SYNAPS: A LIBRARY FOR DEDICATED APPLICATIONS IN SYMBOLIC NUMERIC COMPUTING

BERNARD MOURRAIN*, JEAN-PASCAL PAVONE*
, PHILIPPE TREBUCHET†, ELIAS P. TSIGARIDAS‡, AND JULIEN WINTZ*

Abstract. We present an overview of the open source library SYNAPS. We describe some of the representative algorithms of the library and illustrate them on some explicit computations, such as solving polynomials and computing geometric information on implicit curves and surfaces. Moreover, we describe the design and the techniques we have developed in order to handle a hierarchy of generic and specialized data-structures and routines, based on a view mechanism. This allows us to construct dedicated plugins, which can be loaded easily in an external tool. Finally, we show how the design of the library allows us to to embed the algebraic operations, as a dedicated plugin, to the external geometric modeler AXEL.

Key words. symbolic-numeric computation, software

AMS(MOS) subject classifications. Primary 1234, 5678, 9101112.

The aim of this paper is to give an overview of the software library SYNAPS[1]. It is an open source project, the objective of which is to provide a coherent and efficient library for symbolic and numeric computations. It implements data-structures and classes for manipulating basic algebraic 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 and gcd computations, etc. The main motivation behind this project, is the need to combine symbolic and numeric computations, which is ubiquitous in many problems. Starting with an exact description of the equations, in most cases, eventually, we 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 these cases, we are not dealing with a solely system but with a neighborhood of an 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 both from a theoretical and a practical point of view, that lie in the frontier between Algebra and Analysis and witnesses the emergence of new investigations. In order to develop efficient implementations for such problems we have to combine algorithms from numeric and symbolic computation and to develop and manipulate data structures

*GALAAD, INRIA, BP 93, 06902 Sophia-Antipolis, France
†SPIRAL, LIP6, 8 rue du capitaine Scott 75015 Paris, France
‡Department of Informatics & Telecommunications, National Kapodistrian University of Athens, Panepistimiopolis 15784, Greece
[1] `http://synaps.inria.fr/`

that are on one hand generic and on the other are easily tuned to specific problems. Moreover, the reusability of external or third-party libraries, such as LAPACK (Fortran library for numerical linear algebra), GMP (C library for extended arithmetic) has to be considered carefully. Specialized routines provided by these external tools have to coexist with generic implementation. Therefore, the software should be designed so that it can connect, in an automatic and invisible to end-user way, the appropriate implementation with the needed operation.

In this paper, we first describe representative algorithms available in the library, and illustrate them by some explicit computations. We begin with a description of the solvers of polynomial equations. These tools are used as black boxes in geometric computations on implicit curves and surfaces. We show how the first level of data structures and polynomial solving implementations are composed to build such algorithms. Such higher level operations on geometric objects are embedded in the geometric modeler AXEL[2], as a dedicated plugin. We describe the design and techniques we have developed to handle a hierarchy of generic and specialized implementations, based on a view mechanism. This approach is extended to build plugins, which provide the equivalent functions in an interactive environment. In particular, we show how template mechanisms can be exploited to transform static strongly typed code into dynamic polymorphic and generic functions, assembled into a plugin that can be loaded easily in an external tool.

1. Solvers of Polynomial Equations. A critical operation, which we will have to perform in geometric computations on curves and surfaces, is to solve polynomial equations. In such a computation, we start with input polynomial equations (possibly with some incertitude on the coefficient) and we want to compute an approximation of the (real) roots of these equations or boxes containing these roots. Such operation should be performed very efficiently and with guarantee, since they will be used intensively in geometric computation.

In sections 1.1, 1.2, 1.3, we describe subdivision solvers which are based on *certified* exclusion criteria. In other words, starting from an initial bounded domain, we remove subdomains which are guaranteed not to contain a real solution of the polynomial equations. A parameter $\epsilon > 0$ is controlling the size of the boxes that are kept. For univariate polynomials, existence and uniqueness criteria are applied to produce certified isolation intervals which contain a singleroot. Such criteria also exist in the multivariate case, but are not yet available in our current implementation. The interest of these subdivision methods, compared to homotopy solvers [34], [15] or algebraic solvers [13], [33] is that only local information related to the initial domain are used and it avoids the representation or approximation of all the complex roots of the system. The methods are particularly

[2] `http://axel.inria.fr/`

efficient for systems where the number of real roots is much smaller that the number of complex roots or where the complex roots are far from the domain of interest. However multiple roots usually reduces their performance if their isolation is required, in addition to their approximation.

1.1. Univariate Subdivision Solvers. Let us consider first an exact polynomial $f = \sum_{i=0}^{d} a_i x^i \in \mathbb{Q}[x]$. Our objective is to isolate the real roots of f, i.e. to compute intervals with rational endpoints that contain one and only one root of f, as well as the multiplicity of every real root. The algorithms take these exact input polynomials and output certified isolation intervals of their real roots. Some parts of the computation are performed with exact integer or rational arithmetic (using the library GMP), but some other parts might be implemented using floating point arithmetic. It uses adapted rounding modes, to be able to certify the result. Here is the general scheme of the subdivision solver that we consider, augmented appropriately so that it also outputs the multiplicities. It uses an external function $V(f, I)$, which bounds the number of roots of f in the interval I.

ALGORITHM 1.1. REAL ROOT ISOLATION

INPUT: A polynomial $f \in \mathbb{Z}[x]$, such that $\deg(f) = d$ and $\mathcal{L}(f) = \tau$.
OUTPUT: A list of intervals with rational endpoints, which contain one and only one real root of f and the multiplicity of every real root.

1. Compute the square-free part of f, i.e. f_{red}
2. Compute an interval $I_0 = (-B, B)$ with rational endpoints that contains all the real roots. Initialize a queue Q with I_0.
3. While Q is not empty do
   a) Pop an interval I from Q and compute $v := V(f, I)$.
   b) If $v = 0$, discard I.
   c) If $v = 1$, output I.
   d) If $v \geq 2$, split I into I_L and I_R and push them to Q.
4. Determine the multiplicities of the real roots, using the square-free factorization of f.

Two families of solvers have been developed. One using Sturm theorem, where $V(f, I)$ returns the exact number (counted without multiplicities) of the real roots of f in I. The second one based on Descartes' rule and Bernstein representation, where $V(f, I)$ bounds the number of real roots of f in I (counted with multiplicities). As analyzed in [10], the bit complexity of both approaches is in $\widetilde{\mathcal{O}}_B(d^4\tau^2)$, if $f \in \mathbb{Z}[x]$, $\deg(f) = d$ is the degree of f and $\mathcal{L}(f) = \tau$ the maximal bitsize of its coefficients. Notice that with the same complexity bound, we can also compute the multiplicities of the real roots. However in practice, the behavior is not exactly the same, as we will illustrate.

