INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A framework for symbolic and numeric computations

B. Mourrain & H. Prieto

N° ????

15 Septembre 2000

THÈME 2

Rapport de recherche

ISSN 0249-6399

A framework for symbolic and numeric computations

B. Mourrain & H. Prieto

Thème 2 — Génie logiciel
et calcul symbolique
Projet Saga

Rapport de recherche n° ???? — 15 Septembre 2000 — 96 pages

Abstract: The need to combine symbolic and numeric computations is ubiquitous is many problems such as the forward and inverse kinematics of robots, motion planning, the analysis of the geometric structure of molecules, computational geometry, geometric and solid modelling, graphics, computer-aided design, computer vision, signal processing ...Starting with an exact or approximate description of the equations, we will eventually have to compute a numerical approximation of the solutions. This leads to new, interesting and challenging questions either from a theoretical or practical point of view at the frontier between algebra and analysis.

The objective of this report is to describe a framework for symbolic and numeric computations, called ALP, which can be used to validate these new developments and to build devoted applications or solvers. This environment allows us to define parameterised but efficient classes of basic algebraic objects such as vectors, matrices, monomials and polynomials, ...which can be used easily in the construction of more elaborated algorithms. We pay a special attention to genericity and to reusability of external libraries such as LAPACK, SUPERLU, GMP ...In this report, we describe the structure of the software and the main tools or classes which are available.

Key-words: vector, matrix, polynomial, equations, resultant, eigenvalue, solver, data-structure, library

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 — Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

Un environnement pour le calcul symbolique et numérique

Résumé : Le besoin de combiner des calculs numériques et symboliques est présent dans beaucoup de problèmes, allant de la robotique à la géométrie algorithmique en passant par l'analyse de structures moléculaires, la vision par ordinateur, le traitement du signal ... Dans ces applications, les équations peuvent être décrites de manière exacte ou avec des coefficients approchés et une bonne approximation numérique des coordonnées des points doit être calculée. Ces problèmes soulèvent de nouvelles et intéressantes questions à la fois théoriques et pratiques, à la frontière entre l'algèbre et l'analyse. L'objectif de ce rapport est de décrire un environnement permettant de combiner le symbolique et le numérique, en vue de valider ces nouveaux développements ou de mettre en place des solveurs dédiés à certains type de problèmes. Celui-ci s'appuie sur des classes paramétrées de vecteurs, de matrices, de polynômes qui peuvent être agencées et instancier afin de construire des applications ou algorithmes efficaces. Nous avons apporté une attention particulière à la généricité des codes et la connexion avec des bibliothèques externes et spécialisées telles que LAPACK, SUPERLU, GMP ... Dans ce rapport, nous décrivons la structure de la bibliothèque ainsi que les principaux outils disponibles.

Mots-clés : vecteur, matrice, polynome, équations, résultant, valeur propre, solveur, structure de données, bibliothèque

Contents

Chapter 1

Introduction

The objective of this report is to describe a framework for symbolic and numeric computations, called ALP[1]. The need to combine these two domains is ubiquitous is many problems. Starting with an exact description of the equations, in most of the cases, we will eventually have to compute an approximation of the solutions. Even worse, in many problems the coefficients of the equations may be known with some inaccuracy (due, for instance, to measurement errors). In this case, we are not dealing with a solely system but with a neighbourhood of systems and we have to take into account the continuity of the solutions with respect to the input coefficients. This leads to new, interesting and challenging questions [50, 53] either from a theoretical or practical point of view at the frontier between algebra and analysis and witnesses the emergence of a new investigation [65, 41].

Many of these developments have applications in the forward and inverse kinematics of robots and mechanisms as well as the computation of their motion plans [14, 56], the geometric structure of molecules [3, 29], computational geometry, geometric and solid modelling, graphics, and computer-aided design [34, 2, 39, 46], computer vision, signal processing [53] as well as quantifier elimination [20, 57, 15, 4], and the solution of systems of inequalities [37]. The tools that we describe here and which are based on matrix manipulations, [64, 31] are also used in effective algebraic geometry [27], in complexity theory [62, 33, 51]. They appear to be fundamental in several other domains such as residue theory [63, 43] and complex analysis [7, 6, 1]. Their applications from an algorithmic point of view are illustrated in [17, 5, 19, 28]. We refer to [31] for a more detailed description of these methods.

Because of all these different domains of application, this environment has to provide various internal representations for the objects that will be manipulated. Thus, this library allows to define parameterised but efficient classes of basic algebraic objects such as vectors, matrices, monomials and polynomials, . . . which can be used easily in the construction of more elaborated algorithms. We pay a special attention to genericity or more precisely to parameterised classes (templates) which allow us to tune easily the data-structure to

[1] `http://www.inria.fr/saga/logiciels/ALP/`

the problem to be solved. So called *template expression* are used to guide the expansion of code during the compilation and thus to get very efficient implementations. The usual developments of libraries, in particular for scientific computations, is usually understood as a collection of "passive" routines. The approach that we adopt here is inspired by a new paradigm of software developments called *active library* [22, 68, 25] and illustrated by the library STL [54]. The software components of such active libraries take part to the configuration and generation of codes (at compilation time), which combine parametrisation and efficiency. Projects like BLITZ[2], SciTL[3], MTL[4] are built on these ideas. The present environment is also a step in this direction, with an orientation toward algebraic data-structures such as polynomials.

We also consider carefully the problem of reusability of external libraries. Currently, we have connected LAPACK (fortran library for numerical linear algebra), UMFPACK, SparseLib (respectively fortran and C++ libraries for sparse matrices), SUPERLU (C library for solving sparse linear system), GMP (C library for extended arithmetics), GB, RS, ... Thus, specialised implementation for instance using LAPACK routines, can coexist with generic one, the choice being determined by the internal representation. We also want this library to be easy to use. Once the basic objects have been defined by specialisation of the parameters, including maybe the optimisation of some critical routines, we want to be able to write the code of an algorithm "as it is on the paper". The polymorphism and genericity of C++ allows us to work in this direction.

We have distinguished three levels of structure, implemented either as classes or namespaces: **containers, views** and **modules**. We are going to describe them now. Next, we will go through a tutorial on how to use them and then we will present the tools that are currently available in the environment. For a complete description, we refer to [52].

1.1 Containers

They specify the internal representation of the objects. They provide methods or functions for accessing, creating, transforming this representation and thus depend closely on the data-structure. They are implemented as classes (often parameterised by coefficient types or index types), with few functions so that they can be rewritten or extended easily.

We follow the approach of the STL library [54], which implements basic structures such as `list<R>, vector<R>, set<R>, deque<R>, ...` New containers have been added such as one-dimensional arrays `array1d<C>` with generic coefficients, two-dimensional arrays `array2d<C>` for dense matrices, `lapack<C>` for the connection to the Lapack library, `sparselu<C>` for the connection to SuperLU, ...

[2] `http://monet.uwaterloo.ca/blitz/`
[3] `http://www.acl.lanl.gov/SciTL/`
[4] `http://www.lsc.nd.edu/research/mtl/`

1.2 Views

They specify how to manipulate or to see the containers as mathematical objects and provide operations on the objects which are independent of the container. They are implemented as classes parameterised by the container type and sometimes by *trait* classes which precise the implementation. The only data available in such a class, called `rep`, is of container type.

For instance a one-dimensional array of `double` numbers can be seen as a vector, using the type

```
VectStd<array1d<double> >.
```

Among the operations defined in this class, we have for example, the vector operators `+=, -=, +, -` and the scalar multiplications `*, *=`. But such an array can also be viewed as a univariate polynomial. In this case, we will define the following type :

```
UPolyDense<array1d<double> >
```

which extends the operations given in `VectStd<array1d<double> >` by adding the polynomial multiplication operators `*=, *` and changing the output operator. The implementations of these views are defined in a module and can easily be specialised (see next section).

Here are the main examples of view classes that are implemented in the current version of the library, `R` standing for the container type: `VectStd<R>` (standard vectors), `MatDense<R>` (dense matrices), `MatStruct<R>` (structured matrices), `MatSparse<R>` (sparse matrices), `UPolyDense<R>` (dense univariate polynomials), `UPolyQuot<R>` (quotiented univariate polynomials), `Monom<C,E>` (monomials with coefficients in `C` and exponents in `E`), `MPoly<R,O>` (multivariate polynomials where `O` defines the order on the monomials).

1.3 Modules

A module is a collection of implementations which apply to a family of objects sharing common properties (such as vectors, matrices, univariate polynomials, ...). They are implemented as *namespaces* and thus can be extended easily. They are used to define generic implementations. For instance in the module (or *namespace*) `VECTOR`, we have gathered generic implementations for vectors independent of the internal representation, such as `Print`, `Norm`, ...

The main modules of the library are `VECTOR`, `MATRIX`, `LINALG` (linear algebra implementations), `UPOLYNOMIAL` (univariate polynomials), `MPOLYNOMIAL` (multivariate polynomials).

As explained before, we need to be able to specialise some critical functions of our classes. For the univariate polynomials `UPolyDense<array1d<C> >`, we would like for instance, to replace the naive multiplication by a more efficient one, based eg. on FFT. This can be done easily by extending the namespace `UPOLYNOMIAL` and by defining the function:

```
template<class C>
void UPOLYNOMIAL::mult(array1d<C> & r, array1d<C> & a, const array1d<C> & b)
   {...}
```

The module `VECTOR` is detailed in section 3.2.

1.4 Template expression

Template expressions are type manipulations used to guide the compiler to produce optimised code. We illustrate its use on the addition of vectors. Consider for instance the following instruction:

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

where `v, v1, v2, v3` are vectors (say of type `Vector`). Usual implementations would defined the operator

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

so that the previous instruction will produce the temporary vector (say `vtmp1`) for the addition of `v1` and `v2`, another one (say `vtmp2`) for the addition of `vtmp1` and `v3` and will use the operator

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

to assign `v` to the value of `vtmp2`, using a loop

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

In such an implementation, two temporary vectors are allocated and deleted, which is time and memory consuming, especially if the vectors are of large size. A more optimised implementation would consist in writing the code

```
for(index_type i = 0; i <v.size(); i++) v[i] = v1[i] + v2[i] + v3[i];
```

This code can be generated directly, at compilation time, using template expression techniques. Instead of defining the operator `+`, as before, we define it with the following signature:

```
VAL<Op<'+',Vector,Vector> > operator+(const Vector & v1, Vector & v2);
```

where `VAL<Op<'+',Vector,Vector> >` is the type of a data-structure which contains references to the two vectors `v1, v2`, but which does not compute the sum of this two vectors. The instruction `v1+v2+v3` that we are considering, produces a data of type

```
VAL<Op<'+', Vector, VAL<Op<'+', Vector, Vector> > > >;
```

Now, the assignment

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

will involve the operator

```
template<class R> Vector & operator=(const VAL<R> & v)
```

which is implemented as

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

The key point is that the compiler will expand the call to the inline function `_elem(v,i)` into `v1[i]+v2[i]+v3[i]`, because the i^{th} element of an element `v` of type `VAL<Op<'+',...,...> >` is the sum of the i^{th} of its two components. Here are the arithmetic operations implemented with template expressions and their meaning:

- `Op<'+',A,B>`, sum of two terms.
- `Op<'-',A,B>`, subtraction of two terms.
- `Op1<'-',A>`, opposite of a term.
- `Op<'*',A,B>`, multiplication of two terms.
- `Op<'.',A,B>`, multiplication of a term by a scalar
- `Op<'/',A,B>`, division of two terms.
- `Op<'%',A,B>`, division of a term by a scalar.
- `Op<'^',A,B>`, power of a term by the other.

Such arithmetic operations will return an unevaluated arithmetic tree and the computation will be performed only when the assignment operator will be used.

This mechanism is also used when a function `f` has to return an object say of type `R`, that need to allocate its memory space. Instead of returning a copy of the computed object, we use the following mechanism:

1. Allocation of a pointer to the required memory space: `R* result= new R(...);`
2. Computation of its value `*result =...;`
3. Return `VAL<R*>(result)`, which has to be understood as the value of the object associated to the pointer.

This allows for instance to avoid copies of large objects, when the result of such function is assigned to a variable. Usually only *copies of pointers* are needed at this stage.

This mechanism is a new and very interesting programming feature of C++. It can be seen as an extension of the functional approach. The functions are not returning an object but a way to compute or to recover this object. The computation is performed only when required, that is, in most of the cases, in the assignments. We exploit it intensively in the development of the library.

1.5 Historical background

The development of this library started during the Frisco project (LTR 21.024, 1996-1999) with the work of F. Livigni. At the beginning, it was structured along abstract classes, virtual functions and derivation mechanisms. As this approach suffers from efficiency, and as the template expressions of `C++` where becoming stronger we decided to rewrite it completely, in terms of parametrised classes (or *templates*). We spend some times (more than six months), in tests, experimentations, ... before we decided to adopt the structure described in this report. This evolution benefits from many discussions with G. Dos Reis, P. Aubert, T. Papadopoulos, that we would like particularly to thank.

1.6 A tutorial

Step 1: Using the standard library

The basic data structures of the standard `C++` library are available, once you have included the correct header files. Many of our constructions will be based on such containers. If you want for instance to manipulate lists of, say integer, you will use the type:

```
list<int>
```

which is a parametrised type `list<T>` in which we instantiate the type `T` by `int`.

```
1  #include <list.h>
2  typedef list<int>::iterator iterator;
3
4  int main(){
5
6   list<int> l;
7   l.push_back(1); l.push_back(2);  l.push_back(3);
8
9   for(iterator it=l.begin(); it!=l.end(); it++) cout<<*it<<" ";
10  cout <<endl;
11 }
```

Let us detail what is going on here:

1: As we use the list of the standard library, we include the header file `list.h`.

2: We define an abbreviation `iterator` for the iterators on the list of `int`.

6: We declare `l` as a list of `int`.

7: We append at the end of the list, the three elements $1, 2, 3$.

9: We print it. We use an `iterator` to scan this list, from the beginning `l.begin()` to the end `l.end()`. See [54, 25] for more information on these iterators and their correlation with `generic` programming.

10: We print a new line.

Once you have written this code in a file `fich.cc`, you can compile it with the usual compiler command

```
g++ -o fich.ex fich.cc
```

and run the executable `fich.ex` or you can directly use the command:

```
alp fich.cc
```

which will also produce the executable `fich.ex`. The result of this first execution should look like:

```
   1 2 3
```

For more information, on the basic data structures available in the STL, we refer to chapter 2 or to [54], [58].

Step 2: linear algebra

The main categories of objects for doing linear algebra are

- `VectStd`: Standard dense vectors (3.1.2),
- `MatDense`: Dense matrices (3.3.2),
- `MatStruct`: Structured matrices (3.4.2),
- `MatSparse`: Sparse matrices (3.5.1).

Before using these classes you will have to chose the container for the internal representation as illustrated now: If you want to construct standard vectors with `double` coefficients, you can use for instance the definition:

```
typedef VectStd<array1d<double> > Vect;
```

The container `array1d<C>` (3.1.4) is a one-dimensional array of `C`, with a field specifying its size. But you can also used the containor `vector<C>` of the standard library or any other convenient container. See section 3.1.2 for more details. A vector can be constructed, for instance, from an array:

```
double u[]={1,-2,1}; Vect V(3,u);
```

For the definition of dense matrices with `double` coefficients, you can use the definition:

```
typedef MatDense<lapack<double> > Mat;
```

Here we could also have used the container `array2d<C>` (3.3.3) which extends `array1d<C>`, by specifying the number of rows and columns, but we prefer the `lapack<C>` cointainer (3.3.4) which will allow us to use the routines of the LAPACK library. The internal representation is also a one-dimensional array with a number of rows, a number of columns, a size bigger than the product of these two last numbers and a way to access to the element of index (i, j). There are many ways to build a matrix (see 3.3.2). Here we parse a string:

```
Mat A(3,3,"1 2 3 4 5 6 7 8 9 10");
```

It defines the following matrix:

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

Check how are read the elements (in this construction it is by row). The usual operations on matrices and vectors are available:

```
Vect W = A*V; cout <<W<<endl;
```

```
[0,0,0]
```

```
Mat B = A*A; cout <<B<<endl;
```

```
[ 30 36 42 ]
[ 66 81 96 ]
[ 102 126 150 ]
```

```
Mat C(-A); C += B + A*2; cout <<C<<endl;
```

```
[ 31 38 45 ]
[ 70 86 102 ]
[ 109 134 159 ]
```

Views on vectors and matrices

These linear algebra classes also offers local views on their internal representation. A row (or a column) of a matrix, for instance, is a view on the corresponding elements of the matrix and share the properties of a vector. Indeed, it is a `VectStd` with a special container and can be manipulated as such. The types of the columns and the rows are accesible as

```
MatDense<lapack<double> >::row_type
MatDense<lapack<double> >::col_type
```

The characteristic of these views is that performing operations on the vectors will affect the object the view is looking at.

```
Col(A,1)-= Col(A,0)*2;
```

transforms `A` into:

```
[ 1  0 3 ]
[ 4 -3 6 ]
[ 7 -6 9 ]
```

It must be remembered that the memory space of the views `Row(A,i)`, `Col(A,i)` is not duplicated but shared with the matrix `A`. It can be copied into a standard vector (each element is copied):

```
W = Col(A,0);
```

Be carefull when you use these local views:

```
Col(A,0)=Col(A,1);
```

means that the temporary object which is returned by `Col(A,1)` is assigned to the temporary object returned by `Col(A,0)`. It does not mean that the coefficients viewed by `Col(A,0)` will be set to those viewed by `Col(A,1)`. If this is what you want, here is the command to be used:

```
Col(A,0)=Eval(Col(A,1));
```

