_a$ be the matrix of multiplication by a in a basis B of $\mathcal{A}$ such that $a(\zeta_i) \neq 0$ for $i = 1, \ldots, d$. By its definition, M_a is invertible and a is invertible in $\mathcal{A}$.

Let $\mathbf{v}_0 = \mathbf{w}_0$ be the coordinate vector of an element of $\widehat{\mathcal{A}}$ in the dual basis of B.

$$\mathbf{w}_k = \frac{1}{\|\mathbf{w}_{k-1}\|}\mathtt{M}_a^t \mathbf{w}_{k-1} \text{ et } \mathbf{v}_k = \frac{1}{\|\mathbf{v}_{k-1}\|}(\mathtt{M}_a^t)^{-1}\mathbf{v}_{k-1},$$

$k = 1, 2, \ldots$, respectively.
If ζ is a simple root maximizing (resp. minimizing) $|a|$ and if $\mathbf{w} = \mathbf{v} = (\zeta^\alpha)$ is the monomial basis evaluated in ζ, we have

$$\lim_{k\to\infty} \mathbf{w}_k = \mathbf{w} \text{ (resp. } \lim_{k\to\infty} \mathbf{v}_k = \mathbf{v}).$$

This gives us a way to select an eigenvector of $\mathtt{M}_a^t$ corresponding to the root maximizing (resp. minimizing) the modulus of a on $\mathcal{Z}(I)$.
Remarks :

- If there are k roots minimizing the modulus de a on $\mathcal{Z}(I)$, it is also possible to recover the eigenvectors corresponding to these roots from the successive vectors $v_n, \ldots, v_{n+k-1}$.

- If the root ζ is multiple, the process also converge probalistically to an eigenvector of $\mathtt{M}_a$ or $\mathtt{M}_a^t$ but more slowly.

In the next chapter, we are going to consider this method adapted to the univariate case.

Chapter 5

Univariate case

Before describing the **Sebastio e Silva method**, which is a derivative of the **classical power** method, let us see some particularities of the univariate case.

5.1 Univariate case - particularities

Here we consider p, a polynomial in one variable x with coefficients in $\mathbb{K}$ of degree d, and $\mathcal{A} = \mathbb{K}[x]/(p)$.
A traditionnal basis of the quotient algebra $\mathcal{A}$ is the monomial basis :

$$M = \{1, x^1, \ldots, x^{d-1}\}$$

We have another basis of interest, i.e the Horner basis, involving the polynomials appearing in the Horner scheme associated to p :

$$H = \{h_{d-i} \;=\; a_0 x^{d-i} + a_1 x^{d-i-1} + \ldots + a_{d-i},\ 1 \leq i \leq d-1\}$$

A great interest of the Horner basis is that only 2 convolutions for squaring f^2 in $\mathcal{A}$, and 4 for the product fg are required (see [8] p 170).

Another interest of this last basis is that if α is a simple root, we have the idempotent $\mathbf{e}_\alpha = \frac{1}{p^{'}(\alpha)} \frac{p(x)}{x-\alpha}$, which decomposes in the Horner basis as $\mathbf{e}_\alpha = \frac{1}{p^{'}(\alpha)}(1, \alpha, \alpha^2, \ldots, \alpha^{d-1})$. It means that if a method converges to some $\mathbf{e}_\alpha$ then we will not only obtain the approximate root α itself but also all its powers, up to α^{d-1}. This can be used to evaluate any algebraic expression such as $g(\alpha)$, not only for a polynomial g of degree $d-1$ but also for every polynomial expression since every polynomial can be reduced modulo p to a polynomial of degree less or equal to $d-1$.

5.2 The Sebastiao e Silva method

The **Sebastiao e Silva method** ([8]) is an iterative process for approximating numerically one root of a polynomial p. It is essentially the same as the **power method** described above, but it is an accelerated process: Instead of computing the successive powers f^n, we compute the successive squaes f^{2^n}.

Algorithm 5.2.1

1. *Choose a starting value f of $\mathcal{A}$.*
2. *Iterate* $f_{n+1} = \frac{(f_n)^2}{\|(f_n)^2\|}$ *in $\mathcal{A}$.*

And then, if there exists a simple root α maximizing $|f|$, we obtain

$$\lim_{n\to\infty} f_{n+1} = \mathbf{e}_\alpha.$$

Here, we have choosen the euclidian norm for the vectors. The process will be stopped for an error between two iterates less than a value ϵ fixed in advance.

Moreover, if the process is written in the Horner basis, an additionnal interest is to have a simple way for selecting the root which is nearest to a given point, as illustred below.
Let us assume that, for a given point $t \in \mathbb{K}$, there is only one simple root α closest to t. If we start the algorithm from $J_t = (1, t, t^2, \ldots, t^{d-1}) = \frac{p(x) - p(t)}{(x-t)p'(t)}$, so that $J_t(\alpha) = \frac{-p(t)}{\alpha - t}$, then the evaluation of J_t at the roots of p is maximal when $|\alpha - t|$ is minimal. Thus if we want the smallest root of p we start the process from J_0. If we want the largest we should reverse p and look for its smallest root (see [8]).

If we want to obtain other roots after computing a first root α, we have to proceed to a deflation of the polynomial p. We have two different deflation processes, the first is the most natural, the explicit deflation, which is to compute explicitly $\frac{p(x)}{x-\alpha}$. The second deflation process, called implicit deflation, is the following. One root α is computed with a starting value f, and the method produced a first idempotent $\mathbf{e}_\alpha$. To get a second root we choose any g of $\mathcal{A}$ (possibly f again) and start the method from the initial value $(g - g\mathbf{e}_\alpha)$. Since $(g - g\mathbf{e}_\alpha)(\alpha) = 0$, α is no longer dominant, and the process cannot converge again to $\mathbf{e}_\alpha$. Then, when exiting at all, we will obtain another idempotent $\mathbf{e}_\beta$ where β is the dominant root associated to g, α will be ignored. For the next step, we choose h (possibly f or g) and start the method from the initial value $h - h(\mathbf{e}_\alpha + \mathbf{e}_\beta)$... Continuing in this way will ultimately collect as many roots of p as desired (for more details see [8]).

For this work we have choosen the implicit deflation.

5.3 Tests

Here are some results of experimentation for the **Sebastio e Silva method**. Some polynomials with random coefficients real and complex have been generated, and we applied the method for computing one root for a degree of 10000 and 100000. We have also computed all the root for a polynomial with random complex coefficients of degree 10000. One can observe that, for a polynomial p with random coefficients, as its degree increases, the roots of p will come closer and closer to the unit circle of the complex plane.

One root has also been computed for some univariate polynomials called **truncated exponential**, and this for different degrees. These polynomials have the following characteristics,

$$\sum_{i=0}^{n} \frac{x^i}{i!} \tag{5.1}$$

This example is supplied by D.A. Bini (FRISCO test suite, see [3]), Oct. 1996).

1. The polynomial is obtained by truncating the series of e^x.
2. The roots of this polynomial are located along a curve. To have a chance to obtain a correct result for a degree > 20, it is necessary to use a multiple precision library. Here it is the class **SCL** of **ALP** which uses the **GNU MP** library ([11]).

In the following table we have the results for

1. Polynomials with real random coefficients, of degrees 10000 and 100000.
2. Polynomials with complex random coefficients, of degrees 10000, 100000 and 1000000.
3. Exponential polynomials $\sum_i \frac{x^i}{i!}$, for some degrees of 100, 150, 200.

	d	x0	k	T
`double`	10000	0.0	8	22.03s
`double`	100000	0.0	8	346.52s
`complex`	10000	(0.4,0.6)	6	16.86s
`complex`	100000	(0.4,0.6)	6	292.02s
`complex`	1000000	(0.4,0.6)	7	2520s
`exp`	100	(0.0,0.8)	6	45s
`exp`	150	(0.0,0.8)	6	59s
`exp`	200	(0.0,0.8)	6	101s

where d is the degree of the polynomial, $x0$ the starting value for the iterative process, k is the number of iterations to compute one root and T is the total time of the computation. These experimentations have been carried on a Dec workstation with 496M of local memory.

Chapter 6

Multivariate case

In this section, we are going to extend the approach of the **power method** to the multivariate case. But before we mention some of the main particular difficulties of the multivariate case.

6.1 Multivariate case - particularities

We lose the notion of the leading monomial of the highest degree.
We have no natural monomial basis for representing the quotient modulo a set of polynomials.
The homogenization of the polynomials might introduce spurious solutions, ("the solutions at infinity"), to a polynomial system of equations (see [5]).

For these reasons and some others, the study needs to be restricted to the generic cases of two types, which are the **Macaulay case** (see [16] [12]) and the **toric case** (see [7] [22]). In this work, only the first case is considered.

Now we consider the quotient algebra $\mathcal{A} = \mathbb{K}[\mathbf{x}_1, \ldots, \mathbf{x}_n]/(f_1, \ldots, f_m)$ where $f_1, \ldots, f_m$ are m polynomials in n variables.

6.2 Matrices of multiplication in $\mathcal{A}$

In order to extend the **power method** we have to compute the matrix of multiplication $\mathtt{M}_{f_0}$ for an element $f_0 \in \mathcal{A}$. Usually, this matrix is not available directly from the input equations. However it can be recovered from a Sylvester-like matrix $\mathtt{S}$ which is a resultant matrix. We will see later how such a matrix can be built. For our method we will construct matrices $\mathtt{S}$ with the following properties :

Hypotheses 6.2.1 *The matrix $\mathtt{S}$ is a square matrix and have the following form :*

$$\mathtt{S} = \begin{pmatrix} \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} \end{pmatrix} \tag{6.1}$$

such that

1. *its rows are indexed by monomials $(\mathbf{x}^\alpha)_{\alpha \in F}$,*
2. *the set of monomials $B_0 = (\mathbf{x}^\alpha)_{\alpha \in E_0}$ indexing the rows of the block $(\mathtt{A}\ \mathtt{B})$ is a basis of $\mathcal{A} = R/(f_1, \ldots, f_m)$,*
3. *the columns of $\begin{pmatrix} \mathtt{A} \\ \mathtt{C} \end{pmatrix}$ represents the elements $\mathbf{x}^\alpha f_0$ for $\alpha \in E_0$, expressed as linear combinations of the monomials $(\mathbf{x}^\beta)_{\beta \in F}$,*

4. the columns of $\begin{pmatrix} \mathtt{B} \\ \mathtt{D} \end{pmatrix}$ *represent some multiples of the polynomials* $f_1, \ldots, f_m$*, expressed as linear combinations of the monomials* $(\mathbf{x}^{\beta})_{\beta \in F}$*,*

5. the block $\mathtt{D}$ *is invertible.*

For any matrix $\mathtt{S}$ satisfying these hypotheses, we may obtain the map of multiplication by f_0 modulo $f_1, \ldots, f_n$ as, in the basis $B_0 = (\mathbf{x}^{\alpha})_{\alpha \in E_0}$ of $\mathcal{A}$, $\mathtt{M}_{f_0}$ is the Schur complement of $\mathtt{D}$ in $\mathtt{S}$:

$$\mathtt{M}_{f_0} = \mathtt{A} - \mathtt{B}\,\mathtt{D}^{-1}\,\mathtt{C}.$$

(for more details see [5])

6.3 Sylvester-like matrices

This construction generalizes the Sylvester map for two polynomials in one variable. In this work we consider the construction due to Macaulay of a resultant matrix for $n+1$ polynomials $f_0 \ldots, f_n$, in n variables.

