The ALP reference manual

B. Mourrain & H. Prieto

June 7, 2000

Contents

1 A framework for symbolic and numeric computations **2**
- 1.1 Containers . . . 2
- 1.2 Views . . . 3
- 1.3 Modules . . . 3
- 1.4 Template expression . . . 4

2 A tutorial **7**

3 Basic data structures **16**

4 Arithmetic **17**
- 4.1 Complex numbers . . . 17
- 4.2 Big numbers . . . 18
- 4.3 Modular numbers . . . 21
- 4.4 Algebraic closure . . . 22
- 4.5 Real field . . . 22

5 Linear Algebra **23**
- 5.1 Vectors . . . 23
  - 5.1.1 Introduction . . . 23
  - 5.1.2 Implementation . . . 23
  - 5.1.3 Containers for Vectors . . . 25
  - 5.1.4 An example of containor . . . 26
  - 5.1.5 How to use them . . . 27
- 5.2 The module `VECTOR` . . . 28
- 5.3 Dense matrices . . . 30
  - 5.3.1 Introduction . . . 30
  - 5.3.2 Implementation . . . 30
  - 5.3.3 Containers for dense matrices . . . 35
  - 5.3.4 The standard container . . . 35
  - 5.3.5 The container for the connection with Lapack . . . 36
  - 5.3.6 How to use them . . . 40
- 5.4 The module `MATRIX` . . . 43
- 5.5 The module `LINALG` . . . 44

5.6 Structured matrices . . . 45
5.6.1 Introduction . . . 45
5.6.2 Implementation . . . 45
5.6.3 A container for Toeplitz matrices . . . 47
5.6.4 A container for Hankel matrices . . . 48
5.6.5 How to use them . . . 49
5.7 Sparse matrices . . . 51
5.7.1 Implementation . . . 51
5.7.2 Containers for sparse matrices . . . 53
5.7.3 A standard container for sparse matrices . . . 53
5.7.4 Container for the connection with SuperLU . . . 53
5.8 The linear algebra package . . . 54
5.8.1 Fast Fourier Transform . . . 54
5.8.2 Triangular decomposition . . . 54
5.8.3 Solving linear systems . . . 55
5.8.4 Determinants . . . 55
5.8.5 Rank . . . 56
5.8.6 Eigenvectors and eigenvalues . . . 57
5.8.7 Exemple . . . 58

6 Univariate Polynomial **60**
6.1 Dense univariate polynomials . . . 60
6.1.1 Implementation . . . 60
6.1.2 Containers for univariate polynomials . . . 63
6.1.3 Example of container . . . 64
6.1.4 Example . . . 64
6.2 Quotient of univariate polynomials . . . 66
6.2.1 Implementation . . . 66
6.2.2 Containers for quotiented polynomials . . . 68
6.2.3 Example . . . 68
6.3 The module `UPOLYNOMIAL` . . . 70
6.4 Solving univariate polynomials . . . 71
6.4.1 Aberth method . . . 71
6.4.2 Sebastio a Sylva method . . . 72
6.4.3 Exclusion methods . . . 75
6.5 Sylvester and Bezout matrices . . . 76
6.5.1 Introduction . . . 76
6.5.2 Implementation . . . 76

7 Multivariate polynomials **78**
7.1 Monomials . . . 78
7.1.1 Implementation . . . 78
7.1.2 Containers for exponents . . . 80
7.1.3 Monomial orders . . . 80
7.2 Multivariate polynomials . . . 81
7.2.1 Implementation . . . 81

7.2.2 Containers for multivariate polynomials . . . 83
7.2.3 Example . . . 83
7.3 The module MPOLYNOMIAL . . . 85
7.4 Multivariate Resultants . . . 85
7.4.1 Sylvester-like matrices . . . 87
7.4.2 Projective resultant . . . 87
7.4.3 Toric resultant . . . 90
7.4.4 Bezoutian resultant . . . 91
7.5 Solving polynomial systems by matrix computations . . . 92
7.5.1 Solving from the operators of multiplication . . . 93
7.5.2 Solving from matrix of univariate polynomials . . . 95
7.5.3 Computing the matrix of multiplication from resultant matrices . . . 96
7.5.4 An example based on resultant matrices . . . 97
7.6 Solving by controlled iterative methods . . . 98
7.6.1 The classical power method and inverse power method . . . 98
7.6.2 Implicit iterative method . . . 99
7.6.3 Results . . . 100

8 Distributed computations **102**
8.1 Using PVM . . . 102
8.1.1 Getting started . . . 102
8.1.2 The PVM console . . . 103
8.1.3 Programming in C/C++ with PVM . . . 104
8.1.4 What is the class Parallel? . . . 104
8.1.5 How to use it? . . . 104
8.1.6 The main concept . . . 104
8.1.7 A first exemple . . . 105
8.1.8 Implementation . . . 106

A How to install the library **109**
A.1 Get the files by ftp . . . 109
A.2 Installing the script alp . . . 110
A.3 How to use it . . . 111
A.4 Using a makefile . . . 111

Chapter 1

A framework for symbolic and numeric computations

The objective of the library ALP[1] is to provide a framework for symbolic and numeric computations. It 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 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. 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 fortan 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**.

1.1 Containers

They specify the internal representation of the objects. They provide methods or functions for accessing, creating, transforming this representation and thus

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

depend 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 [?], 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 LAPACK library, `sparselu<C>` for the connection to SUPERLU, ...

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) {...}
```

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` 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.

Chapter 2

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 [?] 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 use the command `alp`:

```
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 3 or to [?], [?].

Step 2: linear algebra

The main categories of objects for doing linear algebra are

- `VectStd`: Standard dense vectors (5.1.2),
- `MatDense`: Dense matrices (5.3.2),
- `MatStruct`: Structured matrices (5.6.2),
- `MatSparse`: Sparse matrices (5.7.1).

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>` (5.1.4) 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 5.1.2 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>` (5.3.4) which extends `array1d<C>`, by specifying the number of rows and columns, but we prefer the `lapack<C>` cointainer (5.3.5) 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 5.3.2). 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 if 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 `extttalp` in order to get an executable file `.ex`:

```
1   #include "linalg.H"
2
3   typedef VectStd<array1d<double> >   Vect;
4   typedef MatDense<lapack<double> >   Mat;
5   typedef Mat::col_type               col_type;
6
7   int main(int argc, char** argv) {
8
9     Vect V(3,"1 -2 1");                cout<<V<<endl;
10    Mat A(3,3,"1 2 3 4 5 6 7 8 9 10"); cout<<A<<endl;
11    Vect W(A*V);                       cout<<W<<endl;
12    Mat B(A * A);                      cout<<B<<endl;
13    Mat C(-A); C += B + A*2;           cout<<C<<endl;
14    Col(A,1)-= Col(A,0)*2;             cout<<A<<endl;
15    W = Eval(Col(A,0));                cout<<W<<endl;
16    Col(A,0)=Eval(Col(A,1));           cout<<A<<endl;
17    Mat D(2,2,"1 0 2 0");
18    Col(A,0)=Col(D,0)*10;              cout<<A<<endl;
19    W[Range(0,1)]=Eval(W[Range(1,2)]); cout <<W<<endl;
20    Col(A,0)[Range(0,1)]=Eval(Col(A,1)[Range(1,2)]); cout <<A<<endl;
21
22  }
```

Step 3: univariate polynomials

Univariate polynomials are derived from the vectors. They have a multiplication, a degree and are printed using the variable `x` (6.1.1). 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 `arra1d` by storing the degree. The coefficients are of type `Z<31>` that is integers modulo 31 (4.3). 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 6.1.1):

```
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>`[1].

Next is another example where we use big integers, defined from the file `bignum.h` (4.2):

```
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
   +(-19225174782095530740314432 96)*x^3
   +(10827576861722625150704513999916)*x^2
   +(-20327015669035171107600087000151776)*x
   +(12720241876172641933421003452296326504)
```

Here is the complete file for this section:

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

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

```
4
5   typedef UPolyDense<upar<Z<31> > >      Pol;
6   typedef UPolyDense<upar<Scl<MPZ> > >   ZPol;
7
8   int main(int argc, char** argv) {
9
10    Pol p("x^3-34*x+23;");                      cout<<p<<endl;
11
12    Pol q = p*p;                                cout<<q<<endl;
13    p+= Pol("x+1;");  Pol r= q%p;               cout<<r<<endl;
14
15    cout <<q(Z<31>(1))<<endl;
16
17    ZPol a("x^2-1243432232*x+2334345435434;"); cout<<a<<endl;
18    ZPol b=a^3;                                 cout<<b<<endl;
19  }
```

Step 4: multivariate polynomials

Multivariate monomials are defined by a type `C` for the coefficients and a type `E` for the exponents (7.1.1):

```
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 (7.2.1):

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

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

```
1   #include "mpoly.H"
2
3   typedef Monom<double, dynamicexp<'x'> > Mon;
4   typedef MPoly<list<Mon>, Dlex<Mon> >    Pol;
5
6   int main (int argc, char **argv)
7   {
8      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 7.1.2).

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 7.1.3).

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 7.2.1.

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
```

Troubleshooting

Things which are not allowed for the moment:

- Avoid

  ```
  p = p*p; p*=p; b = A*b; ..
  ```

  Instead, use

  ```
  p = Pol(p*p); p *= Pol(p); b = Vect(A*b); ..
  ```

  which make a copy of the right hand side, in order to avoid problems when assigning a value to the left hand side, which could depend on this left hand side.

- ...

How to read the documentation

- For the interesting classes, the signature of the associated methods are displayed.
- For the interesting functions, their exact signature is also displayed.
- Not all the functions are documented.
- The name of the file where these functions can be found (in the `src` directory of `ALP`) is printed on the right-hand side.

Chapter 3

Basic data structures

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 [**?**], for more details.

Chapter 4

Arithmetic

The arithmetic classes provide at least the following signatures:

```
self_t(int n);

self_t& self_t::operator += (const self_t&);
self_t& self_t::operator -= (const self_t&);
self_t& self_t::operator *= (const self_t&);
self_t& self_t::operator /= (const self_t&);

self_t operator - (const self_t& x);
self_t operator +(const self_t& x, const self_t& y);
self_t operator -(const self_t& x, const self_t& y);
self_t operator *(const self_t& x, const self_t& y);
self_t operator /(const self_t& x, const self_t& y);

bool operator ==(const self_t& x, const self_t& y);
bool operator !=(const self_t& x, const self_t& y);
```

4.1 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)
```

4.2 Big numbers

They are based on GMP developed by T. Granlund [?], MPFR developed by P. Zimmermann [?], 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("10002323254545452323223434211111114343434355555"),
     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;
```

$$pf - qf = 0.10002323254545452 32e46$$

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

$$-qf = 0.1000232325454545232e46$$

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

$$pf/qf = 0.391403766564095 1799e41$$

```
//Big integers
BZ pz("10002323254545452323223434211111114343434355555"),
   qz(25555),rz;
cout<<"pz   = "<<pz<<endl;
```

$$pz = 10002323254545452323223434211111114343434355555$$

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

$$qz = 25555$$

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

$$pz * qz = 2556093707699090341199748612649445270464649560 8025$$

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

$$pz + qz = 10002323254545452323223434211111114343434611 10$$

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

$$pz - qz = 10002323254545452323223434211111114343434100 00$$

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

$$-qz = 10002323254545452323223434211111114343434100 00$$

```
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;
```

$$rz* = (pz*qz-qz\%pz) = 713150144448046205194729862929195230459637226750913 00$$

```
//Big rationals
BQ pq("10002323254545452323223434211111114343434355555"),
   qq(25555),rq;

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

$$pq/ = qq = 2000464650909090464644686842222222868686871111/5111$$

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

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

```
}
```

4.3 Modular numbers

arithm/Zp.H

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

Integers modulo p. The number p should be a prime number, otherwise inversion and exponentiation may not be valid.