Operations between rows (resp. columns) of different matrices are also possible:

```
Mat D(2,2,"1 0 2 0");  Col(A,0)=Col(D,0)*10;
```

yields:

```
[ 10  0 3 ]
[ 20 -3 6 ]
[ -6 -6 9 ]
```

Similarly subvectors also correspond to views on one-dimensional arrays. The operation

```
W[Range(0,1)] = Eval(W[Range(1,2)]);
```

transforms the vector `W=[1,4,7]` into `W=[4,4,7]`. Or you can use operations such as

```
Col(A,0)[Range(0,1)] = Eval(Col(A,1)[Range(1,2)]);
```

which transforms `A` into

```
[ -3  0 3 ]
[ -6 -3 6 ]
[ -6 -6 9 ]
```

Here is the complete file for these examples, that you will compile with the command `alp` in order to get an executable file `.ex`:

```
#include "linalg.H"

typedef VectStd<array1d<double> >   Vect;
typedef MatDense<lapack<double> >   Mat;
typedef Mat::col_type               col_type;

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

  Vect V(3,"1 -2 1");                cout<<V<<endl;
  Mat A(3,3,"1 2 3 4 5 6 7 8 9 10"); cout<<A<<endl;
  Vect W(A*V);                       cout<<W<<endl;
  Mat B(A * A);                      cout<<B<<endl;
  Mat C(-A); C += B + A*2;           cout<<C<<endl;
  Col(A,1)-= Col(A,0)*2;             cout<<A<<endl;
  W = Eval(Col(A,0));                cout<<W<<endl;
  Col(A,0)=Eval(Col(A,1));           cout<<A<<endl;
  Mat D(2,2,"1 0 2 0");
  Col(A,0)=Col(D,0)*10;              cout<<A<<endl;
  W[Range(0,1)]=Eval(W[Range(1,2)]); cout <<W<<endl;
  Col(A,0)[Range(0,1)]=Eval(Col(A,1)[Range(1,2)]); cout <<A<<endl;

}
```

Step 3: univariate polynomials

Univariate polynomials are derived from the vectors. They have a multiplication, a degree and are printed using the variable `x` (4.1.1). Here is an example of definition:

```
typedef UPolyDense<upar<Z<31> > > Pol;
```

We use the view of dense univariate polynomials `UPolyDense`, with the container `upar` which extends `array1d` by storing the degree. The coefficients are of type `Z<31>` that is integers modulo 31 (2.4). In order to be able to use this class, we must first include the following files:

```
#include "upoly.H"       // univariate polynomials
#include "arithm/Zp.H"   // modular numbers
```

A polynomial can be constructed from an array of coefficient or parsed from a string (ending by a `;`):

```
Pol p("x^3-34*x+1;");
```

yields the polynomial

```
(1)*x^3+(-3)*x+(23)
```

The "natural" operations of polynomials are available (see 4.1.1):

```
Pol q = p*p;
```

yields the polynomial

```
(1)*x^6+(-6)*x^4+(15)*x^3+(9)*x^2+(-14)*x+(2)
```

Here, all the operations on the coefficients are performed modulo 31. These polynomials also have an euclidean division:

```
p+= Pol("x+1;");  Pol r= q%p;
```

yields the remainder `r`

```
(1)*x^2+(-29)*x+(1)
```

which is also $(x+1)^2\ mod\ 31$. Notice that the operator `%` will yield the remainder of the Euclidean division if the leading term is invertible (here it is the case for 31 is prime). The quotient is given by the operator `/`. Univariate polynomials can also be evaluated:

```
cout <<q(Z<31>(1))<<endl;
```

yields `9`, which is of type `Z<31>`. The output of the evaluation is of the same type as the input type, if it is compatible with the coefficient type. For instance, `q(1)` will not be valid here, for the evaluation tries to output an `int` from arithmetic operations between `int` and `Z<31>`[5].

Next is another example where we use big integers, defined from the file `bignum.h` (2.3):

```
typedef UPolyDense<upar<Scl<MPZ> > >   ZPol;
ZPol a("x^2-1243432232*x+2334345435434;");
ZPol b=a^3;    cout <<b<<endl;

   (1)*x^6+(-3730296696)*x^5
   +(4638378149765811774)*x^4
   +(-1922517478209553074031443296)*x^3
   +(1082757686172262515070451399916)*x^2
   +(-20327015669035171107600087000151776)*x
   +(12720241876172641933421003452296326504)
```

[5]In a next version, we plan to implement traits classes which will simplify these problems.

Here is the complete file for this section:

```
#include "upoly.H"       // univariate polynomials
#include "arithm/Zp.H"   // modular numbers
#include "bignum.H"

typedef UPolyDense<upar<Z<31> > >      Pol;
typedef UPolyDense<upar<Scl<MPZ> > >   ZPol;

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

  Pol p("x^3-34*x+23;");                      cout<<p<<endl;

  Pol q = p*p;                                cout<<q<<endl;
  p+= Pol("x+1;");  Pol r= q%p;               cout<<r<<endl;

  cout <<q(Z<31>(1))<<endl;

  ZPol a("x^2-1243432232*x+2334345435434;"); cout<<a<<endl;
  ZPol b=a^3;                                 cout<<b<<endl;
}
```

Step 4: multivariate polynomials

Multivariate monomials are defined by a type `C` for the coefficients and a type `E` for the exponents (5.1.1):

```
Monom<C,E>
```

Multivariate polynomials are represented as sorted list of monomials, defined by a container `R` and a trait class type `O` for the order on the monomials (5.2.1):

```
MPoly<R,O>
```

Let us detail their use on the following example (file `step4.cc`):

```
#include "mpoly.H"

typedef Monom<double, dynamicexp<'x'> > Mon;
typedef MPoly<list<Mon>, Dlex<Mon> >    Pol;

int main (int argc, char **argv)
{
   Pol p("x1^2*x2+2*x1+3;"), q("x1^2+x3;");
```

```
 9       cout<<p<<endl;
10       p *= q;
11       cout<<p<<endl;
12       for(Pol::iterator it=p.begin(); it !=p.end(); it++)
13         cout<<it->GetCoeff()<<" ";
14       cout<<endl;
15   }
```

1: Inclusion of the necessary header files, for the manipulation of multivariate polynomials.

3: We define an abbreviation for the data-structure, corresponding to monomials. These are monomials with `double` coefficients and `dynamicexp<'x'>` for the exponent type (see 5.1.2).

4: We define an abbreviation for the data-structure, corresponding to the polynomials. They are represented by a list of monomials, sorted by Degree Lexicographic ordering (see 5.1.3).

6: We define the `main` procedure.

8: We build the polynomial `p`, by parsing the string `"x[1]^2*x[2]+x[1]+1"`. Other techniques for constructing `p` are detailed in section 5.2.1.

10: We multiply `p` by `q`.

11: We output the new value of the variable `p`.

12: We scan the monomials of `p` from the beginning to the end.

13: And we output the coefficients of the monomials.

Once you have typed this file, you compile it with the command `alp`, run it and obtain the following result:

```
x1^2*x2+(2)*x1+(3)
x1^4*x2+x1^2*x2*x3+(2)*x1^3+(3)*x1^2+(2)*x1*x3+(3)*x3
1 1 2 3 2 3
```

Chapter 2

Basic data structures

2.1 Using the STL

We use the STL implementation, which provides the following optimized data-structures:

- list<T> generic list.
- vector<T> generic vector.
- deque<T> generic double-ended queue.
- set<T, order<T> > ordered generic set, with no repetition of elements.
- muliset<T, order<T> > ordered generic set with repetition of the elements.
- pair<I,T> generic pair of elements ((i,t).
- map<I,T, order<I> > generic set of ((i,t), ordered according to I. No repetition for the elements (key) of I.
- multimap<I,T, order<I> > generic set of ((i,t), ordered according to I. Duplicate keys are allowed.
- stack<T,C> generic lifo stack of elements in T, implemented with the container C (eg. list<T>).
- queue<T,C> generic fifo stack of elements in T, implemented with the container C (eg. deque<T>).

See the STL reference manual [54], for more details.

2.2 Complex numbers

We use the complex numbers of the standard library:

```
#include <complex>
```

Here is an example of construction of a Complex number with `double` real and imaginary parts:

```
complex<double> a(1.0,2.0)
```

2.3 Big numbers

They are based on GMP developed by T. Granlund [36], MPFR developed by P. Zimmermann [70], and MPC an extension of complex numbers by D. Bini and G. Fiorentino.

arithm/Scl.H

- `Scl<MPZ>` which corresponds to big integers, based on the GMP type `mpz`.
- `Scl<MPQ>` which corresponds to big rational, based on the GMP type `mpq`.
- `Scl<MPF>` which corresponds to big floats, based on the GMP type `mpf`.
- `Scl<MPFR>` which corresponds to reliable big floats, based on the MPFR library.
- `Scl<MPC>` which corresponds to big complexes, based on the type `mpc`.

Example

```
#include "bignum.H"

typedef Scl<MPZ> BZ;
typedef Scl<MPF> BF;
typedef Scl<MPQ> BQ;

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

  //Big floats
  BF pf("1000232325454545232322343421111111434343435555"),
     qf(25555),rf;
  cout<<"pf  = "<<pf<<endl;
```

$$pf = 0.1000232325454545232e46$$

```
cout<<"qf  = "<<qf<<endl;
```

$$qf = 0.25555e5$$

```
rf =pf*qf; cout<<"pf*qf= "<<rf<<endl;
```

$$pf * qf = 0.2556093707699090341e50$$

```
rf =pf+qf; cout<<"pf+qf= "<<rf<<endl;
```

$$pf + qf = 0.1000232325454545232e46$$

```
rf =pf-qf; cout<<"pf-qf= "<<rf<<endl;
```

$$pf - qf = 0.1000232325454545232e46$$

```
rf =pf-qf; cout<<"-qf =  "<<rf<<endl;
```

$$-qf = 0.1000232325454545232e46$$

```
rf =pf/qf; cout<<"pf/qf= "<<rf<<endl;
```

$$pf/qf = 0.3914037665640951799e41$$

```
//Big integers
BZ pz("1000232325454545232322343421111111434343435555"),
   qz(25555),rz;
cout<<"pz   = "<<pz<<endl;
```

$$pz = 1000232325454545232322343421111111434343435555$$

```
cout<<"qz   = "<<qz<<endl;
```

$$qz = 25555$$

```
rz =pz*qz; cout<<"pz*qz= "<<rz<<endl;
```

$$pz * qz = 25560937076990903411997486126494452704646495608025$$

```
rz =pz+qz; cout<<"pz+qz= "<<rz<<endl;
```

$$pz + qz = 1000232325454545232322343421111111434343461110$$

```
rz =pz-qz; cout<<"pz-qz= "<<rz<<endl;
```

$$pz - qz = 1000232325454545232322343421111111434343410000$$

```
rz =pz-qz; cout<<"-qz  = "<<rz<<endl;
```

$$-qz = 1000232325454545232322343421111111434343410000$$

```
rz =pz/qz; cout<<"pz/qz= "<<rz<<endl;
```

$$pz/qz = 391403766564095179934393825517946168790 23$$

```
rz =pz%qz; cout<<"pz\%qz= "<<rz<<endl;
```

$$pz\%qz = 2790$$

```
rz+=(pz*qz);cout<<"rz+=(pz*qz)= "<<rz<<endl;
```

$$rz+ = (pz * qz) = 25560937076990903411997486126494452704646495610815$$

```
rz-=(pz*qz);cout<<"rz-=(pz*qz)= "<<rz<<endl;
```

$$rz- = (pz * qz) = 2790$$

```
rz*=(pz*qz-qz%pz);cout<<"rz*=(pz*qz-qz\%pz)= "<<rz<<endl;
```

$$rz* = (pz * qz - qz\%pz) = 7131501444480462051947298629291952304596372267509130 0$$

```
//Big rationals
BQ pq("1000232325454545232322343421111111434343435555"),
   qq(25555),rq;

cout<<"pq/=qq   = "<<(pq/=qq)<<endl;
```

$$pq/ = qq = 20004646509090904646446868422222228686868 7111/5111$$

```
  rq=pq/qq+qq;   cout<<"pq/qq+qq= "<<rq<<endl;
```

$$pq/qq + qq = 200046465090909046464468684222225624648252886/130611605$$

```
}
```

2.4 Modular numbers

arithm/Zp.H

template <int p, class T=int> class Z

They are represented by numbers of type `T`. The computations are performed modulo `p`. The number `p` should be a prime number, otherwise inversion and exponentiation may not be valid.

Chapter 3

Linear Algebra

We describe here the main features of the linear algebra package. More information can be found in [52].

3.1 Vectors

3.1.1 Introduction

Vectors are one-dimensional arrays in which all the coefficients are stored (even if they are zero). They provide classical arithmetic operations such as addition, substraction, multiplication by scalars. ...

3.1.2 Implementation

linalg/VectStd.H

template <class R> struct VectStd

The standard class of vectors.

3.1.3 Containers for Vectors

The container type `R` should provide the following signature:

```
typedef size_type;   // type of the index.
typedef value_type;  // type of the coefficients.
typedef iterator;    // type of the iterator.

R(size_type n);
```

```
  R(size_type n, value_type* t);
  R(iterator b,iterator e);

  R& R::operator=(const R &)

  size_type    R::size()
  value_type & R::operator[] (size_type i);
  value_type   R::operator[] (size_type i) const;

  void R::reserve(size_type);
```

The constructor `R(size_type i)` allocate the memory but nothing is supposed to be known about the allocated values.

The constructor `R(iterator_t b, iterator_t e)` allocates the memory and fill it with the values from the iterator from `b` to `e`.

The constructor `R(size_type n, const char* str)` build the array of size `n` from the string `str`.

The function `void R::reserve(size_type s)` reallocate the memory if necessary and copies the elements of `this`, or use the current memory space if `s` is less that the actual space.

3.1.4 An example of containor

linalg/array1d.H

template <class C> struct array1d

Unidimensional array of coefficients with a size.

3.1.5 How to use them

```
#include "linalg.H"
  // General header file for linear algebra

typedef VectStd<array1d<double> > VECT;
  // Vector definition based on array1d container, with double coefficients.

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

  VECT U(4,"0 2 1 3"), V(4,"1 2 3 4"), W;

  cout<<"U="<<U<<", V ="<<V<<endl;
```

$$U = [0, 2, 1, 3], V = [1, 2, 3, 4]$$

```
cout<<V*2 << ", " << V/4<<endl;
```

$$([1, 2, 3, 4]).(2), ([1, 2, 3, 4])/(4)$$

```
// Yes, the operations are performed only when the result is assigned
// to a variable.

V *= 2;    cout<<V<<endl;
```

$$[2, 4, 6, 8]$$

```
V /= 4;    cout<<V<<endl;
```

$$[0.5, 1, 1.5, 2]$$

```
V += V;    cout<<V<<endl;
```

$$[1, 2, 3, 4]$$

```
U-=(V+V);  cout<<U<<endl;
```

$$[-2, -2, -5, -5]$$

```
W=U+V;     cout<<W<<endl;
```

$$[-1, 0, -2, -1]$$

```
W=V*2;     cout<<W<<endl;
```

$$[2, 4, 6, 8]$$

```
W=W*2-3*U; cout<<W<<endl;
```

$$[10, 14, 27, 31]$$

```
  W = V; V[2]= -1;  cout<<W << ", " << V<<endl;
```

$$[1, 2, 3, 4], [1, 2, -1, 4]$$

```
  W= V[Range(1,2)]; cout<<W<<endl;
```

$$[2, -1]$$

```
  // The indexes are starting from 0.
}
```

3.2 The module VECTOR

Here is an illustration of the type of code provided by a module (in this case for vectors). It corresponds to generic implementations, that could be specialized for specific datatype containors.

linalg/VECTOR.H

Generic implementations for vectors are gathered in the namespace `VECTOR`. These are generic functions, that can be used to build new linear classes. The type `R` must provide the following definitions or methods:

```
typename index_t;
typename value_type;
index_t this-> size();
value_type this-> operator[](index_t);
```

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

Output function for general vectors: `[v1, v2, ...]`.

template <class R> inline istream & VECTOR::Read(istream & is, R & V)

Input operator for standard vectors. The input format is of the form `s c0 c1 ...` *where* `s` *is the number of elements.*

template <class V, class W> void VECTOR::assign(V & a, const W & b)

Assignation of a vector to another.

template <class V, class I> void VECTOR::reserve(V & a, I n)

Reserve the space in `a` *for* `n` *entries.*

template <class R, class I, class C> void VECTOR::reserve(vector<R> & a, I n, const C & v)

Reserve the space in `a` *for* `n` *entries set to* `v`.

template <class V, class W> void VECTOR::plus(V & a, const W & b, const W & c)

Addition of two vectors.

template <class V, class W> void VECTOR::plus(V & a, const W & b)

Addition of two vectors.

template <class V, class W> void VECTOR::minus(V & a, const W & b, const W & c)

Substraction of two vectors.

template <class V, class W> void VECTOR::minus(V & a, const W & b)

Substraction of two vectors.

template <class V, class W> void VECTOR::mult(V & a, const V & b, const W & c)

Scalar multiplication.

template <class V, class W> void VECTOR::mult(V & a, const W & c)

Inplace scalar multiplication.

template <class V, class W> void VECTOR::div(V & a, const W & c)

Inplace scalar division.

template <class C, class R> C VECTOR::norm(const R & v)

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

template <class R> RealOf<typename R::value_type>::TYPE VECTOR::norm(const R & v, unsigned int n)

The L_2 *norm of the subvector vector of size* n.

template <class C,class R> C VECTOR::norm(const R & v, int p)

Norm L_p *of a vector.*

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

Innerproduct of two vectors.

3.3 Dense matrices

3.3.1 Introduction

Dense matrices are two-dimensional arrays in which all the coefficients are stored (even if they are zero). They provide classical arithmetic operations such as addition, subtraction, multiplication by scalars, by vectors, by matrices ...

3.3.2 Implementation

linalg/MatDense.H

template< class R> struct MatDense

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 through template expression. The differents operations of this class are implemented in the module* `MATRIX`.

The `value_type` is the type of the coefficients. The `size_type` is the type of the indices used to access the elements. The `row_iterator` is the type of the forward iterators on the elements of a row. The `column_iterator` is the type of the forward iterators on the elements of a column.The `row_type` is the type of the row views. The `col_type` is the type of the column views. The internal representation type `R` is `rep_type`.

3.3.3 The standard container

The default container is defined here

linalg/array2d.H

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

Class of array2d 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>`.