1.1.1. Sturm Subdivision Solver. We recall here the main ingredients related to Sturm(-Habicht) sequences computations.

Let $f = \sum_{k=0}^{p} f_k x^k, g = \sum_{k=0}^{q} g_k x^k \in \mathbb{Z}[x]$ where $\deg(f) = p \geq q = \deg(g)$ and $\mathcal{L}(f) = \mathcal{L}(g) = \tau$. We denote by $\mathtt{rem}(f,g)$ and $\mathtt{quo}(f,g)$ the remainder and the quotient, respectively, of the Euclidean division of f by g, in $\mathbb{Q}[x]$.

Definition 1.2. [35, 3] *The* signed polynomial remainder sequence *of f and g, denoted by* $\mathbf{SPRS}(f,g)$*, is the polynomial sequence*

$$R_0 = f, R_1 = g, R_2 = -\,\mathtt{rem}(f,g), \ldots, R_k = -\,\mathtt{rem}(R_{k-2}, R_{k-1}),$$

with k the minimum index such that $\mathtt{rem}(R_{k-1}, R_k) = 0$*. The* quotient sequence *of f and g is the polynomial sequence* $\{Q_i\}_{0 \leq i \leq k-1}$*, where* $Q_i = \mathtt{quo}(R_i, R_{i+1})$ *and the* quotient boot *is* $(Q_0, Q_1, \ldots, Q_{k-1}, R_k)$.

Another construction yields the Sturm-Habicht sequence of f and g, i.e. $\mathbf{StHa}(f,g)$, which achieves better bounds on the bit size of the coefficients.

Let M_j be the matrix which has as rows the coefficient vectors of the polynomials fx^{q-1-j}, fx^{q-2-j}, $\ldots, fx, f$, $g, gx, \ldots, gx^{p-2-j}, gx^{p-1-j}$ with respect to the monomial basis $x^{p+q-1-j}$, $x^{p+q-2-j}$, $\ldots, x, 1$. The dimension of M_j is $(p+q-2j) \times (p+q-j)$. For $l = 0, \ldots, p+q-1-j$ let M_j^l be the square matrix of dimension $(p+q-2j) \times (p+q-2j)$ obtained by taking the first $p+q-1-2j$ columns and the $l + (p+q-2j)$ column of M_j.

Definition 1.3. *The Sturm-Habicht sequence of f and g, is the sequence* $\mathbf{StHa}(f,g) = (H_p = H_p(f,g), \ldots, H_0 = H_0(f,g))$*, where* $H_p = f$, $H_{p-1} = g$, $H_j = (-1)^{(p+q-j)(p+q-j-1)/2} \sum_{l=0}^{j} \det(M_j^l) x^l$.

For two polynomials of degree p and q and of bit size bounded by τ, such sequences and their evaluation at a rational point a, where $\mathsf{a} \in \mathbb{Q} \cup \{\pm\infty\}$ and $\mathcal{L}(\mathsf{a}) = \sigma$ can be done respectively with complexity $\widetilde{\mathcal{O}}_B(p^2 q \tau)$ and $\widetilde{\mathcal{O}}_B(q \max\{p\tau, q\sigma\})$. For more details, see [35, 3, 18, 19].

The structure `SturmSeq` encodes these Sturm sequences in SYNAPS. Several constructions are implemented, specified by a class in the constructor, Euclidean, primitive and subresultant polynomial remainder sequences. Let us present an example of code for constructing the Sturm-Habicht sequence `s` of two polynomials `p`, `q` $\in \mathbb{Z}[x]$. The implementation corresponds to a variant of the inductive construction described in [3].

```
UPolDse<ZZ> p("3*x^5+23*x^3-x^2+234"), q("10*x^4+200*x^2-13243");
SturmSeq<ZZ> s(p,q,HABICHT());
```

The result is a sequence of polynomials, with coefficients in the initial ring ZZ:

```
[3*x^5+23*x^3-x^2+234, 10*x^4+200*x^2-13243,
   3700*x^3+100*x^2-397290*x-23400, -174378300*x^2-4685100*x+1813200700,
   796993090279590*x+51961697166600, -37867420670503735668763]
```

Such a sequence can be used to count roots in an interval. Let $W_{(f,g)}(\mathsf{a})$ denote the number of modified sign changes of the evaluation of $\mathbf{StHa}(f,g)$ at a.

THEOREM 1.1. [3, 36] *Let $f, g \in \mathbb{Z}[x]$, where f is square-free and $f^{'}$ is the derivative of f and its leading coefficient $f_d > 0$. If $\mathsf{a} < \mathsf{b}$ are both non-roots of f and γ ranges over the roots of f in (a, b), then $W_{(f,g)}(\mathsf{a}) - W_{(f,g)}(\mathsf{b}) = \sum_{\gamma} \text{sign}\,(f^{'}(\gamma)g(\gamma))$.* If $g = f^{'}$ then $\mathbf{StHa}(f, f^{'})$ is a Sturm sequence and Th. 1.1 counts the number of distinct real roots of f in (a, b).

1.1.2. Bernstein Subdivision Solver. In this section, we recall the background of Bernstein polynomial representation and how it is used in the subdivision solver. Given an arbitrary univariate polynomial function $f(x) \in \mathbb{K}$, we can convert it to a representation of degree d in Bernstein basis, which is defined by:

$$f(x) = \sum_i b_i B_i^d(x), \text{ and } B_i^d(x) = \left(\begin{array}{c} d \\ i \end{array} \right) x^i (1-x)^{d-i} \tag{1.1}$$

where b_i is usually referred as controlling coefficients. Such conversion is done through a basis conversion [11]. The above formula can be generalized to an arbitrary interval $[a, b]$ by a variable substitution $x' = (b-a)x + a$. We denote by $B_d^i(x; a, b) = \binom{d}{i}(x-a)^i(b-x)^{d-i}(b-a)^{-d}$ the corresponding Bernstein basis on $[a, b]$. There are several useful properties regarding Bernstein basis given as follows:

- *Convex-Hull Properties*: Since $\sum_i B_d^i(x; a, b) \equiv 1$ and $\forall x \in [a, b]$, $B_d^i(x; a, b) \geq 0$ where $i = 0, ..., d$, the graph of $f(x) = 0$, which is given by $(x, f(x))$, should always lie within the convex-hull defined by the control coefficients $(\frac{i}{d}, b_i)$ [11].
- *Subdivision* (de Casteljau): Given $t_0 \in [0, 1]$, $f(x)$ can be represented piece-wisely by:

$$f(x) = \sum_{i=0}^{d} b_0^{(i)} B_d^i(x; a, c) = \sum_{i=0}^{d} b_i^{(d-i)} B_d^i(x; c, b), \text{ where} \tag{1.2}$$

$$b_i^{(k)} = (1-t_0) b_i^{(k-1)} + t_0 b_{i+1}^{(k-1)} \text{ and } c = (1-t_0)a + t_0 b. \tag{1.3}$$

