SYNAPS,

a library for symbolic-numeric computation

B. Mourrain & J.P. Pavone & Ph. Trébuchet & E. Tsigaridas

February 28, 2005

The need to combine symbolic and numeric computations is ubiquitous in many problems. Starting with an exact description of the equations, in most cases, we will eventually have to compute an approximation of the solutions. Even more, in many problems the coefficients of the equations may only be known with some inaccuracy (due, for instance, to measurement errors). In this case, we are not dealing with a solely system but with a neighborhood of the "exact" system and we have to take into account the continuity of the solutions with respect to the input coefficients. This leads to new, interesting and challenging questions either from a theoretical or practical point of view at the frontier between Algebra and Analysis and witnesses the emergence of new investigations.

The aim of the open source project SYNAPS

`http://www-sop.inria.fr/galaad/logiciels/synaps/index.html`

is to provide a coherent and efficient library for symbolic *and* numeric computation. It implements data-structures and classes for the manipulation of basic objects, such as (dense, sparse, structured) vectors, matrices, univariate and multivariate polynomials. It also provides fundamental methods such as algebraic number manipulation tools, different types of univariate and multivariate polynomial root solvers, resultant computations, ...

1 Structure and concepts

As SYNAPS is targeting different domains of application, it has to provide various internal representations for the abstract data-types required by algorithms specifications. Thus, this library makes it possible to define parameterized but efficient data structures for fundamental 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 parameterized types (templates in C++ parlance) which allow us to tune easily the data structure to each specific problem at hand. So-called *template expression* are used to guide the expansion of code during program translation and thus to get very efficient implementations. Traditional developments of libraries, in particular for scientific computations, are usually understood as a collection of "passive" routines. The approach taken here is inspired by a recent paradigm of software developments called *active library* and illustrated by the library STL. The software components of such libraries take active parts of the configuration and generation of codes (at compile time), which combine parameterization and efficiency. The present environment is a step in that direction, with an orientation toward algebraic data-structures (eg. polynomials).

We also carefully consider the problem of reusability of external or third-party libraries, such as LAPACK (Fortran library for numerical linear algebra), GMP (C library for extended arithmetics), or MPSOLVE (C univariate solver implementation, using extended multiprecision arithmetic). For instance LAPACK routines — can coexist with generic ones, the choice being determined by the internal representation. We also want this library to be easy to use. Once the basic objects are defined by specialization of the parameters, including maybe the optimization of some critical routines, we want to be able to write the code of an algorithm "as it is on the paper". Tools featured by the C++ programming language to support polymorphism allows us to make a great step towards achieving the goals described above.

Let us briefly present how these antagonist objectives are achieved. We distinguish several levels in the implementation of the library.

1.1 The containers and domains

The *containers* specify the internal representation. They provide methods or functions for accessing, scanning, creating, transforming this representation. Examples are: `linalg::rep1d<C>; linalg::rep2d<C>; lapack::rep2d<C>; ...` The containers are attached to *domains* (implemented as namespaces): `linalg; lapack; upol; bezier; ...` They allow to define (by an external user) specialized algorithms for given containers (through the scope name mechanism). For instance, writting the following code

```
namespace lapack {
template <class C>
   void solve(LU,linalg::rep1d<C> & x, const rep2d<C> & A,
                  const linalg::rep1d<C> & b);
}
```

one defines a specialized `solve` function for the `lapack::rep2d<C>` container.

1.2 Modules

It is a collection of *generic* implementations which apply to a family of objects (or containers), sharing common properties. These functions can thus be applied to third-party containers, provided that they satisfy some constraints, imposed on the parameters of these generic functions. These constraints define a *category* of containers. Modules can be combined or extended naturally. The main modules of the library are `VECTOR, MATRIX, UPOLDAR, MPOLDST ...`

1.3 Views

They specify how to manipulate or to see the containers as mathematical objects. The internal data is available, via the method `rep()`. They are usualy classes, parameterised by the container type `R` and sometimes by trait classes which precise the implementation:

```
template<class C, class R> struct VectDse;
```

The implementations of these views are based on modules.

We present now some specific features of the library and applications of them in geometric problems.