```
{
private:
  T rep;

public:
  Z(): rep(0) {}
  Z(int n): rep(n%p){}
  Z(char * s) {
    Scl<MPZ> tmp(s);
    tmp %= Scl<MPZ>(p);
    rep=mpz_get_si(&tmp.z);
    rep%=p;
  }
  Z(const Z<p,T> & z): rep(z.rep) {}

  Z<p,T>&  operator= (const Z<p,T>  z) {rep=z.rep; return *this;}
  Z<p,T>&  operator+=(const Z<p,T>  z) {(rep += z.rep)%= p; return *this;}
  Z<p,T>&  operator-=(const Z<p,T>  z) {(rep -= z.rep)%= p; return *this;}
  Z<p,T>&  operator*=(const Z<p,T>  z) {(rep *= z.rep)%= p; return *this;}
  Z<p,T>&  operator/=(const Z<p,T>  z);
  bool   operator==(const Z<p,T>  z) const {return ((rep-z.rep)%p)==0;}
  bool   operator!=(const Z<p,T>  z) const {return !(*this==z);}

  friend inline
  Z<p,T> operator -(const Z<p,T> a) {return Z<p,T>(-a.rep);}
  friend inline
  Z<p,T> operator+(const Z<p,T>  a, const  Z<p,T> b){return (Z<p,T>(a.rep+b.rep));}
  friend inline
  Z<p,T> operator-(const Z<p,T>  a, const  Z<p,T> b){return (Z<p,T>(a.rep-b.rep));}
  friend inline
  Z<p,T> operator*(const Z<p,T>  a, const  Z<p,T> b){return (Z<p,T>(a.rep*b.rep));}
  friend inline
  Z<p,T> operator/(const Z<p,T>  a, const  Z<p,T> b){return (Z<p,T>(a)/=b);}

   friend
  ostream &  operator<<(ostream&  os, const Z<p,T>  & z) {return (os << z.rep);}
  friend
  istream & operator>>(istream&  is, Z<p,T> & z) {return (is >> z.rep);}

};
```

4.4 Algebraic closure

arithm/AlgClos.H

This class is a traits class used to specify the type corresponding to the algebraic closure. The complex type for a class `T` is `AlgClos<T>::TYPE`.

template <class T> struct AlgClos

By default the algebraic closure class of `T` *is* `complex<T>`.

template<class T> struct AlgClos<complex<T> >

The algebraic closure of the `complex<T>` *is the class itself.*

4.5 Real field

arithm/RealOf.H

The class `RealOf` is a traits class used to specify the type corresponding to the real closure. The real class type of a class `T` is `RealOf<T>::TYPE`.

template <class T> struct RealOf

The default value is the empty class `Warning_Value_Type_Not_Defined`.

template <> struct RealOf<double>

The real class type of `double` *is* `double`.

template <> struct RealOf<int>

The real class type of `int` *is* `int`.

template <class T> struct RealOf<complex<T> >

The real class type of `complex<T>` *is* `T`.

Chapter 5

Linear Algebra

5.1 Vectors

5.1.1 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. ...

5.1.2 Implementation

linalg/VectStd.H

template < class R> struct VectStd

The standard class of vectors.

```
{

  R rep;

  typedef typename R::size_type        size_type;
  typedef typename R::value_type       value_type;
  typedef typename R::iterator         iterator_t;
  //  typedef typename R::reverse_iterator iterator_r;
  typedef R                            rep_type;
  typedef VectStd<R>                   self_t;

  VectStd(): rep() {}
  VectStd(const R & r):rep(r){}
  VectStd(size_type s): rep(s){}
  VectStd(size_type s, value_type* t): rep(t,t+s){}
  VectStd(iterator_t b,iterator_t e): rep(b,e){}
  VectStd(size_type m, const char * str);
```

```
  // Copy constructor
  VectStd(const self_t & r): rep(r.rep) {}
  template<class S>
  VectStd(const VectStd<S> & P) {assign(*this,P);}
  template<class S>
  VectStd(const VAL<S> & P) {assign(*this,P.rep);}

  iterator_t begin()        {return rep.begin();}
  iterator_t begin()  const {return rep.begin();}
  iterator_t end()          {return rep.end();}
  iterator_t end()    const {return rep.end();}
//   iterator_r rbegin()       {return rep.rbegin();}
//   iterator_r rbegin() const {return rep.rbegin();}
//   iterator_r rend()         {return rep.rend();}
//   iterator_r rend()   const {return rep.rend();}

  // Assignement
  self_t & operator =(const self_t & V){rep=V.rep; return *this;}
  template<class S>
  self_t & operator=(const VectStd<S> & V) {assign(*this,V); return *this;}
  template<class S>
  self_t & operator=(const VAL<S> & V) {assign(*this,V.rep); return *this;}

  self_t & operator+=(const self_t & v);
  self_t & operator-=(const self_t & v);
  self_t & operator*=(const value_type & c);
  self_t & operator/=(const value_type & c);

  value_type & operator[] (size_type i)       {return rep[i];}
  value_type   operator[] (size_type i) const {return rep[i];}

  VectStd<seq1d<value_type,iterator_t> > operator[](const Range & I);

  size_type size() const {return rep.size();}

};
```

The `value_type` is the type of the coefficients. The `size_type` is the type of the indices used to access the elements of the vectors. The `iterator_t` is the type of the forward iterators on the vectors. The internal representation type `R` is `rep_type`.

template <class R> inline VAL<OP<'+',VectStd<R>,VectStd<R> > > operator+(const VectStd<R> & a, const VectStd<R> & b)

Addition operator.

template <class R> inline VAL<OP<'-',VectStd<R>,VectStd<R> > > operator-(const VectStd<R> & a, const VectStd<R> & b)

Substraction operator.

template <class R> inline VAL<OP<'-',VectStd<R>,VectStd<R> > > operator-(const VectStd<R> & a)

Unary substraction operator.

template <class R> inline VAL<OP<'.',VectStd<R>,typename VectStd<R>::value_type> > operator*(const VectStd<R> & a, const typename VectStd<R>::value_type & c)

Scalar multiplication on the right.

template <class R> inline VAL<OP<'.',VectStd<R>,typename VectStd<R>::value_type> > operator*(const typename VectStd<R>::value_type & c,const VectStd<R> & a)

Scalar multiplication on the left.

template <class R,class C> inline VAL<OP<'/',VectStd<R>,C> > operator/(const VectStd<R> & a, const C & b)

Scalar division on the right.

template<class R> ostream & operator<<(ostream & os, const VectStd<R> & V)

Output operator.

template<class R> inline istream & operator>>(istream & is, VectStd<R> & V)

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

linalg/VectStd.C

template <class R> VectStd<R>::VectStd(size_type m, const char * str): rep(m)

Constructor for a size `m` *and a string containing the elements. Example:* `VectStd(3,"1 2 3")`.

5.1.3 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.

5.1.4 An example of containor

linalg/array1d.H

template < class C> struct array1d

Unidimensional array of coefficients with a size.

```
{

  typedef C            value_type;
  typedef unsigned int size_type;
  typedef C*           iterator;
  typedef C*           const_iterator;

  size_type size_;
  C * tab_;

  array1d() : size_(0), tab_(0) {}
  array1d(const array1d<C> & v);
  array1d(size_type i);
  array1d(size_type i, C* t);
  array1d(C* b, C* e);
  array1d(size_type, const char * nm);

  void reserve(size_type i);
  array1d<C> & operator=(const array1d<C> & v);
  array1d<C> & operator=(array1d<C> * v);

  size_type size() const {return size_;}
```

```
  iterator        begin()       { return iterator(tab_); }
  const_iterator  begin() const { return const_iterator(tab_); }

  iterator        end()         { return iterator(tab_+size_); }
  const_iterator  end()   const { return const_iterator(tab_+size_); }

  inline       C & operator[] (size_type i)
    {return tab_[i];}
  inline const C & operator[] (size_type i) const
    {return tab_[i];}
  // No test {if(i<size_) return tab_[i]; else return C(0);}

  ~array1d () { delete[] tab_;}
};
```

5.1.5 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.
}
```

5.2 The module VECTOR

linalg/VECTOR.H

Generic implementations for vectors are gathered in the namespace . 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.

5.3 Dense matrices

5.3.1 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, vectors, matrices ...

5.3.2 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` *(5.4).*

```
{
  R rep;

  typedef typename R::size_type     size_type;
  typedef typename R::value_type    value_type;
  typedef typename R::row_iterator  row_iterator;
  typedef typename R::col_iterator  col_iterator;

  typedef VectStd<seq1d<value_type,row_iterator> > row_type;
  typedef VectStd<seq1d<value_type,col_iterator> > col_type;

  typedef R                         rep_type;
  typedef MatDense<R>               self_t;

  MatDense():rep() {}
  MatDense(const R & r): rep(r) {}
```

```
  MatDense(size_type m, size_type n): rep(m,n){}
  MatDense(size_type m, size_type n, const value_type & c);
  MatDense(size_type m, size_type n, value_type * t, ByRow);
  MatDense(size_type m, size_type n, value_type * t, ByCol);
  MatDense(size_type m, size_type n, const char * str, ByRow);
  MatDense(size_type m, size_type n, const char * str, ByCol);

  // Copy constructors.
  MatDense(const self_t & M): rep(M.rep) {}
  template<class T>
  MatDense(const VAL<T> & M) {assign(*this,M.rep);}

  //Assignement operators
  self_t & operator=(const self_t & M) {rep=M.rep; return *this;}
  template<class T>
  self_t & operator=(const VAL<T> & M) {assign(*this,M.rep); return *this;}

  //Number of rows and columns
  size_type nbrow() const {return rep.nbrow();}
  size_type nbcol() const {return rep.nbcol();}

  value_type & operator() (size_type i, size_type j)       {return rep(i,j);}
  value_type   operator() (size_type i, size_type j) const {return rep(i,j);}

  VAL<OP<'s',self_t,Range2d> > operator()(Range I, Range J);

  //Inplace arithmetic operations
  self_t & operator+=(const self_t & a);
  self_t & operator-=(const self_t & a);
  self_t & operator*=(const value_type & c);
  self_t & operator/=(const value_type & c);
  self_t & operator*=(const self_t& a);

  // Rows and columns operations.
  self_t & swapRow(size_type i, size_type j);
  self_t & swapCol(size_type, size_type);
  self_t & permRow(size_type *pivot);
  self_t & permCol(size_type *pivot);
  self_t & addRow(size_type i,size_type j, const value_type & c = 1);
  self_t & addCol(size_type i,size_type j, const value_type & c = 1);
  self_t & multRow(size_type i, const value_type & c = 1);
  self_t & multCol(size_type i, const value_type & c = 1);

  // Inplace transposition.
  void transpose();

};
```

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`.

template < class R> inline VAL< OP< '+',MatDense< R> ,MatDense< R> > > operator+(const MatDense< R> & a, const MatDense< R> & b)

Addition operator.

template < class R> inline VAL< OP< '-',MatDense< R> ,MatDense< R> > > operator-(const MatDense< R> & a, const MatDense< R> & b)

Substraction operator.

template < class R> inline VAL< OP< '-',MatDense< R> ,MatDense< R> > > operator-(const MatDense< R> & a)

Unary substraction operator.

template < class R> inline VAL< OP< '*',MatDense< R> ,MatDense< R> > > operator*(const MatDense< R> & a, const MatDense< R> & b)

Matrix multiplication operator.

template < class R,class S> inline VAL< OP< '*',MatDense< R> ,VectStd< S> > > > operator*(const MatDense< R> & a, const VectStd< S> & b)

Matrix-vector multiplication operator.

template < class R> inline VAL< OP< '.',MatDense< R> ,typename MatDense< R> ::value_type> > operator*(const MatDense< R> & a, const typename MatDense< R> ::value_type & c)

Right scalar multiplication operator.

template < class R> inline VAL< OP< '.',MatDense< R> ,typename MatDense< R> ::value_type> > operator*(const typename MatDense< R> ::value_type & c, const MatDense< R> & a)

Left scalar multiplication operator.

template < class R> inline VAL< OP< '/',MatDense< R> ,MatDense< R> > > operator/(const MatDense< R> & a, const MatDense< R> & b)

Scalar division operator (on the right).

template < class R> inline VAL< OP< '^',MatDense< R> ,int> > operator^(const MatDense< R> & a, int i)

Power operator.

template < class R> inline typename MatDense< R> ::row_type Row(const MatDense< R> & a, int i)

Row of index `i` of a matrix (local view).

template <class R> inline VAL<OP<'s',MatDense<R>,Range2d> > Row(const MatDense<R> & a, const Range & r)

Submatrix corresponding to the rows of indices in the range `r`.

template <class R> inline typename MatDense<R>::col_type Col(const MatDense<R> & a, int i)

Column of index `i` *of a matrix (local view).*

template <class R> inline VAL<OP<'s',MatDense<R>,Range2d> > Col(const MatDense<R> & a, const Range & r)

Submatrix corresponding to the columns of indices in the range `r`.