Another interesting property of this representation related to Descartes rule of signs is that there is a simple and yet efficient test for the existence of real roots in a given interval. It is based on the number of sign variation $V(\mathbf{b})$ of the sequence $\mathbf{b} = [b_1, \ldots, b_k]$ that we define recursively as follows:

$$V(\mathbf{b}_{k+1}) = V(\mathbf{b}_k) + \left\{ \begin{array}{l} 1, \text{if } b_i b_{i+1} < 0 \\ 0, \text{else} \end{array} \right. \tag{1.4}$$

With this definition, we have:

PROPOSITION 1.1. *Given a polynomial $f(x) = \sum_i^n b_i B_i^d(x; a, b)$, the number N of real roots of f on $]a, b[$ is less than or equal to $V(\boldsymbol{b})$, where $\boldsymbol{b} = (b_i)_{i=1,\ldots,n}$ and $N \equiv V(\boldsymbol{b}) mod\, 2$.* With this proposition,

- if $V(\mathbf{b}) = 0$, the number of real roots of f in $[a, b]$ is 0;

- if $V(\mathbf{b}) = 1$, the number of real roots of f in $[a, b]$ is 1.

This function V yields another variant of subdivision algorithm 1.1. In order to analyze its behavior, a partial inverse of Descartes' rule and lower bounds on the distance between roots of a polynomial have been used. It is proved that the complexity of isolating the roots of a polynomial of degree d, with integer coefficients of bit size $\leq \tau$ is bounded by $\mathcal{O}\left(d^4\tau^2\right)$ up to some poly-logarithmic factors. See [8, 10] for more details.

Notice that this localization algorithm extends naturally to B-splines, which are piecewise polynomial functions [11].

The approach can also be extended to polynomials with interval coefficients, by counting 1 sign variation for a sign sub-sequence $+, ?, -$ or $-, ?, +$; 2 sign variations for a sign sub-sequence $+, ?, +$ or $-, ?, -$; 1 sign variation for a sign sub-sequence $?, ?$, where ? is the sign of an interval containing 0. Again in this case, if a family $\overline{\underline{\mathrm{f}}}$ of polynomials is represented by the sequence of intervals $\overline{\underline{\mathbf{b}}} = [\overline{\underline{\mathrm{b}}}_0, \ldots, \overline{\underline{\mathrm{b}}}_d]$ in the Bernstein basis of the interval $[a, b]$

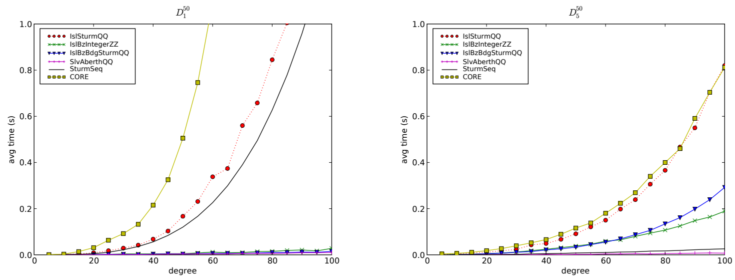
- if $V(\overline{\underline{\mathbf{b}}}) = 1$, all the polynomials of the family $\overline{\underline{\mathrm{f}}}$ have one root in $[a, b]$,
- if $V(\overline{\underline{\mathbf{b}}}) = 0$, all the polynomials of the family $\overline{\underline{\mathrm{f}}}$ have no roots in $[a, b]$.

This subdivision algorithm, using interval arithmetic, yields either intervals of size smaller than ϵ, which might contain the roots of $f = 0$ in $[a, b]$ or isolating intervals for all the polynomials of the family defined by the interval coefficients.

A variant of such approach in the monomial basis is called Uspensky's method, e.g. [8, 29] and references therein. Another variant, using Cauchy's lower bound on the positive roots of the polynomial, isolates the real roots by computing their continued fraction expansion, c.f. [9] and references therein. The expected complexity of this variant is the same as the worst case bound of the subdivision solvers, i.e $\widetilde{\mathcal{O}}_B(d^4\tau^2)$.

1.1.3. Experimentation. In this section, we describe the experimental behavior of some of the implemented subdivision solvers on specific data sets. The solvers take as input, a polynomial `f` with integer, rational or interval coefficients and output intervals with rational endpoints. All use the same initial interval, given by Cauchy bound.

The following graphs illustrate the behavior of various univariate solvers, on random (D_1) and Mignotte (D_5) polynomials of maximum coefficient bit size 50 bits:



The different solvers are: `IslSturmQQ` based on the construction of the Sturm-Habicht sequence and subdivisions, using rational numbers or large integers; `IslBzIntegerZZ` implementing the Bernstein subdivision solver, using extended integer coefficients; `IslBzBdgSturmQQ` combining two solvers (in a first part, the polynomial is converted to the Bernstein representation on the initial interval, using rational arithmetic and its coefficients are rounded to `double` intervals. The Bernstein solver is applied on the polynomial with interval coefficients. If the size of the domain is too small, the Sturm solver is launched), `CORE` [16] and `SlvAberthQQ` corresponding to MPSOLVE, a numerical solver based on Aberth's method [4] and implemented by G. Fiorentino and D. Bini.

The average time over 100 runs is in seconds. The experiments were performed on a Pentium (2.6 GHz), using g++ 3.4.4 (Suse 10). The extended arithmetic is based on the library GMP. For polynomials with few, distinct and well separated real roots (D_1), we observe that the Bernstein subdivision solver perform well. When there are roots that are very close (D_5), the computation time of the Sturm-Habicht sequence is negligible. A combined solver based on numerical solvers such as MPSOLVE and subdivision techniques using for instance the Bernstein representation seems to be the most efficient approach. For more details, the reader may refer to [10].

1.2. 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, i.e. those real numbers that satisfy a polynomial equation with integer coefficients, form a real closed field denoted by $\mathbb{R}_{alg} = \overline{\mathbb{Q}}$. From all integer polynomials that have an algebraic number α as root, the primitive one (the gcd of the coefficients is 1) with the minimum degree is called *minimal*. The minimal polynomial is unique (up to a sign), primitive and irreducible, e.g. [36]. For the computation with real algebraic numbers, we use Sturm-Habicht sequences, hence it suffices to deal with algebraic numbers, as roots of a square-free polynomial and not as roots of their minimal one. In order to represent a real algebraic number we choose the *isolating interval representation*, i.e. 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 the geometric applications (topology of algebraic curves and surfaces, arrangement computation, ...) that we are targeting, this representation is enough. This is the reason why, we have not considered towers of algebraic extensions (algebraic numbers which defining polynomials are also algebraic numbers).