3.3.4 The container for the connection with Lapack

linalg/Lapack.H

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

Container for the connection with the fortran library LAPACK*. Class derived from* `array2d<C>`*, with a new* `lda_` *field. It indicates the leading dimension array, that is the number of rows in the internal allocated array. It can differ from the number of rows returned by* `nbrow()` *which is smaller than* `lda_`*. This field is used only in an internal way.*

3.3.5 How to use them

Here is an example of dense matrices, with the container `lapack`.

```
#include "linalg.H"     // General header file for linear algebra.
#include "lapack.H"     // We will use lapack containers.
#include "maple.H"      // For output in maple format.

typedef VectStd<array1d<double> > VECT;
  // Vector definition based on array1d container, with double coefficients.
typedef MatDense<lapack<double> > MAT;
  // Marix definition based on lapack container, with double coefficients.
  // This will allow us to use the routines of the lapack library.

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

  double v[]={1,2,1,1,3,4,2,2,5,5,5,5,-2, 1,2,3,1, -2.8,-2.4,1,.2,5.8};
  MAT A(4,4,v,ByRow()),
      B(4,4,"1 2 1 1 3 4 2 2 5 5 5 5 -2  1 2 3 1 .2 5.8", ByCol());

  cout<<"A="<<A<<", B="<<B<<endl;
```

$$A = \begin{bmatrix} 1 & 2 & 1 & 1 \\ 3 & 4 & 2 & 2 \\ 5 & 5 & 5 & 5 \\ -2 & 1 & 2 & 3 \end{bmatrix}, B = \begin{bmatrix} 1 & 3 & 5 & -2 \\ 2 & 4 & 5 & 1 \\ 1 & 2 & 5 & 2 \\ 1 & 2 & 5 & 3 \end{bmatrix}$$

```
  // Arithmetic operations.
  MAT C;
  C= A+B; cout<<C<<endl;
```

$$\begin{bmatrix} 2 & 5 & 6 & -1 \\ 5 & 8 & 7 & 3 \\ 6 & 7 & 10 & 7 \\ -1 & 3 & 7 & 6 \end{bmatrix}$$

```
  C= A-B; cout<<C<<endl;
```

$$\begin{bmatrix} 0 & -1 & -4 & 3 \\ 1 & 0 & -3 & 1 \\ 4 & 3 & 0 & 3 \\ -3 & -1 & -3 & 0 \end{bmatrix}$$

```
  C= A*B; cout<<C<<endl;
```

$$\begin{bmatrix} 7 & 15 & 25 & 5 \\ 15 & 33 & 55 & 8 \\ 25 & 55 & 100 & 20 \\ 5 & 8 & 20 & 18 \end{bmatrix}$$

```
C= A*B*A; cout<<C<<endl;
```

$$\begin{bmatrix} 167 & 204 & 172 & 177 \\ 373 & 445 & 372 & 380 \\ 650 & 790 & 675 & 695 \\ 93 & 160 & 157 & 175 \end{bmatrix}$$

```
C= 2*A; cout<<C<<endl;
```

$$\begin{bmatrix} 2 & 4 & 2 & 2 \\ 6 & 8 & 4 & 4 \\ 10 & 10 & 10 & 10 \\ -4 & 2 & 4 & 6 \end{bmatrix}$$

```
// Construction of a Vector from to iterators.
VECT V(v+1,v+5);  cout<<V<<endl;
```

$$[2, 1, 1, 3]$$

```
VECT W(A*V);      cout<<W<<endl;
```

$$[8, 18, 35, 8]$$

```
//Extractions of sub matrices of vectors:
W = Col(A,1);                  cout<<"Col(A,1) = "<<W<<endl;
```

$$Col(A, 1) = [2, 4, 5, 1]$$

```
W = Row(A,2);                  cout<<"Row(A,2) = "<<W<<endl;
```

$$Row(A, 2) = [5, 5, 5, 5]$$

```
B = A(Range(1,3),Range(1,3));  cout<<"A(1..3,1..3) = "<<B <<endl;
```

$$A(1..3, 1..3) = \begin{bmatrix} 4 & 2 & 2 \\ 5 & 5 & 5 \\ 1 & 2 & 3 \end{bmatrix}$$

```
B = Row(A,Range(1,2));          cout<<B<<endl;
```

$$\begin{bmatrix} 3 & 4 & 2 & 2 \\ 5 & 5 & 5 & 5 \end{bmatrix}$$

```
B = Col(A,Range(1,2));          cout<<B<<endl;
```

$$\begin{bmatrix} 2 & 1 \\ 4 & 2 \\ 5 & 5 \\ 1 & 2 \end{bmatrix}$$

```
//Addition of the row 2 times 1.5 into row 1
A.addRow(1,2,1.5);              cout<<A<<endl;
```

$$\begin{bmatrix} 1 & 2 & 1 & 1 \\ 10.5 & 11.5 & 9.5 & 9.5 \\ 5 & 5 & 5 & 5 \\ -2 & 1 & 2 & 3 \end{bmatrix}$$

```
A.addCol(1,2,-0.5);             cout<<A<<endl;
```

$$\begin{bmatrix} 1 & 1.5 & 1 & 1 \\ 10.5 & 6.75 & 9.5 & 9.5 \\ 5 & 2.5 & 5 & 5 \\ -2 & 0 & 2 & 3 \end{bmatrix}$$

```
// Output to the maple format.
cmpl <<"A  := "<<A<<endl;
//A  := linalg[matrix](4,4, [[1,1.5,1,1], [10.5,6.75,9.5,9.5], [5,2.5,5,5], [-2,0,2,3]]);


};
```

3.4 Structured matrices

3.4.1 Introduction

Structured matrices are two-dimensional arrays which can be represented by $O(n)$ values, where n is the size of the matrix. This class provides classical arithmetic operations such as addition, substraction, multiplication by scalars, vectors, matrices ... The product of two structured matrices is usually not a structured matrices (at least of the same type) and therefore is not implemented. The product of a structured matrices by a vector can usually be performed in almost linear time, ie. in $O(n\log(n))$ ops (or $\mathcal{O}(n\log(n)\log^2(n)$ according to the field), using for instance FFT. See (3.6.1) and [12].

3.4.2 Implementation

linalg/MatStruct.H

template< class R> struct MatStruct

Structured matrices.

3.4.3 A container for Toeplitz matrices

A matrix `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 $t_{i,j} = t_{i+1,j+1}$ 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 a univariate polynomial $\in \mathbb{K}[x]$ (see [51]). An interesting point of this definition is that the product of a $n\times n$ Toeplitz matrix by a vector is a subvector of the coefficient vector of the product of two polynomials of $\mathbb{K}[\mathbf{x}]$. This product can be done within $\mathcal{O}(n\log(n))$ (or $\mathcal{O}(n\log(n)\log^2(n)$ according to the field) operations using **FFT**. See (3.6.1) and [12].

linalg/toeplitz.H

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

Containers for Toeplitz matrices, as a subclass of `array1d`*. Defined by its number of row* `nbrow_`*, its number of columns* `nbcol_`*. The elements of the first row and first column are stored, starting from* `(0,nbcol)` *up to* `(nbrow,0)`*. The product of a Toeplitz matrix by a vector is performed by FFT computations (3.6.1).*

3.4.4 A container for Hankel matrices

A matrix `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 $\mathbb{K}[x,x^{-1}]$, that is a polynomial in both the variables x and x^{-1}). So we can compute the product of a $n\times n$ Hankel matrix by a vector as a subvector of the product of a fixed polynomial $h(x)$ by a polynomial in x^{-1}. This can also be done within $\mathcal{O}(n\log(n))$ ops (or $\mathcal{O}(n\log(n)\log^2(n)$ according to the field), using for instance FFT. See (3.6.1) and [12].

linalg/hankel.H

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

Containers for Hankel matrices, as a subclass of `array1d`. *Defined by its number of row* `nbrow_`, *its number of columns* `nbcol_`. *The elements of the first row and last column are stored, starting from* `(0,0)` *up to* `(nbrow_,nbcol_)`. *The product of a Hankel matrix by a vector is performed by FFT computations (3.6.1). .*

3.4.5 How to use them

```
#include "linalg.H"
#include "linalg/toeplitz.H"
#include "linalg/hankel.H"

typedef VectStd<array1d<double> > VECT;
typedef hankel<double> rep; //Definition of the container.
typedef MatStruct<rep> MAT; //Definition of Hankel matrices.

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


  double v[]={1,2,1,1,3,4,2,2,5,5,5,5,-2, 1,2,3,1, -2.8,-2.4,1,.2,5.8};
  MAT A(4,4,v), B(A);
  B.transpose();

  cout<<"A  := "<<A<<", B   = "<<B<<endl;
```

$$A := \begin{bmatrix} 1 & 2 & 1 & 1 \\ 2 & 1 & 1 & 3 \\ 1 & 1 & 3 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}, B = \begin{bmatrix} 1 & 2 & 1 & 1 \\ 2 & 1 & 1 & 3 \\ 1 & 1 & 3 & 4 \\ 1 & 3 & 4 & 2 \end{bmatrix}$$

```
  MAT C;
  C= A+B; cout<<C<<endl;
```

$$\begin{bmatrix} 2 & 4 & 2 & 2 \\ 4 & 2 & 2 & 6 \\ 2 & 2 & 6 & 8 \\ 2 & 6 & 8 & 4 \end{bmatrix}$$

```
  C= A-B; cout<<C<<endl;
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
C= A*2; cout<<C<<endl;
```

$$\begin{bmatrix} 2 & 4 & 2 & 2 \\ 4 & 2 & 2 & 6 \\ 2 & 2 & 6 & 8 \\ 2 & 6 & 8 & 4 \end{bmatrix}$$

```
VECT V(v+1,v+5); cout<<V<<endl;
```

$$[2, 1, 1, 3]$$

```
//Multiplication using FFT.
VECT W(A*V);     cout<<"W(A*V)   = "<<W<<endl;
```

$$W(A * V) = [8, 15, 18, 15]$$

```
//Submatrices
W = Col(A,1);  cout<<W<<endl;
```

$$[2, 1, 1, 3]$$

```
W = Row(A,2);  cout<<W<<endl;
```

$$[1, 1, 3, 4]$$

```
B = A(Range(1,3),Range(1,3));  cout<<"A(1..3,1..3) = "<<B <<endl;
```

$$A(1..3, 1..3) = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 3 & 4 \\ 3 & 4 & 2 \end{bmatrix}$$

```
B = Row(A,Range(1,2));  cout<<B<<endl;
```

$$\begin{bmatrix} 2 & 1 & 1 & 3 \\ 1 & 1 & 3 & 4 \end{bmatrix}$$

```
  B = Col(A,Range(1,2));  cout<<B<<endl;
```

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \\ 1 & 3 \\ 3 & 4 \end{bmatrix}$$

```
};
```

3.5 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 [61]). In such applications it is not rare to treat (sparse) matrices of size $10^4 \times 10^4$ or even $10^5 \times 10^5$, which could not be managed as dense matrices.

3.5.1 Implementation

`linalg/MatSparse.H`

template< class R> struct MatSparse

Sparse matrices.

3.5.2 A standard container for sparse matrices

`linalg/sparse2d.H`

template< class C> struct sparse2d: public array1d< C>

3.5.3 Container for the connection with SuperLU

`linalg/superlu.H`

template< class C> struct superlu : public SuperMatrix

Container for the SuperLU package [23], which will be called to solve a linear system by LU-factorisation, as illustrated in the following example:

```
MatSparse<superlu<double> >A(...);
VectStd<array1d<double> >V(...), W;
W=Solve(A,V,LU());
```

3.6 The linear algebra package

3.6.1 Fast Fourier Transform

linalg/FFT.H

template< class T> inline void Root_Unity(complex< T> &c, int n)

Root of Unity. Replace `c` *by the primitive root of order* `n`.

inline unsigned int BitReverse(int k, int n)

Functions which permute an array by Bit Reversal method.

template <class T> inline void Permut(T *Tab, int N)

Shuflle on N entries.

template <class C, class U> void FFT(int n, const complex< U> & w, complex< C> *W, int inverse)

Implementation of `FFT` *computation of a complex array. It is an "in-place" transformation, based on the Butterfly algorithm. A complex array W, of size* `n` *(a power of 2), is replaced by the FFT of its values. The complex number* `w` *is a primitive root of unity of order n.*

3.6.2 Triangular decomposition

linalg/Triang.H

template< class A, class R> VAL< R*> Triang(const R & mat)

Triangulation of the matrix `m`. *The first argument of template specifies the type of triangulation which is applied. It should called as follows* `Triang<Method>(mat);`. *The function* `LINALG::triang(Method(), ...)` *is called on a copy of* `mat`. *The different methods available are* `Gauss, QR, Bareiss,...` *Example:*

```
T = Triang<Gauss>(mat);
```

template <class M> inline VAL< M*> USchur(M & m, int n)

Compute the upper schur complement of size `n` *of the matrix* `mt`, *given by the formula $A - B\, D^{-1}\, C$ where A is the first diagonal submatrix block of size* `n`. ***USchur***

3.6.3 Solving linear systems

linalg/Solve.H

template <class M, class R> inline VAL<VectStd<R>*> Solve(M & A, const VectStd<R> & b, LU mth)

Resolution by LU decomposition. Calling the function `LINALG::solve_lu`. *`Solve, LU`*

template <class M, class R> inline VAL<R*> Solve(M & A, R & B, LU mth)

Resolution of `A X = B` *by LU decomposition. Calling the function* `LINALG::solve_lu(X.rep,A.rep)`, *where* `X` *is a copy of* `B`. *`Solve, LU`*

template <class M, class V> inline VAL<V*> Solve(M & A, V & b, LS mth)

Resolution by LeastSquare method. Calling the function `LINALG::solve_ls(X.rep,A.rep,b.rep)` *where* `X` *is a copy of A.* *`Solve, LS`*

3.6.4 Determinants

Different methods for computing the determinant of a matrix are discribed here: Gauss, Bareiss or Berkowitz methods.

- Gauss method uses classical pivoting techniques (ie. LU decomposition) to compute the determinant.
- Bareiss method applies the following pivoting scheme:

$$a_{i,j}^{(k)} = \frac{a_{i,j}^{(k-1)} a_{k,k}^{(k-1)} - a_{i,k}^{(k-1)} a_{k,j}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}} \tag{3.1}$$

  where $A^{(k)} = (a_{i,j}^{(k)})$ is the transformed matrix at step k and $A^{(0)}$ is the initial matrix. At the end of this process, after some permutation of the rows, the matrix is upper triangular. If A is a square matrix, the last entry $A_{n,n}^{(n)}$ on the diagonal at the end of the process is the determinant of the matrix A. Moreover, if A has its coefficients in a ring, it is the same for the matrices $A^{(k)}$, since the division in (3.1) is exact.
- Berkowitz method is based on the following recursion formula, based on Cayley-Hamilton identity:

$$d_0 := Trace(A);\ A_0 := A;\ A_{k+1} := A_k * A - \frac{1}{k+1} Trace(A^k)\, A$$

- A mixed strategy, which expands the determinant with respect to the first row and Bareiss method is applied to the subdeterminants.

linalg/Det.H

template <class R> typename R::value_type Det(R & M)

Compute the determinant of `M`*, using Bareiss method. Example:*

```
typedef MatDense< array2d< double> > MAT;
MAT M(3,3,"1 2 3 4 5 6 7 8 9", ByRow());
double det;
det=Det(M);
```

template <class R> typename R::value_type Det(const Mixed & mth, R & M)

Compute the determinant of `M` *by mixing two methods: A pseudo-triangulation is first computed. If the size of* `M` *is less or equal to 4, the determinant is computed by expansion along the columns. Else a development is done along the first row with triangulation by Bareiss method of the submatrix corresponding to each column that is removed. Example:*

```
typedef MatDense< array2d< double> > MAT;
MAT M(3,3, "1 2 3 4 5 6 7 8 9",ByRow());
double det=Det(Mixed(),M);
```

template<class R> typename R::value_type Det(const Berko & mth, const R & M)

Compute the determinant of `M` *by Berkowitz method. Example:*

```
typedef MatDense< array2d< double> > MAT;
MAT M(3,3,"1 2 3 4 5 6 7 8 9");
double det=Det(Berko(), M);
```

3.6.5 Rank

`linalg/Rank.H`

template<class R> inline typename R::size_type Rank(const R & M)

Compute the rank of `M`*, using* `LINALG::rank`*.*

template<class R,class V> typename R::size_type Rank(const R & M, V & I, V & J)

Compute the rank of `M`*, using* `LINALG::rank`*. The columns (resp. row) indices of a permutation which put the non-zero minor in the upper corner are stored in* `permut_col` *and* `permut_row`*. They have to be initialized before, to a size larger than* `min(M.nbcol(), M.nbrow())`*.*

template<class R,class I> VAL<R*> Extract(R & M, const I & ipvt, const I & jpvt, const typename R::size_type & rank)

Extraction of a square submatrix of size rank from the matrix `M`*. This is performed by keeping the columns whose index is one of the first* `rank` *elements of* `jpvt`*, and the rows whose index is one of the first rank elements of* `ipvt`*, in the same orders.*

3.6.6 Eigenvectors and eigenvalues

linalg/Eigen.H

Classical eigenproblem.

```
template <class M> inline VAL<VectStd<array1d<AlgClos<typename M::value_type>::TYPE > >*> Eigenval(M & A)
```

Compute the eigenvalues of A.

```
template <class M> inline VAL<VectStd<array1d<typename M::value_type> >*> Eigenval(M & A, Real)
```

Compute the real eigenvalues of A.

```
template <class M> inline VAL<VectStd<array1d<AlgClos<typename M::value_type>::TYPE> >*> Eigenval(M & A, VectStd<array1d<typename M::value_type> > & Er)
```

Compute the eigenvalues of A and the error `Er` *on these eigenvalues.*

```
template <class M> inline VAL<MatDense<lapack<AlgClos<typename M::value_type>::TYPE> >*> Eigenvct(M & A)
```

Compute the eigenvectors of A.

```
template <class M> inline VAL<MatDense<lapack<typename M::value_type> >*> Eigenvct(M & A, Real)
```

Compute the real eigenvectors of A.