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

Output operator for standard matrices.

template<class R> inline istream & operator>>(istream & is, MatDense<R> & M)

Input operator for standard matrices. The input format is of the form `m n c00 c01 ...` *where* `m` *is the number of rows,* `n` *is the number of columns and the coefficients are read by rows.*

linalg/MatDense.C

template<class R> MatDense<R>::MatDense(size_type m, size_type n, value_type* t, ByRow):rep(m,n)

Construct a `m` *by* `n` *matrix from an array of coefficients. These coefficients are read row by row.*

template<class R> MatDense<R>::MatDense(size_type m, size_type n, value_type* t, ByCol):rep(m,n)

Construct a `m` *by* `n` *matrix from an array of coefficients. These coefficients are read column by column.*

template<class R> MatDense<R>::MatDense(size_type m, size_type n, const value_type & c): rep(m,n)

Construct a `m` *by* `n` *matrix with entries all equal to* `c`.

template<class R> MatDense<R>::MatDense(size_type m, size_type n, const char* str, ByRow): rep(m,n)

Construct a `m` *by* `n` *matrix from a string containing the coefficients of the matrix. These coefficients are read row by row.*

template<class R> MatDense<R>::MatDense(size_type m, size_type n, const char* str, ByCol): rep(m,n)

Same as above but the coefficients are read column by column.

template< class R> inline VAL< OP< 's',MatDense< R> ,Range2d> > MatDense< R> ::operator()(Range I, Range J)

Submatrix from row and column ranges.

template< class R> inline MatDense< R> & MatDense< R> ::operator+=(const MatDense< R> & a)

Inplace addition.

template< class R> inline MatDense< R> & MatDense< R> ::operator-=(const MatDense< R> & a)

Inpace substraction.

template< class R> MatDense< R> & MatDense< R> ::operator*=(const typename MatDense< R> ::value_type & c)

Inplace multiplication by a scalar.

template< class R> MatDense< R> & MatDense< R> ::operator/=(const typename MatDense< R> ::value_type & c)

Inplace division by a scalar.

template< class R> inline MatDense< R> & MatDense< R> ::operator*=(const MatDense< R> & a)

Inplace multiplication by a matrix.

template< class R> MatDense< R> & MatDense< R> ::swapRow(const typename MatDense< R> ::size_type i, const typename MatDense< R> ::size_type j)

Swap rows `i` *and* `j`.

template< class R> MatDense< R> & MatDense< R> ::swapCol(const typename MatDense< R> ::size_type i, const typename MatDense< R> ::size_type j)

Swap columns `i` *and* `j`.

template< class R> MatDense< R> & MatDense< R> ::addRow(const typename MatDense< R> ::size_type i, const typename MatDense< R> ::size_type j, const typename MatDense< R> ::value_type & c)

Add `c` *times row* `j` *into row* `i`.

template< class R> MatDense< R> & MatDense< R> ::addCol(const typename MatDense< R> ::size_type i, const typename MatDense< R> ::size_type j, const typename MatDense< R> ::value_type & c)

Add `c` *times column* `j` *into column* `i`.

template< class R> MatDense< R> & MatDense< R> ::multRow(const typename MatDense< R> ::size_type i, const typename MatDense< R> ::value_type & c)

Multiply by `c` *the row* `i`.

template< class R> MatDense< R> & MatDense< R> ::multCol(const typename MatDense< R> ::size_type i, const typename MatDense< R> ::value_type & c)

Multiply by `c` *the column* `i`.

template < class T> inline void MatDense< T> ::transpose(void)

Inplace transposition.

5.3.3 Containers for dense matrices

The container `R` should provide the following definitions or methods:

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

R(size_type i, size_type j, value_type* t);
R(size_type i, size_type j, value_type* t, char c);
R(size_type s, Alloc);
R(size_type s, C *t);
R(iterator b, iterator e);
void R::reserve(size_type m, size_type m);

      value_type & operator() (size_type i, size_type j);
const value_type & operator() (size_type i, size_type j) const;

size_type nbrow() const;
size_type nbcol() const;
```

Possible containers are `array2d` (5.3.4) and `lapack` (5.3.5).

5.3.4 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>` *.*

{

```
typedef C*                                col_iterator;
typedef stride_iterator<C,C*,size_type> row_iterator;
```

```
  size_type nbrow_, nbcol_;

  array2d(): array1d<C>(){}
  array2d(const array2d<C> & v):
      array1d<C>(v), nbrow_(v.nbrow_), nbcol_(v.nbcol_) {}
  array2d(size_type i,size_type j=1):
      array1d<C>(i*j), nbrow_(i),nbcol_(j) {}
  array2d(size_type i, size_type j, C* t):
      array1d<C>(i*j,t), nbrow_(i), nbcol_(j) {}
  array2d(size_type i, size_type j, C* t, char c);
  array2d(const char * nm);

  void reserve(size_type i,size_type j);

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

  C &       operator()(size_type i, size_type j) {return tab_[i+nbrow_*j];}
  const C & operator()(size_type i, size_type j) const {return tab_[i+nbrow_*j];}

  size_type nbrow() const {return nbrow_;}
  size_type nbcol() const {return nbcol_;}

};
```

5.3.5 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.*

```
{
  unsigned int lda_;

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

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

```
  void reserve(unsigned int i,unsigned int j);

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

linalg/Lapack.C

integer LINALG::dcmp_qr(lapack< double> & A)

Compute the QR decomposition of the matrix A, without pivoting. The operation is done inplace. The output matrix is upper-triangular.

integer LINALG::dcmp_qr(lapack< double> & A, double * tau)

Compute the QR decomposition of the matrix `A`*. The operation is done inplace, without pivoting. The matrix Q is represented by the lower part of* `A` *and the coefficients in* `tau`*. Use the* LAPACK *routine* `dgeqrf`*.*

integer LINALG::dcmp_qr (lapack< double> & A, unsigned int * pivot, double * tau)

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

integer LINALG::dcmp_lu(lapack< double> & A)

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

integer LINALG::dcmp_lu(lapack< double> & A, lapack< double> & U)

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

integer LINALG::dcmp_lu(lapack< double> & A, integer *pivot)

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

template < class C> C LINALG::triang(Gauss m, lapack< C> & A)

Compute a triangulation of A, based on the LU decomposition of the matrix `A`*. Return the determinant, which is the product of terms on the diagonal.*

void LINALG::solve(LU m, array1d< double> & x, const lapack< double> & A, const array1d< double> & b)

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

void LINALG::solve(LU m, lapack< double> & X, const lapack< double> & A)

Solve the system $AX = B$ where `B` *is a matrix by LU decomposition of the matrix* `A`*. Use the* LAPACK *routine* `dgesv`*.*

void LINALG::solve_ls(lapack< double> & A, array1d< double> & b)

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

void LINALG::solve_ls(array1d< double> & x, const lapack< double> & A, const array1d< double> & b)

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

void LINALG::svd(array1d< double> & x,const lapack< double> & A)

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

void LINALG::svd(array1d< double> & x, const lapack< double> & A, lapack< double> & U)

Compute the Singular values of the matrix, calling LAPACK *subroutine* `dgesvd`*. The result, put in* `x`*, is the vector of singular values in decreasing order. The matrix* `A` *is copied. The singular left vectors are stored in* `U`*.*

void LINALG::svd(array1d< double> & S, const lapack< double> & A, lapack< double> & U, lapack< double> & V)

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

void LINALG::svdx(array1d< double> & S, const lapack< double> & A, lapack< double> & U, lapack< double> & V)

The same but we have `A=U` $\mathtt{V}^t$*.*

void LINALG::eignval(array1d< complex< double> > & V, const lapack< double> & A)

Compute the eigenvalues of `A`*, put them in the complex vector* `V`*. Use the* LAPACK *routine* `dgeev`*. The matrix* `A` *is copied.*

void LINALG::eignval(array1d< double> & V, const lapack< double> & A, Real)

Compute the real eigenvalues of `A` *and put them in the real vector* `V`*. Use the* LAPACK *routine* `dgeev`*. The matrix* `A` *is copied.*

void LINALG::eignval(array1d< complex< double> > & V, const lapack< double> & A, array1d< double> & error)

Compute the eigenvalues of `A`*, put them in the complex vector* `V` *and store in* `error[i]` *a bound on the error on the eigenvalue* `V[i]`*. Use the* LAPACK *routine* `dgeevx`*. The matrix* `A` *is copied.*

void LINALG::eignval(array1d< complex< double> > & V, const lapack< double> & A, const lapack< double> & B)

Compute the eigenvalues of the pencil $A - z\,B$ *and put it in the complex vector* `V`*. Use the* LAPACK *routine* `dgegv`*. The matrices* `A` *and* `B` *are copied.*

void LINALG::eignval(array1d< double> & V, const lapack< double> & A, const lapack< double> & B, Real real)

Compute the real eigenvalues of the pencil $A - z\,B$ *and put it in the complex vector* `V`*. Use the* LAPACK *routine* `dgegv`*. The matrices* `A` *and* `B` *are copied.*

void LINALG::leigvct(lapack< complex< double> > & E, const lapack< double> & A)

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

void LINALG::leigvct(lapack< complex< double> > & E, array1d< complex< double> > & V, const lapack< double> & A)

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

void LINALG::reigvct(lapack< complex< double> > & E, const lapack< double> & A)

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

void LINALG::reigvct(lapack< double > & E, const lapack< double> & A, const Real &)

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

void LINALG::reigvct(lapack< complex< double> > & E, array1d< complex< double> > & V, const lapack< double> & A)

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

void LINALG::reigvct(lapack< double> & E, const lapack< double> & A, const lapack< double> & B, const Real & real, unsigned int M)

Compute the real eigenvectors of the pencil $A - z\,B$ and keep only their m first coordinates. The matrices `A` *and* `B` *are copied.*

void LINALG::reigvct(lapack< double> & E, const lapack< double> & A, const lapack< double> & B, const Real &, array1d< double> & V)

Compute the real eigenvectors of the pencil $A - z\,B$. Put the eigenvalues in `V`*. The matrices* `A` *and* `B` *are copied.*

template< class C> typename lapack< C>::size_type LINALG::rank(const lapack< C> & M)

Compute the rank of `M`*, using a QR-decomposition (use the routine* `dgeqpf`*). The matrix* `M` *is copied.*

template< class C> typename lapack< C>::size_type LINALG::rank(const lapack< C> & M, typename lapack< C>::size_type * permut_row, typename lapack< C>::size_type * permut_col)

Compute the rank of `M`*, using a QR-decomposition (use the routine* `dgeqpf`*. The matrix* `M` *is copied. The permutation of rows and columns needed to get a leading square matrix of maximal rank are stored in* `permut_row` *and* `permut_col`*.*

integer LINALG::inv_tr(lapack< double> & A, char s, char u)

Inplace inversion of triangular matrices. The value of `s` *should be* `'L'` *for lower or* `'U'` *for upper triangular matrix. The value of* `u` *should be* `'U'` *for unitary (implicitly 1 on the diagonal) or* `'N'` *for the default. Return an integer which is* 0 *if the computation runs correctly.*

5.3.6 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]]);
```

```
};
```