Let $\mathcal{V}_0, \ldots, \mathcal{V}_n$, $n+1$ vector spaces generated by $\mathbf{x}^{E_i} = \{\mathbf{x}^{\alpha}, \alpha \in E_i\}$, where E_i is the set of exponents specified hereafter,

$$E_i = \{\beta_{i,1}, \beta_{i,2}, \ldots\}.$$

Let $\mathcal{V}$ be the vector space generated by all the monomials of the polynomials $f_i \mathbf{x}^{\beta_i}$ for $\beta_i \in E_i$. This set of monomials is denoted by $\mathbf{x}^F = (\mathbf{x}^{\beta})_{\beta \in F}$. We define the following map :

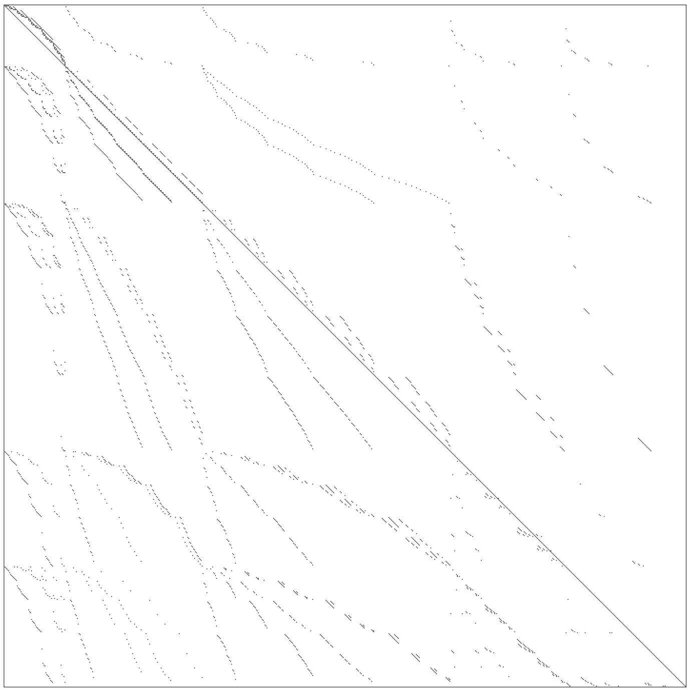
$$\begin{array}{rcl} S : \mathcal{V}_0 \times \cdots \times \mathcal{V}_n & \to & \mathcal{V} \\ (q_0, \ldots, q_n) & \mapsto & \displaystyle\sum_{i=0}^{n} f_i\, q_i. \end{array} \tag{6.2}$$

The matrix $\mathtt{S}$ associated to S in the monomial basis of $\mathcal{V}_0 \times \cdots \times \mathcal{V}_n$ and $\mathcal{V}$ is

$$\begin{array}{cc} & \begin{array}{ccc} \overbrace{\qquad\qquad}^{\mathcal{V}_0} & \qquad\qquad & \overbrace{\qquad\qquad}^{\mathcal{V}_n} \end{array} \\ \mathcal{V}\left\{\begin{array}{l} \mathbf{x}^{\alpha_1} \\ \cdot \\ \cdot \\ \cdot \\ \mathbf{x}^{\alpha_N} \end{array}\right. & \left[\begin{array}{c|c|c} \cdot & & \cdot \\ \cdot & & \cdot \\ \mathbf{x}^{\beta_{0,1}} f_0 \quad \cdots & \cdots\cdots\cdots & \mathbf{x}^{\beta_{n,1}} f_n \quad \cdots \\ \cdot & & \cdot \\ \cdot & & \cdot \end{array}\right]. \end{array}$$

In the construction due to Macaulay, if $d_0, \ldots, d_n$ are the degrees of the polynomials $f_0, \ldots, f_n$, and $\nu = d_0 + \ldots + d_n - n$, the set $\mathbf{x}^F$ is the set of all the monomials of degree $\leq \nu$ in the variables $x_1, \ldots, x_n$ and E_i is the subset of the monomials of degree $\nu - d_i$. We can check that the hypothesis 6.2.1 are well respected. In this resultant construction, it may happen that for specific values of the coefficients of the input polynomials, the matrix $\mathtt{D}$ become singular. In this case another construction must be applied (e.g. **Gröbner basis**).

Here is a example of such a matrix of size 715 :



System of 4 equations in 3 variables, of degrees 4, 4, 2 and 2.

Now we can applied the **implicit inverse power method** for generic systems of equations.

6.4 Implicit inverse method

First let us recall that this method allows to compute a root of our system that minimizes the modulus of f_0. An advantage versus the **power method** is the possible use of shifts of the variable, that is the transformation from the matrix $\mathtt{M}_{f_0}^{-1}$ to the matrix $(\mathtt{M}_{f_0} - \sigma I)^{-1}$, where σ is closed to the selected eigenvalue of $\mathtt{M}_{f_0}$, for the convergence acceleration (see [13]).

So under the hypotheses 6.2.1 and for $\mathtt{S}$ invertible of the form

$$\mathtt{S}^{-1} = \begin{pmatrix} \mathtt{U} & \mathtt{V} \\ \mathtt{Z} & \mathtt{W} \end{pmatrix}$$

We have $\mathtt{M}_{f_0}$ invertible, and $\mathtt{U} = \mathtt{M}_{f_0}^{-1}$.

This gives the following iterative algorithm,

Algorithm 6.4.1 IMPLICIT INVERSE POWER METHOD.

1. *Choice of a random vector* $\mathbf{u}_0$.
2. *Solving the system* $\mathtt{S}^t \begin{bmatrix} \mathbf{v}_{n+1} \\ \mathbf{v}'_{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{u}_n \\ 0 \end{bmatrix}$.
3. *If* $v_{n+1,1}$ *is the first coordinate of* v_{n+1} *and if it is not zero, then* $u_{n+1} = \frac{1}{v_{n+1,1}} v_{n+1}$, *else stop.*

We obtain, if $\zeta \in \mathcal{Z}(I)$ is a simple root minimizing $|f_0|$ and $\sigma = (\zeta^\alpha)_{\alpha \in E_0}$,

$$\lim_{n\to\infty} u_n = \sigma.$$

To accelerate this algorithm, we will first compute the LU-decomposition of the matrix $\mathtt{S}^t$. Moreover the matrix $\mathtt{S}^t$ is sparse since the number of non-zero terms per columns is bounded by the number of monomials $f_0, \ldots, f_m$, which is small compared to the size of the matrix. This allows to perfom each step of the previous algorithm in $\mathcal{O}(C\,N + N\log^2(N))$ arithmetic operations, where C is the bound on the number of arithmetic operations required to multiply $\mathtt{S}$ by a vector. Practically, if $\mathtt{S}$ has $\mathcal{O}(N)$ non-zero coefficients, then $C = \mathcal{O}(N)$ and each step can be performed by using $\mathcal{O}(N^2)$ operations versus known bounds of order N^3. Moreover the matrix $\mathtt{S}$ and its block $\mathtt{A}$, $\mathtt{B}$, $\mathtt{C}$, $\mathtt{D}$ are structured matrices, structure which can be used to simplify multiplication of such a matrix by a vector (see [4] [7]). But in this work, we do not use this structure, nevertheless we may multiply such matrices by vectors and solve linear system effectively, based on exploiting the sparsity and the previous remark.

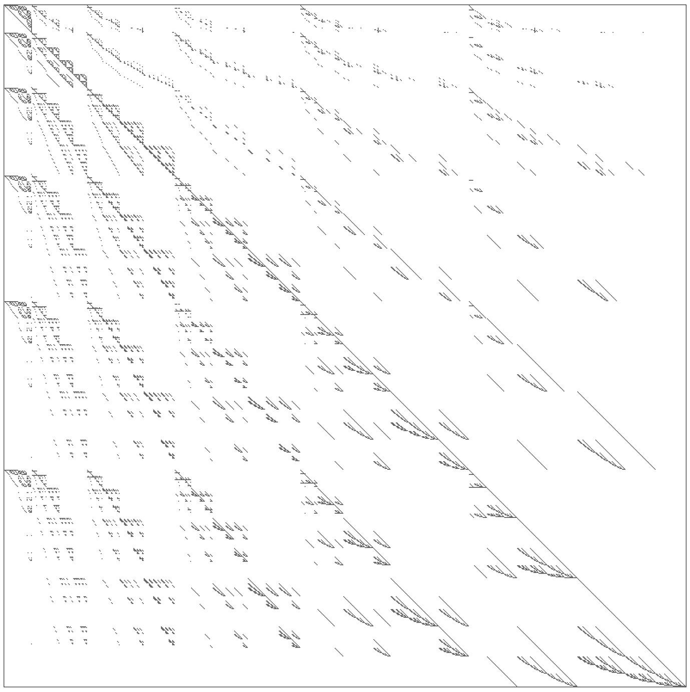
Before presenting some results of different tests, we are going to see a particular case of overconstrained system.

6.5 Overconstrained system

We can also construct a matrix $\mathtt{S}$ for an overconstrained system, that is a system of equations $f_1, \ldots, f_m$ with $m > n$ defining a finite number of roots, by a generalization of the construction previously described. We construct a matrix $\mathtt{S}$ for the polynomials $f_1, \ldots, f_n$ and add new columns corresponding to the multiples of the polynomials $f_{n+1}, \ldots, f_m$. By extracting a square matrix of this rectangular matrix we obtain a matrix $\mathtt{S}$ which follows hypotheses 6.2.1.

In this work we have considered a particular case, which is the case where $\mathcal{A}$ is of dimension 1. So there is only one root, $\chi = (\chi_1, \ldots, \chi_n)$. A basis of $\mathcal{A}$ is 1, and the matrix of multiplication by x_i is $[\chi_i]$. Then for any matrix $\mathtt{S}$ satisfying hypotheses 6.2.1 with $f_0 = x_i$, $\mathtt{A}$ is a one-by-one matrix and we have $[\chi_i] = \mathtt{A} - \mathtt{B}\,\mathtt{D}^{-1}\mathtt{C}$. In this case, only one resolution of a linear system is required. For solving this kind of system, we have extracted a matrix $\mathtt{S}'$ from $\mathtt{S}$ by removing its first row and putting the first column in a vector c, so we obtain a matrix which has one row less and one column less than $\mathtt{S}$. Then the system to solve is $\mathtt{S}'u = -c$ where c is the first column of $\mathtt{S}$. And we directly obtain $(\zeta_1, \ldots, \zeta_n)$.

Here is the matrix of size 792, obtained by this last method, for a system of 6 equations in 5 variables coming from the Kruppa auto-calibration problem,



System of Kruppa equations

It appears in a problem of reconstruction in Computer vision, where any pair of points, in correspondence to the images, gives a polynomial equation. Thus we can find almost as many equations as we want, but there is only one solution (see [10]).

6.6 Tests

The following table contains the results of our experiment for the **implicit inverse method** using the structures implemented during this work. These different classes will be presented in the next part of this report.
Many of the following examples have two results, one, $*G$, by using the **GMRES** solver ([15]) developped by R. Pozo and his library **SparseLib++** ([19]), the other, by $*U$ using a direct method with **UMFPACK** routines ([23]) for solving the sparse linear system $\mathtt{S}x = b$.
The examples $s22G$ and $s22U$ are both the following system ([9]) :

$$-11 + 2x^2 + 3y^2 = 0$$
$$-3 + x^2 - y^2 = 0$$

$s44*$ are systems of two equations in two variables, both of degree 4. $s4422*$ are systems of four equations in 4 variables, of degree 4, 4, 2 and 2.
$sph*$ are the intersection of a sphere with an other quadric and a plane ([9]).

$$x^2 + y^2 + z^2 = 4$$
$$xy + z^2 - 1 = 0$$
$$x + y - z = 0$$

Examples $cyc3*$ are the following, for $n = 3$: Cyclic n-roots (see [2])

$$\begin{gathered} x_1 + x_2 + \cdots + x_n = 0 \\ x_1x_2 + x_2x_3 + \cdots + x_nx_1 = 0 \\ \vdots \\ x_1 \cdots x_{n-1} + x_2 \cdots x_n + \cdots + x_nx_1 \cdots x_{n-2} = 0 \\ x_1x_2 \cdots x_n = 1 \end{gathered}$$

Results $kat6U$ (resp. $kat8U$) are, for $n = 6$ (resp. $n = 8$): Katsura(N) (see [2])

$$\begin{gathered} u_m - \sum_{i=-N}^{N} u_i u_{m-i}; m = -N+1..N+1 \\ 1 - \sum_{i=-N}^{N} u_i \\ u_{-m} = u_m; m = 1..2N-1 \\ u_m = 0; m = N+1..2N-1 \end{gathered}$$

The last result *kruppa* corresponds to the Kruppa equations of a reconstruction problem in Computational Vision (see [10]) reduced to an overconstrained system of 6 quadrics in a space of dimension 5. The way to find the root was explained above.

	N	S	D	n	k	T
`s22G`	10	30	4	2	2	0.020s
`s22U`	10	30	4	2	2	0.010s
`s44G`	36	108	16	2	4	0.020s
`s44U`	36	108	16	2	4	0.010s
`s4422G`	715	3449	64	4	46	149s
`s4422U`	715	3449	64	4	4	0.67s
`sphG`	20	67	4	3	9	0.060s
`sphU`	20	67	4	3	9	0.060s
`cyc3G`	35	117	6	3	17	0.080s
`cyc3U`	35	117	6	3	17	0.040s
`kat6U`	1716	26460	64	6	13	22.53s
`kat8G`	24310	556086	256	8	46	125mn
`kruppa`	792	15840	1	5	1	1.170s

In this table, N is the dimension of the matrix `S` (that is, the matrix has size $N \times N$), S is the number of non-zero entries of the matrix `S`, D is the dimension of $\mathcal{A}$, n is the number of variables, k is the number of iterations required for an error less than $\epsilon = 10^{-4}$, and T is the total time of the computation, it includes the time of construction of the resultant matrix (for instance, 153.91 seconds for the test $kat8G$).