Generalized eigenproblem.

```
template <class M> inline VAL<VectStd<array1d<AlgClos<typename M::value_type>::TYPE> >*> Eigenval(const M & A, const M & B)
```

Compute the eigenvalues of (A,B).

```
template <class M> inline VAL<VectStd<array1d<typename M::value_type> >*> Eigenval(const M & A, const M & B, Real real)
```

Compute the real eigenvalues of (A,B).

```
template <class M,class V> inline VAL<MatDense<lapack<typename M::value_type> > *> Eigenvct(MatDense<M> & A, MatDense<M> & B, Real r, VectStd<V> & v)
```

Compute the real eigenvectors of `(A,B)` *and the eigenvalues* `V`.

```
template <class M> inline VAL<MatDense<lapack<typename M::value_type> >*> Eigenvct(MatDense<M> & A, MatDense<M> & B, Real real, const typename M::size_type & m)
```

Compute the real eigenvectors of `(A,B)`. *The first* `m` *coordinates of these eigenvectors are returned.*

3.6.7 Exemple

```
#include "linalg.H"     // General header file for linear algebra.
#include "lapack.H"     // We will use lapack containers.

typedef VectStd<array1d<double> > VECT;
  // Vector definition.
typedef VectStd<array1d<complex<double> > > VECTC;
  // The same with complex coefficients.
typedef MatDense<lapack<double> > MAT;
  // Marix definition.
typedef MatDense<lapack<complex<double> > > MATC;
  // The same with complex coefficients.

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

  double v[]={1,-3,1,2,3,4,2,2,4,5,5,5,5,-2, 1,2,3,1, -2.8,-2.4,1,.2,5.8};
  MAT A(4,4,v, ByRow()),
      B(4,4,"1 2 1 1 3 4 2 2 5 5 5 5 -2  1 2 3 1  .2 5.8", ByCol());

  cout<<"A="<<A<<", B="<<B<<endl;
```

$$A = \begin{bmatrix} 1 & -3 & 1 & 2 \\ 3 & 4 & 2 & 2 \\ 4 & 5 & 5 & 5 \\ 5 & -2 & 1 & 2 \end{bmatrix}, B = \begin{bmatrix} 1 & 3 & 5 & -2 \\ 2 & 4 & 5 & 1 \\ 1 & 2 & 5 & 2 \\ 1 & 2 & 5 & 3 \end{bmatrix}$$

```
  VECT V(v+1,v+5);  cout<<V<<endl;
```

$$[-3, 1, 2, 3]$$

```
  VECT W(A*V);      cout<<W<<endl;
```

$$[2, 5, 18, -9]$$

```
  W =Svd(A);        cout<<"Svd(A)   = "<<W<<endl;
```

$$Svd(A) = [11.4091, 6.03278, 2.53035, 0.189481]$$

```
  cout<<"Rank(A)  = "<<Rank(A)<<endl;
```

$$Rank(A) = 4$$

```
//Triangulation by Gauss method.
B = Triang<Gauss>(A);  cout<<B<<endl;
```

$$\begin{bmatrix} 5 & -2 & 1 & 2 \\ 0 & 6.6 & 4.2 & 3.4 \\ 0 & 0 & 2.45455 & 2.93939 \\ 0 & 0 & 0 & 0.407407 \end{bmatrix}$$

```
//Determinant computed by Bareiss method.
cout<<"Det      = "<<Det(A)<<endl;
```

$$BareissDet = -33$$

```
//Eigenvalue and eigenvector computations.
VECTC L=Eigenval(A); cout<<L<<endl;
```

$$[(-1.21002, 0), (6.26876, 1.12049), (6.26876, -1.12049), (0.672512, 0)]$$

```
MATC  E=Eigenvct(A); cout<<E<<endl;
```

$$\begin{bmatrix} 0.377172 & -0.132449 + \mathbf{i} * (0.0803325) & -0.132449 + \mathbf{i} * (-0.0803325) & -0.0578894 \\ -0.09358 & 0.457738 + \mathbf{i} * (-0.0040333) & 0.457738 + \mathbf{i} * (0.0040333) & -0.105242 \\ 0.470337 & 0.855002 & 0.855002 & 0.820628 \\ -0.792318 & -0.134821 + \mathbf{i} * (0.131372) & -0.134821 + \mathbf{i} * (-0.131372) & -0.558698 \end{bmatrix}$$

```
};
```

Chapter 4

Univariate Polynomial

4.1 Dense univariate polynomials

A dense univariate polynomial $f = f_0 + f_1\, x + \cdots + f_d\, x^d$ is represented by an array of coefficients $[f_0, \ldots, f_d]$. All its coefficients are stored. It is assumed here that the coefficient ring has the basic arithmetic operations.

4.1.1 Implementation

upoly/UPolyDense.H

template< class R> struct UPolyDense

Univariate polynomials, with a dense representation given by the container `R`*. Forward and backward iterators are provided, as well as the arithmetic operations* `+,-,+=,-=,*,*=,/, /=, % ,%=`*. The operators* `%` *and* `/` *correspond respectively to the quotient and remainder computation in the Euclidean division. The* `operator()(const C &)` *corresponds to the evaluation of the polynomial.*
The function `UPOLYNOMIAL::checkdegree` *is called at the end of an arithmetic operation, to reajust the variable* `degree_`*, if it exists in the container* `R`*.*
UPolyDense

4.1.2 Containers for univariate polynomials

The container `R` should provide the following signatures:

```
typename R::value_type coeff_t;
typedef typename R::size_type;
typedef typename R::iterator;
```

```
R(size_type s, Alloc);
R(size_type s, C *t);
R(iterator b, iterator e);
void R::reserve(size_type n);
iterator  R::begin()       ;
iterator  R::begin() const ;
iterator  R::end()         ;
iterator  R::end()   const ;
iterator  R::rbegin()      ;
iterator  R::rbegin() const;
iterator  R::rend()        ;
iterator  R::rend()   const;
size_type R::size();
int  Degree(const R &);
```

The container `upar` (4.1.3) and `vector` of the STL can be used here.

4.1.3 Example of container

upoly/upar.H

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

`upar<C>`, *univariate polynomials representation, as a subclass of* `array1d<C>`.

4.1.4 How to use them

```
cout<<"p0 ="<<p0<<", p1 ="<<p1<<endl;
```

$$p0 = (1) * x^2 + (1) * x + (1), p1 = (1) * x^3 + (-1) * x^2 + (2)$$

```
cout<<"p0((0,1)) ="<<p0(complex<double>(0,1))<<endl;
```

$$p0((0,1)) = -1$$

```
cout<<"p1(1)     ="<<UPOLYNOMIAL::eval(p1.rep,1.)<<endl;
```

$$p1(1) = 2$$

```
Pol p2(p0+p1);        cout<<p2<<endl;
```

$$(1)*x^3+(1)*x+(3)$$

```
p2 =p0-p1;            cout<<p2<<endl;
```

$$(-1)*x^3+(2)*x^2+(1)*x+(-1)$$

```
p2=p0*p1;             cout<<p2<<endl;
```

$$(1)*x^5+(1)*x^2+(2)*x+(2)$$

```
p2+=p1-p0;            cout<<p2<<endl;
```

$$(1)*x^5+(1)*x^3+(-1)*x^2+(1)*x+(3)$$

```
p2=p1; p2*=p0;        cout<<p2<<endl;
```

$$(1)*x^5+(1)*x^2+(2)*x+(2)$$

```
p2+=(p1-p0)*(p1-p0*2);cout<<p2<<endl;
```

$$(1)*x^6+(-4)*x^5+(3)*x^4+(8)*x^3+(2)$$

```
p2=p0-(p0+p1);        cout<<p2<<endl;
```

$$(-1)*x^3+(1)*x^2+(-2)$$

```
p2=p0 +p1*2;          cout<<p2<<endl;
```

$$(2)*x^3+(-1)*x^2+(1)*x+(5)$$

```
p2=(p1+p0)*(p1-p0*2); cout<<p2<<endl;
```

$$(1)*x^6+(-3)*x^5+(-1)*x^4+(-11)*x^2+(-6)*x$$

```
//Quotient in the Euclidean division
p2=p1/p0;             cout<<p2<<endl;
```

$$(1)*x+(-2)$$

```
//Remainder in the Euclidean division
p2=p1%p0;              cout<<p2<<endl;
```

$$(1) * x + (4)$$

```
p1%=p0;                cout<<p1<<endl;
```

$$(1) * x + (4)$$

```
}
```

4.2 Quotient of univariate polynomials

This class represents dense univariate polynomial modulo a fixed polynomial p, that is elements in $\mathcal{A} = \mathbb{K}[x]/(p)$. If d is the degree of p, an element is represented by a vector of size d, corresponding to its coefficients in the Horner basis:

$$f = f_0\, H_0(x) + f_1\, H_1(x) + \cdots + f_{d-1}\, H_{d-1}(x).$$

where

$$H_i(x) = \pi_+(x^{-d+i}\, p)$$

is the i^{th} Horner polynomial of degree i. In this basis, the square of a polynomial can be performed with two convolutions (eg. based on FFT (3.6.1)). See [18] for more details.

4.2.1 Implementation

upoly/UPolyQuot.H

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

Class of polynomials modulo a polynomial `UPolyQuot<R>::poly`. *Elements are represented in the Horner basis. All the arithmetic operations are performed in this basis. Only the display of a polynomial needs a change to the monomial basis.*

4.2.2 Containers for quotiented polynomials

The container `R` should provide the same signatures as for dense univariate polynomials. It can be `upar` (4.1.3) for instance. The container `vector` of the stl can also be used.

4.2.3 How to use them

```
#include "upoly.H"

typedef double C;
typedef UPolyQuot<upar<C> > Pol;

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

  // We define the quotient by the following polynomial
  C u[]={-3,-2,-1,1};
  Pol::poly= upar<C>(4,u);
  cout<<UPolyDense<upar<C> >(Pol::poly)<<endl;
```

$$(1) * x^3 + (-1) * x^2 + (-2) * x + (-3)$$

```
  // Here are the polynomials
  C u1[]={0,0,1,0,0};
  Pol p0(u1,u1+3), p1(u1+1,u1+4), p2;

  cout<<"p0 ="<<p0<<endl;
```

$$p0 = (1) * x^2 + (-1) * x + (-2)$$

```
  cout<<"p1 ="<<p1<<endl;
```

$$p1 = (1) * x + (-1)$$

```
  p2 = p0+p1; cout<<p2<<endl;
```

$$(1) * x^2 + (-3)$$

```
  p2 = p0*p0; cout<<p2<<endl;
```

$$(-2) * x^2 + (5) * x + (1)$$

```
  p2 = p0*p1; cout<<p2<<endl;
```

$$(-1) * x^2 + (1) * x + (5)$$

```
}
```

4.3 Solving univariate polynomials

4.3.1 Aberth method

This is a connection to the C-implementation by D. Bini and G. Fiorentino for solving univariate polynomials. See [10] for more details.

upoly/solve/Aberth.H

template< class T> VAL< VectStd< array1d< Scl< MPC> > >*> Solve(const T &t, const Aberth & A)

Solve the univariate polynomial of type T, by the Aberth method. The complex roots are output in a vector of `Scl<MPC>` *complex numbers.*
The default constructor of the class `Aberth` *has to be used when we assume no error on the input coefficients and we want the default precision (*`2*DBL_DIG` *digits) on the output roots. The constructor* `Aberth(i)` *is used when we assume an input precision of* `i` *digits and the default output precision for the approximation of the roots. The constructor* `Aberth(i,o)` *is used when we assume an input precision of* `i` *digits and we want an output precision of* `o` *digits on the roots.*

Example:

```
#include <string.h>
#include "upoly.H"
#include "bignum.H"
#include "util/Clock.H"
#include "upoly/solve.H"

typedef Scl<MPZ> C;
typedef Scl<MPC> CC;
typedef UPolyDense<upar<C> > upoly;
typedef VectStd<array1d<CC> > Vect;
int main (){

  upoly q; cin >>q; cout<<q<<endl;
```

$$\begin{aligned}
&(1) * x^{30} + (-465) * x^{29} + (103385) * x^{28}\\
&+(-14631225) * x^{27} + (1480321269) * x^{26}\\
&+(-114009431445) * x^{25} + (6949189247325) * x^{24}\\
&+(-344092707928125) * x^{23} + (14097793282984515) * x^{22}\\
&+(-484338676679532675) * x^{21} + (14090257524223082475) * x^{20}\\
&+(-349600545868057540875) * x^{19} + (7435941626111727234855) * x^{18}\\
&+(-136055808711963322871175) * x^{17} + (2145883249334501452139775) * x^{16}\\
&+(-29197210605623737977801375) * x^{15} + (342563613932937660652700640) * x^{14}\\
&+(-3460266110493898677911394000) * x^{13} + (30006513636556697864066736800) * x^{12}\\
&+(-222457423246962063058403076000) * x^{11} + (1401937624086807501691142239744) * x^{10}\\
&+(-7454161471690660700139655157760) * x^{9} + (33114629767614997850763390570240) * x^{8}\\
&+(-121365366674745136523074652102400) * x^{7} + (360930788158836812805614538878976) * x^{6}\\
&+(-851899888505423112503184251412480) * x^{5} + (1547794975254719737111781253120000) * x^{4}\\
&+(-2070792202024594683660866641920000) * x^{3} + (1902893785240928209998216560640000) * x^{2}\\
&+(-1059681761389533859949327155200000) * x + (265252859812191058636308480000000)
\end{aligned}$$

```
  Clock c;  c.start();
  Vect r =Solve(q, Aberth());

  cout<<"Roots:"<<endl;
  for (int i = 0; i < q.degree(); i++) cout<<"#"<<i+1<<": "<<r[i]<<endl;
}
Roots:
#1: (0.1e1, -0.2587701782179289593134832644809658102 6e-61)
#2: (0.1999999999999999999999999999999999984e1, -0.4655668099047146989121409994203166998 7e-76)
#3: (0.4000000000000000000000000000000364749e1, -0.2053588880106791405450432425815111975 9e-62)
#4: (0.2999999999999999999999999999999998682e1, -0.1690014420565147123865015402332071313e-67)
#5: (0.6000000000000000000000000010308179885e1, -0.1706934445611403858249406353160190079 7e-55)
#6: (0.4999999999999999999999999999761733257e1, 0.5611733654493964377194479967852111609 6e-47)
#7: (0.9999999999999999999999978482696323121 2e1, 0.1677037890900643849669321014094637549e-45)
#8: (0.9000000000000000000000003251998595299 1e1, -0.1050909587512575036338572745521223566 6e-47)
#9: (0.1400000000000000000000084025537027538 28e2, 0.2015435089101158787266077152350015994e-45)
#10: (0.6999999999999999999999999991601077172e1, 0.2539098035463788120807462118988236530 7e-37)
#11: (0.1099999999999999999999988532508203455 8e2, -0.5723510095298753906316094722370261252 9e-40)
#12: (0.7999999999999999999999999943143321769 9e1, -0.1946737550048761708201723963372232392 7e-41)
#13: (0.1299999999999999999999963478192535910 34e2, 0.3635391788056614475132696584392314993 5e-48)
#14: (0.1200000000000000000000009707954139115 16e2, 0.1046543376615059783234815611049713609 5e-28)
#15: (0.1600000000000000000000121692969865195 616e2, 0.6128240095605522619366513385760703913 9e-44)
#16: (0.1699999999999999999999973219365009465 1459e2, 0.2309095887500468055812263958690829855e-38)
#17: (0.1799999999999999999999968238309659498 7337e2, 0.1681261609471063016278307244227484241 6e-30)
#18: (0.1500000000000000000000021899514173736 43e2, -0.7490939520500240247661306054365703158e-32)
#19: (0.1999999999999999999999929658687614171 1257e2, 0.8517603524887230547265355986117284082 4e-31)
#20: (0.1899999999999999999999952978828117844 1923e2, 0.5328214420713721315495917226495627331 2e-40)
#21: (0.2199999999999999999999929734800697816 3749e2, 0.3694342413384924565273333687905897800 1e-37)
#22: (0.2299999999999999999999928791497583120 7769e2, 0.2973274119173276971088778393518350330 1e-41)
#23: (0.2599999999999999999999983204532100663 1775e2, 0.2058034301084154126361739346269656541 4e-34)
#24: (0.2099999999999999999999986316013796828 98985e2, 0.8541649542379073113036814406378440837 2e-38)
#25: (0.2899999999999999999999995118582495120 79e2, 0.5877547835601912146537799999723852364 6e-29)
#26: (0.2500000000000000000000018252201658343 1122e2, 0.1702818085845692419455710925989157286e-36)
#27: (0.2800000000000000000000010977922438105 99e2, -0.2792228857483794425653906143850768552 8e-31)
#28: (0.2400000000000000000000051274372584682 1381e2, -0.1835942875556097466567822357481980858 3e-30)
#29: (0.2700000000000000000000036044721387544 108e2, 0.4960968173627055722752075804709944549 6e-31)
#30: (0.2999999999999999999999999913709517042 125e2, 0.8297490426240627975026037961445853149 5e-31)
```

4.3.2 Sebastio a Sylva method

The **Sebastiao e Silva method** (see [18]) is an iterative process for computing a selected approximate root of a polynomial p. It is essentially the same as the **power method** of matrices, but it is accelerated: instead of computing the successive powers f^n, we compute the successive squares f^{2^n}.

4.3.1. **An iterative process based on the Sebastiao e Silva method**

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

If there exists a simple root ζ maximizing $|f|$, we obtain

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

Here, we have choosen the euclidian norm for the vectors. The process stops if the error between two iterations is 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 to select the root which is the 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 ζ closest to t. We start the algorithm from the vector $f_t = (1, t, t^2, \ldots, t^{d-1})$, that is the coefficient vector in the Horner basis of $\frac{p(x) - p(t)}{(x-t)p'(t)}$. We have $f_t(\zeta) = \frac{-p(t)}{\zeta - t}$ for all the roots of p, so that the evaluation of f_t at the roots of p is maximal when $|\zeta - t|$ is minimal. If we want the smallest root of p, we start the process from f_0. If we want the largest root, we should reverse p and look for its smallest root.

If we want to obtain other roots after computing a first root ζ, we apply a "deflation" of the polynomial p. There are two different deflation processes, the first one is the most natural; it is called the explicit deflation, and consists in computing explicitly $\frac{p(x)}{x-\zeta}$. The second deflation process, called implicit deflation, is as follows. One root ζ is computed with a starting value f_0, and the method produced a first idempotent $\mathbf{e}_\zeta$. To get a second root, we choose any g of $\mathcal{A}$ (possibly f again) and start the method from the initial value $(g - g\mathbf{e}_\zeta)$. Since $(g - g\,\mathbf{e}_\zeta)(\zeta) = 0$, ζ is no longer dominant, and the process cannot converge again to $\mathbf{e}_\zeta$. Thus we will obtain another idempotent $\mathbf{e}_{\zeta'}$ where ζ' is the dominant root associated to g, ζ will be ignored. For the next step, we choose h (possibly f or g) and start the method from the initial value $h - h(\mathbf{e}_\zeta + \mathbf{e}_{\zeta'})$. Continuing in this way, we will ultimately collect as many roots of p as desired (for more details see [18]).