5.4 The module MATRIX

linalg/MATRIX.H

It contains generic implementations on matrices, that can be used to build new linear algebra classes. The following signatures or definitions must at least be available for the type R:

```
typedef typename size_type;    //type of the indices.
typedef typename value_type;   //type of the coefficients.

size_type  R::nbrow();
size_type  R::nbcol();
value_type R::operator()(size_type i, size_type j);
```

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

Output function for general matrices. It is called by the output operator << of the classes of matrices.

template<class R> inline istream & MATRIX::Read(istream & is, R & M)

Input function for general matrices. The input format is of the form `m n c00 c0 ...` *where* `m` *is the number of rows,* `n` *is the number of columns and the coefficients are read row by row.*

template<class R,class I> inline void MATRIX::reserve(R & a, I m, I n)

Reserve in `a` *the place to store a* $m \times n$ *matrix.*

template<class R,class I, class C> inline void MATRIX::reserve(R & a, I m, I n, const C & c)

Reserve in `a` *the place to store a* $m \times n$ *matrix and set all its entries to* `c`.

template < class R1, class R2> inline void MATRIX::setequalto(R1 & r, const R2 & m)

Assignement of a dense matrix to another one, not necessarily of the same type.

template< class R> inline typename R::value_type MATRIX::trace(const R & M)

Trace of a matrix.

template< class R,class S> inline void MATRIX::plus(R & a, const S & b)

Inplace addition of matrices.

template< class R,class S> inline void MATRIX::plus(R & a, const S & b, const S & c)

Addition of matrices of the same size. The required space should be allocated for `a`.

template< class R,class S> inline void MATRIX::minus(R & a, const S & b)

Inplace substraction of matrices of the same size. The required space should be allocated for `a`.

template< class R> inline void MATRIX::minus(R & a, const R & b, const R & c)

Substraction of matrices of the same size. The required space should be allocated for `a`.

template< class R,class S,class T> inline void MATRIX::mult(R & a, const S & b, const T & c)

Multiplication of matrices. The required place should be allocated in `a`.

template< class R,class S> inline void MATRIX::mult(R & a, const S & b)

Inplace multiplication of matrices.

template< class V,class M,class T> inline void MATRIX::mult_vect(V & a, const M & b, const T & v)

Multiplication by a vector. The vector `a` *should be first initialised with the correct size and zero entries.*

template < class R> void MATRIX::transpose(R & M)

Inplace transposition.

5.5 The module `LINALG`

linalg/LINALG.C

template< class Vect, class Mat> void LINALG::svdx(Vect & S, const Mat & A, Mat & Us, Mat & V)

Singular value decomposition of the matrix `A`. *The singular values are output in* `S`. *In output, the matrix* `Us` *is the product of an orthogonal matrix by the diagnonal matrix of singular values,* `V` *is an orthogonal matrix and we have* `A = Us * Transpose(V)`. *The matrix* `A` *is not modified but copied first in* `Us`. *If* `A` *is an* $m \times n$ *matrix,* `Us` *is* $m \times m$ *and* `V` *is* $n \times n$.
The necessary memory space must be allocated *in* `S` *and* `V`, *before calling this function:* `S` *should be of size* `Max(A.nbrow(),A.nbcol())` *and* `V` *of size* `(A.nbcol(),A.nbcol())`. *The matrix* `Us` *need not be preallocated, for it is assigned to a copy of* `A` (`Us=A`).
It requires the functions or operators `Norm`, `sqrt`, `+,-,*,/, <, =` *for the coefficient field.*

template < class R> typename R::value_type LINALG::triang(Bareiss m,R & M)

Bareiss method. The operations are performed inplace. The output value is the last element on the diagonal. If the matrix is invertible, it's its determinant.

template< class R> typename R::size_type LINALG::rank(const R & M)

Rank. Based on svd computations.

5.6 Structured matrices

5.6.1 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 (5.8.1) and [**?**].

5.6.2 Implementation

linalg/MatStruct.H

template< class R> struct MatStruct

Structured matrices.

```
{

  R rep;
```

```
  typedef typename R::size_type  size_type;
  typedef typename R::value_type value_type;
  typedef MatStruct<R>           self_t;

  MatStruct() {}
  MatStruct(const R  & r): rep(r)                  {}
  MatStruct(int m, int n) : rep(n,m)               {}
  MatStruct(int m, int n, value_type* t):rep(m,n,t){}
  MatStruct(const char * str): rep(str)            {}

  //Copy constructor
  MatStruct(const self_t & M): rep(M.rep)          {}
  template<class T>
  MatStruct(const VAL<T> & M) {assign(*this,M.rep);}

  //Assignement
  self_t & operator=(const self_t & M)  {rep=M.rep; return *this;}
  template<class T>
  self_t & operator=(const VAL<T> & M)  {assign(*this,M.rep); return *this;}

  size_type nbrow() const {return rep.nbrow();}
  size_type nbcol() const {return rep.nbcol();}

  value_type   operator() (size_type i, size_type j) const {return rep(i,j);}

  VAL<OP<'s',self_t,Range2d> > operator()(Range I, Range J);

  self_t & operator+=(const self_t & a);
  self_t & operator-=(const self_t & a);
  self_t & operator*=(const value_type & c);

  self_t & transpose() {rep.transpose(); return *this;}

};
```

No modification operator `(i,j)` is implemented.

template < class R> inline VAL< OP< '+',MatStruct< R> ,MatStruct< R> > > operator+(const MatStruct< R> & a, const MatStruct< R> & b)

Addition operator.

template < class R> inline VAL< OP< '-',MatStruct< R> ,MatStruct< R> > > operator-(const MatStruct< R> & a, const MatStruct< R> & b)

Substraction operator.

template < class R> inline VAL< OP< '-',MatStruct< R> ,MatStruct< R> > > operator-(const MatStruct< R> & a)

Unary substraction operator.

template < class R,class S> inline VAL< OP< '*',MatStruct< R> ,VectStd< S> > > operator*(const MatStruct< R> & a, const VectStd< S> & b)

Matrix vector mutiplication.

template < class R> inline VAL< OP< '.',MatStruct< R> ,typename MatStruct< R> ::value_type> > operator*(const MatStruct< R> & a, const typename MatStruct< R> ::value_type & c)

Scalar multiplication.

template < class R> inline VAL< OP< 's',MatStruct< R> ,Range2d> > Row(const MatStruct< R> & a, const Range & r)

Submatrix from a range of rows.

template < class R> inline VAL< OP< 's',MatStruct< R> ,Range2d> > Col(const MatStruct< R> & a, const Range & r)

Submatrix from a range of columns.

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

Output operator for standard matrices.

template< class R> inline istream & operator> > (istream & is, MatStruct< R> & M)

Input operator for standard matrices. The input format is of the form `m n c00 c0 ...` *where* `m` *is the number of rows,* `n` *is the number of columns and the coefficients are read by rows.*

linalg/MatStruct.C

5.6.3 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 [**?**]). 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 (5.8.1) and [**?**].

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 (5.8.1).*

```
{

  unsigned int nbrow_, nbcol_;

  typedef C           coeff_t;
  typedef array1d<C>  row_t;
  typedef array1d<C>  col_t;

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

  void reserve(unsigned int i, unsigned int j);

  C   operator()(unsigned int i, unsigned int j) const
        {return tab_[i-j+nbcol_-1];}

  void transpose() ;

  size_type nbrow() const {return nbrow_;}
  size_type nbcol() const {return nbcol_;}
};
```

5.6.4 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 (5.8.1) and [?].

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 (5.8.1).* .

```
{

  unsigned int nbrow_, nbcol_;

  typedef C           coeff_t;
  typedef array1d<C>  row_t;
  typedef array1d<C>  col_t;

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

  void reserve(unsigned int i, unsigned int j)
    {
      this->array1d<C>::reserve(i+j-1); nbrow_ = i; nbcol_=j;
    }

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

  size_type nbrow() const {return nbrow_;}
  size_type nbcol() const {return nbcol_;}

  void transpose() { swap(nbcol_,nbrow_);};
```

5.6.5 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;
```

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

```
};
```

5.7 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 [**?**]). 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.

5.7.1 Implementation

linalg/MatSparse.H

template< class R> struct MatSparse

Sparse matrices.

```
{
  R rep;

  typedef typename R::size_type  size_type;
  typedef typename R::value_type value_type;
  typedef MatSparse<R>          self_t;

  MatSparse() {}
  MatSparse(const R  & r): rep(r)               {}
  MatSparse(int m, int n, int nz) : rep(n,m,nz) {}
  MatSparse(int m, int n, int nz, value_type* t, size_type* idx):
    rep(m,n,nz,t,idx) {}

  MatSparse(const char * str): rep(str)         {}
  MatSparse(const self_t & M): rep(M.rep)       {}
  template<class T>
  MatSparse(const VAL<T> & M) {assign(*this,M.rep);}

  self_t &  operator=(const self_t & M)
    {rep=M.rep; return *this;}
  template<class T>
  self_t & operator=(const VAL<T> & M) {assign(*this,M.rep); return *this;}

  self_t & transpose () {rep.transpose(); return *this;}

  size_type nbrow() const {return rep.nbrow();}
  size_type nbcol() const {return rep.nbcol();}
  size_type nzero() const {return rep.nzero();}

  value_type operator() (size_type i, size_type j) const {return rep(i,j);}

  self_t & operator*=(const value_type & c);
  self_t & operator/=(const value_type & c);
};
```

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

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

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

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

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

Output operator for standard matrices.

linalg/MatSparse.C

5.7.2 Containers for sparse matrices

The container `R` should provide the following methods or definitions:

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

size_type R::nbrow();
size_type R::nbcol();
size_type R::nbnz();

      value_type & R::operator()(size_type,size_type);
const value_type & R::operator()(size_type,size_type) const;

void   R::transpose();
```

5.7.3 A standard container for sparse matrices

John-John Ili & Cédric Peyruqeou

linalg/sparse2d.H

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

5.7.4 Container for the connection with SuperLU

Ph. Trebuchet

linalg/superlu.H

template< class C> struct superlu : public SuperMatrix

Container for the SuperLU package [?], 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());
```

```
{
  typedef unsigned int  size_type;
  typedef double        value_type;
  superlu();
  superlu(const int nrow, const int ncol);
  superlu(const int nrow, const int ncol, const int nz);
  superlu(size_type nrow, size_type ncol, size_type nz,
          value_type* coeffs, size_type *idx);
```

```
  ~superlu();

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

  inline void destroystore();
  size_type nbrow() const {return nrow;}
  size_type nbcol() const {return ncol;}
  void reserve(size_type, size_type, size_type);
};
```

5.8 The linear algebra package

5.8.1 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)

Shuffle 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.*

5.8.2 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 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`.

5.8.3 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`.

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`.

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.

5.8.4 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:

$$a_{i,j}^{(k)} = \frac{a_{i,j}^{(k-1)} a_{k,k}^{(k-1)} - a_{i,k}^{(k-1)} a_{k,j}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}} \tag{5.1}$$

  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 (5.1) 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);
```

5.8.5 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.*

5.8.6 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.*

5.8.7 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;
```

$$>> RankRank(A) = 4$$

```
//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 6

Univariate Polynomial

6.1 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 defined in section 4.