2 Algebraic numbers

Algebraic numbers are of particular importance in geometric problems such as arrangement or topology computation. In geometric modeling the treatment of algebraic curves or surfaces leads implicitly or explicitly to the manipulation of algebraic numbers. A package of the library is devoted to such problems. It is dealing with real algebraic numbers, which are real roots of polynomials, whose coefficients are integers. We represent such a real algebraic number, by a square-free polynomial and an isolating interval, that is an interval with rational endpoints, which contains only one root of the polynomial:

$$r \equiv (f(X), [a, b])\,, \text{where } f \in \mathbb{Z}[X] \text{and } a, b \in \mathbb{Q}$$

In order to achieve high performance for problems involving small degree polynomials (typically geometric problems for conics), the treatment of polynomials and algebraic numbers of degree up to 4, is preprocessed. We use precomputed discrimination systems (Sturm-Habicht sequences) in order to determine the square-free polynomial and to compute the isolating interval as function in the coefficients of the polynomial (and to compare algebraic numbers).

For polynomials of higher degree, we use a Sturm-like solver in order to isolate the roots of the polynomial, but we can use any other solver that can return isolating intervals.

The implementation of the algebraic numbers is in the namespace `ALGEBRAIC`. Since algebraic numbers need a lot of information concerning the ring and the field number type, the number type of the approximation etc, we gathered all this information into a struct called `ALG_STURM<RT>`, which takes only the ring number type `RT` as parameter and from this class we can determine all the other types. The class of the algebraic number which implements the real algebraic numbers is `root_of<RT>`. It provides construction functions (such as `Solve`, `RootOf`), comparison functions, sign function and extensions to bivariate problems, considered as univariate over a univariate polynomial ring.In order to compare two algebraic numbers, filtering techniques improving the numerical approximation combined with explicit methods based on Sturm's theorem are used.

The use of these tools will be illustrated on examples, issued from the computation of arrangement of arcs of curves.

3 Resultant-based methods

A projection operator is an operator which associates to an overdetermined polynomial system in several variables a polynomial depending only on the coefficients of this system, which vanishes when the system has a solution. This projection operation is a basic ingredient of many methods in EAG. It has important applications in CAGD (Computer Aided Geometric Design), such as for the problem of implicitization of parametric surfaces, or for surface parameterisation inversion, intersection, and detection of singularities of a parameterized surface. The library implements a set of resultants

Such approach based on resultant constructions yields a preprocessing step in which we generate a dedicated code for the problem we want to handle. The effective resolution, which then requires the instantiation of the parameters of the problems and the numerical solving, is thus highly accelerated. The library SYNAPS contains several types of resultants constructions, such the projective resultant, the toric resultant (based on an implementation by I. Emiris), and the Bezoutian resultant.

We illustrate the use of such tools on the following CAD modeling problem: the extraction of a set of geometric primitives properly describing a given 3D point cloud obtained by scanning a real scene. If the extraction of planes is considered as a well-solved problem, the extraction of circular cylinders, these geometric primitives are basically used to represent "pipes" in an industrial environment, is not easy and has been recently addressed. In this section, we describe an application of resultant based method to this problem which has been experimented in collaboration with Th. Chaperon from the MENSI company. It proceeds as follows: First, we devise a polynomial dedicated solver, which given 5 points randomly selected in our 3D point cloud, computes the cylinders passing through them (recall that 5 is the minimum number of points defining generically a finite number of cylinders, actually 6 in the *complex* numbers). Then we apply it for almost all (or randomly choosen) sets of 5 points in the whole point could, and extract the "clusters of directions" as a primitive cylinder. This requires the intensive resolution of thousands of small polynomial systems. Classical resultant or residual resultant constructions are used in this case, to compute quickly theirs roots, even in the presence of multiple roots. We detail the implementation of this approach, using the tools provided by the library SYNAPS.

4 Subdivision solvers

In many problems, real root-finding is restricted to a specific subdomain of $\mathbb{R}^n$. In some cases, even the existence of a solution is of importance. The library SYNAPS provides such method, based on subdivision techniques. We consider here the problem of computing the solutions of a polynomial system in a box $I := [a_1, b_1] \times \cdots \times [a_n, b_n] \subset \mathbb{R}^n$. The subdivision solver that we describe here, uses the representation of multivariate polynomials in the Bernstein basis, analysis of sign variations and univariate solvers to localise the real roots of a polynomial system. The output is a set of small-enough boxes, which may contain these roots.