For this work we have choosen the implicit deflation.

Implementation

upoly/solve/Sebastio.H

struct Sebastiao

Class which gives the parameters of the Sebastiao method: `nbroot` *number of roots that should be computed,* `nbiter` *maximal number of iterations allowed,* `epsilon` *tolerance to stop the iteration.*

```
{
  int nbroot;           //number of roots.
  unsigned nbiter;      //number of iterations.
  double   epsilon;     //tolerance.

  Sebastiao():nbroot(1),nbiter(40),epsilon(Tolerance){}
  Sebastiao(const Sebastiao & m):
    nbroot(m.nbroot),nbiter(m.nbiter),epsilon(m.epsilon){}
  Sebastiao(int n):
    nbroot(n),nbiter(40),epsilon(Tolerance) {}
  Sebastiao(int n, unsigned nit):
    nbroot(n),nbiter(nit),epsilon(Tolerance) {}
  Sebastiao(int n, unsigned nit, double eps):
    nbroot(n),nbiter(nit),epsilon(eps) {}
};
```

template <class R> inline VAL<UPolyQuot<R> *> StartingValue(UPolyDense<R> p, const typename R::value_type x0)

Returns $B_{1,p}(x, x_0)$ in the dual basis. Used to get a starting value of the iterative process.

template< class R> VAL< UPolyQuot< R> *> IterFrom(const UPolyQuot< R> & p0, Sebastiao mth)

Apply Sebastiao e Silva iteration from the starting polynomial `p0`. *It stops either if the norm of the distance between two iterations is smaller than* `mth.epsilon`, *or if the number of iterations goes up to* `mth.nbiter`.

template < class R> inline typename R::value_type SolveOne(UPolyDense< R> & p, const typename R::value_type & x0, Sebastiao mth)

Compute the root of the polynomial p which is the closest (in modulus) to x_0, using Sebastiao e Silva method. If there are two roots closest to x_0, the sequence will have two accumulation points and the output solution will not be correct. This is not detected for the moment.

template < class R> inline typename R::value_type* Solve(UPolyDense< R> & p, const typename R::value_type & x0, Sebastiao mth)

Compute $k = mth.nbroot$ roots of the polynomial p, using Sebastiao e Silva method. It starts with the root the closest to x_0. If $k = -1$, all the roots of p are computed.

Tests:

Here are some results of experimentation for the **Sebastiao e Silva method**. Some polynomials with random coefficients real and complex have been generated, and we applied the method for the computation of 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 are of the form:

$$\sum_{i=0}^{n} \frac{x^i}{i!} \tag{4.1}$$

See the examples supplied by D.A. Bini in the FRISCO test suite [11].

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<MPF>** based on **GNU MP** ([36]).

In the following table, we present 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	x_0	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, x_0 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 performed on a Dec workstation with 496M of local memory.

4.3.3 Exclusion methods

The methods of this family are based on a test which answers

- No root in the interval.
- I don't know.

If we are in the first case, we remove the interval. Otherwise, the interval is splitted into two subintervals, to which we apply the test recursively.

The test implemented below use the function

$$\Delta_f(x_0, t) = |f(x_0)| - \sum_{k=1}^{d} \frac{|f^{(k)}(x_0)|}{k!} t^k$$

where f is a polynomial of degree d. We can check that if $\Delta_f(x_0, r) \geq 0$, the interval $[x_0 - r, x_0 + r]$ does not contain a root of f.

upoly/solve/Exclusion.H

template< class R> typename R::value_type Exclude(const R & p, const typename R::value_type & x0, const typename R::value_type & r)

Test if the box `B(x0,r)` *does not contain a (real) root. If the result is positive, this is the case. Otherwise, we cannot say.* `p` *is a univariate polynomial,* `x0` *is the center of the box, and* `r` *its radius* template< class POL, class C> VAL< vector< C> *> Locate(const POL& p,const C& a,const C& b,const C & epsilon)

Output a sequence of points, such that the real roots of the univariate polynomial `p` *in the interval* `[a,b]`*, are at a distance less than* `epsilon`*, from one of these points.*

template< class POL, class C> inline VAL< vector< C> *> Locate(const POL& p, const box< C> & b, const C & epsilon)

The same but with a box `b`.

4.4 Sylvester and Bezout matrices

4.4.1 Introduction

Let $f = f_0 + \cdots + f_m\, x^m$ and $g = g_0 + \cdots + g_n\, x^n$ be two polynomials of degree m and n respectively. The Sylvester matrix of f and g is the matrix of

$$f, x\, f, \ldots, x^{d_1-1}\, f, g, x\, g, \ldots, x^{d_0-1}\, g$$

in the monomial basis. It has the following form:

$$\left[\begin{array}{ccc|ccc} f_0 & & \mathbf{0} & g_0 & & \mathbf{0} \\ \cdot & \ddots & & \cdot & \ddots & \\ \cdot & & f_0 & \cdot & & g_0 \\ f_m & & \cdot & g_n & & \cdot \\ & \ddots & \cdot & & \ddots & \cdot \\ \mathbf{0} & & f_m & \mathbf{0} & & g_n \end{array}\right]$$

The Bezoutian of f and g is the bivariate polynomial

$$\Theta_{f,g}(x,y) = \frac{f(x)\,g(y) - f(y)\,g(x)}{x-y} = \sum_{0 \le i,j \le d-1} \theta_{i,j}^{f,g}\, x^i y^j$$

and its matrix is

$$B_{f,g} = \begin{bmatrix} \theta_{0,0}^{f,g} & \cdots & \theta_{0,d-1}^{f,g} \\ \vdots & & \vdots \\ \theta_{d-1,0}^{f,g} & \cdots & \theta_{d-1,d-1}^{f,g} \end{bmatrix}$$

where $d = \max(m,n)$.

4.4.2 Implementation

upoly/UniReslt.H

template <class REP, class R> VAL<REP*> Bezout(const UPolyDense<R> & P, const UPolyDense<R> & Q)

The Bezout matrix of `P` *and* `Q`.

template <class R> VAL<MatStruct<hankel<typename R::value_type> >*> Bezout(const R & P)

The Bezout matrix of `P` *and 1. The result is a Hankel matrix.*

template <class REP, class R> VAL<REP*> Sylvester(const UPolyDense<R> & P, const UPolyDense<R> & Q)

The Sylvester matrix of `P` *and* `Q`.

template <class R, class REP> VAL<REP*> Sylvester(const UPolyDense<R> & P, const UPolyDense<R> & Q, char c)

The Sylvester matrix, with the maximum degree coefficient at the beginning.

template <class REP, class R> VAL<REP*> Companion(const UPolyDense<R> & P)

The companion matrix of a polynomial `P`.

Chapter 5

Multivariate polynomials

5.1 Monomials

They are defined by a coefficient and an exponent.

5.1.1 Implementation

mpoly/Monom.H

template <class C, class R> struct Monom

Implementation of monomials. The monomial class has tow template parameters, `C` *the arithmetic class of the coefficients and* `R` *a container of int to store the powers.*

5.1.2 Containers for exponents

Possibilities are `dynamicexp`, `numexp`, ...

mpoly/dynamicexp.H

template< char X,class E=char> struct dynamicexp

Dynamic exponent.

mpoly/numexp.H

template< char X, int D=2, class T=int> struct numexp

Numeric exponent. To be used with caution for the moment.

5.1.3 Monomial orders

mpoly/Plex.H

template <class M> struct Plex

Class specifying the lexicographic order on the monomials of type `M`.

mpoly/Dlex.H

template <class M> struct Dlex

Class specifying the degree inverse lexicographic order on the monomials of type `M`. $m_1 < m_2$ *if either the degree of* $degree(m_1) < degree(m_2)$ *or* $degree(m_1) = degree(m_2)$ *and* m_2 *is greater than* m_1 *for the inverse lexicographic ordering.*

mpoly/Vlex.H

template <class M> struct Vlex

Class defining the following order on the monomials m_1 *and* m_2*;* $m_1 < m_2$ *if either the degree of* $degree(m_1) > degree(m_2)$ *or* $degree(m_1) = degree(m_2)$*, or they are equal and* m_1 *is greater for the inverse lexicopgraphic ordering. In a sequence of decreasing monomial for this order, the degree are increasing.*

mpoly/SQVlex.H

template <class M> struct SQVlex

Class defining the following order on the monomials m_1 *and* m_2*;* $m_1 < m_2$ *if either the maximal of the degrees of each variable is strictly greater in* m_1 *than in* m_2*, or they are equal and* m_1 *is greater for the inverse lexicopgraphic ordering refined by the degree.*

5.2 Multivariate polynomials

They are represented by a sorted list of monomials.

5.2.1 Implementation

mpoly/MPoly.H

template <class R, class O> class MPoly

Interface for multivariate polynomials. The type `R` *is the representation of the polynomial (eg. a list of monomial). The type* `O` *is the class specifying the order on the monomials.*

5.2.2 Containers for multivariate polynomials

It could be `list`, `vector`, ...

5.2.3 Example

```
#include "mpoly.H"

typedef Monom<double, dynamicexp<'y'> > Mon;
typedef MPoly<vector<Mon>, Dlex<Mon> > Pol;

int main (int argc, char **argv)
{
  Pol p0("x1+x2+1;");
  Pol p1("x1^2+x2+1;");

  cout<<"p0 ="<<p0<<endl<<",  p1        ="<<p1<<endl;
```

$$p0 = y1 + y2 + (1), p1 = y1^2 + y2 + (1)$$

```
  Pol p2(p0+p1);        cout<<p2<<endl;
```

$$y1^2 + y1 + (2) * y2 + (2)$$

```
  p2+=p1-p0;            cout<<p2<<endl;
```

$$(2) * y1^2 + (2) * y2 + (2)$$

```
  p2 =p0-p1;            cout<<p2<<endl;
```

$$(-1) * y1^2 + y1$$

```
  p2=p0-(p0+p1);        cout<<p2<<endl;
```

$$(-1) * y1^2 + (-1) * y2 + (-1)$$

```
  p2=p0*p1;             cout<<p2<<endl;
```

$$y1^3 + y1^2 * y2 + y1^2 + y1 * y2 + y2^2 + y1 + (2) * y2 + (1)$$

```
  p2=p0 +p1*2;          cout<<p2<<endl;
```

$$(2) * y1^2 + y1 + (3) * y2 + (3)$$

```
  p2=p0 -p1*Mon(2);     cout<<p2<<endl;
```

$$(-2) * y1^2 + y1 + (-1) * y2 + (-1)$$

```
  p2=(p1+p0)*(p1-p0*2); cout<<p2<<endl;
```

$$y1^4+(-1)*y1^3+y1^2*y2+(-1)*y1^2+(-5)*y1*y2+(-2)*y2^2+(-5)*y1+(-4)*y2+(-2)$$

```
  cout<<(p1+p0)*p1<<endl;
```

$$((y1^2 + y2 + (1)) + (y1 + y2 + (1))) * (y1^2 + y2 + (1))$$

```
  // Yes, the operations are performed only when assigned to a variable.
}
```

5.3 Multivariate Resultants

Let us consider a system of $n + 1$ generic polynomials

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

where

- $\mathbf{c} = (c_{i,j})$ are parameters
- $\mathbf{x}$ is a point of the projective variety $X \subset \mathbb{P}^N$, of dimension n,
- The vector functions $\mathcal{L}_i(\mathbf{x}) = (\psi_{i,j}(\mathbf{x}))_{j=0,\ldots,k_i}$ are regular functions on X independent (see [38]) of the parameters $\mathbf{c}$.

The elimination problem consists, in this case, in finding some necessary and sufficient conditions on the parameters $\mathbf{c} = (c_{i,j})$ such that the equations $f_0 = 0, \ldots, f_n = 0$ have a common root in X.

In the classical case, $\mathcal{L}_i(\mathbf{x})$ is the vector of all the monomials of degree d_i and $X{=}\mathbb{P}^n$. The necessary and sufficient condition on the parameters $\mathbf{c} = (c_{i,j})$ such that the homogeneous polynomials $f_0, \ldots, f_n$ have a common root in $X = \mathbb{P}^n$ is $Res_{\mathbb{P}^n}(f_0, \ldots, f_n) = 0$ where

$Res_{\mathbb{P}^n}$ is the *classical projective resultant*. From a geometric point of view, we are looking for the set of parameters $\mathbf{c} = (c_{i,j})$ such that there exists $x \in X$ with $\sum_j c_{i,j}\psi_{i,j}(\mathbf{x}) = 0$ for $i = 0, \ldots, n$. This means that $\mathbf{c}$ is the projection of the point $(\mathbf{c}, \mathbf{x})$ of the *incidence variety*

$$W_X = \{(\mathbf{c}, \mathbf{x}) \in \mathbb{P}^{k_0} \times \cdots \times \mathbb{P}^{k_n} \times X \ ; \ \sum_{j=0}^{k_i} c_{i,j}\psi_{i,j}(\mathbf{x}) = 0 \ ; \ i = 0, \ldots, n\}.$$

We denote by

$$\begin{aligned} \pi_1 : W_X &\rightarrow \mathbb{P}^{k_0} \times \cdots \times \mathbb{P}^{k_n}, \\ \pi_2 : W_X &\rightarrow X. \end{aligned}$$

the two natural projections. The image of W_X by π_1 is the set of parameters $\mathbf{c}$ for which the system has a root. The image by π_2 of a point of W_X is a root in X of the associated system. Thus we have the following definition (see [27]):

Definition 5.3.1 *We denote $Z = \pi_1(W_X)$. If Z is a hypersurface, then its equation (unique up to a scalar) is called the* resultant *of $f_0, \ldots, f_n$ over X. It is denoted by $Res_X(f_0, \ldots, f_n)$.*

In order to be in this case, we impose the following conditions (see [26]):

Hypotheses 5.3.2

1. *X is a projective irreducible variety.*
2. *The regular functions $\mathcal{L}_i, i = 0, \ldots, n$ do not vanish identically at a point of X.*
3. *For generic values of $\mathbf{c}$, the system $f_0, \ldots, f_n$ has no common root in X, and n of these equations (say $f_1, \ldots, f_n$) have a finite number of common solutions.*

Thus, the system $f_0 = \cdots = f_n = 0$ has a solution in X if and only if $Res_X(f_0, \ldots, f_n) = 0$. This generalizes, under the conditions 5.3.2, the definition of the classical resultant over $\mathbb{P}^n$ to other projective varieties.

The approaches to compute effectively the resultant are based on matrix constructions whose determinant will give the resultant or a non-trivial multiple. These resultant matrices can be classified into two main families: the class of *Sylvester matrices*, which generalize the construction of Sylvester (1835, see [66]) to the multivariate case and the class of *Bezoutian matrices* generalizing the construction of E. Bézout (1779, see [9]) to the multivariate case.

5.3.1 Sylvester-like matrices

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

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 :

$$\begin{aligned} S : \mathcal{V}_0 \times \cdots \times \mathcal{V}_n &\rightarrow \mathcal{V} \\ (q_0, \ldots, q_n) &\mapsto \sum_{i=0}^{n} f_i\, q_i. \end{aligned} \tag{5.1}$$

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

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

In this matrix of type 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 5.4.2 are well respected.

5.3.2 Projective resultant

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

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 map (5.1) :

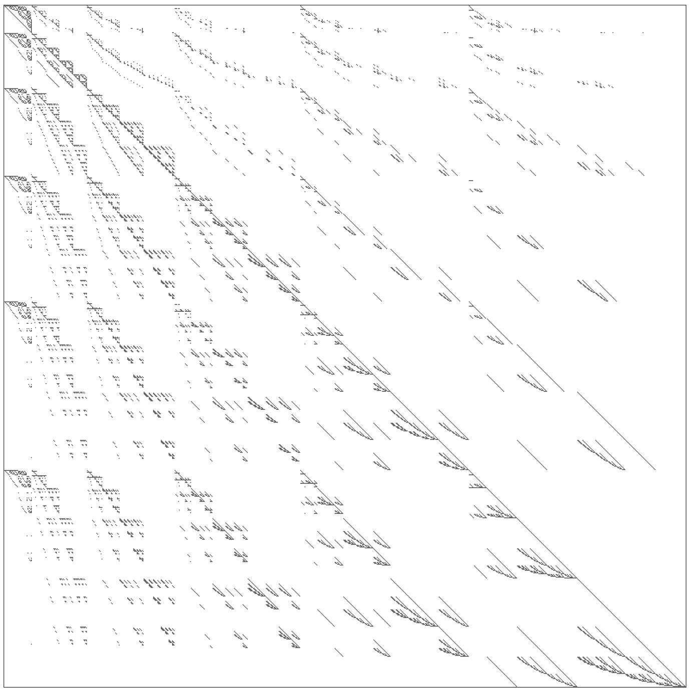
$$\begin{array}{rcl} S : \mathcal{V}_0 \times \cdots \times \mathcal{V}_n & \to & \mathcal{V} \\ (q_0, \ldots, q_n) & \mapsto & \displaystyle\sum_{i=0}^{n} f_i\, q_i. \end{array} \tag{5.2}$$

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

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

In this Macaulay matrix, 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 5.4.2 are well respected. In this resultant construction, it may happen that for specific values of the coefficients of the input polynomials, the matrix D become singular. In this case another construction must be applied.