6.1.1 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 &)` *correspond 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`.

```
{

  typedef typename R::value_type       value_type;
  typedef typename R::size_type        size_type;
  typedef R                            rep_type;
  typedef typename R::iterator         iterator;
  typedef typename R::reverse_iterator reverse_iterator;
  typedef UPolyDense<R>                self_t;

  R rep;

  UPolyDense() : rep() {}
```

```
UPolyDense(const R & r) : rep(r) {}
UPolyDense(const int n);
UPolyDense(size_type n, Alloc A) {VECTOR::reserve(this->rep,n);}
UPolyDense(value_type c, size_type n);
UPolyDense(size_type n, value_type * t): rep(n,t) {}
UPolyDense(iterator b, iterator e): rep(b,e) {}

UPolyDense(char *);
UPolyDense(const self_t & P) : rep(P.rep) {TRACE<self_t,'C'>(P);}

template<class S>
UPolyDense(const VAL<S> & P) {assign(*this,P.rep);}

iterator begin()        {return rep.begin();}
iterator begin()  const {return rep.begin();}
iterator end()          {return rep.end();}
iterator end()    const {return rep.end();}
reverse_iterator rbegin()       {return rep.rbegin();}
reverse_iterator rbegin() const {return rep.rbegin();}
reverse_iterator rend()         {return rep.rend();}
reverse_iterator rend()   const {return rep.rend();}

self_t & operator=(const self_t & P)
  {TRACE<self_t,'='>(P);rep=P.rep; return *this;}

template<class S>
self_t & operator =(const VAL<S> & M)
       {assign(*this,M.rep);return *this;}

bool operator==(int n) const;
bool operator!=(int n) const {return !(*this==n);}

self_t & operator+= (const self_t  & P);
self_t & operator-= (const self_t  & P);
self_t & operator*= (const self_t  & P);
self_t & operator*= (const value_type & c);
self_t & operator/= (const self_t  & Q);
self_t & operator/= (const value_type & c);
self_t & operator%= (const self_t  & Q);

int   degree()       { return (Degree(rep));}
int   degree() const { return (Degree(rep));}
size_type size() const { return (rep.size());}

inline       value_type &  operator[] (size_type i)
  {
    if(i< max((size_type)(degree()+1),(size_type)0))
    return rep[i]; else return *new value_type(0);return rep[i];
  }
inline       value_type    operator[] (size_type i) const
```

```
    {
     if(i< max((size_type)(degree()+1),(size_type)0))
      return rep[i]; else return 0;
    }
  template<class T> T operator()(const T & c);

};
```

template < class R> int Degree(const UPolyDense< R> & p)

Output the degree, that is the index of the last non-zero coefficient.

template < class R> inline VAL< OP< '+',UPolyDense< R> ,UPolyDense< R> > > operator+(const UPolyDense< R> & a, const UPolyDense< R> & b)

Addition of polynomials.

template < class R> inline VAL< OP< '-',UPolyDense< R> ,UPolyDense< R> > > operator-(const UPolyDense< R> & a, const UPolyDense< R> & b)

Substraction of polynomials.

template < class R> inline VAL< OP< '*',UPolyDense< R> ,UPolyDense< R> > > operator*(const UPolyDense< R> & a, const UPolyDense< R> & b)

Product of polynomials.

template < class R> inline VAL< OP< '.',UPolyDense< R> ,typename UPolyDense< R> ::value_type> > operator*(const UPolyDense< R> & a, const typename UPolyDense< R> ::value_type & c)

Scalar multiplication.

template < class R> inline VAL< OP< '/',UPolyDense< R> ,UPolyDense< R> > > operator/(const UPolyDense< R> & a, const UPolyDense< R> & b)

Quotient of polynomials.

template < class R> inline VAL< OP< '%',UPolyDense< R> ,UPolyDense< R> > > operator%(const UPolyDense< R> & a, const UPolyDense< R> & b)

Remainder of polynomials.

template < class R> inline VAL< OP< '^',UPolyDense< R> ,unsigned int> > operator^(const UPolyDense< R> & a, const unsigned int & n)

Power of a polynomial.

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

Output operator.

template < class R> inline istream & operator> > (istream & is, UPolyDense< R> & P)

Input operator. The variable is `x` *or* `x0`*. Ended by a* `;`*.*

template < class R> UPolyDense< R> ::UPolyDense(char * s)

Constructor of a univariate polynomial from a string, ending by `;`. *Example:* `UPolyDense< R> ("x^2+x^3-1;")`.

template< class R> inline bool UPolyDense< R> ::operator==(int n) const

Equality test to 0,1,...

template< class R> inline UPolyDense< R> & UPolyDense< R> ::operator/= (const typename UPolyDense< R> ::value_type & c)

Inplace division by a scalar.

template < class R> inline UPolyDense< R> & UPolyDense< R> ::operator /= (const UPolyDense< R> & Q)

Inplace quotient in the Euclidian division.

template < class R> inline UPolyDense< R> & UPolyDense< R> ::operator %= (const UPolyDense< R> & Q)

Inplace remainder in the Euclidian division.

template < class R> template< class T> inline T UPolyDense< R> ::operator()(const T & c)

Evaluation of a polynomial, using Horner scheme. The output value is specified by the type of `c`.

6.1.2 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 (6.1.3) and vector of the STL can be used here.

6.1.3 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>.

```
{

  typedef C*                   iterator;
  typedef reverse_iterator<C*> const_reverse_iterator;
  typedef reverse_iterator<C*> reverse_iterator;

  int degree_;

  upar(): array1d<C>(), degree_(-1) {}
  upar(size_type s, Alloc): array1d<C>(s), degree_(s-1) {}
  upar(size_type s, C *t): array1d<C>(s,t), degree_(s-1)
    {UPOLYNOMIAL::checkdegree(*this);}
  upar(C* b, C *e): array1d<C>(b,e)
    {degree_=size_-1;UPOLYNOMIAL::checkdegree(*this);}
  upar(const upar<C> & p): array1d<C>(p), degree_(p.degree_) {}

  void reserve(size_type n)
    {this->array1d<C>::reserve(n);degree_=n-1;}

  iterator          begin()       { return iterator(tab_); }
  iterator          begin() const { return iterator(tab_); }
  iterator          end()         { return iterator(tab_+degree_+1); }
  iterator          end()   const { return iterator(tab_+degree_+1); }
  reverse_iterator  rbegin()      { return reverse_iterator(tab_+degree_+1); }
  reverse_iterator  rbegin() const{ return reverse_iterator(tab_+degree_+1); }
  reverse_iterator  rend()        { return reverse_iterator(tab_); }
  reverse_iterator  rend()   const{ return reverse_iterator(tab_); }

  upar<C> & operator=(const upar<C> & v)
    {this->array1d<C>::operator=(v); degree_=v.degree_; return *this;}

  upar<C> & operator=(upar<C> * r)
    {this->array1d<C>::operator=(r); degree_ = r.degree_;return *this;}
};
```

6.1.4 Example

```
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;
```

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

```
}
```

6.2 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 (5.8.1)). See [?] for more details.

6.2.1 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.*

```
{

  static R poly;

  typedef typename R::value_type       value_type;
  typedef typename R::size_type        size_type;
  typedef R                            rep_type;
  typedef typename R::iterator         iterator;
  typedef typename R::reverse_iterator iterator_r;
  typedef UPolyQuot<R>                 self_t;

  UPolyQuot() {}
  UPolyQuot(const R & r): UPolyDense<R>(r) {}
  UPolyQuot(int n): UPolyDense<R>(n){}
  UPolyQuot(size_type n, Alloc A): UPolyDense<R>(n,A){}
  UPolyQuot(size_type n, value_type * t, bool mbs = false):
    UPolyDense<R>(n,t) {} //{UPOLYNOMIAL::montohorn(*this,*this,mbs);}
  UPolyQuot(iterator b, iterator e, bool mbs =true):
    UPolyDense<R>(b,e) {}

  //  UPolyQuot(const char * str): UPolyDense<R>(str) {}
  UPolyQuot(const UPolyQuot<R> & P):  UPolyDense<R>(P.rep) {}
  template <class T>
  UPolyQuot(const VAL<T> & P) {assign(*this,P.rep);}

  // Returns the quotient dimension
  static size_type dim() {return Degree(poly);}

  UPolyQuot<R> & operator=(const UPolyQuot<R> & P)
    {rep=P.rep; return *this;}

  template<class T>
  UPolyQuot<R> & operator=(const VAL<T> & P)
    {assign(*this,P.rep); return *this;}
};
```

template < class R> inline VAL< OP< '+',UPolyQuot< R> ,UPolyQuot< R> > > operator+(const UPolyQuot< R> & a, const UPolyQuot< R> & b)

Addition operator.

template < class R> inline VAL< OP< '-',UPolyQuot< R> ,UPolyQuot< R> > > operator-(const UPolyQuot< R> & a, const UPolyQuot< R> & b)

Substraction operator.

template < class R> inline VAL< OP< '-',UPolyQuot< R> ,UPolyQuot< R> > > operator-(const UPolyQuot< R> & a)

Unary susbtraction operator.

template < class R> inline VAL< OP< '*',UPolyQuot< R> ,UPolyQuot< R> > > operator*(const UPolyQuot< R> & a, const UPolyQuot< R> & b)

Product of polynomials.

template <class R> inline VAL<OP<'.',UPolyQuot<R>,typename UPolyQuot<R>::value_type> > operator*(const UPolyQuot<R> & a, const typename UPolyQuot<R>::value_type & c)

Scalar multiplication.

template <class R> inline VAL<OP<'/',UPolyQuot<R>,UPolyQuot<R> > > operator/(const UPolyQuot<R> & a, const UPolyQuot<R> & b)

Quotient of polynomials.

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

Output operator.

template <class R> inline istream & operator>>(istream & is, UPolyQuot<R> & P)

Input operator. The variable is `x` *or* `x0`*. Ended by* `;`*.*

upoly/UPolyQuot.C

template <class R> void UPOLYNOMIAL::quo_square(R & x,const R & f)

Square `f` *in* $\mathcal{A}$ *and put it* `x`*.*

template <class R> void UPOLYNOMIAL::quo_mult(R & x, const R & a, const R& b)

Multiply `a` *by* `b` *in* $\mathcal{A}$ *and put it* `x`*.*

template <class R> void UPOLYNOMIAL::montohorn(R &x, const R & y, bool MtoH)

Express the polynomial x from a basis (Monomial or Horner) in the other basis. From the Horner basis to the monomial basis (`MtoH=false`*), the Bezoutian matrix* $B_{1,p}$ *is used. For the other case (*`MtoH=true`*), the matrix* H_{τ_p} *is used.*

6.2.2 Containers for quotiented polynomials

The container `R` should provide the same signatures as for dense univariate polynomials. It can be `upar` (6.1.3) for instance. The container `vector` of the stl can also be used.