Here is an example of program written in order to solve the system described above (not used for the kruppa equations),

```
#define RepMMon "RepDTab.H"
#define RepMPol "RepList.H"
// Definition of standard vector with arrai1d<C> container
#define DsVector(C) Vector<Std<array1d<C> > >
// Definition of monomial with coefficients of type double
// and a DynamicArray container
#define Mon MMon<double,DynamicArray>
// Definition of polynomial with monomials of type MMon
// and lexicographic order Plex
```

```
#define Pol MPol<list<Mon >, Plex<Mon > >
// Used by the parser
#define YYSTYPE Pol
list<Pol> l;
// Definition of sparse matrix with a compcol_mat_double container
#define MAT Matrix<Sps<compcol_mat_double> >
// Definition of sparse matrix with a umfpack<double> container
#define MAT Matrix<Sps<umfpack<double> > >
int yywrap(void);

int main (int argc, char **argv)
{
  // Construction of the list of polynomials, which has been
  // read from a file with the help of a parser, yparse();
      yyparse();
      int nb =0;
      for (list<Pol>::const_iterator i = l.begin(); i != l.end(); ++i){
        nb++;
        cout << *i << endl; }

  // Construction of the vector containing the polynomials and
  // which will pass to the solver.
      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);
      CPUClock T;

  // Fist we call the function of resolution with an instance of
  // compcol_mat_double, in order to use the GMRES solver.
      cout << MPSolve(L,compcol_mat_double()) << endl ;

  // Now we call the function of resolution with an instance of
  // umfpack<double>, in order to use the UMFPACK routines.
      cout << MPSolve(L,umfpack<double>()) << endl ;

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

Part II

Programming work

In the presentation of the programming work, first we describe the library **ALP**(*Algèbre Linéaire pour les Polynômes*), in which this work takes place. Next we will see how the algebraic objects and the methods described in the previous part have been expressed in the programming language.

Chapter 7

Library ALP

The library **ALP** is a tool to manipulate some objects of linear algebra like vectors, matrices, monomials and polynomials. Its development has begun in 1995 during the practical period of third year ESSI of Fabrice Luvigny. **ALP** has the objectif to be as generic as possible, that is, for instance, to have the possibility to index the rows and/or the columns of a matrix by monomials or polynomials as well as by integers. So the principal word which characterizes this library is **template**. Moreover other concerns are a minimum occupation of the space memory and a minimum computational cost. For this, as less as possible copies of objects are allowed. An interesting way is the non-evaluation of some operations such as multiplication (or addition, etc...) of a matrix by a matrix or of a matrix by a vector. Only one evaluation is done for a necessary element at the necessary moment. That is, instead of computing an object which contains the evaluation of an operation, it returns the way to get and compute separatly the elements. It is equivalent to virtual function mechanism but with efficiency. For this purpose, there are special invisible classes of matrices and vectors which cannot be instancied, described in section 13.1.3. Another way to prevent too many copies of objects is to have some functions which return pointers, easily deletable. For this special classes of matrix and vector are implemented, only used as return type of functions, Matrix<Ref<...> > and Vector<Ref<...>.

We have 3 levels of structures :

1. Interfaces, like Matrix<...>, Vector<...>, UPol<...>, MPol<...>, MMon<...>,..., which are the link between the C code and the C++ objects.
2. Specific implementations, like Std<...>, Str<...>, Ref<...>, Hdl<...>, ...
3. Representations, definitions and elementary operations of the objects (e.g. **array1d<...>**, **upar<....>** which are one-dimension arrays).

For instance, a one-dimensional array of double seen as a vector, using the standard implementation, will be defined as
Vector<Std<array1d<double> > >.
A one-dimensional array of double derived from array1d<double> seen as an univariate polynomial with a specialized implementation, will be defined as
UPol<Spl<upar<double> > >.
Monomials are defined by an interface, the type of their coefficients (like double, int,...) and a container for the set of their exponents (like Dynamicarray, RepTab). For the definition of multivariate polynomials we have to specify the type of storage of the monomials, like list<MMon<...,...> > for a list of monomials, this is the standard type of list in the **STL** (Standard Template Library) of C++. We have also to specify an order on the monomials, like Plex<MMon<...,...> > for the lexicographic order Plex. So a definition of polynomial may be MPol<list<Mon >, Plex<Mon >

>.
The content of the library **ALP** is listed below,

7.1 General

The details of this list is reported in the annexes (chapter 13) in the different sections indicated.

1. MATRICES

   (a) Matrix<...>, general interface for matrices, see 13.1.1 (p 61).
   (b) Matrix<Std<...> >, interface for dense matrices, see 13.1.2 (p 73).
   (c) Matrix<Str<...> >, interface for structured matrices, see 8.1 (p 43).
   (d) Matrix<Sps<...> >, interface for sparse matrices, see 8.2 (p 45).
   (e) Matrix<Hdl<...> >, interface for non-evaluated matrices, see 13.1.3 (p 75).
   (f) Matrix<Ref<...> >, interface for referenced matrices, see 13.1.4 (p 77).
   (g) Matrix<Spl<...> >, interface for specialized matrices, see 13.1.5 (p 77).
   (h) Std<lapack<...> >, Spl<lapack<...> >, implementation of Lapack matrices, see 13.2.1 (p 84).
   (i) MONOMIALS
      i. MMon<...>, interface for monomials, see 13.1.12 (p 83).
   (j) PENCILS
      i. Pencil

2. POLYNOMIALS

   (a) UPol<...>, general interface for dense univariate polynomials, see 13.1.10 (p 81).
   (b) MPol<...>, general interface for multivariate polynomials, see 13.1.11 (p 82).

3. VECTORS
   The hierarchy for the classes of vectors are the same as for the classes of matrices.

   (a) Vector<...>, general interface for vectors, see 13.1.6 (p 78).
   (b) Vector<Std<...> >, interface for standard vectors, see 13.1.7 (p 79).
   (c) Vector<Hdl<...> >, interface for non-evaluated vectors, see 13.1.8 (p 79).
   (d) Vector<Ref<...> >, interface for referenced vectors, see 13.1.9 (p 81).

7.2 Classes of containers

1. array1d<...>, one-dimensional array, see 13.3.1 (p 86).
2. array2d<...>, bi-dimensional array, see 13.3.2 (p 87).
3. toeplitz<...>, container for Toeplitz matrices as a subclass of array1d<...>, see 8.1.1 (p 43).
4. hankel<...>, container for Hankel matrices as a subclass of array1d<...>, see 8.1.1 (p 44).
5. compcol_mat_double, container for connection with SparseLib++ library, see 8.2.1 (p 46).

6. umfpack<...>, container for connection with UMFPACK library, see 8.2.2 (p 48).
7. lapack<...>, container for connection with LAPACK library, see 13.3.3 (p 88).
8. DynamicArray, dynamic representation of monomials, see 13.3.4 (p 88).
9. RepTab<...,int>, internal representation of monomials, see 13.3.5 (p 89).

7.3 Classes of monomial ordering

1. Plex<...>, lexicographic order on monomials, see 13.4.1 (p 90).
2. Vlex<...>, valuation lexicographic order see 13.4.2 (p 91).
3. SQVlex<...>, square valuation lexicographic order see 13.4.3 (p 91).

7.4 Special functions, Macros

1. Functions for multivariate polynomials, like number of terms, degrees..., see 13.5.1 (p 92).
2. Macaulay, see 13.5.2 (p 93).

   List of monomials.
   Return the polynomial, which is the sum of all monomials in n variables of degree k, with coefficient 1. The output is stored in the variable ML.

   Construction of the Macaulay matrix, from a vector of $n + 1$ polynomials in n variables. The order choosed to sort the monomials is the same as the order associated to the polynomials POL> The second argument specifies the type of the output matrix. For instance, Macaulay(V, array2d<double>()); will output a matrix of type Matrix<Ref<array2d<double> > >.

3. Bezoutian, see 13.5.3 (p 94).

   Bezout matrices.

   - template <class R> Matrix<Ref<array2d<typename R::coef_type> > > Bezout(const UPol<Std<R> > & P, const UPol<Std<R> > & Q)
   Bezout matrix.

   - template <class R> Matrix<Ref<hankel<typename R::coef_type> > > Bezout(const UPol<Spl<R> > & P)
   Bezout matrix of P and 1.

4. Gaussian

   Gaussian elimination, Bareiss method and computation of the determinant.

   - template <class R> void Triang(Matrix<Std<R> > & M, Gauss mth) Gaussian triangulation, with row pivoting. The operations are performed inplace.

   - template <class R> void Triang(Matrix<Std<R> > & M, Bareiss mth) Bareiss method. The operations are performed inplace.

5. Choose, List of monomials
6. Diff, Derivation of a polynomial with respect to a variable.
7. DiscJac, Discrete Jacobian of n+1 polynomials in n variables
8. Egal

9. IsComparable, Comparison function. True if the exponents are equal.
10. IsNull, Test if the coefficients are null.
11. IsNull, Test wether of the coefficients of the matrix are null
12. IsNull, Test wether of the coefficients of the matrix are null
13. MatrixOf, Construction of the bezoutian matrix
14. MonoDiv, Division of a monomial by another
15. Search, Search in a polynomial P, the monomial m and return its index.
16. SplitByDegree, Lists of monomials indexing the Macaulay Matrix
17. SplitMonome, Split the monomial in two monomials, from the variable of index NbVar
18. Solve, Solving a system $\mathtt{A}u = b$.

Chapter 8

Classes of matrices

In order to exploit the advantages of the structure or the sparsity of matrices, it was necessary to add some new classes for representing such objects. Here are presented the interface for structured matrices, written by **Bernard Mourrain**, and the implementations of Toeplitz and Hankel Matrices, this last written during this work. After one can find the description of two classes of sparse matrices.

8.1 Classes of structured matrices

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

□ **template<class R> struct Matrix< Str<R> > : public Matrix< Std<R> >**

Structured matrices.

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

  Matrix() {}
  Matrix(const R & r): Matrix< Std<R> >(r) {}
  Matrix(unsigned int n, unsigned int m, coef_type * t):
    Matrix< Std<R> >(n,m,t){}
  Matrix(const char * fl): Matrix< Std<R> >(fl) {}
  Matrix(const Matrix< Str<R> > & M): Matrix< Std<R> >(M.rep){}
  Matrix< Str<R> > &  operator=(const Matrix< Str<R> > & M);
  template<class S>
  Matrix< Str<R> > &  operator=(const Matrix<S> & M);
  Matrix< Str<R> > & operator*=(const Matrix< Str<R> >& B);
  Matrix< Str<R> > & rightMult(const Matrix< Str<R> >& M) ;
  Matrix<Str<R> > & leftMult (const Matrix< Str<R> >& M);
};
```

8.1.1 Class of Toeplitz matrices

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

Container

□ **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): array1d(v),
    nbrow_(v.nbrow_), nbcol_(v.nbcol_){}
  toeplitz(unsigned int i, unsigned int j):
    array1d<C>(i+j-1),nbrow_(i),nbcol_(j) {}
  toeplitz(unsigned int i, unsigned int j, C* t):
    array1d<C>(i+j-1,t),nbrow_(i),nbcol_(j){}

  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 {return nbrow_;}
  int nbcol() const {return nbcol_;}
};
```

Implementation

□ **template <class C> class Str<toeplitz<C> >**

Structured matrix specialization for `toeplitz<C>` *representation.*

```
{
  template<class S>
  static void assign(toeplitz<C> & r, const Matrix<S> & M);
  static void mult(array1d<C> & r, const toeplitz<C> & a, const array1d<C>& v);
  static void mult_fft(array1d<C> & r, const toeplitz<C> & a,
       const array1d<C>& b);
  static void add(toeplitz<C> & r, const toeplitz<C> & a,
  const toeplitz<C>& b);
};
```

8.1.2 Hankel matrices

Implementation of Hankel matrices.
In this particular case, the product of a $(d+1)\times(d+1)$ Hankel matrix
by a vector as a subvector of the product of two polynomials, by using FFT computations.

Container

□ **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(): array1d(){}
  hankel(const hankel<C> & v): array1d(v),
    nbrow_(v.nbrow_), nbcol_(v.nbcol_){}
  hankel(unsigned int i, unsigned int j):
    array1d<C>(i+j-1),nbrow_(i),nbcol_(j) {}
  hankel(unsigned int i, unsigned int j, C* t):
    array1d<C>(i+j-1,t),nbrow_(i),nbcol_(j){}

  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;
};
```

Implementation

□ **template<class C> struct Str<hankel<C> >**

Structured matrix specialization for `hankel<C>` *representation.*

```
{
  template<class S>
  static void assign(hankel<C> & r, const Matrix<S> & M);
  static void add(hankel<C> & r, const hankel<C> & a,
  const hankel<C>& b);
  static void mult(array1d<C> & r, const hankel<C> & a, const array1d<C>& v) ;
  static void mult_fft(array1d<C> & r, const hankel<C> & a,
       const array1d<C>& b);
};
```

8.2 Classes of sparse matrices

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

Interface

□ **template<class R> struct Matrix< Sps<R> > : public Matrix< Std<R> >**

Sparse matrices. 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.

```
{
  Matrix() {}
  Matrix(const R & r): Matrix< Std<R> >(r) {}
  Matrix(int n, int m, int nz, double *val, int *r, int *c, int base=0);
  Matrix(int n, int m, int nz, coef_type *val, int *ind);
  Matrix(unsigned int n, unsigned int m, coef_type * t);
  Matrix(const char * fl);
  Matrix(const Matrix< Sps<R> > & M);

  unsigned int nzero();

  Matrix< Sps<R> > &  operator=(const Matrix< Sps<R> > & M);
  Matrix<Sps<R> > & operator=(const Matrix<Ref<R> > & M) ;
  Matrix< Sps<R> > & operator*=(const Matrix< Sps<R> >& B);
  Matrix< Sps<R> > & rightMult(const Matrix< Sps<R> >& M) ;
  Matrix<Sps<R> > & leftMult (const Matrix< Sps<R> >& M);
  Matrix<Sps<R> > & transpose () ;
  Vector<Ref<col_rep> > col (const unsigned int i) const;
};
```

8.2.1 Class of sparse matrices with a derivated SparseLib++ representation

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.