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. Evidently, a real algebraic number is constructed by solving (in our case by isolating) the real roots of an integer univariate polynomial.

Let us demonstrate the capabilities of the library by an example:

```
#include <synaps/usolve/Algebraic.h>

using namespace::SYNAPS;
using namespace::std;

typedef ZZ NT;
typedef Algebraic<NT> SOLVER;

int main() {
  SOLVER::Poly f("x^9-29*x^8+349*x^7-2309*x^6+9307*x^5-23771*x^4
                 +38511*x^3-38151*x^2+20952*x-4860");
  Seq<SOLVER::RO_t> sol= solve(f, SOLVER());
  for (unsigned i=0; i< sol.size(); ++i)
   cout << "(" << i << ") " << sol[i] << endl;
 return 0; }
```

First the user declares the number type of the coefficients of the polynomials that he wants to deal with. In our case we use `ZZ`, which correspond to GMP integers. In the sequel, he declares the solve algorithm, which means that he chooses an algorithm in order to isolate the real roots a integer polynomial. There are many solvers in SYNAPS and each of them has a similar structure. In our example, we choose the Sturm subdivision solver. For the various univariate solvers available in SYNAPS, the reader may refer to the previous section or to [10]. Inside the main routine, the user constructs a polynomial and solves it using the `solve` function. This function constructs a sequence of real algebraic numbers that are printed subsequently.

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 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. For the complexity of these operations, the reader may refer to [10] and references therein.

Moreover, projection-based algorithms exists for constructing pairs of real algebraic numbers that are real solutions of bivariate polynomials systems, as well as functions for the computing the sign of a bivariate integer polynomial, evaluated over two real algebraic numbers.

1.3. Multivariate Bernstein Subdivision Solver. We consider here the problem of computing the solutions of a polynomial system

$$\begin{cases} f_1(x_1, \dots, x_n) = 0, \\ \vdots \\ f_s(x_1, \dots, x_n) = 0, \end{cases}$$

in a box $B := [a_1, b_1] \times \cdots \times [a_n, b_n] \subset \mathbb{R}^n$. The method for approximating the real roots of this system, that we describe now uses the representation of multivariate polynomials in Bernstein basis, analysis of sign variations and univariate solvers (Section 1.1.2). The output is a set of small-enough, which contain these roots. The boxes which are removed are guaranteed to contain no root, but the existence and uniqueness of a root in each output box is not provided, neither the multiplicity. The computation is done which floating point arithmetic, using controlled rounding modes.

The subdivision solver [24] that we describe now, can be seen as an improvement of the *Interval Projected Polyhedron* algorithm in [31].

In the following, we use the Bernstein basis representation of a multivariate polynomial f of the domain $I := [a_1, b_1] \times \cdots \times [a_n, b_n] \subset \mathbb{R}^n$:

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

Definition 1.4. *For any* $f \in \mathbb{R}[\mathbf{x}]$ *and* $j = 1, \ldots, n$, *let*

$$m_j(f; x_j) = \sum_{i_j=0}^{d_j} \min_{\{0 \le i_k \le d_k, k \ne j\}} b_{i_1,\ldots,i_n} \; B_{d_j}^{i_j}(x_j; a_j, b_j)$$

$$M_j(f; x_j) = \sum_{i_j=0}^{d_j} \max_{\{0 \le i_k \le d_k, k \ne j\}} b_{i_1,\ldots,i_n} x \; B_{d_j}^{i_j}(x_j; a_j, b_j).$$

Theorem 1.2 (Projection Lemma). *For any* $\mathbf{u} = (u_1, \ldots, u_n) \in I$, *and any* $j = 1, \ldots, n$, *we have*

$$m(f; u_j) \le f(\mathbf{u}) \le M(f; u_j).$$

As a direct consequence, we obtain the following corollary:

Corollary 1.1. *For any root* $\mathbf{u} = (u_1, \ldots, u_n)$ *of the equation* $f(\mathbf{x}) = 0$ *in the domain* I, *we have* $\underline{\mu}_j \le u_j \le \overline{\mu}_j$ *where*

- $\underline{\mu}_j$ *(resp.* $\overline{\mu}_j$*) is either a root of* $m_j(f; x_j) = 0$ *or* $M_j(f; x_j) = 0$ *in* $[a_j, b_j]$ *or* a_j *(resp.* b_j*) if* $m_j(f; x_j) = 0$ *(resp.* $M_j(f; x_j) = 0$*) has no root on* $[a_j, b_j]$,
- $m_j(f; u) \le 0 \le M_j(f; u)$ *on* $[\underline{\mu}_j, \overline{\mu}_j]$.

The general scheme of the solver implementation consists in

1. applying a preconditioning step to the equations;
2. reducing the domain;
3. if the reduction ratio is too small, to split the domain

until the size of the domain is smaller than a given epsilon.

The following important ingredients of the algorithm parametrize its implementation:

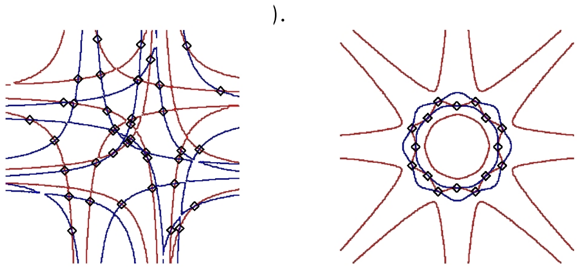
Preconditioner. It is a transformation of the initial system into a system, which has a better numerical behavior. Solving the system $\mathbf{f} = 0$ is equivalent to solving the system $M\,\mathbf{f} = 0$, where M is an $s \times s$ invertible matrix. As such a transformation may increase the degree of some equations, with respect to some variables, it has a cost, which might not be negligible in some cases. Moreover, if for each polynomial of the system not all the variables are involved, that is if the system is sparse with respect to the variables, such a preconditioner may transform it into a system which is not sparse anymore. In this case, we would prefer a partial preconditioner on a subsets of the equations sharing a subset of variables. We consider Global transformations, which minimize the distance between the equations, considered as vectors in an affine space of polynomials of a given degree and Local straightening (for $s = n$), which transform locally the system $\mathbf{f}$ into a system $J^{-1}\mathbf{f}$, where $J = (\partial_{x_i} f_j(\mathbf{u}_0)_{1 \le i,j \le s}$ is the Jacobian matrix of $\mathbf{f}$ at a point $\mathbf{u}$ of the domain I, where it is invertible.

It can be proved that the reduction based on the polynomial bounds m and M behaves like Newton iteration near a simple root, that is we have a quadratic convergence, with this transformation.