6.2.3 Example

```
#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;
```

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

```
}
```

6.3 The module UPOLYNOMIAL

upoly/UPOLYNOMIAL.H

It contains generic implementations on univariate (dense) polynomials. This set of functions apply when R provides the following methods or definitions:

```
typename R::value_type;
typename R::size_type;
typename R::iterator;
typename R::reverse_iterator;

iterator_t R::begin();
iterator_t R::end();
value_type R::operator[](int);
```

template <class R> inline void UPOLYNOMIAL::plus(R & r, const R & a, const R & b)

Addition of two polynomials. r *should be allocated to the correct size and with 0 entries.*

template <class R> inline void UPOLYNOMIAL::minus(R & r, const R & a, const R & b)

Substraction of two polynomials. r *should be allocated to the correct size and with 0 entries.*

template <class R,class A, class B> inline void UPOLYNOMIAL::mult_index(R & r, const A & a, const B & b)

Naive multiplication of two polynomials using indices. r *should be allocated to the correct size and with 0 entries.*

template <class R> inline void UPOLYNOMIAL::mult(R & r, const R & a, const R & b)

Naive multiplication of two polynomials using iterators. r *should be allocated to the correct size and with 0 entries.*

template <class R> inline void UPOLYNOMIAL::mult_index(R & a, const R & b)

Naive inplace multiplication of a *by* b *using indices. The result is put in* a*. A copy of* a *is made*

template <class R> inline void UPOLYNOMIAL::mult(R & a, const R & b)

Naive inplace multiplication of a *by* b *using iterators. The result is put in* a*. A copy of* a *is made*

template <class R,class C> inline void UPOLYNOMIAL::mult_scalar(R & r, const R & a, const C & c)

Multiplication by a scalar.

template < class C, class R> inline C UPOLYNOMIAL::eval(const R & p, const C & c)

The value of a polynomial `p` *at* `x`*, using Horner scheme and reverse iterators.*

template< class R> inline void UPOLYNOMIAL::checkdegree(R & w)

Function called at the end of an arithmetic operation to readjust a degree variable if it exists.Its default value is .

6.4 Solving univariate polynomials

6.4.1 Aberth method

D. Bini & G. Fiorentino

This is a connection to the C-implementation by D. Bini and G. Fiorentino for solving univariate polynomials. See [?] 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;
```

$$(1) * x^{30} + (-465) * x^{29} + (103385) * x^{28} + (-14631225) * x^{27} + (1480321269) * x^{26} + (-114009431445) * x^{25}$$
$$+(6949189247325) * x^{24} + (-344092707928125) * x^{23} + (14097793282984515) * x^{22} + (-484338676679532675) * x^{21} + (1409025752422308247\text{5}) * x^{20} + (-349$$
$$+(74359416261111727234855) * x^{18} + (-13605580871196332287117\text{5}) * x^{17} + (21458832493345014521397\text{75}) * x^{16} + (-29197210605623737978013\text{75}) * x^{15} +$$
$$+(30006513636556697864066736800) * x^{12} + (-22245742324696206305840307\text{6000}) * x^{11} + (14019376240868075016911422397\text{44}) * x^{10} + (-7454161471690660$$
$$+(3609307881588368128056145388789\text{76}) * x^{6} + (-85189988850542311250318425141248\text{0}) * x^{5} + (15477949752547197371117812531200\text{00}) * x^{4} + (-207079220$$
$$+(26525285981219105863630848000\text{0000})$$

```
  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, -0.258770178217928959313483264480965810 26e-61)
#2: (0.19999999999999999999999999999999999984e1, -0.46556680990471469891214099942031669987e-76)
#3: (0.40000000000000000000000000000000364749e1, -0.20535888801067914054504324258151119759e-62)
#4: (0.29999999999999999999999999999998682e1, -0.16900144205651471238650154023320713 13e-67)
#5: (0.60000000000000000000000000001030817988 5e1, -0.17069344456114038582494063531601900797e-55)
#6: (0.49999999999999999999999999997617333257e1, 0.56117336544939643771944799678521116096e-47)
#7: (0.99999999999999999999999978482696323121 2e1, 0.16770378909006438496693210140946375 49e-45)
#8: (0.90000000000000000000000032519985952991e1, -0.10509095875125750363385727455212235666e-47)
#9: (0.14000000000000000000084025537027538 28e2, 0.20154350891011587872660771523500159 94e-45)
#10: (0.69999999999999999999999999161077172e1, 0.25390980354637881208074621189882365307e-37)
#11: (0.10999999999999999999999988532508203455 8e2, -0.57235100952987539063160947223702612529e-40)
#12: (0.79999999999999999999999943143323176 99e1, -0.19467375500487617082017239633722323927e-41)
#13: (0.12999999999999999999963478192535910 34e2, 0.36353917880566144751326965843923149935e-48)
#14: (0.12000000000000000000009707954139115 16e2, 0.10465433766150597832348156110497136095e-28)
#15: (0.16000000000000000001216929698651956 16e2, 0.61282400956055226193665133857607039139e-44)
#16: (0.16999999999999999997321936500946514 59e2, 0.23090958875004680558122639586908298 55e-38)
#17: (0.17999999999999999996823830965949873 37e2, 0.16812616094710630162783072442274842416e-30)
#18: (0.15000000000000000000021899514173736 43e2, -0.74909395205002402476613060543657031 58e-32)
#19: (0.19999999999999999999296586876141711 257e2, 0.85176035248872305472653559861172840824e-31)
#20: (0.18999999999999999995297882811784419 23e2, 0.53282144207137213154959172264956273312e-40)
#21: (0.21999999999999999992973480069781637 49e2, 0.36943424133849245652733336879058978001e-37)
#22: (0.22999999999999999999287914975831207 769e2, 0.29732741191732769710887783935183503301e-41)
#23: (0.25999999999999999998320453210066317 75e2, 0.20580343010841541263617393462696565414e-34)
#24: (0.20999999999999999998631601379682898 985e2, 0.85416495423790731130368144063784408372e-38)
#25: (0.28999999999999999999995118582495120 79e2, 0.58775478356019121465377999997238523646e-29)
#26: (0.25000000000000000000182522016583431 122e2, 0.17028180858456924194557109259891572 86e-36)
#27: (0.28000000000000000000010977922438105 99e2, -0.27922288574837944256539061438507685528e-31)
#28: (0.24000000000000000000051274372584682 1381e2, -0.18359428755560974665678223574819808583e-30)
#29: (0.27000000000000000000036044721387544 108e2, 0.49609681736270557227520758047099445496e-31)
#30: (0.29999999999999999999999137095170421 25e2, 0.82974904262406279750260379614458531495e-31)
```

6.4.2 Sebastio a Sylva method

The **Sebastiao e Silva method** ([?]) 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}.

6.4.1. **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 iteration 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 we should reverse p and look for its smallest root (see [**?**]).

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 [**?**]).

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:

$$\sum_{i=0}^{n} \frac{x^i}{i!} \tag{6.1}$$

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

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** ([**?**]).

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.

6.4.3 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`*.*

6.5 Sylvester and Bezout matrices

6.5.1 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:

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

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

$$B_{f,g} = \left[\begin{array}{ccc} \theta_{0,0}^{f,g} & \cdots & \theta_{0,d-1}^{f,g} \\ \vdots & & \vdots \\ \theta_{d-1,0}^{f,g} & \cdots & \theta_{d-1,d-1}^{f,g} \end{array}\right]$$

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

6.5.2 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 7

Multivariate polynomials

7.1 Monomials

They are defined by a coefficient and an exponent.

7.1.1 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.*

```
{

  // Type definitions
  typedef int                    index_t;
  typedef typename R::exponent_t exponent_t; //typename R::coeff_t exponent_t;
  typedef C                      coeff_t;
  typedef Monom<C,R>             self_t;

  // Constructors
  Monom():coeff(0),rep(){}
  Monom(const C & c, const R & r):coeff(c),rep(r){}
  Monom(const C & c):coeff(c),rep(){} // monome constant
  Monom(int i):coeff(i){}
  Monom(int i, int d);                 // monome x_i^d
  Monom(const C & c, int i, int d =1); // monome c*x_i^d
  Monom(const C & c, int s, int *t):coeff(c), rep(s){
    for(int i=0;i<s;i++) rep.setExponent(i,t[i]);
  }
```

```
  //Copy constructor
  Monom(const Monom & m):coeff(m.coeff),rep(m.rep) {TRACE<self_t,'C'>(m);}
  template<class S>
  Monom(const VAL<S> & M)  {assign(*this,M.rep);}

  // Destructor
  // not dynamic allocation at this level

  // Assignement
  self_t & operator = (const self_t & m);
  template<class S>
  self_t & operator = (const VAL<S> & M)
    {assign(*this,M.rep); return *this;}

  // Accessors
  //  int size() const {return rep.size;}
  C GetCoeff () const { return coeff; }
  void SetCoeff (const C & c) { coeff = c; }
  int Lvar() const { return lvar(rep); }

  exponent_t operator[] (int i) const { return rep[i]; }
  exponent_t GetDegree  (int i) const { return rep[i]; }

  void setExponent (int i, int d) { rep.setExponent(i,d); }

  // boolean opeartors and functions
  bool operator==(int n) const;
  bool operator!=(int n) const {return !(*this==n);}

  friend bool operator == (const self_t & m1, const self_t & m2) {
    return m1.coeff == m2.coeff && m1.rep == m2.rep;
  }
  friend bool IsComparable (const self_t & m1, const self_t & m2){
    return m1.rep == m2.rep;
  }


  // Arithmetic operators
  self_t   operator  - () const {return self_t(-coeff,rep);}
  self_t & operator += (const self_t &);
  self_t & operator -= (const self_t &);
  self_t & operator *= (const self_t &);
  self_t & operator *= (const C &);
  self_t & operator /= (const C &);

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

mpoly/Monom.C

template < class C, class R> Monom< C,R> ::Monom(int i, int d):coeff(1)

Construction of the monomial x_i^d.

template < class C, class R> Monom< C,R> ::Monom(const C & c, int l, int d =1):coeff(c), rep(l+1)

Construction of the monomial $c\,x_l^d$.

7.1.2 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.

7.1.3 Monomial orders

mpoly/Plex.H

template < class M> struct Plex

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

```
{
  static bool less (const M &, const M &);
  bool operator() (const M &, const M &) const {return less(m1,m2);}
};
```

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.*

```
{
  static bool less (const M &, const M &);
  bool operator() (const M & m1, const M & m2) const {return less(m1,m2);}
};
```

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.

```
{
  static bool less(const M &, const M &);
  bool operator() (const M & m1, const M & m2) const {return less(m1,m2);}
};
```

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.

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

7.2 Multivariate polynomials

They are represented by a sorted list of monomials.