Here is an example of such a matrix of size 792×792, coming from the autocalibration problem of a camera with Kruppa's equations:



mpoly/resultant/Macaulay.H

unsigned int SizeOfS(unsigned int n, unsigned int k)

Compute the size of the Sylvester-like matrix.

template< class POL> int Choose(int k, int n, POL & ML)

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`*.*

template< class R, class L> VAL< R*> Macaulay(const L & f, char z='N')

Construction of the Macaulay matrix, for the projective resultant of $n+1$ *polynomials in* n *variables. It will be of minimal degree in the coefficients of* `f[0]`*, that is the product of the degree of* `f[1]`*, ...,* `f[n]`*. The order choosed to sort the monomials indexing the rows of the matrix is the* reverse *of the order associated to the polynomials of the sequence* `f`*.* `L` *is the type of a sequence of polynomials.* `R` *specifies the type of the output matrix. For instance,* `Macaulay<Mat>(f);` *will output a matrix of type* `VAL<Mat*>`*. It should have a constructor* `Mat(m,n)` *which allocates the necessary place to store a* $n \times m$ *matrix and the method* `Mat::operator()(size_type i, size_type j)`*. If* `z` *is* `'T'`*, the transposed matrix is constructed.*

template< class R, class L> VAL< R*> Macaulay(const L & f, unsigned int nu, unsigned int nv, char z='T')

Construction of the matrix of all multiples of degree `nu` *of all the polynomials* `f[0]`, `f[1]`, ..., `f[n]`. *The argument* `nv` *is the number of variables. The order choosed to sort the monomials indexing the rows of the matrix is the* reverse *order associated to the polynomials of the sequence* `f`. *The second template argument specifies the type of the output matrix. For instance,* `Macaulay<Mat>(f);` *will output a* `VAL<Mat*>`. *The usual Macaulay matrix is a submatrix of this one, for* $\nu = \sum_{i=0}^{n} \deg(f_i) - n + 1$.

5.3.3 Toric resultant

We define the *Newton polytope* of $f \in S$ as the convex hull of the set $A \subset \mathbb{Z}^n$ of all monomial exponents corresponding to nonzero coefficients is the *support* of f.

The mixed volume of polytopes $Q_1, \dots, Q_n$ is the coefficient of $\lambda_1 \cdots \lambda_n$ in the volume of $Q = \lambda_1 Q_1 + \lambda_2 Q_2 + \cdots + \lambda_n Q_n$.

An interesting fact is that the mixed volume $MV(f_1, \dots, f_n)$ counts the (generic) number of roots of $\mathcal{Z}(f_1, \dots, f_n)$ in $(\overline{\mathbb{K}}^*)^n$ [8]. This bound is known as the BKK bound to underline the contributions of Kushnirenko and Khovanskii in its development and proof [44, 42].

Theorem 5.3.3 *Let* $f_1, \dots, f_n \in \mathbb{K}[x_1, x^{-1}, \dots, x_n, x_n^{-1}]$ *with Newton polytopes* $Q_1, \dots, Q_n$. *The number of isolated common zeros in* $(\overline{\mathbb{K}}^*)^n$*, multiplicities counted, is either infinite, or does not exceed* $MV(Q_1, \dots, Q_n)$*, where* $\overline{\mathbb{K}}$ *is the algebraic closure of* $\mathbb{K}$.

For the construction of Toric Resultant matrices, we consider $n + 1$ Laurent polynomials $f_0, \dots, f_n \in \mathbb{K}[\mathbf{x}, \mathbf{x}^{-1}]$, with support respectively in the polytopes $A_1, \dots, A_{n+1}$. Here is a short description of the algorithm used (see [16] for more details):

5.3.2. **Construction of the Newton matrices.**

1. Subdivide the monomials of the Minkowski sum $Q = A_0 + \cdots A_n$ into cells (mixed subdivision).
2. Decompose each of these cells as a sum $\mathbf{a}_{i_0} + B_{i_0}$ of a vertex of some A_{i_0} and faces of the other polytopes A_i, $i \neq i_0$. This gives a partition of $\mathbf{x}^B$ as an union of sets $\mathbf{x}^{\mathbf{a}_{i_0}} \mathbf{x}^B$ (to be compared with the partition of $\mathbf{x}^B$ into the union of the sets $x_i^{d_i} \mathbf{x}^{B_i}$ in the previous section).
3. For all these cells, replace the monomial $\mathbf{x}^{\mathbf{a}_{i_0}}$ by the polynomial f_{i_0} and construct the corresponding coefficient matrix of all these polynomials.

This matrix is also the matrix of the following map:

$$\begin{array}{rcl} S : \langle \mathbf{x}^{B_0} \rangle \times \cdots \times \langle \mathbf{x}^{B_n} \rangle & \to & \langle \mathbf{x}^B \rangle \\ (g_0, \dots, g_n) & \mapsto & g = \sum_{i=0}^{n} g_i f_i, \end{array}$$

Here are the functions connecting the C-implementation for mixed volume and toric resultant of I.Z. Emiris.

mpoly/resultant/Toric.H

template< class POL> int MixedVolume (const list< POL> & l, int numvars)

Input a list of polynomials, and the number of variables. Output the mixed volume of the polytopes for the variables `1..numvars`.

template< class POL> inline int MixedVolume (const list< POL> & l)

Mixed Volume. The number of variables is supposed to be the size of the list.

template <class R,class POL> VAL< R*> ToricResultant(const list< POL> & l)

Compute the Toric resultant by the incremental algorithm [].

5.3.4 Bezoutian resultant

Here we recall some definitions and properties of the Bezoutian theory (for more details, see [19, 26]).

To the set of variables $\mathbf{x}$ we add a new set of variables $\mathbf{y} = (y_1, \ldots, y_n)$ and we denote $\mathbf{x}^{(0)} = \mathbf{x}$, $\mathbf{x}^{(1)} = (y_1, x_2, \ldots, x_n), \ldots,$ $\mathbf{x}^{(n)} = \mathbf{y}$.

For a polynomial $q \in R$, we define $\theta_i(p) = \frac{p(\mathbf{x}^{(i)}) - p(\mathbf{x}^{(i-1)})}{y_i - x_i}$.

Definition 5.3.4 *For $n+1$ polynomials $f = (f_0, f_1, \ldots, f_n) \in R$, we call* Bezoutian *of $f_0, f_1, \ldots, f_n$ the polynomial in $\mathbf{x}$ et $\mathbf{y}$ def ined by:*

$$\Theta(f_0, f_1, \ldots, f_n) = \Theta_f(f_0) = \det \begin{pmatrix} f_0(\mathbf{x}) & \theta_1(f_0) & \cdots & \theta_n(f_0) \\ \vdots & \vdots & & \vdots \\ f_n(\mathbf{x}) & \theta_1(f_n) & \cdots & \theta_n(f_n) \end{pmatrix} = \sum_{i,j} \theta_{i,j}^{f}\, \mathbf{x}^{u_i}\, \mathbf{y}^{v_j}, \tag{5.3}$$

It is a polynomial of degree at most $\sum_{i=0}^{n} deg(f_i) - n$.

In the monomial basis $\mathbf{x}^\alpha$ and $\mathbf{y}^\beta$, Θ_f can be written:

$$\Theta_f(f_0) = \sum_{\alpha \in E, \beta \in F} t_{\alpha,\beta} \mathbf{x}^\alpha \mathbf{y}^\beta$$

.

Definition 5.3.5 *the set of monomials* $\mathbf{x}^{\alpha}$ *(resp.* $\mathbf{y}^{\beta}$*) is called the* $\mathbf{x}$-support *(resp.* $\mathbf{y}$-support*) of* $\Theta_f(f_0)$.

Let the following map:

$$\begin{array}{rcl} \Phi_f(f_0) : \widehat{R} & \to & R \\ \Lambda & \mapsto & \sum_{\alpha\in E, \beta\in F} t_{\alpha\,\beta}\mathbf{x}^{\alpha}\Lambda(\mathbf{y}^{\beta}). \end{array}$$

Definition 5.3.6 *We denote by* $\mathtt{B}_{f_0}(f)$*, the matrix of* $\Phi_f(f_0)$ *from the dual basis of* $\widehat{R}$ *into the monomial basis* $(\mathbf{x}^{\alpha})$ *of* R*. This matrix is the* Bezoutian *matrix of the polynomials* $f_0, f_1, \ldots, f_n$.

mpoly/resultant/MBezout.H

```
template <class POL> inline POL MBezout(const list<POL> & l, typename POL::monom_t::index_t c=0)
```

Compute the polynomial in the two set variables, which is the determinant associated with the list of polynomials `l`*. The* `c` *first variables are considered as parameters. The* n *next variables (where* n *is the length of* `l-1`*) are the variables* `x` *and the next* n *variables are the* `y`.

```
template <class POL, class RM> VAL<RM*> DeltaOf(const POL & P, int l0, int l1, int l2, RM M)
```

Compute the matrix associated with the polynomial `p`*, assuming the variables up to* `l0` *are hidden, the first block of variables is of size* `l1` *and the second one of size* `l2`*. The monomials in the first block of variables are indexing the rows, the monomials in the second block of variables are indexing the columns. The entries of this matrix are polynomials in the variables* $x_0, ..x_{l_0}$*. The type* `RM` *specifies the the type of the output matrix.*

5.4 Solving polynomial systems by matrix computations

Let $\mathcal{A} = \mathbb{K}[\mathbf{x}]/I$ be the quotient algebra by the ideal $I = (f_1, \ldots, f_m)$, which is a finite $\mathbb{K}$-dimensional vector space iff $\mathcal{Z}(I)$ is 0 dimensional.

5.4.1 Solving from the operators of multiplication

For each $a \in \mathcal{A}$, consider:

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

defining the operator of multiplication by a in $\mathcal{A}$, and the $\mathbb{K}$-linear application associated to M_a:

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

The matrix of M_a^t in the dual basis of $\widehat{\mathcal{A}}$ is the transposed of the matrix of M_a. So they share their eigenvalues.

Our matrix approach to solve polynomial systems is based on the following theorem (for more details and proof, see [49]) :

Theorem 5.4.1 *Assume that $\mathcal{Z}(I)$ is zero-dimensional.*

1. *The eigenvalues of the linear operator M_p (resp. M_p^t) are $\{p(\zeta_1), \ldots, p(\zeta_d)\}$.*
2. *The common eigenvectors of $(M_{x_i}^t)$ are (up to a scalar) $\mathbf{1}_{\zeta_1}, \ldots, \mathbf{1}_{\zeta_d}$.*
3. *When $m = n$, the common eigenvectors of (M_{x_i}) are (up to a scalar) $J(\mathbf{x})\mathbf{e}_1, \ldots, J(\mathbf{x})\mathbf{e}_d$, where $J(\mathbf{x})$ is the Jacobian of $p_1, \ldots, p_n$.*

By [13], theorem 2.1, we know that 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 $\mathtt{M}_{a,i}$ is of the form

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

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,

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

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:

First, note that for any pair $a, b \in \mathcal{A}$, we have

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

so that $\mathbf{1}_{\zeta_i}$ is an eigenvector of M_a^{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 $i = 1, \ldots, d$ ([13], proposition 2.3).

If a root ζ_i is simple, the eigenvector of M_a^{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:

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

1. Compute the transposed 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

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

Implementation:

mpoly/solve/EigenMult.H

template <class M> VAL<MatDense<lapack<AlgClos<typename M::value_type>::TYPE > > *> Solve(M & A, int n, Eigen mth)

Compute the simple roots from the first $n+1$ first coordinates of the eigenvectors of the matrix of multiplication `A`. *It is assumed that the roots are simple and that the first $n+1$ elements of the basis of the quotient ring are $1, x_1, \ldots, x_n$.*

```
E =Solve(M,n,Eigen());
```

where `M` *is a matrix of multiplication and* `n` *the number of variable. The matrix* `E` *will be the matrix of complex coordinates of the (simple) roots.*

template <class M> inline VAL<MatDense<lapack<AlgClos<typename M::value_type>::TYPE > > *> Solve(M & A, const array1d<typename M::size_type> & t, Eigen method)

Compute the simple roots from $n+1$ coordinates of the eigenvectors of the matrix of multiplication `A`. *It is assumed that the roots are simple. The array* `t` *is the array of positions of the monomials $1, x_1, \ldots, x_n$.* ***`Solve, Eigen`***

5.4.2 Solving from matrix of univariate polynomials

To reduce the resolution of our system to a univariate and eigenvector problems, several methods can be applied:

- If the system of n equations $f_1 = \cdots = f_n = 0$, where f_i is considered as a polynomial in $x_1, \ldots, x_{n-1}$, with parameter x_n has a solution, then the "resultant" (to be precised) vanishes. So any multiple of the resultant must vanish. Let $\mathtt{M}(x_n)$ be a resultant matrix of the system (see 5.3), where we consider the variable x_n as hidden in the coefficients field. By construction, $\mathtt{M}(x_n)$ is the matrix of multiples of $(f_1, \ldots, f_n)$ in a monomial basis $w = (\mathbf{x}^\alpha)_{\alpha \in F}$. Thus we have $w\, M(x_n) = 0$ at a root of our system (and usually by construction of $\mathtt{M}$, $w \neq 0$). If we solve $det(\mathtt{M}(x_n)) = 0$, we obtain the values of the coordinnate x_n of the roots. If we compute the corresponding w, we obtain the other coordinates.

- Another approach consists in adding a new polynomial, eg $f = u_0 + u_1 x_1 + \ldots + u_n x_n$, where u_0 is an indeterminate. Then we compute a resultant matrix $\mathtt{M}(u_0)$ of these $n+1$ polynomials, and solve the generalized eigenproblem $w\, \mathtt{M}(u_0) = 0$.

In this approach the matrix may be nonlinear in the hidden variable, so $\mathtt{M}$ is a matrix polynomial $\mathtt{M}(x_n) = \mathtt{M}_d\, x_n^d + \cdots + \mathtt{M}_1\, x_n + \mathtt{M}_0$, for some $d \geq 1$. Our problem reduces to the following eigenproblem:

$$\left(\begin{bmatrix} 0 & \mathbb{I} & 0 & \cdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \mathbb{I} \\ -\mathtt{M}_0^t & -\mathtt{M}_1^t & \cdots & -\mathtt{M}_{d-1}^t \end{bmatrix} - \beta \begin{bmatrix} \mathbb{I} & 0 & \cdots & 0 \\ 0 & \ddots & 0 & \cdots \\ \vdots & 0 & \mathbb{I} & 0 \\ 0 & \cdots & 0 & \mathtt{M}_d^t \end{bmatrix} \right) \begin{bmatrix} w \\ \beta w \\ \vdots \\ \beta^{d-1} w \end{bmatrix} = 0$$

If $\mathtt{M}_d$ is numerically singular, we may change variable x_n to $\frac{(t_1 x_n + t_2)}{(t_3 x_n + t_4)}$.

Implementation:

mpoly/solve/EigenHidden.H

template< class Mat, class MatPol> VAL< Mat*> Solve(const MatPol & MP, Compagnon method)

Compute the real solutions of $M(x).v = 0$ by transforming this problem into an eigenvector problem $(A - xB)\, w = 0$. It is assumed that the pencil has only simple real eigenvectors. The case of multiple roots is not handled for the moment. This case can be detected by checking if all the elements of the first line are 1 or not.

```
E = Solve<MatC>(Mx, Compagnon());
```

where `MatC` *is a type of matrix with complex coefficients.*

5.4.3 Computing the matrix of multiplication from resultant matrices

As we have seen, the matrix of multiplication $\mathtt{M}_{f_0}$ by an element $f_0 \in \mathcal{A}$ yields a way to recover the roots of the system. Usually, this matrix is not available directly from the input equations. However it can be computed from a Sylvester-like matrix `S` (we will see later that we can take for `S` a resultant matrix, in the generic case) satisfying the following properties:

Hypotheses 5.4.2 *The matrix* `S` *is a square matrix and have the following form :*

$$\mathtt{S} = \begin{pmatrix} \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} \end{pmatrix} \tag{5.4}$$

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* `D` *is invertible.*

For any matrix `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 `D` in `S`:

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

(for more details see [13]).

Implementation:

mpoly/solve/EigenRslt.H

template< class Mat, class LPol> VAL< MatDense< lapack< AlgClos< typename Mat::value_type> ::TYPE > > *> Solve(const LPol & l, Projectiv method)

Solve the system of equations `l`, *by computing the tranposed matrix of multiplication by* `f0` *modulo* `l` *as the Schur complement of the Macaulay matrix of the* $n+1$ *polynomials. It is assumed that* the monomials $1, x_1, ..., x_n$ are smaller than any of the other monomials of degree ≥ 2.
The following function can be used for this purpose is (see 3.6.2):

```
M= USchur(S);
```

where `S` is a matrix satisfying the hypotheses 5.4.2 (eg. a resultant matrix of a generic system).

5.4.4 An example based on resultant matrices

```
#include <list>
#include "mpoly.H"
#include "mpoly/resultant.H"
#include "lapack.H"
#include "linalg.H"

typedef Monom<double, dynamicexp<'x'> > Mon;
typedef MPoly<vector<Mon>, Dlex<Mon> > Pol;
typedef MatDense<lapack<double> > Matr;
typedef VectStd<array1d<double> > Vect;
typedef VectStd<array1d<complex<double> > > VectC;

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

  int n;
  cin >>n;

  array1d<Pol> l(n);
  for(int i=0; i< n;i++) cin >> l[i];

  cout <<"Polynomes:"<<endl;
  for(int i=0; i< n;i++) cout<<l[i]<<";"<<endl;

  int d=1;
  for(int i=1; i< n;i++) d*=Degree(l[i]);
  cout <<"Bezout bound:"<<d<<endl;

  Matr S=Macaulay<lapack<double> >(l,'N');
  cout <<"Macaulay matrix of size "<< S.nbrow()<<endl;

  for(unsigned int i=d;  i<S.nbrow(); i++)
    if(S(i,i)==0)
      cout<<"Probleme d'ordre des polynomes "<<i<<" "<<i <<endl;

  Matr D = S(Range(d,S.nbrow()-1),Range(d,S.nbcol()-1));
```

```
  cout <<"Lower diag. block of size "<< D.nbrow()<<endl;

  cout <<"Singular values of D: "<<endl;
  Vect Sv= Svd(D);
  cout << Sv<<endl;

  //  cout <<"Singular values of S: "<<endl;
  //  cout <<Eval<Vect>(Svd(S))<<endl;

  Matr U= USchur(S,d);
  cout <<"Eigenvalues of the Schur complement: "<<endl;
  cout << Eigenval(U)<<endl;

}
```

5.5 Solving by controlled iterative methods

5.5.1 The classical power method and inverse power method

We have seen above how reducing the resolution of a polynomial system to eigenvector computations. Now, let us have a look to the classical (inverse) power method for matrices (for more details see [69], [13]).

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, $\mathtt{M}_a$ 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.

5.5.2 Implicit iterative 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 [35]).

So under the hypotheses 5.4.2 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,

5.5.4. 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 [51]). 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.