Container

□ 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;

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

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

Implementation

□ template <> class Sps<compcol_mat_double>

Specialisation of the sparse matrices implementation for `compcol_mat_double` *representation.*

```
{
  typedef double coef_type;

  template<class S>
  static void assign(compcol_mat_double & r, const Matrix<S> & M);
  static void mult(array1d<double> & r,
    const compcol_mat_double & a, const array1d<double>& v);
  static void mult_trans(array1d<double> & r,
    const compcol_mat_double & a, const array1d<double>& v);
  static void solve_gmres(array1d<double> & u,
    const compcol_mat_double & a, const array1d<double>& bs,
    double tol = 1.e-6,int restart = 32,int  maxit = 150 );
  static void transpose(compcol_mat_double &a);
  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 row(typename compcol_mat_double::row_rep & c,
    const compcol_mat_double & a, const unsigned int i);
  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);
};
```

- `void Sps<compcol_mat_double>::transpose(compcol_mat_double &a)`

Transpose of a sparse matrix with `compcol_mat_double` representation

- `void Sps<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 Sps<compcol_mat_double>::mult_trans(array1d<double> & r, const compcol_mat_double & a, const array1d<double>& v)`

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

- `void Sps<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 Sps<compcol_mat_double >::submatrix(compcol_mat_double & 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.

- `void Sps<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 Sps<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 Sps<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 Sps<compcol_mat_double>::solve_gmres(array1d<double> & u, const compcol_mat_double & a, const array1d<double>& b, double tol, int restart, int maxit)`

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.

8.2.2 Class of sparse matrices with a representation adapted for UMFPACK routines

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

Container

□ **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 actual size of the array,* `index_` *field which contains rows indices 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, 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;
};
```

Implementation

□ **template <class C> class Sps<umfpack<C> >**

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

```
{
  typedef C coef_type;

  template<class S>
  static void assign(umfpack<C> & r, const Matrix<S> & M);
  static void transpose(umfpack<C>  & A);
  static void mult(array1d<C> & r, const umfpack<C> & a,
     const array1d<C>& v);
  static void solve_umf(array1d<C> & u, const umfpack<C> & a,
```

```
      const array1d<C>& b);
  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 & f,
      const unsigned int & l, 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 <>`
`void Sps<umfpack<double> >::solve_umf(`
`array1d<double> & u, const umfpack<double> & a, const array1d<double>& b)`
Real implementation. Solves the system $Ax = b$ 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`.
- `template <>`
`void Sps<umfpack<complex<double> > >::solve_umf(`
`array1d<complex<double> > & u, const umfpack<complex<double> > & a,`
`const array1d<complex<double> >& b)`
Complex implementation. Solves the system $Ax = b$ 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`.

8.3 Some matrices constructions

Like it was explained in the theorical part of this report, solving a multivariate system by the **implicit inverse method** needs the construction of Sylvester-like matrices, so a function returning such a matrix and its transposed for a polynomial system has been written for the construction due to **Macaulay** (see section 6.3), it uses the same algorithm that the existing function `Macaulay(...)` but adapted for the sparse matrices representations. But before we are going to see some operations of extraction of submatrices.

8.3.1 Submatrices extraction

To solve the system of Kruppa equations described in chapter 6, section 5, by solving the system 6.5, we need some operations of extractions of submatrices and columns of matrices.

Different extractions of submatrices.

- `template<class R, class T>`
`inline void SubMatrix(Matrix<R> & M, const unsigned int & first,`
`const unsigned int & last, T from)`
In place extraction of a submatrix of M, the type of extraction (some columns or some rows) is given by the parameter $from$, which is `ByCol()` to remove columns f to l and `ByRow()` to remove rows f to l.
- `template<class R>`
`inline void Split(const Matrix<R> & S, Matrix<R> & Sp,`
`Vector<Std<typename Matrix<R>::col_rep> > & c)`
Extraction of the first column of the matrix S in the vector c, at exit the matrix Sp contains the matrix S minus its first column. S is unchanged.
- `template<class R>`
`inline void ShiftCol(Matrix<R> & S, unsigned int n)`

In place extraction of a submatrix by removing the columns 0 to $n-1$ of the matrix S, so S is changed.

- ```
  template<class R>
  inline void ShiftRow(Matrix<R> & S, unsigned int n)
  ```

In place extraction of a submatrix by removing the rows 0 to $n-1$ of the matrix S, so S is changed.

- ```
  template<class R>
  inline void Shift(Matrix<R> & S, unsigned int c, unsigned int r)
  ```

In place shift of c columns and r rows of the matrix S, so S is changed.

8.3.2 Sylvester-like matrices construction

Construction of a Sparse Sylvester-like matrix and its transposed from the Macaulay matrix associated to $n+1$ polynomials in n variables. The order chosen to sort the monomials is the same as the order associated to the polynomials `POL`. In all cases, the second argument specifies the type of the output matrix. For instance, `SylvesterMc(V, compcol_mat_double());` will output a matrix of type `Matrix<Ref<compcol_mat_double > >`.

- ```
  template <class POL>
  Matrix<Ref<compcol_mat_double> >
  SylvesterMc(const Vector<Std<array1d<POL> > > & f,
  compcol_mat_double S)
  ```

Sylvester-like matrix under the specialized form Matrix<Sps<compcol_mat_double> > >

- ```
  template <class POL>
  Matrix<Ref<compcol_mat_double> > SylvesterMc_T(const Vector<Std<array1d<POL> > > &
  f,
  compcol_mat_double S)
  ```

Transposed Sylvester-like matrix under the specialized form Matrix<Sps<compcol_mat_double> > >

- ```
  template <class POL, class C>
  Matrix<Ref<umfpack<C> > >
  SylvesterMc(const Vector<Std<array1d<POL> > > & f,
  umfpack<C> S)
  ```

Sylvester-like matrix under the specialized form Matrix<Sps<umfpack<C> > > >

- ```
  template <class POL, class C>
  Matrix<Ref<umfpack<C> > >
  SylvesterMc_T(const Vector<Std<array1d<POL> > > & f,
  umfpack<C> S)
  ```

Transposed Sylvester-like matrix under specialized the form Matrix<Sps<umfpack<C> > > >

Chapter 9

Univariate polynomials

Two representations of univariate polynomials have been added, one in the aim of using FFT for the product of two polynomials, the other for representing polynomials modulo a fixed polynomial, that are the elements of the quotient algebra $\mathcal{A}$ as explained in the theorical part.

9.1 A special representation of dense polynomials

A special implementation of univariate polynomials.
Multiplication of two of these polynomials is computed by FFT.

Container

□ **template <class C> struct upar : public array1d<C>**

`upar<C>`, *container of univariate polynomials, a subclass of* $array1d < C >$.

```
{

  unsigned int degree_;

  typedef C            coef_type;
  typedef unsigned int index_type;

  upar(): array1d(), degree_(-1){}
  upar(index_type s): array1d(s), degree_(s-1){}
  upar(index_type s, C *t): array1d(s,t), degree_(s-1){}
  upar(const upar<C> & p): array1d(p), degree_(p.degree_){}

  index_type degree();
  index_type degree() const;

  void checkdegree();
  void scopy(const upar<C> & r);
};
```

Implementation

□ **template<class C> struct Spl<upar<C> >**

Specialization of `UPol<R>` *for container* `upar<C>`.

```
{

  typedef typename upar<C>::coef_type coef_type;
  typedef typename upar<C>::index_type index_type;

  template<class S>
  static void assign(upar<C> & a, const UPol<S> & b);
  static void add(upar<C> & x, const upar<C> & a, const upar<C>& b);
  static void sub(upar<C> & x, const upar<C> & a, const upar<C>& b);
  static void add(upar<C> & x, const upar<C> & a);
  static void sub(upar<C> & x, const upar<C> & a);
  static void mult(upar<C> & x, const coef_type & c);
  static void mult(upar<C> & x, const upar<C> & a, const coef_type & c);
  static void divd(upar<C> & x, const upar<C> & a, const coef_type & c);
  static void mult(upar<C> & x, const upar<C> & a, const upar<C>& b);
  static void mult(upar<C> & x, const upar<C>& b);
  static void mult_fft(upar<C> & x, const upar<C> & a, const upar<C>& b);
  static void reduce(upar<C> & p, const index_type & e);
};
```

- `template <class C> void Spl<upar<C> >::mult_fft(upar<C> & r, const upar<C> & a, const upar<C>& b)`

Multiplication of a specialized polynomial by another specialized polynomial Result, given by the first parameter r, is a polynomial of the same type as the two polynomials. The method of multiplication is fft.

- `template <class C> void Spl<upar<C> >::reduce(upar<C> & p, const index_type & e)`

Reduction a polynomial p modulo e.

9.2 Class of elements of the quotient algebra

Interface
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 UPol<Quot<R> > : public UPol<Std<R> >**

Class of polynomial of a quotient algebra.

```
{

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

  UPol() {}
```

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

  static index_type dim();

  UPol<Quot<R> > & operator=(const UPol<Quot<R> > & P);
  UPol<Quot<R> > & operator=(const UPol<Ref<R> > & P)
  template <class S>
  UPol<Quot<R> > & operator=(const UPol<Hdl<S> > & P)
    {
      rep.rebuild(P.degree()+1);
      Quot<R>::assign(rep,P);
      Quot<R>::MontoHorn(rep,true);
      return *this;
    }

  template <class S>
  UPol<Quot<R> > & operator=(const UPol<Spl<S> > & P)
    {
      rep.rebuild(P.degree()+1);
      Quot<R>::assign(rep,P);
      Quot<R>::MontoHorn(rep,true);
      return *this;
    }

  UPol<Quot<R> > & operator+=(const UPol<Quot<R> >& a);
  UPol<Quot<R> > & operator-=(const UPol<Quot<R> >& a);
  UPol<Quot<R> > & operator*=(const coef_type & c);
  UPol<Quot<R> > & operator*=(const UPol<Quot<R> >& b);
};
```

Implementation
Univariate polynomials.
Implementation of the quotient algebra associated to a polynomial. Here, polynomials representation is $upar < C >$.
We assume that the elements of the quotient are of degree at most equal to the associated polynomial minus one.
Specific operations are implemented.

□ **template<class C> struct Quot<upar<C> >**

`poly` *is the polynomial which generates the principal ideal quotienting the algebra of the complex polynomials. It is assigned by* `assign(..)` *method, which is a class method.*

```
{
  static upar<C> poly;

  typedef typename upar<C>::coef_type coef_type;
  typedef typename upar<C>::index_type index_type;
```

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

  // Allows to assign a polynomial to a container.
  // It is a hard copy, not a shared one.
  template<class S>
  static void assign(upar<C> & a, const UPol<S> & b) ;
  static void assignpoly(upar<C> & p);
  static void add(upar<C> & x, const upar<C> & a, const upar<C>& b);
  static void sub(upar<C> & x, const upar<C> & a, const upar<C>& b);
  static void add(upar<C> & x, const upar<C> & a);
  static void sub(upar<C> & x, const upar<C> & a);
  static void mult(upar<C> & x, const coef_type & c);
  static void div(upar<C> & x, const coef_type & c);
  static void mult(upar<C> & x, const upar<C> & a, const coef_type & c);

  /*
    Computes squares in
```

$$\mathcal{A} = \frac{C[x]}{(poly)}$$

```
    Allows to compute products in A by the formula :
```

$$fg \;=\; \frac{1}{4}((f+g)^2 - (f-g)^2).$$

```
  */
  static void square(upar<C> & x,const upar<C> & f);

  /*
    Product a*b,  using only squares,
    by the formula
```

$$a * b \;=\; \frac{1}{4}((a+b)^2 - (a-b)^2).$$

```
  */
  static void mult(upar<C> & x, const upar<C> & a, const upar<C>& b);

  /*
    Expresses the polynomial x from a basis (Monomial or Horner)
    in the other basis.
    For this, computes B for the passage of the Horner basis
    to the monomial basis. Computes its inverse, H, in the other case.
  */
  static void MontoHorn(upar<C> &x, boolean MtoH);

  static void mult_fft(upar<C> & x, const upar<C> & a, const upar<C>& b);
};
```

A problem of this implementation is that we cannot work with several quotient algebras at the same time since the polynomial generating this algebra is represented by a class variable. But in the application of finding roots of a univariate polynomial it is not a problem. Moreover, it possible to change the quotient algebra, that is quotienting the ring of polynomials by another polynomial, by a call to the function `assignpoly(..)` of the class `Quot<..>`. In this we have to be aware that the instances previously declared become unavailable without a new assignment.