Reduction strategy. It is the technique used to reduce the initial domain, for searching the roots of the system. It can be based on Convex hull properties as in [31] or on Root localization, which is a direct improvement of the convex hull reduction and consists in computing the first (resp. last) root of the polynomial $m_j(f_k; u_j)$, (resp. $M_j(f_k; u_j)$), in the interval $[a_j, b_j]$. The current implementation of this reduction steps allows us to consider the convex hull reduction, as one iteration step of this reduction process. The guarantee that the computed intervals contain the roots of f, is obtained by controlling the rounding mode of the operations during the de Casteljau computation.

Subdivision strategy. It is technique used to subdivide the domain, in order to simplify the forthcoming steps, for searching the roots of the system. Some simple rules that can be used to subdivide a domain and reduce its size. The approach, that we are using in our implementation is the parameter domain bisection: The domain B is then split in half in a direction j for which $|b_j - a_j|$ is maximal. But instead of choosing the size of the interval as a criterion for the direction in which we split, we may choose other criterion depending also on the value the functions m_i, M_j or f_j (for instance where $M_j - m_j$ is maximal). A bound for the complexity of this method is detailed in [24]. It involves metric quantities related to the system $\mathbf{f} = 0$, such as the Lipschitz constant of $\mathbf{f}$ in B, the entropy of its near-zero level sets, a bound d on the degree of the equations in each variable and the dimension n.

Examples. Here are some comparisons of the different strategies, describing the number of iterations in the main loop, the number of subdivision of a domain, the number of boxes produced by the method, the time it takes. We compare the method **sbd**, a pure subdivision approach, **rd** a method doing first reduction and based on a univariate root-solver using the Descarte's rule. **sbds** a subdivision approach using the global preconditioner, **rds** a reduction approach using the global preconditioner, **rdl**, a reduction approach using the Jacobian preconditioner. The first example is a bivariate system, with equations of degree 12) in each variable, the second ex).



(a) (b)

Example a:

method	iterations	subdivisions	results	time (ms)
sbd	4826	4826	220	217
rd	2071	1437	128	114
sbds	3286	3286	152	180
rds	1113	748	88	117
rdl	389	116	78	44

Example b:

method	iterations	subdivisions	results	time (ms)
sbd	84887	84887	28896	3820
rd	82873	51100	20336	4553
sbds	6076	6076	364	333
rds	1486	920	144	163
rdl	1055	305	60	120

For more details on this solver, see [28], [24].

1.4. 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 Effective Algebraic Geometry. It has important applications in CAGD (Computer Aided Geometric Design), such as for the problem of implicitization of parametric surfaces, or for surface parametrization inversion, intersection, and detection of singularities of a parametrized 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. Such solvers, exploiting adequately tools from numerical linear algebra, are numerically robust and compute efficiently approximations of all the roots, even if they have multiplicities. They can be used directly in geometric applications (see eg. [20]) or embedded in an algorithm which validates afterward the approximation. Such validation step is not yet provided in our current implementation of these resultant based methods.

The library SYNAPS contains several types of resultant constructions, such that the projective resultant, the toric resultant (based on an implementation by I. Emiris), and the Bezoutian resultant. Using these resultant matrix formulations, solving a polynomial problem can be reduced to solving the generalized eigenvector problem $T^t(x)\mathbf{v} = 0$, where $T(x)$ is a matrix of size $N \times N$ with polynomial coefficients, or equivalently a polynomial with $N \times N$ matrix coefficients.

If $d = \max_{i,j}\{\deg(T_{ij}(x))\}$, we obtain $T(x) = T_d x^d + T_{d-1}x^{d-1} + \cdots + T_0$, where T_i are $n \times n$ matrices. The problem is transformed into a generalized eigenvalue problem $(A - \lambda B)\,\mathbf{v} = 0$ where

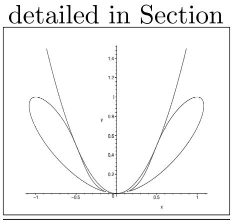
$$A = \begin{pmatrix} 0 & I & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & I \\ T_0^t & T_1^t & \cdots & T_{d-1}^t \end{pmatrix},\ B = \begin{pmatrix} I & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & I & 0 \\ 0 & \cdots & 0 & -T_d^t \end{pmatrix},$$

where A, B are $N \times d$ constant matrices. And we have the following interesting property :

$$T^t(x)\,\mathbf{v} = 0 \Leftrightarrow (A - xB) \begin{pmatrix} \mathbf{v} \\ x\,\mathbf{v} \\ \vdots \\ x^{d-1}\,\mathbf{v} \end{pmatrix} = 0.$$

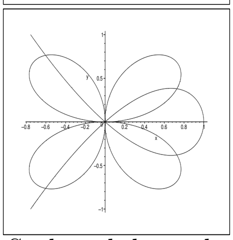
We apply it for implicit curve intersection problems in [5]. Given two polynomials $p, q \in \mathbb{Q}[x, y]$, we compute their resultant matrix, with respect to y. This yields a matrix $T(x)$, from which we deduce the coordinates of the intersection points by solving the generalized eigenvector problem $T(x)^t\,\mathbf{v} = 0$. The case of multiple roots in the resultant and of intersection points with the same x-coordinates is analyzed in details, so that we can recover their multiplicities.

Examples. Here are two examples which illustrate the behavior of the implementation. The eigenvalue computation is using the routine `dgegv` from the FORTRAN library LAPACK. The connection with this external library is performed transparently, through the mechanisms of views, as detailed in Section 3.



$$\begin{cases} p = x^4 - 2x^2y + y^2 + y^4 - y^3 \\ q = y - 2x^2 \end{cases}$$

```
(x1, y1) = (1.6e-09,0) of multiplicity 4
(x2, y2) = (-0.5,0.5)  of multiplicity 2
(x3, y3) = (0.5,0.5)   of multiplicity 2
Execution time = 0.005s; Eps =  10^-6.
```



$$\begin{cases} p = x^6 + 3x^4y^2 + 3x^2y^4 + y^6 - 4x^2y^2 \\ q = y^2 - x^2 + x^3 \end{cases}$$

```
(x1, y1) = (-3e-16,2e-17) of multiplicity 8
(x2, y2) = (-0.60,-0.76)  of multiplicity 1
(x3, y3) = (-0.60,0.76)   of multiplicity 1
(x4, y4) = (0.72, -0.37)  of multiplicity 1
(x5, y5) = (0.72,0.37)    of multiplicity 1
Execution time = 0.011s; Eps = 10^-3.
```

Such tools have also been used in 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. 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 chosen) 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 their roots, even in the presence of multiple roots.

1.5. Generalized Normal Form. The solver that we describe now computes all the complex roots of a zero-dimensional polynomial systems in $\mathbb{C}^n$. It proceeds in two steps:

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

Hereafter, the polynomials are in the ring $\mathbb{K}[x_1, \ldots, x_n]$ with coefficients in the field $\mathbb{K}$ (eg. $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$) and variables $x_1, \ldots, x_n$.