5.5.3 Results

The following table contains the results of our experiment for the **implicit inverse method** using the structures implemented during this work.

Many of the following examples have two results, one, $*G$, by using the **GMRES** solver ([40]) developped by R. Pozo and his library **SparseLib++** ([55]), the other, by $*U$ using a direct method with **UMFPACK** routines ([67]) for solving the sparse linear system $\mathtt{S}x = b$.
The examples $s22G$ and $s22U$ are both the following system ([21]) :

$$
\begin{gathered}
-11 + 2x^2 + 3y^2 = 0 \\
-3 + x^2 - y^2 = 0
\end{gathered}
$$

$s44*$ 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 interchapter of a sphere with an other quadric and a plane ([21]).

$$
\begin{gathered}
x^2 + y^2 + z^2 = 4 \\
xy + z^2 - 1 = 0 \\
x + y - z = 0
\end{gathered}
$$

Examples $cyc3*$ are the following, for $n = 3$: Cyclic n-roots (see [11])

$$
\begin{gathered}
x_1 + x_2 + \cdots + x_n = 0 \\
x_1 x_2 + x_2 x_3 + \cdots + x_n x_1 = 0 \\
\vdots \\
x_1 \cdots x_{n-1} + x_2 \cdots x_n + \cdots + x_n x_1 \cdots x_{n-2} = 0 \\
x_1 x_2 \cdots x_n = 1
\end{gathered}
$$

Results $kat6U$ (resp. $kat8U$) are, for $n = 6$ (resp. $n = 8$): Katsura(N) (see [11])

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

The last result *kruppa* corresponds to the Kruppa equations of a reconstruction problem in Computater Vision (see [32]) 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 $kat8G$).

Chapter 6

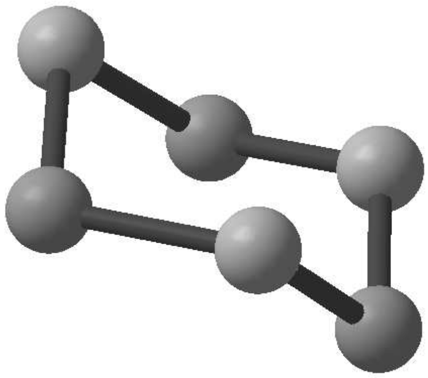
Applications

Here are examples of applications that has been developped inside this framework. We mentionned also the implementation of a solver for the inverse kinematic problem of $6R$ serial robot [60], that will be reported in another work.

6.1 The case of a Six-Atom Molecule

6.1.1 Problem

Our problem is to compute all configurations of a molecule. Only the *dihedral* angles, that is the solid angles between two consecutive plans, each defined by an atom and its two links, are allowed to vary, while keeping bond lengths and bond angles fixed. We consider cyclic molecules of six atoms:



This figure shows one possible configuration of such a molecule. The relationship between geometry and robotics is obvious, once the bonds are thought of as rigid joints and atoms as mechanism's links or articulations. The only movement allowed is the rotation around the bond axes, so that the problem of identifying a configuration reduces to finding the respective poses. In kinematic terms, the molecule is equivalent to a serial mechanism in which each pair of consecutive axes intersects at a link. This implies that the link offsets are zero for all six links, which will allow us to reduce the 6-dimensional problem to a system of 3 polynomials in 3 unknowns.

6.1.2 Algebraic formulation

The molecule studied has a cyclic backbone of 6 atoms, typically of carbon, that determines our problem, carbon-hydrogen bonds or other bonds outside the backbone being ignored. The bond lengths and angles provide the constraints while the six dihedral angles are allowed to vary.

Two formulations are generally used for this problem, one using distances, another using angles. We present only last formulation used in this study [30].

$$\alpha_{11} + \alpha_{12} cos\theta_2 + \alpha_{13} cos\theta_3 + \alpha_{14} cos\theta_2 cos\theta_3 + \alpha_{15} sin\theta_2 sin\theta_3 = 0 \tag{6.1}$$
$$\alpha_{21} + \alpha_{22} cos\theta_3 + \alpha_{23} cos\theta_1 + \alpha_{24} cos\theta_3 cos\theta_1 + \alpha_{25} sin\theta_3 sin\theta_1 = 0 \tag{6.2}$$
$$\alpha_{31} + \alpha_{32} cos\theta_1 + \alpha_{13} cos\theta_2 + \alpha_{14} cos\theta_1 cos\theta_2 + \alpha_{35} sin\theta_1 sin\theta_2 = 0 \tag{6.3}$$
$$cos^2\theta_1 + sin^2\theta_1 = 1 \tag{6.4}$$
$$cos^2\theta_2 + sin^2\theta_2 = 1 \tag{6.5}$$
$$cos^2\theta_3 + sin^2\theta_3 = 1 \tag{6.6}$$

where α_{ij} are input coefficients.
For our resultant approach we prefer an equivalent formulation with a smaller number of polynomials, obtained by applying the standard transformation to half-angles that gives rational equations in the new unknowns t_i :

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

This transformation takes automatically the last three equations. By multiplying both sides of the i^{th} equation by $(t+t_j^2)(1+t_k^2)$, where (i,j,k) is a permutation of $1,2,3$, we obtain the following system:

$$f_1 = \beta_{11} + \beta_{12} t_2^2 + \beta_{13} t_3^2 + \beta_{14} t_2 t_3 + \beta_{15} t_2^2 t_3^2 = 0 \tag{6.7}$$
$$f_2 = \beta_{21} + \beta_{22} t_3^2 + \beta_{23} t_1^2 + \beta_{24} t_3 t_1 + \beta_{25} t_3^2 t_1^2 = 0 \tag{6.8}$$
$$f_3 = \beta_{31} + \beta_{32} t_1^2 + \beta_{33} t_2^2 + \beta_{34} t_1 t_2 + \beta_{35} t_1^2 t_2^2 = 0 \tag{6.9}$$

where β_{ij} are input coefficients.

6.1.3 Resolution - Results

The results which are presented here have been obtained with the resultant-based method presented in [30]. The following specialisation of our parameters has the number maximal of solutions, namely 16, which are all real.

$$\begin{array}{c} -x_2^2x_3^2 - x_3^2 + 24x_2x_3 - x_2^2 - 13 \\ -x_1^2x_3^2 - x_3^2 + 24x_1x_3 - x_1^2 - 13 \\ -x_1^2x_2^2 - x_2^2 + 24x_1x_2 - x_1^2 - 13 \end{array}$$

This system has a Bezoutian matrix of rank 28, when we add a generic linear form.

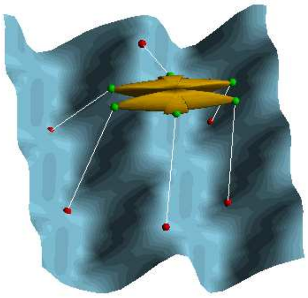
x_1	x_2	x_3
10.850487224	0.779548045	0.779548045
0.779548045	0.779548045	10.850487224
0.779548045	10.853913563	0.779548045
0.779548045	0.779524472	0.779548045
0.332599000	4.625181601	4.625181601
-0.779548045	-0.779548344	-0.779548045
-0.779548045	-0.779548045	-10.859502097
-0.779548045	-10.859502097	-0.779548045
-10.853211545	-0.779548045	-0.779548045
-4.625181602	-4.625181601	-4.625181601
-4.625181601	-4.625181601	-0.331957622
-4.625181601	-0.331957622	-4.625181601
-0.332989621	-4.625181601	-4.625181601
4.625181602	4.625181601	4.625181601
4.625181601	0.332599000	4.625181601
4.625181601	4.625181602	0.332599000

6.2 The direct kinematic problem of a parallel robot

Now we give a brief presentation of the direct kinematic problem of a parallel robot (called the Stewart platform or left hand).

6.2.1 Problem

Consider six fixed points $(X_i)_{1\leq i\leq 6}$ on a fixed solid S_X and six others points Z_i, attached to a moving solid S_Z. The articulations between the two solids S_X and S_Z are extensible bars $(X_i, Z_i)_{1\leq i\leq 6}$ with spherical joints.



The platform can be controlled by changing the lengths of the bars. From a practical point of view, it is easy to measure the length of these articulations but much less obvious to determine the position of the solid S_Z once these lengths are known.

For our approach, we are interessed in the map which associates to a displacement of the solid S_Z, the square of the length $\|X_i, Z_i\|$, or more precisely the number of points in a "generic" fiber. Some previous works [59], [47], [48], [45] have proved that in the generic case the degree of this map (thus the number of solutions) is 40, and another recent work [24] that, for particular platforms, these 40 solutions can be all real.

6.2.2 Algebraic formulation

This formulation is completely described in [47]. We denote $\mathbb{K}$ (in theory $\mathbb{C}$ and in practice $\mathbb{R}$) a field, $\mathbb{A}^n = \mathbb{K}^n$ the affine space of dimension n over $\mathbb{K}$. We consider $\mathbb{A}^n$ as an open set of the projective space $\mathbb{P}(\mathbb{K}^{n+1})$. We denote $\mathcal{D}$ the set of displacements of $\mathbb{A}^3$. It is an algebraic set of dimension 6, and we have $\mathcal{D} = SO_3 \times \mathbb{K}^3$, $\mathbb{K}^3$ where $\mathbb{K}^3$ is the set of translations and SO_3 is the variety of rotations which fixe the origin.
Let (Y_i) be the initial position of the solid S_Y, we are interested by the map:

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

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

The space D is also called the *space of configurations* and $\mathbb{K}^6$ the *space of observations*. The variety SO_3 can be parametrized with quaternions by:

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

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

where $V(a^2+b^2+c^2+d^2=0)$ is the variety defined by the polynomial $a^2+b^2+c^2+d^2$. Furthermore the map ω is bijective. The translation $T \in \mathbb{K}^3$ is a vector of coordinates $T=[T_1,T_2,T_3]$.

We will consider the "generic" fiber (more precisely, the fibers which do not meet the critical locus) of the map ψ. It has the same number of points (counted with multiplicities) then the fiber of $\psi \circ (\omega \times Id)$.
This number which is called the degree of the map $\psi \circ (\omega \times Id)$ corresponds also to the following dimension [47]:

Definition 6.2.1 *The degree d of the map $\psi(\omega \times Id)$ is the dimension of the field $\mathbb{K}(\|\omega(\frac{a}{d},\frac{b}{d},\frac{c}{d}.Y_i+T-X_i\|^2_{a\leq i\leq 6})$ as a K-vector space, where $K=\mathbb{K}(\frac{a}{d},\frac{b}{d},\frac{c}{d},T_1,T_2,T3)$.*

We use the following proposition:

Proposition 6.2.2 *The fiber of a non-critical value is an algebraic set of d distincts points on which the jacobian of the map is of maximal rank.*

The non-critical values are in the complementary of the image of the critical locus. This set is a dense open subset of the space of observations. So taking a random point in the last space, we shall almost be sure that it is a non-critical value. The multiplicity of the distinct points of this fiber is one. As the number of points (with multiplicities) is the degree d of the map, we have exactly d distinct points in a generic fiber of ψ.

We work here, directly in a space of dimension 6, by using the parametrization of rotations by quaternions. Given 6 squares of length δ_i and the points (X_i), (Y_i), we have to determine how many solutions $D=(R,T)$ satisfied the system :

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

for $1\leq i\leq 6$. As the points (Y_i) correspond to the initial positions of the solid S_Z, we may assume without loss of generality than $X_1=[0,0,0]$, $Y_1=[0,0,0]$. So we have $[T,T]=\delta_1=\alpha_1$, and we obtain, after substraction of the first equation to the others, a new equivalent system:

$$\left\{\begin{array}{l}[T,T]=\alpha_1 \\ [R.Y_2,T]-[X_2,T]-[R.Y_2,X_2]=\alpha_2 \\ [R.Y_3,T]-[X_3,T]-[R.Y_3,X_3]=\alpha_3 \\ [R.Y_4,T]-[X_4,T]-[R.Y_4,X_4]=\alpha_4 \\ [R.Y_5,T]-[X_5,T]-[R.Y_5,X_5]=\alpha_5 \\ [R.Y_6,T]-[X_6,T]-[R.Y_6,X_6]=\alpha_6\end{array}\right. \tag{6.10}$$

where $\alpha_i = \frac{1}{2}(\delta_i - \delta_1 - [X_i, X_i] - [Y_i, Y_i])$, for $2 \leq i \leq 6$.

By considering the case where $Y_1 = Y_6$, we know that $Z_1 = RY_1 + T = RY_6 + T$ is on a circle whom the center is the projection of $Y_1 = Y_6$ on the axis (X_1, X_6). This allows us the following parametrization of the translation: $T = [p\ r(\frac{1-t^2}{1+t^2})\ r(\frac{2t}{1+t^2})]$, where r is the radius of the circle and p the projection of Y_1 on (X_1, X_6). Constructing a resultant matrix in the variables $a,\ b,\ c$ *et* d (for the 4 equations not corresponding to the points Y_1, and Y_6) yields a matrix whose determinant is the product of a polynomial of degree 40 by a power of $t^2 + 1$. This first factor vanishes if and only if there exists a rotation such that these 4 equations are satisfied.

To construct a resultant matrix for this system we have used the generalization of Dixon's method described in [47], with $d = 1$. Once the Bezoutian and the multiples of the polynomials are computed, we obtain a square matrix of size 20 whose coefficients are polynomials of degree at most 8. The generalized eigenvalues are, after eliminating of the extraneous factors, the values of t which gives us the coordinates of the translation. The corresponding eigenvectors give us the values of $a,\ b,\ c,\ d$. Now we give an example of such a resolution.

6.2.3 Resolution - Results

We have considered the following geometry of robot:
$X_1 = (0\ 0\ 0)$, $X_2 = (\frac{1}{2}\ \frac{-1}{2}\ 0)$, $X_3 = (\frac{3}{2}\ 1\ 0.3)$, $X_4 = (\frac{-3}{2}\ \frac{1}{2}\ 0)$, $X_5 = (\frac{1}{2}\ \frac{1}{3}\ 0.1)$, $X_6 = (0\ 2\ 0)$,
$Y_1 = (0\ 0\ 0)$, $Y_2 = (1\ 1\ 0)$, $Y_3 = (0\ 1\ 0)$, $Y_4 = (\frac{3}{2}\ 1\ 0)$, $Y_5 = (\frac{-3}{2}\ \frac{1}{2}\ 0)$, $Y_6 = (0\ 0\ 0)$.

And the fixed squares of the arm lengths are: $l_1 = 2$, $l_2 = \frac{15}{2}$, $l_3 = \frac{17}{4}$, $l_4 = \frac{49}{4}$, $l_5 = 14$, $l_6 = 2$.

We denote $u,\ v,\ w$ the coordinates of the translation and $a,\ b,\ c$ the three quaternions variables and we obtain the following system to solve:

$$-l_2a^2 + 3ua^2 + va^2 + 4wab - l_2b^2 + 3ub^2 - 3vb^2 - 4wac + 4ubc + 4l_8bc - l_2c^2 - uc^2 + vc^2 - 4ua + 4l_8a + 2.5a^2 + 4wb + 4.5b^2 + 4wc + 0.5c^2 - l_2 - u - 3v - 4a + 2.5 = 0$$

$$-l_3a^2 - 3va^2 - l_3b^2 - 3vb^2 - 4wal_{12} + 4vbc - l_3c^2 - 4uc^2 - 3vc^2 + 4va + 2.25a^2 + 4wb + 2.25b^2 - 6bc + 6.25c^2 - l_3 - 4u - 3v - 6a + 6.25 = 0$$

$$-l_4a^2 + ua^2 + 6va^2 + 6wab - l_4b^2 + ub^2 - 4wac + 6ubc + 4vbc - l_4c^2 - 3uc^2 + 6vc^2 - 6ua + 4va + 9.25l_{10}^2 + 4wb + 0.25b^2 + 6wc + 3bc + 11.25c^2 - l_4 - 3u + 9a + 2.25 = 0$$

$$-l_5a^2 + 4ua^2 - 4va^2 - 6wab - l_5l_{11}^2 + 4ub^2 + 2vb^2 - 2wac - 6ubc + 2l_8bc - l_5c^2 + 2uc^2 - 4vc^2 + 6ua + 2l_8a + 8a^2 + 2wb + 5b^2 - 6wc - 10bc + 5l_{12}^2 - l_5 + 2u + 2v + 8a + 2 = 0$$

The approlimate solutions are:

t	u	v	w
-0.8559741511414334	1	0.1542734032036417	-0.9880281964923707
0.8559741511414365	1	0.1542734032036381	0.9880281964923713
-0.5208622044322742	1	0.5731959676433147	-0.8194183197106615
0.5208622044322728	1	0.5731959676433166	0.8194183197106604
0.999999980272572	1	1.972742819160095e-08	0.9999999999999998
1.000000019727433	1	-1.972743263130614e-08	0.9999999999999998

a	b	c	d
-0.30022899732068	-0.2524706689758425	-0.008478921420018205	1
0.3002289973206608	0.2524706689758229	-0.00847892142001521	1
0.7526500320258817	-0.5609206400435618	-0.1171358376325	1
-0.7526500320258833	0.5609206400435761	-0.1171358376324918	1
2.219335519681477e-08	7.397784242297935e-09	4.931857996019877e-09	1
-2.219336279616145e-08	-7.397788962834889e-09	-4.931857226361814e-09	1

This last solution can be considered as a double solution. The corresponding configurations of the parallel robot are shown in fig. 6.2.3.