9.3 Reciprocal of a polynomial

We are going to see the algorithm for computing the reciprocal of a polynomial, which is necessary for computing the inverse of a Hankel matrix since such a matrix can be seen as a polynomial (see section 2.1.2), this function has been added in the class of standard (dense) univariate polynomial `UPol<C>` described above. (algorithm from [6], pp 22-27).

- `template <class R>`

`inline void reciprocal(UPol<R> & w,const UPol<R> & p)`
Computing the reciprocal of the polynomial p. 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.

```
{
  UPol<R> w0(p.degree()+1),v(p.degree()+1),xp(p);
  typedef typename UPol<R>::coef_type C;
  typename UPol<R>::index_type K =
  (typename UPol<R>::index_type) (log(p.degree()+1)/log(2)),j=1;
  C p0 = xp[p.degree()];

  xp /= p0;
  w0 = xp/xp[0];
  v=w0*C(-1.0);
  v[0]+=C(2);
  w=v/xp[0];
  w0=w;
  while (j <= K) {
    reduce(w0,pow(2,j+1));
      // reduction modulo z^(2^(j+1)) of the polynomial w0.
    w=xp*w0;
    v=w*C(-1.0);
    v[0]+=C(2);
    reduce(v,pow(2,j+1));
      // reduction modulo z^(2^(j+1)) of the polynomial v.
    w=w0*v;
    w0=w;
    j++;
  }
   reduce(w,p.degree()+1);
   w/=p0;
}
```

Chapter 10

Fast Fourier Transform

The polynomial ring $\mathbb{K}[\mathbf{x}]$ has also a structure of algebra within the product of two elements is a product of convolution :

For each integer k, $k \geq 0$, $c_k = \sum_{i+j=k} a_i bj$.

A basic method for the product of two polynomials of degree n have a cost of $\mathcal{O}(n^2)$ arithmetic operations. But by performing a **Discret Fast Fourier Transform** on the coefficients vectors of these polynomials this cost can be reduced to $\mathcal{O}(nlog(n))$. In this way the product of two polynomials is performed by computing the FFT of the two coefficients vectors, multiplying coefficients wise result and performing a reverse FFT in order to obtain the coefficients vector of the product of the two polynomials.

The algorithm of the `BitReversal(..)`, `Permut(...)` and `FFT(..)` functions described below are available in [24].

Implementation of `2-radix FFT` computation of a complex array. It is an "in-place" transformation.

- `inline unsigned int BitReverse(int k, int n)`

- `template <class T>`
`inline void Permut(T *Tab, int N)`
Function which permutates an array by Bit Reversal method
- `template <class C, class U>`
`void FFT(int n, const complex<U> & w, complex<C> *W, int inverse)`
FFT computation - based on the Butterfly algorithm. A complex array W, of size n (a power of 2), is replaced by the FFT of its values.
Here is the generic recursive function, the argument W is a complex array whom the elements will be replaced by their discret FFT evaluation, if *inverse* is 0, else by their reverse discret FFT. w is a root of the unity.

```
template <class C, class U>
void FFT(int n, const complex<U> & w,  complex<C>  *W, int inverse)
{
  switch (n) {
  case 1:
    break;
  case 2:
    {
```

```
      complex<C> t(W[0]);
      W[0] = t + W[1];
      W[1] = t - W[1];
    }
    break;
  case 4:
    {
       complex<C> x0(W[0]+W[1]), x1(W[0]-W[1]), y0(W[2]+W[3]), y1(W[2]-W[3]);
      if (inverse) {
        W[0] = (x0 + y0);
W[1] = (x1 - complex<C>(-y1.imag(), y1.real()));
W[2] = (x0 - y0);
W[3] =(x1 + complex<C>(-y1.imag(), y1.real()));}
      else {
        W[0] = x0 + y0;
W[1] = x1 + complex<C>(-y1.imag(), y1.real());
W[2] = x0 - y0;
W[3] = x1 - complex<C>(-y1.imag(), y1.real()); }
      }
    break;
  default:
  {
    /* Since the array w has been permutated before the call to FFT,
       elements with even index in the initial array, are now the
       first n/2 elements and elements with odd index the last n/2.
       So the recursif process takes the initial array divided in two parts.
    */

    FFT(n/2,pow(w, 2), W, inverse);
    FFT(n/2, pow(w, 2), W+n/2,inverse);

    for (int k = 0; k< n/2; k++)
      {
complex<C> z = pow(w,k)*W[k+n/2] ;
W[k+n/2] = W[k] - z;
W[k] = W[k] + z;
      }
  }
  }
}
```

Chapter 11

Iterative methods

11.1 Sebastiao e Silva

Description

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>
  inline UPol<Ref<R> > StartingValue(UPol<Spl<R> > p, const typename R::coef_type x0)
  ```

Returns $B(x0, p)$, used to get a starting value to the process.

- ```
  template <class R>
  inline void Solve(typename R::coef_type * x, UPol<R> & p, Sebastiao mth)
  ```

Computes all the roots of polynomial p, using Sebastiao e Silva method. This function uses a default starting value which is 0. Result is given in x.

- ```
  template <class R>
  inline void solve(typename R::coef_type * x, UPol<Spl<R> > & p, const typename R::coef_type
  & x0, Sebastiao mth)
  ```

Computes all the roots of the polynomial p, using Sebastiao e Silva method. Result is given in x. Starting value is $x0$.

- ```
  template <class R>
  inline void solve(typename R::coef_type & x, UPol<Spl<R> > & 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.

- ```
  template <class R>
  inline void solve(UPol<Quot<R> > & x,const UPol<Spl<R> > & p, const UPol<Quot<R> >
  & x0, Sebastiao mth)
  ```

Computes one root of the polynomial p, using Sebastiao e Silva method. Starting value, $x0$, is an element of the quotient algebra associated to the polynomial p. Result is given in x, which is the idempotent element associated to the root found.

Code

Here is the code of these functions,

```
//--------------------------------------------------------------
```
```

```
struct Sebastiao {};

//------------------------------------------------------------
template <class R>
inline UPol<Ref<R> > StartingValue(UPol<Spl<R> > p,
    const typename R::coef_type x0)
{
  UPol<Quot<R> > res(p.degree());

  res[p.degree()-1]= (typename R::coef_type)(1.0);
  for (typename R::index_type i = 0; i < p.degree()-1; i++)
    res[i] =(typename R::coef_type) pow(x0,(p.degree()-1-i));
  return UPol<Ref<R > > (&res.rep,p.degree());
}


//-----------------------------------------------------------
template <class R>
inline void Solve(typename R::coef_type * x, UPol<R> & p,
  Sebastiao mth)
{
  solve(x,p,0,mth);
}

//-----------------------------------------------------------
template <class R>
inline void solve(typename R::coef_type * x, UPol<Spl<R> > & p,
     const typename R::coef_type & x0, Sebastiao mth)
{
  typedef typename R::coef_type C;
  Quot<R>::assignpoly(p.rep);
  UPol<Quot<R> > sum,u0,u,un;
  typename R::index_type k, degree=Quot<R>::dim()-1;
  Complex_Typ<typename R::coef_type>::TYPE bound(Tolerance);
  u0 = StartingValue(p, x0);

  for (typename R::index_type i = 0; i <=degree; i++)
    {
      if (i==0) u=u0;
      else
{
  u=u0;
  un=u*sum;
  u-=un;
}

      un=u^2;
      un/=norme(un);
      u-=un;

      k=1;
      while ((norme(u) > bound) && (k++ <= Max_Iter))
```

```
{
  u=un;
  un=u^2;
  un/=norme(un);
  un/=un[degree];
  u-=un;
}

      x[i] = un[degree-1];

      /* First derivative of Poly estimated in the last root found,
 We use the coefficients of E,
 which are the different powers of this root. */

      C deriv= C(0.0);
      for (typename R::index_type j=1; j<=degree+1;j++)
deriv+=Quot<R>::poly[j]*un[degree+1-j]*C(j*1.0);

      un/=deriv;

       if (i == 0)
sum = un;
      else
sum += un;
    }
}

//----------------------------------------------------------------------
template <class R>
inline void solve(typename R::coef_type & x, UPol<Spl<R> > & p,
  const typename R::coef_type & x0, Sebastiao mth)
{
  typedef typename R::coef_type C;
  Quot<R>::assignpoly(p.rep);
  UPol<Quot<R> > u,un;
  typename R::index_type k=0, degree=Quot<R>::dim()-1;

  u = StartingValue(p, x0);
  un = u^2;
  un/=norme(un);
  u-=un;
  Complex_Typ<typename R::coef_type>::TYPE bound(Tolerance);

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

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

//-----------------------------------------------------------------
template <class R>
inline void solve(UPol<Quot<R> > & x,const UPol<Spl<R> > &  p,
  const UPol<Quot<R> > & x0, Sebastiao mth)
{
  typedef typename R::coef_type C;
  UPol<Quot<R> > u(x0);
  typename R::index_type k=0, degree=Quot<R>::dim()-1;
  Complex_Typ<typename R::coef_type>::TYPE bound(Tolerance);

  x = u^2;
  x/=norme(x);
  u-=x;

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

11.2 Implicit inverse method

Description

- `template <class POL, class RepS>`
`Vector<Ref<array1d<typename RepS::coef_type> > >`
`MPSolve(const Vector<Std<array1d<POL> > > & f, RepS S)`
Finding a root of a polynomial system in n variables by the `Implicit Inverse Power method`. Compute 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 other 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.

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(Vector<R> & x, const Matrix<Sps<M> > & s)`
Implicit inverse for a `Sylvester-like` matrix with a `sparse` representation.

- `template <class R, class S>`
`inline Vector<Ref<R> > Solve(const Matrix<S> & A, const Vector<Std<R> > & b, UMFso`

`meth)`
Solve the linear system $A\,x = b$ with the UMFPACK library routines.

- `template <class R, class S>`
`inline Vector<Ref<R> > Solve(const Matrix<S> & A, const Vector<Std<R> > & b, GMres meth)`
Solve the linear system $A\,x = b$ with the IML++ library `GMRES` method.

Code

```
//--------------------------------------------------------
template <class POL, class RepS>
Vector<Ref<array1d<typename RepS::coef_type> > >
  MPSolve(const Vector<Std<array1d<POL> > > & f, RepS S)
{
  typedef typename RepS::coef_type C;
  Matrix<Sps<RepS> > Syl;

  Syl = SylvesterMc_T(f,S);

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

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

  Vector<Std<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 (unsigned int i = 0; i < f.size()-1; i++) x[i] = Res[i+1];

  return Vector<Ref<array1d<C> > > (new array1d<C>(f.size()-1,x));
}
//--------------------------------------------------------------------------------------------------------

//-----------------------------------------------------
#define hold 1.0e-4    // tolerance for convergence
#define max_iter 100   // maximum number of iterations
//-----------------------------------------------------
template <class R, class M>
unsigned int ImplicitInverse(Vector<R> & x, const Matrix<Sps<M> > & s)
{
  unsigned int dim = s.nbcol(),j=1, e0 = x.size();
  Vector<R> vn(dim), un(dim),u(dim);
  typedef typename Matrix<Sps<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;
}
//-------------------------------------------------------
template <class R, class S>
inline Vector<Ref<R> > Solve(const Matrix<S> & A,
       const Vector<Std<R> > & b, UMFso meth)
{
  R* x = new R(A.nbcol());
  S::solve_umf(*x,A.rep,b.rep);
  return Vector<Ref<R> >(x);
}
//-------------------------------------------------------
template <class R, class S>
inline Vector<Ref<R> > Solve(const Matrix<S> & A,
                       const Vector<Std<R> > & b, GMres meth)
{
  R* x=new R(A.nbcol());
  S::solve_gmres(*x,A.rep,b.rep);
  return Vector<Ref<R> >(x);
}
//-------------------------------------------------------
```

In the function `ImplicitInverse(...)` we assume that the coefficient corresponding to the “1” of the basis choosen for the polynomial system is in first position inside the coefficients vectors. This is the case for the actual orders in **ALP**. It has to be changed for an another type of order.