A classical approach for normal form computation is through Gröbner basis [6]. Unfortunately, their behavior on approximate data is not satisfactory [23]. In SYNAPS, a generalized normal form method is implemented which allows us to treat polynomial systems with approximate coefficients, more safely. Such a normal form computation is available with exact arithmetic (rational numbers or modular numbers) and with extended floating point arithmetic. The numerical stability of the generalized normal form computation is improved by the allowing column pivoting on the coefficient matrices of the involved polynomials, which is not possible for Gröbner basis computation. In the case of floating point arithmetic, no certification is provided yet in the current implementation.

The method constructs a set of monomials B and *a rewriting family* F on the monomial set B, which have the following property: $\forall f, f' \in F$,

- f has exactly **one** monomial that we denoted by $\gamma(f)$ (also called the *leading monomial* of f) in ∂B,
- $\text{supp}(f) \subset B^+$, $\text{supp}(f - \gamma(f)) \subset B$,
- if $\gamma(f) = \gamma(f')$ then $f = f'$,

where $B^+ = B \cup x_1 B \cup \cdots \cup x_n B$, $\partial B = B^+ \backslash B$. Moreover, the monomial set B will be connected to 1: for every monomial m in B, there exists a finite sequence of variables $(x_{i_j})_{j\in[1,l]}$ such that $1 \in B$, $\Pi_{j=1\ldots l'} x_{i_j} \in B$, $\forall l' \in [1, l]$ and $\Pi_{j\in[1,l]}\, x_{i_j} = m$.

The normal form construction is based on the following criteria:

Theorem 1.3. *[22] Let B be a monomial set connected to 1. Let $R_F : B^+ \to B$ be a projection from B^+ to B, with kernel F and let $I = (F)$ be the ideal generated by F. Let $M_i : B \to B$ be defined by $b \mapsto R_F(x_i b)$. Then, the two properties are equivalent:*

1. *For all $1 \leq i, j \leq n$, $M_i \circ M_j = M_j \circ M_i$.*
2. *$\mathbb{K}[x_1, \ldots, x_n] = \langle B \rangle \oplus I$.*

The algorithm consists in constructing degree by degree the set B and in checking at each level, whether the partial operator of multiplication commutes. For more details, see [33, 25, 26, 23].

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 (see [2], [21], [32]):

Algorithm 1.5. Solving in the case of simple roots

Let $a \in \mathcal{A}$ such that $a(\zeta_i) \neq a(\zeta_j)$ if $i \neq j$ (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 $\Lambda_1 \neq 0$, 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). These different types of tools are combined together, in order to obtain an efficient zero-dimensional polynomial system solver. We are currently working on techniques adapted from thoses of [30, 27] to certify *a posteriori* the numerical approximation of the roots obtained by such eigenvalues/eigenvectors computation. These techniques will also allow certified root refinement. Eventually a purely symbolic representation of the roots will be added.

2. Geometric computing with curves and surfaces. A special context where algebraic operations on curves and surfaces are critical is non-linear computational geometry. Shapes are modeled by semi-algebraic sets, where the local primitives are implicit or parametric curves or surfaces. This is typically the case in CAGD, where NURBS or Bspline functions [11] are used standard primitives. The degree of the piecewise polynomial models is usually 3 (or 5) in each variable. In this section, we describe some functions from the module `shape` of the library SYNAPS. We focus on tools for computing the topology of implicit curves and surfaces, which are fundamental in arrangement problems. All these methods rely on polynomial solvers as fundamental ingredients. They guaranty there results, provided the polynomial solvers are able to perform certified root isolation.

2.1. Sweeping of planar curves. Let $\mathcal{C}$ be a planar implicit curve defined by a polynomial equation $f(x, y) = 0$, where $f \in \mathbb{Q}[x, y]$. Here is the scheme of an algorithm, which outputs a graph of points in the plane, isotopic to the curve $\mathcal{C}$, and with the following properties:

ALGORITHM 2.1. TOPOLOGY OF AN IMPLICIT CURVE

INPUT: an algebraic curve $\mathcal{C}$ given by a square-free equation $f(x, y) = 0$ with $f \in \mathbb{Q}[x, y]$.

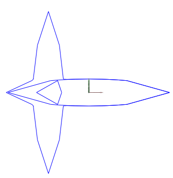
- Choose a direction (say the x-axis direction), and consider a virtual line orthogonal to this direction, sweeping the plane;
- Detect at which critical position, the topology of the intersection of the line and the curve changes (the roots of the resultant $\mathtt{res}_y(f, \partial_y f)(x) = 0$.
- Compute the corresponding intersection points $f(\alpha, y) = 0$ for α a root of $\mathtt{res}_y(f, \partial_y f)(x) = 0$.
- Compute the intersection points for sample lines in between these critical positions.
- Check the generic position (at most one critical point per sweeping line) and connect the computed points by segments, in order to get a graph isotopic to $\mathcal{C}$.

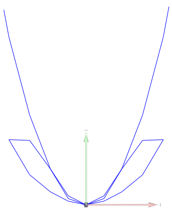
In this algorithm, we compute the Sturm-Habicht sequence of the two polynomials $f, \partial_y f$ with respect to the variable y. The last term of this sequence is their resultant $r(x) \in \mathbb{Q}[x]$. We solve this equation. Let $\alpha_1 < \cdots < \alpha_s$ be the real roots of $r(x) = 0$. For each α_i, we compute the corresponding y such that $f(\alpha_i, y) = 0$. At this point, we also check that the curve is in *generic* position, that is, there is at most one multiple solution of $f(\alpha_i, y) = 0$. This construction involve the manipulation of algebraic numbers and the use of univariate solvers (see Sections 1.1 and 1.2).

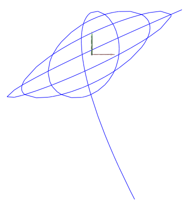
Once these points are computed, we compute intermediate (rational) points $\mu_0 < \mu_1 < \cdots < \mu_s$, such that $\mu_{i-1} < \alpha_i < \mu_i$ and the corresponding y such that $f(\mu_i, y) = 0$. Here again we use a univariate solver, knowing that the roots are simple (using a Bernstein subdivision solver).

All these points are collected and sorted by lexicographic order such that $x > y$. The subset of points, corresponding to two consecutive x-coordinates are connected, according to their y-coordinates. See [17] for more details.

Examples.







Here is how the function and the external viewer AXEL in SYNAPS are called:

```
MPol<QQ> p("5*x^4*y-2*x^6-4*x^2*y^2+y^3+y^5-2*y^4*x^2-y^4+2*x^2*y^3");
topology::point_graph<double> g;
topology(g, p,TopSweep2d<double,SlvBzBdg<double> >());
axel::ostream os("tmp.axl"); os<<g; os.view();
```

It computes a graph of points `g` (of type `topology::point_graph<C>`) which is isotopic to the curve $\mathcal{C}$, defined by the polynomial `p`. The coefficients of the points in the computed graph are of type `C`. The polynomial `p` is converted to a polynomial with rational coefficients, if needed. The class `TopSweep2d<C,SLV>` depends on two parameters:

- `C`, which is the type of coefficients of the points in the result,
- `SLV`, which is the type of univariate solver to be used. The default value is `SlvbzBdg<double>`.