7.2.1 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.*

```
{
public:
  // Types
  typedef typename R::value_type             monom_t;
  typedef typename monom_t::coeff_t          coeff_t;
  typedef O                                  order_t;
  typedef typename R::iterator               iterator;
```

```
typedef typename R::const_iterator         const_iterator;
typedef typename R::reverse_iterator       reverse_iterator;
typedef typename R::const_reverse_iterator const_reverse_iterator;
typedef MPoly<R,O>                         self_t;

// Constructors
MPoly():rep(){}
MPoly(const monom_t & m):rep() { rep.push_back(m); }
MPoly(const coeff_t & c):rep() { if(c !=0) rep.push_back(monom_t(c));}
MPoly(const int & i): rep()    { if(i !=0) rep.push_back(monom_t(i));}
MPoly(int, const monom_t *);
MPoly(const self_t & P)
  {TRACE<self_t,'C'>(P);MPOLYNOMIAL::copy(rep, P.rep);}
MPoly(char *);

template<class S>
MPoly(const VAL<S>& M) {assign(*this,M.rep);}

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

// Assignment operators
self_t & operator =(const self_t &);
self_t & operator =(int n)             {*this=self_t(n); return *this;}
self_t & operator =(const monom_t & m) {*this=self_t(m); return *this;}
template<class S>
self_t & operator =(const VAL<S> & M)  {assign(*this,M.rep);return *this;}

self_t & operator*=(const coeff_t &);
self_t & operator/=(const coeff_t &);

self_t & operator*=(const monom_t &);
self_t & operator/=(const monom_t &);

self_t & operator+=(const self_t &);
self_t & operator-=(const self_t &);
self_t & operator*=(const self_t &);
self_t & operator/=(const self_t & Q);

// Boolean operators and functions
bool operator==(int n) const;
bool operator!=(int n) const {return !(*this==n);}

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

// Iterator functions
iterator                begin ()       {return rep.begin(); }
const_iterator          begin () const {return rep.begin(); }
```

```
  reverse_iterator       rbegin ()       {return rep.rbegin();}
  const_reverse_iterator rbegin () const {return rep.rbegin();}

  iterator                 end ()       {return rep.end(); }
  const_iterator           end () const {return rep.end(); }
  reverse_iterator        rend ()       {return rep.rend();}
  const_reverse_iterator  rend () const {return rep.rend();}

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

  coeff_t operator()(const array1d<coeff_t> & p);

  // Data
  R rep;
};
```

template <class R, class O> inline VAL<MPoly<R,O>*> Diff(const MPoly<R,O> & p, unsigned i)

Derivative of `p` *with respect to* `i`$^{\text{th}}$ *variable.*

mpoly/MPoly.C

7.2.2 Containers for multivariate polynomials

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

7.2.3 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.
}
```

7.3 The module MPOLYNOMIAL

It contains generic implementations for multivariate polynomials.

mpoly/MPOLYNOMIAL.H

template < class R, class O> typename R::iterator MPOLYNOMIAL::insert (R & p, typename R::iterator i, const typename R::value_type & m)

Insert the monomial `m` *in the sorted sequence* `p`*, from the position* `i`*. Output the position where the monomial is inserted. Use the function* `O::less` *to search where to insert* `m`*.*

template < class R, class O> inline void plus(R &r,const R &a, const R& b)

Addition of two polynomials.

template < class R, class O> inline void MPOLYNOMIAL::minus(R &r,const R &a, const R& b)

Substraction of two polynomials.

template < class R, class O> inline void mult(R &r, const typename R::const_iterator & a_b, const typename R::const_iterator & a_e, const typename R::const_iterator & b_b, const typename R::const_iterator & b_e)

Multiplication of non-zero polynomials.

template < class R> int MPOLYNOMIAL::lvar (const R & p)

Leading variable (of maximal index) of a polynomial.

template < class R> int MPOLYNOMIAL::degree (const R & p)

Degree of a polynomial.

template< class R> void MPOLYNOMIAL::div_rem(R & q, R & a, const R & b)

Divide `a` *in place by* `b`*.*

template< class R> void MPOLYNOMIAL::diff(R & r, const R & p, unsigned i)

Derivative of `p` *with respect to* $\mathtt{i}^{\text{th}}$ *variable put in* `r`*.*

7.4 Multivariate Resultants

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

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

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 [**?**]) 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 [**?**]):

Definition 7.4.1 *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 [**?**]):

Hypotheses 7.4.2

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 7.4.2, 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 [?]) to the multivariate case and the class of *Bezoutian matrices* generalizing the construction of E. Bézout (1779, see [?]) to the multivariate case.

7.4.1 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 :

$$\begin{array}{rcl} S : \mathcal{V}_0 \times \cdots \times \mathcal{V}_n & \rightarrow & \mathcal{V} \\ (q_0, \ldots, q_n) & \mapsto & \displaystyle\sum_{i=0}^{n} f_i \, q_i. \end{array} \tag{7.1}$$

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

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

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 7.5.2 are well respected.

7.4.2 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 (7.1) :

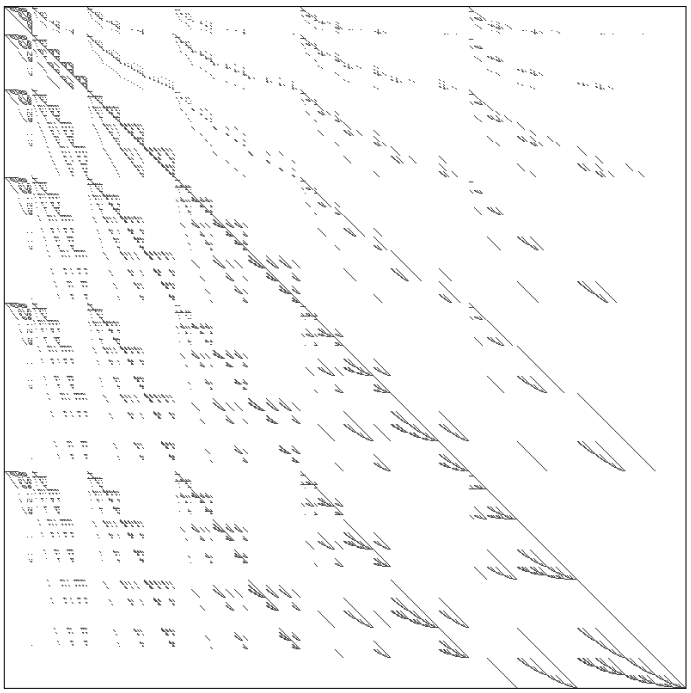
$$\begin{aligned} S : \mathcal{V}_0 \times \cdots \times \mathcal{V}_n &\rightarrow \mathcal{V} \\ (q_0, \ldots, q_n) &\mapsto \sum_{i=0}^{n} f_i \, q_i. \end{aligned} \tag{7.2}$$

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

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

In 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 7.5.2 are well respected. In this resultant construction, it may happen that for specific values of the coefficients of the input polynomials, the matrix $\mathtt{D}$ become singular. In this case another construction must be applied.

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* $\nu = \sum_{i=0}^{n} \deg(f_i) - n + 1$.

7.4.3 Toric resultant

I. Emiris

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 $Q_1, \ldots, Q_n$ 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 $MV(f_1, \ldots, f_n)$ counts the (generic) number of roots of $\mathcal{Z}(f_1, \ldots, f_n)$ in $(\overline{\mathbb{K}}^*)^n$ [**?**]. This bound is known as the BKK bound to underline the contributions of Kushnirenko and Khovanskii in its development and proof [**?**, **?**].

Theorem 7.4.3 *Let* $f_1, \ldots, f_n \in \mathbb{K}[x_1, x^{-1}, \ldots, x_n, x_n^{-1}]$ *with Newton polytopes* $Q_1, \ldots, Q_n$. *The number of isolated common zeros in* $(\overline{\mathbb{K}}^*)^n$, *multiplicities counted, is either infinite, or does not exceed* $MV(Q_1, \ldots, Q_n)$, *where* $\overline{\mathbb{K}}$ *is the algebraic closure of* $\mathbb{K}$.

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

7.4.2. **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:

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

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 [].

7.4.4 Bezoutian resultant

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

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 7.4.4 *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:*

$$\Theta(f_0, f_1, \ldots, f_n) = \Theta_f(f_0) = \det \begin{pmatrix} f_0(\mathbf{x}) & \theta_1(f_0) & \cdots & \theta_n(f_0) \\ \vdots & \vdots & & \vdots \\ f_n(\mathbf{x}) & \theta_1(f_n) & \cdots & \theta_n(f_n) \end{pmatrix} = \sum_{i,j} \theta_{i,j}^{f} \, \mathbf{x}^{u_i} \, \mathbf{y}^{v_j}, \tag{7.3}$$

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 7.4.5 *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:

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

Definition 7.4.6 *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 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* $\mathbf{x}$ *and the next* n *variables are the* $\mathbf{y}$.

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.*

7.5 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.

7.5.1 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:

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

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 [**?**]) :

Theorem 7.5.1 *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 [**?**], 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,

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

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

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

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 $i = 1, \ldots, d$ ([?], 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:

7.5.3. **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* $1, x_1, \ldots, x_n$.

7.5.2 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 $f_1 = \cdots = f_n = 0$, wher 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 7.4), 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 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 $\frac{(t_1 y + t_2)}{(t_3 y + t_4)}$.

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 complexe coefficients.*

7.5.3 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 7.5.2 *The matrix* `S` *is a square matrix and have the following form :*

$$\mathtt{S} = \begin{pmatrix} \mathtt{A} & \mathtt{B} \\ \mathtt{C} & \mathtt{D} \end{pmatrix} \tag{7.4}$$

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 [**?**]).

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 5.8.2):

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

where `S` is a matrix satisfying the hypotheses 7.5.2 (eg. a resultant matrix of a generic system).

7.5.4 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;

}
```

7.6 Solving by controlled iterative methods

7.6.1 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 [?], [?]).

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

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

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

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

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

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

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

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

7.6.2 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 [?]).

So under the hypotheses 7.5.2 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,

7.6.4. 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 `S` and its block `A`, `B`, `C`, `D` are structured matrices, structure which can be used to simplify multiplication of such a matrix by a vector (see [?]). 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.

7.6.3 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 ([?]) developped by R. Pozo and his library **SparseLib++** ([?]), the other, by $*U$ using a direct method with **UMFPACK** routines ([?]) for solving the sparse linear system $\mathtt{S}x = b$.
The examples $s22G$ and $s22U$ are both the following system ([?]) :

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

$s44*$ 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 ([?]).

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

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

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

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

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

The last result $kruppa$ corresponds to the Kruppa equations of a reconstruction problem in Computater Vision (see [?]) 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 8

Distributed computations

8.1 Using PVM

F. Garacci

PVM (Parallel Virtual Machine) is a free software package wich allow to make parallel computing, on a heterogeneous collection of computers linked by a network, in a very easy way. You will find more information about it and how to download and setup it here :"http://www.epm.ornl.gov/pvm/pvm_home.html". The main operation mode of PVM is that you can, from an execution, send the execution of an other binary file in an other computer and, keeping its ID, communicate with it. The complete list of the PVM routine for C/C++ is avaible here :"http://nacphy.physics.orst.edu/PVM/ref/alpha.html"

8.1.1 Getting started

To get started with PVM, once it's installed, you will have to complete different tasks :

1. Find out which computers are available. You can use any computer which fulfills the following requirements :
   - You have an account on it.
   - PVM is installed
2. Edit/create the file .rhosts in your home directory. Each computer that you want to use needs an entry in the .rhosts file consisting of the name of the computer and your login name on that machine, as shown here:

   ```
   penfret fgaracci
   capcoz fgaracci
   molotova fgaracci
   sounion fgaracci
   ```

```
otway fgaracci
korrigan fgaracci
cygnusx1 fgaracci
ophiucus fgaracci
```

3. Create a directory named pvm3 in your home directory in which you'll create two symbolic links : `include->/usr/local/pvm3/include/` and `lib ->/usr/local/pvm3/lib/`, and a directory named bin, then set environment variables :

   - If you use zsh then add the following lines to your .zshenv file,

     ```
     export PVM_ROOT=$HOME/pvm3
     export PVM_ARCH=‘$PVM_ROOT/lib/pvmgetarch‘
     PATH=$PATH:$PVM_ROOT/lib:$PVM_ROOT/lib/$PVM_ARCH:$PVM_ROOT/bin/$PVM_ARCH
     ```

   - If you use csh or tcsh then add the following lines to your .cshrc file,

     ```
     setenv PVM_ROOT '$HOME/pvm3'
     setenv PVM_ARCH $PVM_ROOT/lib/pvmgetarch
     set path=($path $PVM_ROOT/lib)
     set path=($path $PVM_ROOT/lib/$PVM_ARCH)
     ```

8.1.2 The PVM console

The PVM console is an interface between you and your virtual machine; here you can add a host, delete a host, see the hosts statues, start or stop a process ... For a complete list of the available commands for the PVM console see: "http://nacphy.physics.orst.edu/PVM/ref/console.html". Here are the most usefull commands :

- add `hostname` : add the host `hostname` to the virtual machine configuration.
- conf: display the virtual machine configuration.
- ps -a: show the tasks which are running for all hosts.
- reset: kill all the running tasks of the virtual machine.
- delete `hostname` : delete the host `hostename` from the virtual machine.

To start the PVM console, once your environment variables are set, just type `pvm` in a xterm. Then you will see the prompt `pvm>` wich indicate that you're in the PVM console.