11.3 Solving overconstrained polynomial systems with one root

- `template <class POL, class RepS>`
  `Vector<Ref<array1d<typename RepS::coef_type> > >`
  `MPSolveOne(const Vector<Std<array1d<POL> > > & f, RepS S)`

Finding the root of an `overconstrained polynomial system` in n variables and $n+1$ equations, in the special case where A, the `quotient algebra`, is of dimension 1. Computes the `Sylvester-like` matrix S of the system, and solves one system of the form $S'x = -c$, where S' is a submatrix of S constructed by a shift of one column and one row, this first column is saved in c. Polynomial system is passed as a standard vector of polynomials. Result is returned by a vector of size n. The second argument specifies the type of the sparse `Sylvester-like` matrix. For instance, `MPSolveOne(V, umfpack<double>());` will construct a sparse matrix with a representation of type UMFPACK<DOUBLE>, and the system will be solved with the specific library for this representation.

```
//--------------------------------------------------------------
template <class POL, class RepS>
Vector<Ref<array1d<typename RepS::coef_type> > >
 MPSolveOne(const Vector<Std<array1d<POL> > > & f, RepS S)
{
  typedef typename RepS::coef_type C;
  typedef typename RepS::solve_type ST;
  Matrix<Sps<RepS> > Syl;

  // Construction of the Sylvester-like matrix
  Syl = SylvesterMc(f,S);

  cout << " Dimension of S : " << Syl.nbrow() << endl;
  cout << " Number of non-zero entries : " << Syl.nzero() << endl;
  cout << " Dimension of A : " << 1 << endl;

  Vector<Std<typename RepS::col_rep> > X(Syl.nbrow()), Res(Syl.nbcol()-1);

  Split(Syl, Syl,X);

  X.rep.tab_=X.rep.tab_+1;
  X.rep.size_--;
  X*=(-1.);

  ShiftRow(Syl,1);

  Syl.transpose();

  // One resolution of the system, result is put in Res.
  Res=Solve(Syl,X,ST());

  C * x = new C[f.size()-1];
  for (unsigned int i = 0; i < f.size()-1; i++) x[i] = Res[i+1];

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

Part III

Conclusion, Annexes

Chapter 12

Conclusion

12.1 Problems or questions staying

Perhaps it is possible to make some functions or classes written during this work more generic. For instance, see if it would be possible to have only one function of construction of the Sylvester matrix (and of its transposed), the actual problem being the construction of the index array. In the function **ImpPowerMeth(...)** it would be better to have the possibility to use it for all kind of order of the monomial basis. Moreover the name of results are quite heavy to write. For instance to use a dense matrix with entries of double we have to define `Matrix<Std<array2d<double> > >`. A naming like `MatStd<...>` would be simpler, but the notion of general matrix would be lost. During this work, two controlled iterative methods for solving polynomial systems have been applied. There are some other methods for such a problem which could be studied in order to increase efficiency and accuracy, for instance the **Arnoldi** method (see [14]) for the INVERSE POWER METHOD. In order to cover the case where the set of roots of a system is in a toric variety we could use the **Toric resultant** for the construction of smaller resultant matrices (see [7]).

12.2 Personal aspects

I have wished to make a pratical period in this project because I am very interesting in subjects of algebra and geometry, and particularly by linear algebra subjects. Moreover, by this work, I had the possibity to meet again the environment of research, within I have already worked and within I would wish to work in the future. And in this project, the axis of research were close to my interests. For this work I had to study some theorical aspects. I have not a formation of mathematician. ThusI had the possibility to learn a lot and extend my knowledge in these theorical subjects. Another contribution was to discover a different way for programming in C++, less "Object" and more generic (by the way of using template) which has completed the learning in C++ of this second year of ESSI.

Chapter 13

Annexes

13.1 Interfaces

13.1.1 General interface for matrix

Interface for general matrices, with dense representation. The class `R` is a container for the internal representation for the matrix. The methods that show be provided by `R` are

```
unsigned int R::nbrow();
unsigned int R::nbcol();
coef_type R::operator(unsigned int i, unsigned int j)
```

13.1.2 Standard matrix

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 Matrix**

□ **template<class R> struct Matrix<Std<R> >**

```
{
  R rep;

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

  Matrix():rep() {}
  Matrix(const R & r): rep(r) {}
  Matrix(unsigned int n, unsigned int m): rep(n,m){}
  Matrix(unsigned int n, unsigned int m, coef_type * t): rep(n,m,t){}
  Matrix(const char * fl): rep(fl) {}
  Matrix(const Matrix<Std<R> > & M): rep(M.rep) {}
  Matrix(const Matrix<Ref<R> > & M) {rep.scopy(*(M.rep));}
  template <class T>
```

```
  Matrix(const Matrix<T> & M): rep(M.nbrow(),M.nbcol());
  Matrix<Std<R> > & operator=(const Matrix<Std<R> > & M);
  Matrix<Std<R> > & operator=(const Matrix<Ref<R> > & M);
  template<class T>
  Matrix<Std<R> > & operator=(const Matrix<Ref<T> > & M);
  template<class T>
  Matrix<Std<R> > & operator=(const Matrix<Hdl<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;

  Matrix<Std<R> > & operator() (Range I1, Range I2);

  template<class S>
  Matrix<Std<R> > & operator+=(const Matrix<S>& a);

  template<class S>
  Matrix<Std<R> > & operator-=(const Matrix<S>& a);

  Matrix<Std<R> > & operator*=(const coef_type & c);
  Matrix<Std<R> > & operator/=(const coef_type & c);
  Matrix<Std<R> > & operator*=(const Matrix<Std<R> >& B);

  Matrix<Std<R> > & rightMult(const Matrix<Std<R> > & M);
  Matrix<Std<R> > & leftMult (const Matrix<Std<R> > & M);

  Matrix<Ref<R> > operator^(const int n);

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

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

  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();
};
```

13.1.3 Handled matrix

□ **template <class R> struct Matrix<Hdl<R> >**

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 assignement.

```
{
  R rep;

  typedef typename Coef<R>::coef_type coef_type;

  Matrix() {}
  Matrix(const R & r, unsigned int n, unsigned int m):
    rep(r), _nbrow(n), _nbcol(m) {}
  Matrix(const Matrix<Hdl<R> > & r): rep(r.rep),
      _nbrow(r.nbrow()), _nbcol(r.nbcol()) {}

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

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

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

- `template <class R, class S>`
`Vector<Hdl<Mult<Matrix<R>, Vector<S> > > > operator*(const Matrix<R>& A, const Vector<S> & V)`
Multiplication Matrix-Vector. Return an unevaluated Matrix-Vector product of Vector type.
- `template <class R, class S>`
`inline typename Matrix<R>::coef_type _elem(const Vector<Hdl<Mult<Matrix<R>, Vector<S> > > > & V, unsigned int i)`
The ith element of the unevaluated Vector.
- `template <class R, class S>`
`Matrix<Hdl<Mult<Matrix<R>, Matrix<S> > > > operator*(const Matrix<R>& A, const Matrix<S> & B)`
Multiplication Matrix-Matrix. Return an unevaluated Matrix-Matrix product of Matrix type.
- `template <class R, class S> inline typename Matrix<R>::coef_type _elem(const Matrix<Hdl<Mult<Matrix<R>, Matrix<S> > > > & M, unsigned int i, unsigned int j)`
The (i,j) element of the product. Here is an example of operators for such matrices, the operator $+$.

```
//--------------------------------------------------------------------
template <class R, class S>
inline Matrix<Hdl< Plus<Matrix<R>,Matrix<S> > > >
  operator+(const Matrix<R>& a, const  Matrix<S>& b)
{
  return Matrix<Hdl<Plus<Matrix<R>, Matrix<S> > > >
    (Plus<Matrix<R>, Matrix<S> >(a,b), min(a.nbrow(),b.nbrow()),
                                       min(a.nbcol(),b.nbcol()) );
}
```

```
template <class T, class R>
inline typename Matrix<Hdl<Plus<T,R> > >::coef_type
_elem(const Matrix<Hdl<Plus<T,R> > > & v, unsigned int i, unsigned int j)
{
  return v.rep.op1(i,j)+v.rep.op2(i,j);
}
//-----------------------------------------------------------------------
```

another example for the product Matrix-Vector,

```
//-----------------------------------------------------------------------
template <class R, class S>
Vector<Hdl<Mult<Matrix<R>, Vector<S> > > >
operator*(const Matrix<R>& A, const Vector<S> & V)
{
  return Vector<Hdl<Mult<Matrix<R>, Vector<S> > > >
    (Mult<Matrix<R>, Vector<S> >(A,V), A.nbrow());
}

// The i$^th$ element of the unevaluated Vector.
template <class R, class S>
inline typename Matrix<R>::coef_type
_elem(const Vector<Hdl<Mult<Matrix<R>, Vector<S> > > > & V,
       unsigned int i)
{
  typename Matrix<R>::coef_type res=0;
  for(int j=0; j < V.rep.op1.nbcol(); j++) {
    res += V.rep.op1(i,j)*V.rep.op2[j];
  }
  return res;
}
//------------------------------------------------------------------------
```

And the structures associated are :

```
//------------------------------------------------------------------------
template <class A, class B>
struct Plus {
  const A & op1;
  const B & op2;
  Plus () {};
  Plus(const Plus<A,B> &  p): op1(p.op1), op2(p.op2) {};
  Plus(const A & a, const B & b) : op1(a), op2(b) {};
};
template <class A, class B>
struct Mult {
  const A& op1;
  const B& op2;
  Mult() {};
  Mult(const Mult<A,B> &  p): op1(p.op1), op2(p.op2) {};
  Mult(const A & a, const B & b) : op1(a), op2(b) {};
};
//-----------------------------------------------------------------------
```

13.1.4 Referenced matrix

This is the referenced class of matrix.

```
template <class R>
struct Matrix<Ref<R> > {
  R* rep;

  typedef typename R::coef_type  coef_type;

  Matrix() {}
  Matrix(R* r):  rep(r), _nbrow(r->nbrow()), _nbcol(r->nbcol()) {}
  Matrix(R* r, unsigned int n, unsigned int m):
    rep(r), _nbrow(n), _nbcol(m) {}
  Matrix(const Matrix<Std<R> > & r): rep(r.rep),
    _nbrow(r.nbrow()), _nbcol(r.nbcol()) {}

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

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

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

13.1.5 Specialized matrix

Interface for specialized matrices.

□ **template<class R> struct Matrix< Spl<R> > : public Matrix< Std<R> >**

Specialized matrices.

```
{

  typedef typename R::coef_type coef_type;

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

  Matrix< Spl<R> > &  operator=(const Matrix< Spl<R> > & M);
  Matrix< Spl<R> > &  operator=(const Matrix<Ref<R> > & M);

  template<class T>
  Matrix<Spl<R> > & operator=(const Matrix<Hdl<T> > & M);
  Matrix< Spl<R> > & operator*=(const Matrix< Spl<R> >& B);
```

```
  Matrix<Spl<R> > & rightMult(const Matrix< Spl<R> >& M);
  Matrix<Spl<R> > & leftMult (const Matrix< Spl<R> >& M);

  Matrix<Spl<R> > & transpose ();
};
```

- template<class R> inline Matrix<Ref<R> > operator *(const Matrix< Spl<R> > & M1,
const Matrix< Spl<R> > & M2)

13.1.6 General interface for vector

Interface for general vectors. This class is parametrized by the internal representation R of the vectors, and by its referencing properties. The default value for the second parameter is Std.

- template <class R>
inline ostream & operator«(ostream & os, const Vector<R> & v)
Output operator.
- template <class R>
typename Vector<R>::coef_type Norm(const Vector<R> & v, int p =2)
Norm of a vector. The default is the L2 norm.
- template <class R>
typename Vector<R>::coef_type Prod(const Vector<R> & v, const Vector<R> & w)
Innerproduct of two vectors.
- template <class R, class T>
ostream & operator«(ostream & os, const Vector<R,T> & v)
Output operator.
- template <class R, class S, class T, class U>
Vector<Plus<Vector<R,T>,Vector<S,U> >, Hdl>
operator+(const Vector<R,T>& a, const Vector<S,U>& b)
Addition operator. Return a handler for a sum of two vectors.
- template <class T, class R>
double _elem(const Vector<Plus<T,R>,Hdl> & v, unsigned int i)

- template <class R, class S, class T, class U>
Vector<Minus<Vector<R,T>,Vector<S,U> >, Hdl>
operator-(const Vector<R,T>& a, const Vector<S,U>& b)
Substraction operator.
- template <class T, class R>
typename cat<Plus<T,R> >::coef_type
_elem(const Vector<Minus<T,R>, Hdl> & v, unsigned int i)

- template <class R, class T>
Vector<UMinus<Vector<R,T> >, Hdl>
operator-(const Vector<R,T>& a)
Unitary minus operator.
- template <class R>
typename cat<R>::coef_type
_elem(const Vector<UMinus<R>, Hdl> & v, unsigned int i)

- template <class R, class T>
Vector<Mult<Vector<R,T>, typename cat<R>::coef_type>, Hdl>

```
operator*(const Vector<R,T>& a, const typename cat<R>::coef_type & c)
```
Multiplication by a scalar.

- `template <class R,class T>`

```
typename cat<R>::coef_type _elem(const Vector<Mult<R,T>,Hdl >& v, unsigned int i)
```

- `template <class R, class T>`

```
Vector<Divd<Vector<R,T>, typename cat<R>::coef_type>, Hdl>
operator/(const Vector<R,T>& a, const typename cat<R>::coef_type & c)
```
Division by a scalar.

- `template <class R,class T>`

```
typename cat<R>::coef_type
_elem(const Vector<Divd<R,T>,Hdl >& v, unsigned int i)
```

13.1.7 Standard vector

□ **template <class R> struct Vector<Std<R> >**

This is the standard class of vectors

```
{
  R rep;

  typedef typename R::coef_type   coef_type;

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

  Vector<Std<R> > & operator=(const Vector<Std<R> > & V);
  Vector<Std<R> > & operator=(const Vector<Ref<R> > & V);

  template<class S>
  Vector<Std<R> > & operator+=(const Vector<S> & a);
  template<class S>
  Vector<Std<R> > & operator-=(const Vector<S> & V);
  Vector<Std<R> > & operator*=(const coef_type & c);
  Vector<Std<R> > & operator/=(const coef_type & c);

  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();}
};
```

13.1.8 Handled vector

□ **template <class R> struct Vector<Hdl<R> >**

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 is allowed.