2.2. Subdivision method for planar curves. In this section, we consider a curve $\mathcal{C}$ in $\mathbb{R}^2$, defined by the equation $f(x, y) = 0$ with $f \in \mathbb{Q}[x, y]$ and a domain $B = [a, b] \times [c, d] \subset \mathbb{R}^2$.

In order to compute the topology of this curve in a box, we use the notion of regularity:

Definition 2.2. *We say that the curve $\mathcal{C}$ is y-regular (resp. x-regular) in B, if $\mathcal{C}$ has no tangent parallel to the y-direction (resp. x-direction) in B.*

Notice that if $\mathcal{C}$ is x-regular (or y-regular) it is smooth in B since it cannot have singular points in B. A curve is *regular* in B, if it is x-regular *or* y-regular in B.

We use the property that if $\mathcal{C}$ is x-regular in B, then its topology can be deduced from its intersection with the boundary ∂B.

Proposition 2.1. *[17] If $\mathcal{C}$ is regular in B, its topology in B is uniquely determined by its intersection $\mathcal{C} \cap \partial B$ with the boundary of B.*

Here is a simple test for the regularity of a curve in a domain, which extends in some way the criterion in [12]:

Proposition 2.2. *[17] If the coefficients of $\partial_y f(x, y) \neq 0$ (resp. $\partial_x f(x, y) \neq 0$) in the Bernstein basis of the domain $B = [a, b] \times [c, d] \subset \mathbb{R}^2$ have the same sign $\in \{-1, 1\}$, then the curve $\mathcal{C}$ is regular on B.*

In this case, we have the following connection algorithm, which output a set of segments isotopic to the curve in the domain.

- Compute the points of $\mathcal{C} \cap \partial B$, repeating a point if its multiplicity is even.
- Sort them by lexicographic order so that $x > y$: $\mathcal{L} := \{p_1, p_2, \ldots\}$
- Connect them by pair $[p_1, p_2], [p_3, p_4], \ldots$ of consecutive points in $\mathcal{L}$.

This yields the following subdivision algorithm for a planar implicit curve:

ALGORITHM 2.3. TOPOLOGY OF A PLANAR IMPLICIT CURVE

INPUT: A box $B_0 \subset \mathbb{R}^2$, a curve $\mathcal{C}$ defined by the squarefree polynomial equation $f(x, y) = 0$ with $f \in \mathbb{Q}[x, y]$ and $\epsilon > 0$.

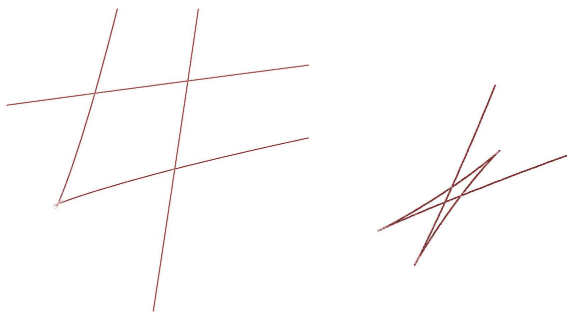
- Compute the x and y critical points of $f(x, y) = 0$.
- $\mathcal{L} = \{B_0\}$
- While $\mathcal{L}$ is not empty,
  - Choose and remove a domain B of $\mathcal{L}$;
  - Test if there is a unique critical point, which is not singular in B;
  - If it is the case, compute the topology of $\mathcal{C}$ in B,
  - else if $|B| > \epsilon$, subdivide the box into subdomains and add them to $\mathcal{L}$,
  - otherwise connect the center of the box to the points of $\mathcal{C} \cap \partial B$.

OUTPUT: a set of points and a set of (smooth) arcs connecting these points.

PROPOSITION 2.3. *For $\epsilon > 0$ small enough, the algorithm 2.3 computes a graph of points which is isotopic to the curve $\mathcal{C} \cap B$.*

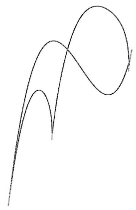
The subdivision level (controlled by ϵ) can be improved significantly by analyzing the number of real branches at a singular point of the curve $\mathcal{C}$, using topological degree computation. The solver used to compute these singular points should be able to isolate them from other extremal points of f (real roots of $\partial_x f = \partial_y f = 0$) to guaranty the topology of the curve.

Example c:



This curve is the discriminant curve of a bivariate system with few monomials used in [7] to give a counter-example to Kushnirenko's conjecture. It is of degree 47 in x and y, and the maximal bit size of its coefficient is of order 300. It takes less that 10 s. (on an Intel M 2.00GHz i686) to compute the topological graph, by rounding up and down the Bernstein coefficients of the polynomial to the nearest double machine precision numbers, applying the subdivision techniques on the enveloping polynomials. We observe a very tiny domain, which at first sight looks like a cusp point, but which contains in reality, 3 cusps points and 3 crossing points. The central region near these cusp points is the region where counter-examples have been found.

Example d:



This curve is the projection onto the (x, y) plane of the curve of points with tangent parallel to the z-direction for a surface of degree 4. It is defined by the equation of degree 12, and has 4 real cusps and 2 real crossing points. The size of the coefficients is small and the topological graph is computed in less than 1 s. It defines 4 connected regions in the plane, one of these being very small and difficult to see on the picture.

2.3. Subdivision approach for space curves. In this section, we consider a curve $\mathcal{C}$ of $\mathbb{R}^3$. We suppose that $I(\mathcal{C}) = (f_1, f_2, \ldots, f_k)$ and for two polynomials $f(x, y, z), g(x, y, z) \in I(\mathcal{C})$, we define $\mathbf{t} = \bigtriangledown(f) \wedge \bigtriangledown(g)$. We are interested in the topology of $\mathcal{C}$ in a box $B = [a_0, b_0] \times [a_1, b_1] \times [a_2, b_2]$. Similar to the 2D case, we can represent f, g and each component of $\mathbf{t}$ in the Bernstein basis for the domain B. As we will see, the sign changes of the resulting Bernstein coefficients will make it possible to test the regularity of the curve with minimal effort.

Here is a criteria of regularity of space curves which allows us to deduce the topology of $\mathcal{C}$ in the domain:

PROPOSITION 2.4. *[17] Let $\mathcal{C}$ be a 3D spatial curve defined by $f = 0$ and $g = 0$. If*

- *$\boldsymbol{t}_x(\boldsymbol{x}) \neq 0$ on B, and*
- *$\partial_y h \neq 0$ on z-faces, and $\partial_z h \neq 0$ and it has the same sign on both y-faces of B, for $h = f$ or $h = g$,*

then the topology of $\mathcal{C}$ is uniquely determined from the points $\mathcal{C} \cap \partial B$.