By a direct extension to the multivariate case, any polynomial $f(x_1, \ldots, x_n) \in \mathbb{R}[x_1, \ldots, x_n]$ of degree d_i in the variable x_i, can be decomposed as:

$$f(x_1, \ldots, x_n) = \sum_{i_1=0}^{d_1} \cdots \sum_{i_n=0}^{d_n} b_{i_1,\ldots,i_n} \, B_{d_1}^{i_1}(x_1; a_1, b_1) \cdots B_{d_n}^{i_n} x(x_n; a_n, b_n).$$

where $(B_{d_1}^{i_1}(x_1; a_1, b_1) \cdots B_{d_n}^{i_n}(x_n; a_n, b_n))_{0 \le i_1 \le d_1, \ldots, 0 \le i_n \le d_n}$ is the tensor product Bernstein basis on the domain $I := [a_1, b_1] \times \cdots \times [a_n, b_n] \subset \mathbb{R}^n$ and $\boldsymbol{b} = (b_{i_1,\ldots,i_n})_{0 \le i_1 \le d_1, \ldots, 0 \le i_n \le d_n}$ are the control coefficients of f on I. The polynomial f is represented in this basis by the n^{th} order tensor of control coefficients $\boldsymbol{b} = (b_{i_1,\ldots,i_n})_{0 \le i \le d_1, 0 \le j \le d_2, 0 \le k \le d_3}$.

De Casteljau algorithm applies in each of the direction x_i, $i = 1, \ldots, n$ so that we can split this representation in these directions. We use the following polynomials to isolate the roots: $m_j(f; x_j) = \sum_{i_j=0}^{d_j} \min_{\{0 \le i_k \le d_k, k \ne j\}} b_{i_1,\ldots,i_n}\, B_{d_j}^{i_j}(x_j; a_j, b_j)$ and $M_j(f; x_j) = \sum_{i_j=0}^{d_j} \max_{\{0 \le i_k \le d_k, k \ne j\}} b_{i_1,\ldots,i_n}\, B_{d_j}^{i_j}(x_j; a_j, b_j)$. By the property $m(f; u_j) \le f(\boldsymbol{u}) \le M(f; u_j)$, the localisation of roots, can be reduced to univariate root-finding. This is the basis of the subdivision solver, which implementation contains the following main ingredients:

- **Preconditioner**: A transformation of the initial system into a system, which has a better numerical behavior.
- **Reduction strategy**: The technique used to reduce the initial domain, for searching the roots of the system.
- **Subdivision strategy**: The technique used to subdivide the domain, in order to simplify the forthcoming steps, for searching of roots of the system.

We will detail how the implementation is parameterised by `class` components, and illustrate the performance on the method on classical benchmarks in robotics and on geometric problems on curves and surfaces in $\mathbb{R}^3$.

5 Generalized normal form

In the case, we are interested in computing all the complex roots of a polynomial systems in $\mathbb{C}^n$, one can consider normal form methods. They proceed in two steps

- Computation of the generalised normal form modulo the ideal $(f_1, \ldots, f_n)$.
- Computation of the roots from eigencomputation.

A classical approach for normal form computation is through Grobner basis. Unfortunately, their numerical behavior is not satisfactory. In SYNAPS, a generalised normal form method is implemented which allows us to treat polynomial systems with approximate coefficients, more safely.

From this normal form N, we deduce the roots of the system as follows. We use the properties of the operators of multiplication by elements of $\mathcal{A} = R/(f_1, .., f_s))$, as follows.

Algorithm: (Solving in the case of simple roots) Let $a \in \mathcal{A}$ such that $a(\zeta_i) \ne a(\zeta_j)$ if $i \ne j$ (which is generically the case).

1. Compute the M_a the multiplication matrix by a in the basis $\boldsymbol{x}^E = (1, x_1, \ldots, x_n, \ldots)$ of $\mathcal{A}$.
2. Compute the eigenvectors $\Lambda = (\Lambda_1, \Lambda_{x_1}, \ldots, \Lambda_{x_n}, \ldots)$ of M_a^t.
3. For each eigenvector Λ with $\Lambda_1 \ne 0$, compute and return the point $\zeta = (\frac{\Lambda_{x_1}}{\Lambda_1}, \ldots, \frac{\Lambda_{x_n}}{\Lambda_1})$.

The case of multiple roots is treated by simultaneous triangulation of several multiplication matrices. The main ingredients which are involved here are sparse linear algebra (implemented from the direct sparse linear solver `superLU`), and eigenvalue and eigenvector computation (based on `lapack` routines). We will show how these different types of tools are combined together, in order to obtain an efficient zero-dimensional polynomial system solver.