```
{
  R rep;

  typedef typename Coef<R>::coef_type coef_type;

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

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

- `template <class R, class S>`

```
inline Vector<Hdl<Plus<Vector<R>,Vector<S> > > > operator+(const Vector<R>& a, const
Vector<S>& b)
```
Addition operator. Return a referenced "sum".

- `template <class T, class R>`

```
inline typename Vector<Hdl<Plus<T,R> > >::coef_type _elem(const Vector<Hdl<Plus<T,R>
> > & v, unsigned int i)
```

- `template <class R, class S>`

```
inline Vector<Hdl<Minus<Vector<R>,Vector<S> > > > operator-(const Vector<R>& a, const
Vector<S>& b)
```
Substraction operator.

- `template <class T, class R>`

```
inline typename Vector<Hdl<Minus<T,R> > >::coef_type _elem(const Vector<Hdl<Minus<T,R>
> > & v, unsigned int i)
```

- `template <class R>`

```
inline Vector<Hdl<UMinus<Vector<R> > > > operator-(const Vector<R>& a)
```
Unitary minus operator.

- `template <class R>`

```
inline typename Vector<Hdl<UMinus<R> > >::coef_type _elem(const Vector<Hdl<UMinus<R>
> > & v, unsigned int i)
```

- `template <class R>`

```
inline Vector<Hdl<Mult<Vector<R>, typename Vector<R>::coef_type> > > operator*(const
Vector<R>& a, const typename Vector<R>::coef_type & c)
```
Multiplication by a scalar.

- `template <class R>`

```
inline typename Vector<R>::coef_type _elem(const Vector<Hdl<Mult<Vector<R>,typename
Vector<R>::coef_type> > >& v, unsigned int i)
```

- `template<class R>`

```
inline Vector<Hdl<Divd<Vector<R>, typename Vector<R>::coef_type> > > operator/(const
Vector<R>& a, const typename Vector<R>::coef_type & c)
```
Division by a scalar.

- `template <class R>`

```
inline typename Vector<R>::coef_type _elem(const Vector<Hdl<Divd<Vector<R>,typename
```

```
Vector<R>::coef_type> > >& v, unsigned int i)
```

13.1.9 Referenced vector

☐ **template <class R> struct Vector<Ref<R> >**

This is the referenced class of vectors

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

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

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

13.1.10 Dense univariate polynomial

Univariate polynomials. Standard (dense) representation.

☐ **template<class R> struct UPol<Std<R> >**

Interface for standard univariate polynomials.

```
{
  R rep;

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

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


  UPol<Std<R> > & operator=(const UPol<Std<R> > & P);
  template <class S>
  UPol<Std<R> > & operator=(const UPol<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>
  UPol<Std<R> > & operator+=(const UPol<S>& a);
  template<class S>
  UPol<Std<R> > & operator-=(const UPol<S>& a);
  UPol<Std<R> > & operator*=(const coef_type & c);
  };
```

- `template <class R, class S>`

`inline typename UPol<R>::coef_type _elem(const UPol<Hdl<Mult<UPol<R>, UPol<S> > > > & M, unsigned int i)`
The i element of the product.

- `template <class R>`

`inline void reciprocal(UPol<R> & w,const UPol<R> & p)`
Computing the reciprocal of the polynomial p. We assume p[0] != 0.

13.1.11 Multivariate polynomial

□ **template <class R, class O> class MPol**

Interface for multivariate polynomials.

```
{
public:
// Types
typedef typename Cat<R>::MMon monom_type;
typedef typename R::iterator iterator;
typedef typename R::const_iterator const_iterator;

// Constructors
MPol(){}
MPol(const monom_type & m) { build(p, m); }
MPol(int, const monom_type *);
MPol(const MPol & P) { copy(p, P.p); }

// Destructor
// no dynamic allocation at this level.

// Assignment operators
MPol & operator=(const MPol &);
MPol & operator+=(const monom_type &);
MPol & operator+=(const MPol &);
MPol & operator-=(const monom_type &);
MPol & operator-=(const MPol &);
MPol & operator*=(const monom_type &);
MPol & operator*=(const MPol &);
MPol & operator /= (const monom_type &);

// Boolean operators and functions
```

```
bool isNull() const{ return IsNull(p); }
friend bool operator == <>(const MPol &, const MPol &);
friend bool operator != <>(const MPol &, const MPol &);
friend bool operator < <>(const MPol &, const MPol &);
friend bool operator <= <> (const MPol &, const MPol &);
friend bool operator > <>(const MPol &, const MPol &);
friend bool operator >= <> (const MPol &, const MPol &);

// Comput operators and functions
MPol operator-() const;
friend MPol operator + <>(const MPol &, const MPol &);
friend MPol operator - <>(const MPol &, const MPol &);
friend MPol operator * <>(const MPol &, const monom_type &);
friend MPol operator * <>(const MPol &, const MPol &);

// Input/Output operator
friend ostream & operator << <>(ostream &, const MPol &);

  // Iterator functions
iterator begin () { return p.begin(); }
const_iterator begin () const { return p.begin(); }
iterator end () { return p.end(); }
const_iterator end () const { return p.end(); }

private:
// Data
R p;

// private member functions
iterator add (const MPol &, iterator);
iterator sub (const MPol &, iterator);
};
```

13.1.12 Multivariate monomial

□ **template <class C, class R> class MMon**

Interface for monomial. It is defined by a set of exponents `rep` *and a set of coefficients* `coef`*.*

```
{
public:
   // Type definitions
  typedef C Coeff;
  typedef R Rep;

   // Constructors
   MMon():coeff(CST<C>::Zero),rep(){}
   MMon(const C & c, const R & r):coeff(c),rep(r){}
   MMon(const C & c):coeff(c),rep(){} // monome constant
  MMon(int i, int d):coeff(1){} // monome x_i^d
  MMon(const C & c, int s, int *t):coeff(c){ // from an array of exponents
      build(rep,s,t);
   }
```

```
   MMon(const MMon & m):coeff(m.coeff),rep(m.rep){} // copy constructor

   // Destructor
   // not dynamic allocation at this level

  // affectation operator
  MMon & operator = (const MMon & m);

  // Accessors
  C GetCoeff () const;
  void SetCoeff (const C & c);
  int Lvar() const;
  int GetDegree () const;
  int GetDegree (int i) const;
  void SetDegree(int i, int d);

  // boolean opeartors and functions
  friend bool IsComparable (const MMon & m1, const MMon & m2);
  friend bool operator == (const MMon & m1, const MMon & m2);

   // Comput operators
  MMon operator - () const;
  MMon & operator += (const MMon &);
  MMon & operator -= (const MMon &);
  MMon & operator *= (const MMon &);
  friend MMon operator + <>(const MMon &, const MMon &);
  friend MMon operator - <>(const MMon &, const MMon &);
  friend MMon operator * <>(const MMon &, const MMon &);

  // Input/output operators
  friend ostream & operator << <>(ostream &, const MMon &);

private:
   C coeff;
   R rep;
};
```

13.2 Implementation

13.2.1 Lapack matrix

□ **template<class C> struct Std<lapack<C> >**

Implementation of standard matrix with a container `lapack<C>`*.*

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

  static void transpose(lapack<double> & A);
```

```
  static void solve_ls(array1d<C> & x,
       const lapack<C> & A, const array1d<C> & b);
  static void solve_ls(lapack<C> & A, array1d<C> & b);

  static void solve_lu(array1d<C> & x,
       const lapack<C> & A, const array1d<C> & b);
  static void svd(array1d<C> & x, const lapack<C> & A);
  static void svd(array1d<C> & x, const lapack<C> & A, lapack<C> & U);
  static void svd(array1d<C> & x, const lapack<C> & A,
  lapack<C> & U, lapack<C> & V);

  static void eignval(array1d<AlgClos<C>::TYPE> & V, lapack<C> & A);
  static void leigvct(lapack<AlgClos<C>::TYPE> & E , lapack<C> & A);
  static void leigvct(lapack<AlgClos<C>::TYPE> & E,
      array1d<AlgClos<C>::TYPE> & V, lapack<C> & A);
  static void reigvct(lapack<AlgClos<C>::TYPE> & E, lapack<C> & A);
  static void reigvct(lapack<AlgClos<C>::TYPE> & E,
      array1d<AlgClos<C>::TYPE> & V, lapack<C> & A);
  static integer dcmp_lu(lapack<C> & A, integer *pivot);
  static integer dcmp_qr(lapack<C> & A);
  static integer dcmp_qr(lapack<C> & A, unsigned int * pivot, C * tau);

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

□ **template<class C> struct Spl< lapack<C> >: public Std<lapack<C> >**

Implementation of specialized matrix with a container `lapack<C>`.

```
{
  static void copy (lapack<C> & A, const lapack<C> & B) {}
  static void scopy(lapack<C> & A, const lapack<C> & B) {}
  static void add (lapack<C> & A, const lapack<C> & B) {}
  static void add (lapack<C> & R, const lapack<C> & A, const lapack<C> & B) {}
  static void sub (lapack<C> & A, const lapack<C> & B) {}
  static void sub (lapack<C> & R, const lapack<C> & A, const lapack<C> & B) {}
  static void mult(lapack<C> & A, const lapack<C> & B) {}
  static void mult(lapack<C> & R, const lapack<C> & A, const lapack<C> & B);
  static void add(lapack<C> & R, const lapack<C> & A, const C & c) {}
};
```

- `template <> void Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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 Std<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.

13.3 Container

13.3.1 array1d<C>, one-dimensional array

□ **template <class C> struct array1d**

Class of array of coefficients with size of the array.

```
{
  int size_;
  C * tab_;

  typedef C            coef_type;
  typedef unsigned int index_type;
```

```
  array1d() : size_(0){}
  array1d(const array1d<C> & v);

  array1d(unsigned int i);
  array1d(unsigned int i, C* t);
  array1d(const char * nm);

  rebuild(unsigned int i);

  array1d<C> & operator=(const array1d<C> & v);

  unsigned int size() const;
  inline C & operator[] (unsigned int i)
  inline C   operator[] (unsigned int i) const;

  void scopy(const array1d<C> & v);
};
```

13.3.2 `array2d<C>`, bi-dimensional array

□ **template <class C> struct array2d : public array1d<C>**

Class of `array2d<C>` *derived from* `array1d<C>`*, including the number of rows* `nbrow_` *and the number of columns* `nbcol_` *of the matrix. The matrix is stored by column in the array* `array1d<C>`*.*

```
{
  int nbrow_, nbcol_;

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

  array2d(): array1d(){}
  array2d(const array2d<C> & v);
  array2d(unsigned int i,unsigned int j=1):;
  array2d(unsigned int i, unsigned int j, C* t);
  array2d(const char * nm);

  void rebuild(unsigned int i,unsigned int j);
  void scopy(const array2d<C> & v);

  array2d<C> & operator=(const array2d<C> & v);

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

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

13.3.3 lapack<C>

Container for connection with the fortran library LAPACK.

□ **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_` *($\leq$* `lda_`*) and its number of columns* `nbcol_`*.*

```
{
  unsigned int lda_;

  lapack(): array2d() {}
  lapack(const lapack<C> & v): array2d(v), lda_(v.lda_){}
  lapack(unsigned int i, unsigned int j=1): array2d<C>(i,j), lda_(i){}
  lapack(unsigned int i, unsigned int j, C* t): array2d<C>(i,j,t), lda_(i){}
  lapack(unsigned int i, unsigned int j, unsigned int k, C* t):
    array2d<C>(i,j,t), lda_(i) {nbrow_=k;}
  lapack(const char * nm):array2d(nm) {}

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

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

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

13.3.4 DynamicArray, multivariate monomial

In this case the set of exponents of a monomial is a pointer.