8.1.3 Programming in C/C++ with PVM

I advice you to create a directory for each architecture you'll use in your virtual machine in your `$HOME/pvm3/bin` directory then you'll put the correct binary file for all your program. To get the PVM code for a computer architechture, you can use the PVM function `pvmgetarch`.

8.1.4 What is the class `Parallel`?

The class Parallel is a class wich allow to use PVM in a more easy way, using ALP objects, and with an object oriented philosophy. Each object instantied from this class symbolize a task. There are four way to create an object :

- The default constructor: create an object which correspond to the current task.
- The constructor taking the name of an executable: send the execution of the executable in any machine of the PVM configuration and create an object which will correspond to this task.
- The constructor taking the name of an executable and the name of a machine: send the execution of the executable in the specified machine of the PVM configuration and create an object which will correspond to this task.
- The constructor taking the constant integer MASTER: create an object which symbolize the parent task.

In order to send a task in a distant computer; you will just have to create a Parallel object with the good parameters and use the operators «, ». See below.

8.1.5 How to use it?

To use the class Parallel, there are two imperatives :

- PVM must be installed and configurate (see: `http://www.epm.ornl.gov/pvm/pvm_home.html` for more details).
- ALP must be installed (see: `http://www-sop.inria.fr/saga/Bernard.Mourrain/ALP/` for more details)

Then you just have to include the Parallel.H file in all your sources in which you want to use the class Parallel.

8.1.6 The main concept

Once you have create some tasks, you will have to communicate between them. This is performed trought the PVM buffer: you just have to put your data in

the buffer and send it to a target task. Then the target task have to recieve the buffer and operate on data. Be carefull: you have to take the data from the buffer in the same order that they were put in it. The operators '«' and '»' are overloaded for all the ALP object. They respectively put an object in the send buffer and take it from the recieved buffer.

8.1.7 A first exemple

This is the file Slave.C which will correspond the the slave task :

```
#include "parallel/PVM.H"
#include "linalg.H"
main()
{
  Parallel Slave;
  Parallel Master(MASTER);

  double t[]={1,2,3,4};
  VectStd<array1d<double> > V(4,t);

  Slave.BuffInit();
  Slave>>V;
  Slave.Send(Master);

  Slave.Recieve(Master);
}
```

And here is the file Master.C which will correspond the the master task :

```
#include <iostream.h>
#include "parallel/PVM.H"
#include "linalg.H"
main()
{
  Parallel Master;

  cout<<"Master "<<Master.Tid()<<" sur : "<<Master.HostName()<<"\n";

  Parallel Slave1("Test_Slave");
  cout<<"Slave1 "<<Slave1.Tid()<<" sur : "<<Slave1.HostName()<<" fils
de  "<<Slave1.TidParent()<<"\n";

  VectStd<array1d<double> > V;

  Master.Recieve(Slave1);
  Master<<V;
  cout<<"\nMaster a recu \n";
```

```
  cout<<V<<endl;
  Master.Kill(Slave1);
}
```

Here is the result of the execution of Master :

```
Master 262150 sur : molotova
Slave1 524290 sur : penfret fils de  262150

Master a recu
[1, 2, 3, 4]
```

8.1.8 Implementation

parallel/PVM.H

class Parallel

Interface for the class `Parallel` *wich allows to make parallel computation with* `ALP`.

```
{
    public:

  // Return the name of the host of this task
  char* HostName();

 // Return the identificator of the task
  int Tid();

 // Return the identificator of the parent task
  int TidParent();

  // Return the number of computer in the parallel virtual machine
  int NbHosts();


  // the following methods put an array of simple type in the active
  // sent buffer
  int Pack(int* IntArray,int Number=1, int Stride=1);
  int Pack(unsigned int* IntArray,int Number=1, int Stride=1);
  int Pack(double* DoubleArray,int Number=1, int Stride=1);
  int Pack(float* FloatArray,int Number=1, int Stride=1);
  int Pack(unsigned short* ShortArray,int Number=1, int Stride=1);
  int Pack(short* ShortArray,int Number=1, int Stride=1);
  int Pack(unsigned long* LongArray,int Number=1, int Stride=1);
  int Pack(long* LongArray,int Number=1, int Stride=1);
  int Pack(char* CharArray,int Number=1, int Stride=1);

  template <class T>
```

```
int Pack(T* ObjArray,int Number=1);

// the following methods take an array of simple type from the active
// recived buffer
int UnPack(int* IntArray,int Number=1, int Stride=1);
int UnPack(unsigned int* IntArray,int Number=1, int Stride=1);
int UnPack(double* DoubleArray,int Number=1, int Stride=1);
int UnPack(float* FloatArray,int Number=1, int Stride=1);
int UnPack(unsigned short* ShortArray,int Number=1, int Stride=1);
int UnPack(short* ShortArray,int Number=1, int Stride=1);
int UnPack(unsigned long* LongArray,int Number=1, int Stride=1);
int UnPack(long* LongArray,int Number=1, int Stride=1);
int UnPack(char* CharArray,int Number=1, int Stride=1);

template <class T>
int UnPack(T* & ObjArray,int* Number);

// Add Host to the virtual machine
int AddHosts(char *HostName);

// Add Hosts to the virtual machine
int AddHosts(char **HostName,int NbHosts);

// Add all the Hosts wich are in the .rhosts file, to the virtual machine
int AddMaxHosts();

// Delete Host to the virtual machine
int DelHosts(char *HostName);

// Delete Hosts to the virtual machine
int DelHosts(char **HostName,int NbHosts);

// Initialisation of the active send buffer
void BuffInit();

// Send the active buffer to the task designed by Target
int Send(Parallel &Target,int Tag=1);

// Receve from the task designed by Sender
int Recieve(Parallel &Sender,int Tag=1);

int Recieve(int Tag=1);

// Catch the output of the slaves tasks
int CatchOut(FILE *OutFile=stdout,int Option=14, int val=0);

// Send the signal Sig to the task target
int SendSig(Parallel &Target,int Sig);

// kill the target task
```

```
  int Kill(Parallel &Target);

  // reset the task simbolized by the calling object, with the same
  // programme on the same host
  int Rebuild();

  // reset the task simbolized by the calling object, with the
  // programme "Prog" on the same host
  int Rebuild(char* Prog);

  // reset the task simbolized by the calling object, with the same
  // programme on the host "Host"
  int Rebuild(char* Host);

  // reset the task simbolized by the calling object, with the
  // programme "Prog" on the host "Host"
  int Rebuild(char* Prog, char* Host);

  // Default constructor, correspond to the current task
  Parallel();

  // Constructor of a task, wich will execute "Prog" on anyone of the
  // available machine
  Parallel(char* Prog);

  // Constructor of a task, wich will execute "Prog" on the machine "Host"
  Parallel( char* Prog, char * Host);

  // Constructor of an object wich will symbolize the parent task
  Parallel(int Master);

  // Destructeur
  ~Parallel();

  private:

  // Name of the host
  char* hostName;
  char* prog;
  // Courrant task identification
  int tid;
  // Parent task identification
  int tidParent;
  // Number of host of the virtual machine
  int nbHosts;
  // Number of different architectures in the virtual machine
  int nbArch;
};
```

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 this 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 the `makefile`.

Bibliography

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

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

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

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

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

[6] 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.

[7] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. Intern. Symp. Applied Algebra, Algebraic Algor. and Error-Corr. Codes (Puerto Rico)*, volume 673 of *Lect. Notes in Comp. Science*, pages 89–104. Springer-Verlag, 1993.

[8] J.P. Cardinal. On two iterative methods for approximating the roots of a polynomial. In J. Renegar, M. Shub, and S. Smale, editors, *Proc. AMS-SIAM Summer Seminar on Math. of Numerical Analysis, (Park City, Utah, 1995)*, volume 32 of *Lectures in Applied Math.*, pages 165–188. Am. Math. Soc. Press, 1996.

[9] 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.

[10] 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.

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

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

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

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

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

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

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

[18] 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++/`.

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

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

[21] 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.

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

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

[24] 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/`.

[25] R. Robson. *Using the STL : the C++ standard template library.* Springer-Verlag, 1998.

[26] Y. Saad. *Iterative methods for sparse linear systems.* Series in Computer Science. PWS, 1996.

[27] 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.

[28] Iain S. Duff Timothy A. Davis. *UMFPACK Unsymmetric-pattern Multifrontal Package.* University of Florida, 1998. `http://www.netlib.org/linalg/umfpack.tar.gz`.

[29] J. Wilkinson. *The Algebraic Eigenvalue Problem.* Oxford Univ. Press, London, 1965.

[30] P. Zimmermann. *GNU Multiple Precision Floating-Point Reliable Library.* Loria, Nancy, 1999. `http://www.loria.fr/projets/mpfr/` .

Index

A

Aberth, 71
arithm/AlgClos.H, 22
arithm/RealOf.H, 22
arithm/Scl.H, 18
arithm/Zp.H, 21

B

Bezout, 76

D

Det
    Bareiss, 45, 56

E

Eigenvector, 94

I

installation, 109

L

Lapack, 36
linalg/array1d.H, 26
linalg/array2d.H, 35
linalg/Det.H, 56
linalg/Eigen.H, 57
linalg/FFT.H, 54
linalg/hankel.H, 48
linalg/Lapack.C, 37
linalg/Lapack.H, 36
linalg/LINALG.C, 45
linalg/MatDense.C, 33
linalg/MatDense.H, 30
linalg/MATRIX.H, 43
linalg/MatSparse.C, 52
linalg/MatSparse.H, 51
linalg/MatStruct.C, 47
linalg/MatStruct.H, 45
linalg/Rank.H, 56
linalg/Solve.H, 55
linalg/sparse2d.H, 53
linalg/superlu.H, 53
linalg/toeplitz.H, 47
linalg/Triang.H, 54
linalg/VECTOR.H, 28
linalg/VectStd.C, 25
linalg/VectStd.H, 23

M

mpoly/Dlex.H, 80
mpoly/dynamicexp.H, 80
mpoly/Monom.C, 80
mpoly/Monom.H, 78
mpoly/MPoly.C, 83
mpoly/MPoly.H, 81
mpoly/MPOLYNOMIAL.H, 85
mpoly/numexp.H, 80
mpoly/Plex.H, 80
mpoly/resultant/Macaulay.H, 89
mpoly/resultant/MBezout.H, 92
mpoly/resultant/Toric.H, 91
mpoly/solve/EigenHidden.H, 95
mpoly/solve/EigenMult.H, 94
mpoly/solve/EigenRslt.H, 97
mpoly/SQVlex.H, 81
mpoly/Vlex.H, 81
Multivariate polynomial, 81

P

parallel/PVM.H, 106

Q

quotient algebra, 66

R

Rank
    SVD, 45

S

Sebastiao e Silva, 72
Singular Value Decomposition, 45
Solve
    Aberth, 71
    Compagnon, 96
    Eigen, 94, 95
    LS, 55
    LU, 55
    Resultant, 97
    Sebastiao, 74
SVD, 45
Sylvester, 76

T

template expression, 4
Triang, 54
    Bareiss, 45
    Gauss Lapack, 37

U

Univariate polynomial
    dense, 60
univariate polynomial, 60, 66
`upoly/solve/Aberth.H`, 71
`upoly/solve/Exclusion.H`, 75
`upoly/solve/Sebastio.H`, 73
`upoly/UniReslt.H`, 76
`upoly/upar.H`, 64
`upoly/UPolyDense.C`, 63
`upoly/UPolyDense.H`, 60
`upoly/UPOLYNOMIAL.H`, 70
`upoly/UPolyQuot.C`, 68
`upoly/UPolyQuot.H`, 66
UPolyDense, 60
USchur, 55

V

VECTOR, 28