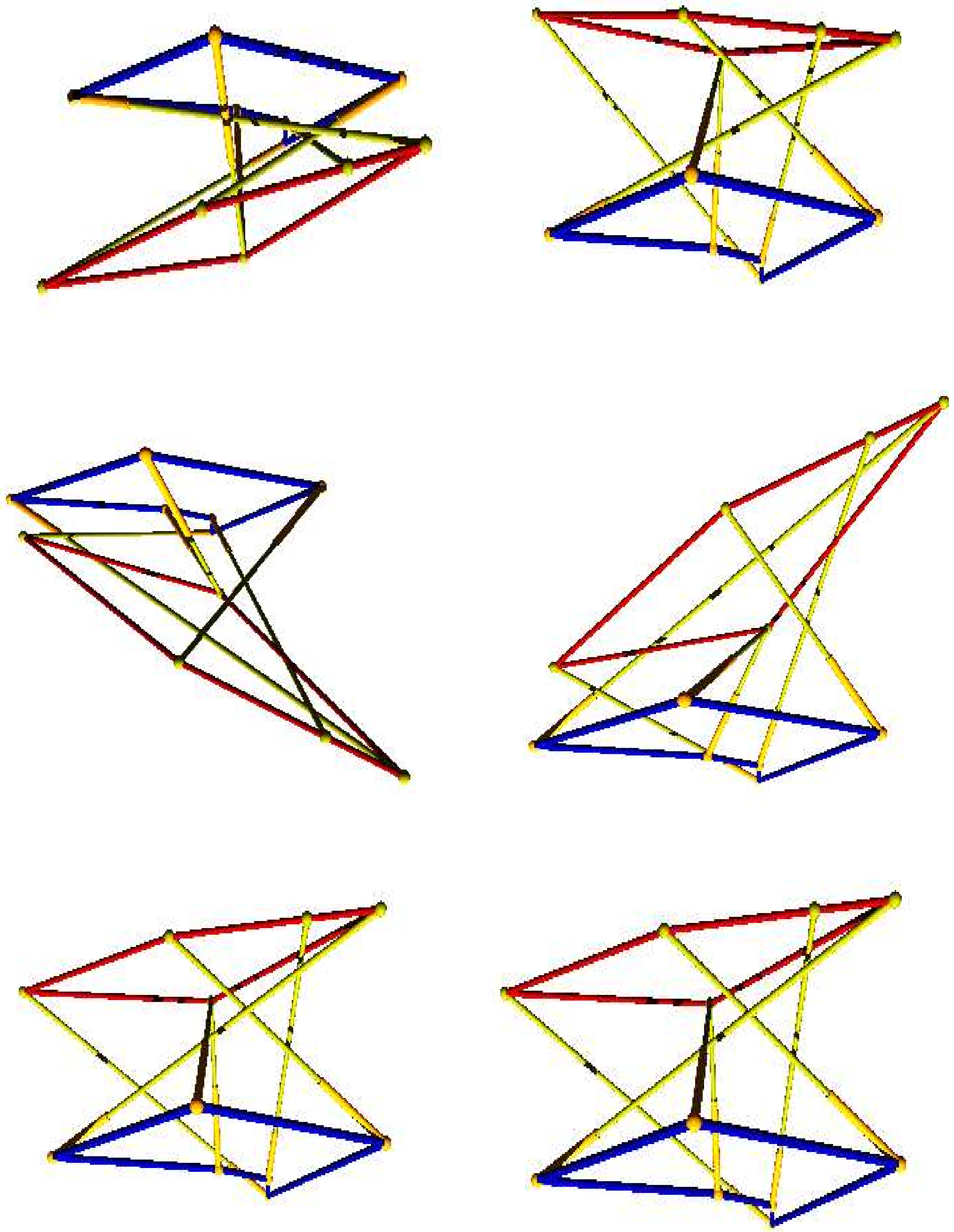


Figure 6.1: Configurations of the 6 real solutions

Appendix A

How to install the library

A.1 Get the files by ftp

The library is available by `ftp` at the following address:

ftp://ftp-sop.inria.fr/saga/ALP.tgz

The web site for the library is

http://www.inria.fr/saga/logiciels/ALP/

The current distribution contains the directories

- `src` for the source codes,
- `doc` for the documentation (`manual.ps.gz`),
- `html` for the documentation in html,
- `expl` containing examples,
- `config` for the configuration files.

Precompiled external libraries involved in ALP are available at:

```
ftp://ftp-sop.inria.fr/saga/mourrain/lib-linux.tgz   (for linux)
ftp://ftp-sop.inria.fr/saga/mourrain/lib-solaris.tgz (for sun solaris)
```

Here are the instructions to install the library:

1. Get the file `ALP.tgz` by ftp.
2. Type

```
tar zxvf ALP.tgz
```

(or gzip -dc ALP.tgz |tar -xvf -). You will otain a directory ALP containing the current distribution, that you will put at your convenience in a place that we will refer hereafter by <ALP>.

3. Move in <ALP> and type make. You will obtain
   - a directory (lib-linux, lib-solaris or lib-dec) containing the libraries for your architecture.
   - the compilation tool alp, configured for your environnement.

If you want to get in addition the precompile libraries, for other architectures, type:

- make lib-linux (for pc linux),
- make lib-solaris (for sun solaris),
- make lib-decosf1 (for dec).

Now, you can try a first example:

```
./alp expl/Linalg.ex
```

Warning: In order to run correctly the installation, check that the following commands are available:

```
cd, mkdir, echo, sed, rm, uname, wget, tar, make
```

The library is working with a compiler version of g++ (g++ -v) greater than

```
g++-2.95
```

If they are not available, ask to your system administrator.

A.2 Installing the script alp

1. Move to the directory config.
2. If necessary, that is, if there is no command called uname, which returns the type of workstation your are working on, edit the second line of the Makefile and replace it by
   - SYSTEM=solaris if you are working on sun stations.
   - SYSTEM=linux if you are working on linux stations.
   - SYSTEM=decosf1 if you are working on dec stations.

This will specify the file `config/Makefile...` that will be used.

If you want to modify the environment variables, edit if necessary the files

```
<ALP>/config/Makefile.*
```

3. Type `make alp`. You obtain the script `alp`. The variable `DIR` is set to the directory of the current version of `alp`. The variable `SYSTEM` specifies the type of workstation.

4. Now add the script `alp` to your path, by moving it to your `bin` directory

```
cp alp ~/bin
export PATH=$PATH:$HOME/bin
```

or

```
export PATH=$PATH:<the complete name of the directory config>
```

A.3 How to use it

Once installed, the library can be used from any position, with the command `alp`. To compile the file `file.cc`, type

```
alp file.cc
```

You get an executable file `file.ex`, that you can run:

```
./file.ex
```

This command calls the usual `make` command, with a Makefile adapted to the configuration (type of machine, libraries, ...). You can define locally your own `Makefile.alp` (eg. for defining `OBJ=...`). In this case, the command alp will also load this `Makefile.alp` and use it to compile the files.

Here are the options of this command:

- `-h` display the help.
- `?` search help on a `word` (not yet ready).
- `-v` display the version of the library.
- `-r` compile and run the executable `*.ex`
- `-f` force the file `*.cc` to be (re)compile and run the executable `*.ex`

A.4 Using a makefile

Here is an example of `Makefile` that you can edit in the directory where you want to compile the files (eg. `fich.cc`) using `ALP`, if you cannot use the compilation command `alp`:

```
#----------------------------------------------------------------------
CCFLAGS= -O2
CCC=g++
#----------------------------------------------------------------------
ALP=DIRECTORY_WHERE_YOU_PUT_ALP
INCPATH= -I$(ALP)/src
LIBPATH= -L. -LDIRECTORY_WHERE_YOU_PUT_THE_LIB
LIB= -lalp -lmps -lgmp -llapack -lblas -lg2c -lm -lincres -lmixvol
#----------------------------------------------------------------------
.SUFFIXES: .ex .cc
.cc.ex:
	$(CCC)  $(CCFLAGS) $(INCPATH) $(LIBPATH) -o $*.ex  $*.cc $(OBJ) $(LIB)
#----------------------------------------------------------------------
```

You can use it as follows:

```
make fich.ex
```

which will compile the file `fich.cc`, with the rule `.cc.ex` of `Makefile`.

Bibliography

[1] L.A. Aizenberg and A. M. Kytmanov. Multdimensional analogues of newtons formulas for systems of nonlinear algebraic equations and some of their applications. *Trans. from Sib. Mat. Zhurnal*, 22(2):19–39, 1981.

[2] C. Bajaj, T. Garrity, and J. Warren. On the applications of multi-equational resultants. Technical Report 826, Purdue Univ., 1988.

[3] L.M. Balbes, S.W. Mascarella, and D.B. Boyd. A perspective of modern methods in computer-aided drug design. *Reviews in Computational Chemistry*, 5:337–379, 1994.

[4] S. Basu, R. Pollack, and M.-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. In *Proc. IEEE Symp. Foundations of Computer Science*, Sante Fe, New Mexico, 1994.

[5] E. Becker, J.P. Cardinal, M.F. Roy, and Z. Szafraniec. Multivariate Bezoutians, Kronecker symbol and Eisenbud-Levin formula. In L. González-Vega and T. Recio, editors, *Algorithms in Algebraic Geometry and Applications*, volume 143 of *Prog. in Math.*, pages 79–104. Birkhäuser, Basel, 1996.

[6] C. A. Berenstein and A. Yger. Effective Bezout identities in $\mathbb{Q}[z_1, \ldots, z_n]$. *Acta. Math.*, 166:69–120, 1991.

[7] C.A. Berenstein, R. Gay, A. Vidras, and A. Yger. *Residue Currents and Bezout Identities*, volume 114 of *Prog. in Math.* Birkhäuser, 1993.

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

[9] E. Bézout. *Théorie Générale des Équations Algébriques.* Paris : Ph.-D. Pierres, 1779.

[10] D. Bini. Numerical computation of polynomial zeros by means of aberth's method. *Numerical Algorithms*, 1997. (to appear).

[11] D. Bini and B. Mourrain. Polynomial test suite, 1998. `http://www.inria.fr/saga/POL/` .

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

[13] D. Bondyfalat, B. Mourrain, and V.Y. Pan. Controlled iterative methods for solving polynomial systems. In O. Gloor, editor, *Proc. ISSAC'98*, pages 252–259. NewYork, ACM Press., 1998.

[14] J. Canny. *The Complexity of Robot Motion Planning.* M.I.T. Press, Cambridge, Mass., 1988.

[15] J. Canny. Improved algorithms for sign determination and existential quantifier elimination. *The Computer Journal*, 36(5):409–418, 1993.

[16] 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 673 of *Lect. Notes in Comp. Science*, pages 89–104. Springer-Verlag, 1993.

[17] J.P. Cardinal. *Dualité et algorithmes itératifs pour la résolution de systèmes polynomiaux.* PhD thesis, Univ. de Rennes, 1993.

[18] 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.

[19] J.P. Cardinal and B. Mourrain. Algebraic approach of residues and applications. In J. Renegar, M. Shub, and S. Smale, editors, *Proc. AMS-SIAM Summer Seminar on Math. of Numerical Analysis, (Park City, Utah, 1995)*, volume 32 of *Lectures in Applied Math.*, pages 189–210. Providence, American Mathematical Society Press, 1996.

[20] A.L. Chistov and D.Y. Grigoryev. *Complexity of Quantifier Elimination in the Theory of Algebraically Closed Fileds.* Lect. Notes Comp. Sci. 176. Springer-Verlag, New York, 1984.

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

[22] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandervoorde, and T. Veldhuizen. Generative Programming And Active Libraries. In *Dagstuhl Seminar on Generic Programming*, volume TBA of Lecture Notes in Computer Science, 1998.

[23] J.W. Demmel, J.R. Gilbert, and Xiaoye S. Li. *SuperLU Users' Guide*, 1997. `http://www.netlib.org/scalapack/prototype/`.

[24] P. Dietmaier. The Stewart-Gough platform of general geometry can have 40 real postures. In *ARK*, pages 7–16, Strobl, 1998.

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

[26] M. Elkadi and B. Mourrain. Approche effective des résidus algébriques. Rapport de Recherche 2884, INRIA, 1996.

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

[28] M. Elkadi and B. Mourrain. Algorithms for residues and Lojasiewicz exponents. *J. of Pure and Applied Algebra*, 2000. To appear.

[29] I.Z. Emiris and B. Mourrain. Polynomial system solving; the case of a 6-atom molecule. Rapport de Recherche 3075, INRIA, Dec. 1996.

[30] I.Z. Emiris and B. Mourrain. Computer algebra methods for studying and computing molecular conformations. *Algorithmica, Special Issue on Algorithms for Computational Biology*, 25:372–402, 1999.

[31] I.Z. Emiris and B. Mourrain. Matrices in Elimination Theory. *J. of Symbolic Computation*, 28(1&2):3–44, 1999.

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

[33] N. Fitchas, M. Giusti, and M. Smietanski. Sur la complexité du théorème des zéros. In J. Gudatt, editor, *Proc. of the second. Int. Conf. on Approximation and Optimization*, Peter Lang Verlag, pages 274–329, La Habana, 1993.

[34] A. Fronville, O. Devillers, M. Teillaud, and B. Mourrain. Algebraic Methods and Arithmetic Filtering for Exact Predicates on Circle Arcs. In *Proc. 16th ACM Symp. on Comp. Geometry*, Hong Kong, Chine, 2000.

[35] G.H. Golub and C.F. Van Loan. *Matrix computations*. John Hapkins, Univ. Press, Baltimore, Maryland, 3^{rd} edition, 1996.

[36] T. Granlund. *GNU Multiple Precision Arithmetic Library*. TMG Datakonsult, 1996. `http://www.swox.com/gmp/` .

[37] D.Y. Grigoryev and N.N. Vorobjov. Solving systems of polynomial inequalities in subexponential time. *J. Symbolic Computation*, 5, Special Issue on Decision Algorithms for the Theory of Real Closed Fields:37–64, 1988.

[38] J. Harris. *Algebraic Geometry, a first course*, volume 133 of *Graduate Texts in Math.* New-York, Springer-Verlag, 1992.

[39] C.M. Hoffmann. *Geometric and Solid Modeling.* Morgan Kaufmann, 1989.

[40] Roldan Pozo Jack Dongarra, Andrew Lumsdaine and Kar in A. Remington. *IML++ Iterative Methods Library.* Oak Ridge National Laboratory and The University of Ten nessee, University of Notre Dame, National Institute of Standards and Technology, 1996. `http://math.nist.gov/iml++/`.

[41] E. Kaltofen. Challenges of symbolic computation: May favorite open problems. *J.S.C.*, 29:891–919, 200.

[42] A.G. Khovanskii. Newton polyhedra and the genus of complete intersections. *Funktsional'nyi Analiz i Ego Prilozheniya*, 12(1):51–61, Jan.–Mar. 1978.

[43] E. Kunz. *Kähler differentials.* Advanced lectures in Mathematics. Friedr. Vieweg and Sohn, 1986.

[44] A.G. Kushnirenko. Newton polytopes and the Bézout theorem. *Funktsional'nyi Analiz i Ego Prilozheniya*, 10(3), Jul.–Sep. 1976.

[45] D. Lazard. Generalized Stewart Platform: How to compute with rigid motions? In *IMACS Symp. on Symbolic Computation*, pages 85–88, Lille, 1993.

[46] D. Manocha and J. Demmel. Algorithms for intersecting parametric and algebraic curves II: Multiple intersections. *Graphical Models and Image Proc.*, 57(2):81–100, 1995.

[47] B. Mourrain. The 40 generic positions of a parallel robot. In M. Bronstein, editor, *Proc. ISSAC*, ACM press, pages 173–182, Kiev (Ukraine), July 1993.

[48] B. Mourrain. Enumeration problems in Geometry, Robotics and Vision. In L. González and T. Recio, editors, *Algorithms in Algebraic Geometry and Applications*, volume 143 of *Prog. in Math.*, pages 285–306. Birkhäuser, Basel, 1996.

[49] B. Mourrain. Computing isolated polynomial roots by matrix methods. *J. of Symbolic Computation, Special Issue on Symbolic-Numeric Algebra for Polynomials*, 26(6):715–738, Dec. 1998.

[50] B. Mourrain. A new criterion for normal form algorithms. In M. Fossorier, H. Imai, Shu Lin, and A. Poli, editors, *Proc. AAECC*, volume 1719 of *LNCS*, pages 430–443, 1999.

[51] B. Mourrain and V.Y. Pan. Multivariate polynomials, duality and structured matrices. *J. of Complexity*, 16(1):110–180, 2000.

[52] B. Mourrain and H. Prieto. The ALP reference manual. Technical report, INRIA, 2000.

[53] B. Mourrain and Ph. Trébuchet. Solving projective complete intersection faster. *Proc. ISSAC*, pages 231–238, 2000.

[54] D.R. Musser and A. Saini. *STL tutorial and reference guide : C++ programming with the standard template library.* Addison-Wesley, 1996.

[55] Roldan Pozo and Andrew Lumsdaine Karin A. Remington. *SparseLib++ Sparse Matrix Class Library.* National Institute of Standards and Technology, Univers ity of Notre Dame, 1996. `http://math.nist.gov/sparselib/`.

[56] M. Raghavan and B. Roth. Solving polynomial systems for the the kinematic analysis of mechanisms and robot manipulators. *ASME J. of Mechanical Design*, 117(2):71–79, 1995.

[57] J. Renegar. On the computational complexity and geometry of the first order theory of reals (I, II, III). *J. Symbolic Computation*, 13(3):255–352, 1992.

[58] R. Robson. *Using the STL : the C++ standard template library.* Springer-Verlag, 1998.

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

[60] O. Ruatta. *Méthodes matricielles pour les systèmes d'équations polynomiales et application à la robotique et à la chimie.* PhD thesis, MDFI, Jun 1999.

[61] Y. Saad. *Iterative methods for sparse linear systems.* Series in Computer Science. PWS, 1996.

[62] J. Sabia and P. Solerno. Bounds for traces in complete intersections and degrees in the Nullstellenstaz. *AAECC-6*, 948:353–376, 1995.

[63] G. Scheja and U. Storch. Über Spurfunktionen bei vollständigen Durschnitten. *J. Reine Angew Mathematik*, 278:174–190, 1975.

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

[65] H.J. Stetter. Principles of numerical polynomial algebra. In *Proc. Workshop Symbolic on Symbolic-Numeric Algebra for Polynomials (SNAP-96)*, Sophia-Antipolis, France, July 1996.

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

[67] Iain S. Duff Timothy A. Davis. *UMFPACK Unsymmetric-pattern Multifrontal Package.* University of Florida, 1998. `http://www.netlib.org/linalg/umfpack.tar.gz`.

[68] T. Veldhuizen and D. Cannon. Active Libaries: Rethinking the roles of compilers and libraries. In *SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*, 1998.

[69] J. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford Univ. Press, London, 1965.

[70] P. Zimmermann. *GNU Multiple Precision Floating-Point Reliable Library.* Loria, Nancy, 1999. `http://www.loria.fr/projets/mpfr/` .

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399