□ **class DynamicArray**

Container class of dynamic array, it is defined by its size `size`*, a pointer on an array of int* `tab`*, which is an array of exponents.*

```
{
public:
DynamicArray():size(0),tab(NULL){}
DynamicArray(const DynamicArray & d):size(d.size){
if (size) {
tab = (int *)malloc(sizeof(int)*size);
memcpy(tab,d.tab,sizeof(int)*size);
}
else
tab = NULL;
}

~DynamicArray(){
if (tab)
```

```
free(tab);
}

DynamicArray & operator = (const DynamicArray & A) {
if (tab)
free(tab);
size = A.size;
if (size) {
tab = (int *)malloc(sizeof(int)*size);
memcpy(tab,A.tab,sizeof(int)*size);
}
else
tab = NULL;
return *this;
}

friend bool operator == (const DynamicArray & A, const DynamicArray & B) {
if (A.size == B.size)
return memcmp(A.tab,B.tab,sizeof(int)*A.size) == 0;
return 0;
}

int size;
int *tab;
};
```

- `ostream & operator « (ostream & os, const DynamicArray & t)`
- `void build(DynamicArray & A, int i, int d)`
- `void build(DynamicArray & A, int s, int *tab)`
- `int GDegree (const DynamicArray & A, int i)`
- `void SDegree (DynamicArray & A, int i, int d)`
- `int lvar (const DynamicArray & A)`
- `void erase (DynamicArray & A)`
- `void add (DynamicArray & A, const DynamicArray & B)`

13.3.5 RepTab, multivariate monomial

In this case the set of exponents of monomial is an array.

□ **template <class C, int n> class RepTab**

Container class for monomial, its is defined by an array of exponents `t_exp` *and a coefficient* `coeff`.

```
{
public:
typedef C Coeff;

RepTab():coeff(0) {
for (int i=0; i<n; ++i)
t_exp[i] = 0;
}
```

```
RepTab(const C &);
RepTab(int, int);
RepTab(const C &, int ,int *);

void copy (const RepTab &);

void add (const RepTab &);
void sub() { coeff = - coeff; }
void sub (const RepTab &);
void mult (const RepTab &);

Coeff GetCoeff() const;
int Degree(int i) const;

void SetCoeff(const Coeff & c);
void SetDegree(int i, int d);

        friend bool operator == <>(const RepTab &, const RepTab &);

friend ostream & operator << <> (ostream &, const RepTab &);
private:
C coeff;
int t_exp[n];
};
```

13.4 Ordering on monomials

13.4.1 Plex<M>, lexical order

□ **template <class M> class Plex**

Class defining the lexicographic order on the monomials $m1$ and $m2$; $m1 < m2$ if in the vector difference $s1 - s2$, where $s1$ and $s2$ are the vectors of exponents of $m1$ and $m2$ resp., the left-most nonzero entry is negative.

```
{
public:
   static bool less (const M &, const M &);
};

template <class M>
bool Plex<M>::less(const M & m1, const M & m2) {
   int d1,d2;
   int n1 = m1.Lvar(), n2 =m2.Lvar();
   int n = max(n1,n2);

   if(n1 <0 && n2>0) return true;
   if(n2 <0 && n1>0) return false;

   int i=n;
   while( m1.GetDegree(i) == m2.GetDegree(i) && i>=0) i--;
```

```
   if(i <0)
     return false;
   else
     return (m1.GetDegree(i)< m2.GetDegree(i));
}

template <class M>
bool operator<(const M & m1, const M & m2) {return  Plex<M>::less(m1,m2);}
```

13.4.2 Vlex<M>, valuation lexical order

□ **template <class M> class Vlex**

Class defining the following order on the monomials m1 and m2; m1 < m2 if either the degree of m1 is strictly greater than the degree of m2, or they are equal and m1 is greater for the inverse lexicographic ordering. In a sequence of decreasing monomial for this order, the degree are increasing.

```
{
public:
   static bool less (const M &, const M &);
};
```

```
template <class M>
bool Vlex<M>::less(const M & m1, const M & m2) {
   int d1,d2;
   int n1 = m1.Lvar(), n2 =m2.Lvar();
   int n = max(n1,n2);

   if(n1 <0 && n2>0) return false;
   if(n2 <0 && n1>0) return true;
   if (m1.GetDegree() < m2.GetDegree()) return false;
   else if (m1.GetDegree() > m2.GetDegree()) return true;
   else {
     int i=n;
     while( m1.GetDegree(i) == m2.GetDegree(i) && i>=0) i--;
     if(i <0)
       return false;
     else
     return (m1.GetDegree(i)> m2.GetDegree(i));
   }
}
```

13.4.3 SQVlex<M>, square valuation lexical order

□ **template <class M> class SQVlex**

Class defining the following order on the monomials m1 and m2; m1 < m2 if either the maximal of the degrees of each variable is strictly greater in m1 than in m2, or they are equal and m1 is greater for the inverse lexicographic ordering refined by the degree.

```
{
public:
   static bool less (const M &, const M &);
};
```

```
template <class M>
bool SQVlex<M>::less(const M & m1, const M & m2) {

   int n1 = m1.Lvar(), n2 =m2.Lvar();
   int n = max(n1,n2);

   if(n1 <0 && n2>0) return false;
   if(n2 <0 && n1>0) return true;
   if(n1 <0 && n2<0) return false;

   int d1=0,d2=0, k, c1, c2;
   for(int i =0 ; i<= n1; i++) {c1 = max(d1,(k=m1.GetDegree(i)));d1+=k;}
   for(int i =0 ; i<= n2; i++) {c2 = max(d2,(k=m2.GetDegree(i)));d2+=k;}

   if      (c1 < c2) return false;
   else if (c1 > c2) return true;
   else if (d1 < d2) return false;
   else if (d1 > d2) return true;
   else {
     int i=n;
     while( m1.GetDegree(i) == m2.GetDegree(i) && i>=0) i--;
     if(i <0)
       return false;
     else
     return (m1.GetDegree(i)> m2.GetDegree(i));
   }
}
```

13.5 Special functions

13.5.1 Functions for multivariate polynomial

- `template <class R, class O> int Lvar (const MPol<R,O> & p)`

Return the number of variables of the monomial of p which has the higher one.

- `template <class R, class O> int Degree (const MPol<R,O> &)`

Return the degree of a polynomial, that is the degree of its monomial with the higher degree.

- `template <class R, class O> int Degree (const MPol<R,O> &, int i)`

Return the degree of a polynomial in the variable x_i, that is the degree of its monomial with the higher degree in x_i.

- `template <class R, class O> int NbCoef (const MPol<R,O> &)`

Return the number of non zero coefficients of a polynomial

- `template <class R, class O> MPol<R,O> Coeff(const MPol<R,O> & p, int i)`

Return a polynomial composed by the monomials of the polynomial p which contain the variable x_i.

- `template <class R, class O> MPol<R,O> Coeff(const MPol<R,O> &, int i, int i)`

Return a polynomial composed by the monomials of the polynomial p which contain the variable x_i, with degree d.

- `template <class P> P Pdiv (const P & p, const typename P::M & m)`

Divide a polynomial p by a monomial m

13.5.2 Macaulay

Construction of the Macaulay matrix, from a vector of $n+1$ polynomials in n variables. The order choosed to sort the monomials is the same as the order associated to the polynomials `POL`$>$ *The second argument specifies the type of the output matrix. For instance,* `Macaulay(V, array2d<double>());` *will output a matrix of type* `Matrix<Ref<array2d<double> > >`.

```
template <class POL, class RepM>
Matrix<Ref<RepM> >
Macaulay(const Vector<Std<array1d<POL> > > & f, RepM  M){

  typedef typename POL::monom_type MON;

  int d[f.size()], n = (f.size()-1), nu = 0;
  POL F;
  // computation of mu
  for(int i=0; i<=n; i++){
    d[i] = Degree(f[i]);
    nu += d[i];
   }
  nu -= n;
  MON* xdi = new MON[n+1];
  POL* D = new POL[n+1];
  xdi[0]= MON(1.0);
  D[0] = POL(xdi[0]);
  for(int j=1; j<=n; j++){
    xdi[j] = MON(j,d[j]);
    for(int i=0;i<d[j];i++) D[j]+= MON(j,i);
  }
  // Construction of the set of monomials of degree mu.
  int taille = Choose(nu, n, F);
  // Construction of the local basis
  POL* E= new POL[n+1];
  POL U(1.),V(1.);
  int mu=nu;
  for(int i=n;i>0;i--){
    mu = nu-d[i];
    for (typename POL::const_iterator j = U.begin(); j!=U.end(); j++){
      Choose(mu-j->GetDegree(),i,V);
      E[i] += V*(*j);
    }
    U*=D[i];
  }
  E[0] = U;
  //Index of monomials
  map<MON,int> index;
  int c =0;
  MON m;
  for(int i=0; i <=n;i++){
    for (typename POL::const_iterator j = E[i].begin(); j!=E[i].end(); j++){
      m = (*j)*xdi[i];
      index[m]=c;
      c++;
```

```
    }
  }

  //Construction of the Macaulay matrix
  RepM *Res = new RepM(taille,taille);
  int u, v=0;
  POL P;
  for(int i=0; i <= n; i++){
    for (typename POL::const_iterator j = E[i].begin(); j!=E[i].end(); j++){
      P = f[i]*(*j);
      for (typename POL::const_iterator k = P.begin(); k!=P.end(); k++)
(*Res)(index[*k],v)=k->GetCoeff();
      v++;
    }
  }
  return Matrix<Ref<RepM> >(Res);
};
```

13.5.3 Bezoutian

Bezout matrices.

- `template <class R> Matrix<Ref<array2d<typename R::coef_type> > > Bezout(const UPol<Std<R> > & P, const UPol<Std<R> > & Q)`

Bezout matrix.

- `template <class R> Matrix<Ref<hankel<typename R::coef_type> > > Bezout(const UPol<Spl<R> > & P)`

Bezout matrix of P and 1.

Bibliography

[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] Dario A. Bini Bernard Mourrain. FRISCO test suite handbook of polynomial systems - multivariate, 1997. `http://www.inria.fr/saga/POL/BASE/2.multipol/index.html`.

[3] Dario A. Bini Bernard Mourrain. FRISCO test suite handbook of polynomial systems - univariate, 1997. `http://www.inria.fr/saga/POL/BASE/1.unipol/index.html`.

[4] Victor Y. Pan Bernard Mourrain. Multivariate polynomials, duality and structured matrices. 1997.

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

[6] D. Bini and V. Pan. *Polynomial and matrix computations, Vol 1 : Fundamental Algorithms.* Birkhäuser, Boston, 1994.

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

[8] J.P. Cardinal. On two iterative methods for approximating the roots of a polynomial. 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 165–188. Am. Math. Soc. Press, 1996.

[9] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms.* Undergraduate Texts in Mathematics. Springer, New York, 1992.

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

[11] Free Software Foundation. GNU multiple precision library, 1996. `ftp://prep.ai.mit.edu/pub/gnu/gmp-M.N.tar.gz`.

[12] I.M. Gelfand, M.M. Kapranov, and A.V. Zelevinsky. *Discriminants, Resultants and Multidimensional Determinants.* Birkhäuser, Boston-Basel-Berlin, 1994.

[13] Golub, Gene H. and Van Loan, Charles F. . *Matrix computations*, volume (third edition). John Hapkins, Univ. Press, Baltimore, Maryland, 1996.

[14] W.B. Gragg. Positive definite Toeplitz matrices, the Arnoldi process for isometric operators and Gaussian quadrature on the unit circle. In E.S. Nikolaev, editor, *Numerical methods in linear algebra*, pages 16–32. Moscow university press, 1982. (In Russian).

[15] Roldan Pozo Jack Dongarra, Andrew Lumsdaine and Karin A. Remington. IML++ iterative methods library, 1996. `http://math.nist.gov/iml++/`.

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

[17] Bernard Mourrain. An introduction to linear algebra methods for solving polynomial equations.

[18] Bernard Mourrain. ALP algèbre linéaire pour les polynômes, 1996. `http://www.inria.fr/saga/Bernard.Mourrain/ALP/`.

[19] Roldan Pozo and Andrew Lumsdaine Karin A. Remington. SparseLib++ sparse matrix class library, 1996. `http://math.nist.gov/sparselib%2B%2B`.

[20] Y. Saad. *Iterative methods for sparse linear systems.* PWS series in computer science. PWS, 1996.

[21] H. J. Stetter. Eigenproblems are at the Heart of Polynomial System Solving. *SIGSAM Bulletin*, 30(4):22–25, 1996.

[22] B. Sturmfels. Sparse elimination theory. In D. Eisenbud and L. Robianno, editors, *Computational Algebraic Geometry and Commutative Algebra*, pages 264–298. Cambridge Univ. Press, 1993. (Proc. Cortona, June 1991).

[23] Iain S. Duff Timothy A. Davis. UMFPACK unsymmetric-pattern multifrontal package, 1998. `http://www.netlib.org/linalg/umfpack.tar.gz`.

[24] Charles VanLoan. *Computational frameworks for the fast Fourier transform*, volume X. Philadelphia : SIAM, 1992.