A similar criterion applies by symmetry, exchanging the roles of the x, y, z coordinates. If one of these criteria applies with $\mathbf{t}_i(x) \neq 0$ on B (for $i = x, y, z$), we will say that $\mathcal{C}$ is i-regular on B.

From a practical point of view, the test $\mathbf{t}_i(x) \neq 0$ or $\partial_i(h) \neq 0$ for $i = x, y$ or z, $h = f$ or g can be replaced by the stronger condition that their coefficients in the Bernstein basis of B have a constant sign, which is straightforward to check. Similarly, such a property on the faces of B is also direct to check, since the coefficients of a polynomial on a facet form a subset of the coefficients of this polynomial in the box.

In addition to these tests, we also test whether both surfaces penetrate the cell, since a point on the curve must lie on both surfaces. This test could be done by looking at the sign changes of the Bernstein coefficients of the surfaces with respect to that cell. If no sign change occurs, we can rule out the possibility that the cell contains any portion of the curve $\mathcal{C}$, and thus terminate the subdivision early. In this case, we will also say that the cell is regular.

This regularity criterion is sufficient for us to uniquely construct the topological graph g of $\mathcal{C}$ within B. Without loss of generality, we suppose that the curve $\mathcal{C}$ is x-regular in B. Hence, there is no singularity of $\mathcal{C}$ in B. Furthermore, this also guarantees that there is no 'turning-back' of the curve tangent along x-direction, so the mapping of $\mathcal{C}$ onto the x axis is injective. Intuitively, the mapped curve should be a series of non-overlapping line segments, whose ends correspond to the intersections between the curve $\mathcal{C}$ and the cell, and such mapping is injective.

This property leads us to a unique way to connect those intersection points, once they are computed in order to obtain a graph representing the topology of $\mathcal{C}$, similar to the 2D method.

In order to apply this algorithm, we need to compute the points of $\mathcal{C} \cap B$, that is to solve a bivariate system of each facet of B. This is performed by applying the algorithm described in Section 1.3.

The special treatment of points of $\mathcal{C}$ on an edge of B or where $\mathcal{C}$ is tangent to a face requires the computation of tangency information at these points. This is performed by evaluating the derivatives of the defining equations of $\mathcal{C}$ at these points.

Collecting these properties, we have the following subdivision algorithm, which subdivides the domain B until some size ϵ, if the curve is not regular in B. It relies on a bivariate solver, for computing the intersection of the curve with the faces of the box.

ALGORITHM 2.4. TOPOLOGY OF A SPACE CURVE

INPUT: a curve $\mathcal{C}$ defined by equations $f_1 = 0, f_2 = 0, \ldots, f_k = 0$ and a domain $B = [a_0, b_0] \times [a_1, b_1] \times [a_2, b_2] \subset \mathbb{R}^3$ and $\epsilon > 0$.

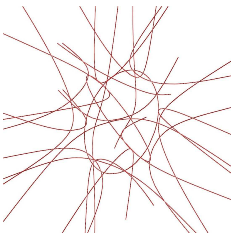
- For $1 \leq i < j \leq k$,
  - compute the Bernstein coefficients of the x, y, z coordinates of $\nabla f_i \wedge \nabla f_j$ in B
  - check that they are of the same sign for one of the coordinates (say x);
  - check the x-regularity condition on the facets of B.
- If such a pair (i, j) satisfying the previous regularity condition exists,
  - Compute the points of $\mathcal{C} \cap \partial B$ and connect them.
- else if $|B| > \epsilon$, subdivide B and proceed recursively on each subdomain.
- otherwise find a point p in $\overset{\circ}{B}$, compute the point $\mathcal{C} \cap \partial B$ and connect them to p.

OUTPUT: a set of points p and a set of arcs connecting them.

As in the 2D case, we have the following "convergence" property:

PROPOSITION 2.5. *For $\epsilon > 0$ small enough, the graph of points and arcs computed by the algorithm has the same topology as $\mathcal{C} \cap B$.*

Example. This curve is defined by $f(x, y, z) = 0$, $\partial_z f(x, y, z) = 0$ where



$$f(x, y, z) = 4(\tau^2 x^2 - y^2)(\tau^2 y^2 - z^2)(\tau^2 z^2 - x^2) \\ -(1 + 2\tau)(x^2 + y^2 + z^2 - 1)^2, \\ \tau = \frac{1}{2}(1 + \sqrt{5}),$$

of degree 6 is defining the surface S called Barth's Sextic. This curve, called the polar curve of S in the z direction, is of degree $30 = 6 \times 5$. We compute the topology by approximating the coefficients of f and $\partial z f$ by floating point numbers.

2.4. Subdivision method for surfaces. In this section, we consider a surface $\mathcal{S}$ defined by the equation $f(x, y, z) = 0$, with $f \in \mathbb{Q}[x, y, z]$. We assume that f is squarefree, that is f has no irreducible factors of multiplicity ≥ 2. For more details, see [1].

Unlike in the 2 dimensional case, the topology of the singular locus and the way to smooth locus is attached to it can be really complicated. Topologically we can characterize the topological situation as follows:

- Near a 2-dimensional stratum the topology is the same as a hyperplane.
- Near a 1-dimensional stratum the topology is the same as a cylinder on a singular planar curve.
- Near a 0-dimensional stratum the topology is the same as a cone with base the surface intersect a small ball containing the 0-dimensional stratum.

Moreover, we know only one of these three situations can and will happen locally. So we just have to design a solution for each one of the above three cases.

For efficiency reasons the criteria we have designed work for situations more general than the limit three cases. The 2-dimensional strata criterion can succeed even with several hyperplanes in the box not only just one. For the 1-dimensional strata we can triangulate even when some patches of the 2-dimensional strata lie in the box even though they are disconnected from the singular locus (in the box). In the case of the 0-dimensional strata we need to have the exact topology in the box. Of course the criteria eventually succeed if the box is small enough. We now describe each one of these criteria and the matching connection algorithm.

Let $B = [a, b] \times [c, d] \times [e, f] \subset \mathbb{R}^3$ and a surface $\mathcal{S} \subset B$. The boundary of B is denoted hereafter by ∂B. The x-faces (resp. y, z-facet) of B are the planar domains of the boundary ∂B of B, orthogonal to the direction x (resp. y, z).

DEFINITION 2.5. *The surface $\mathcal{S}$ is z-regular (resp. y, z-regular) in the domain B if,*

- *$\mathcal{S}$ has no tangent line parallel to the z-direction (reps. x, y-direction),*
- *$\mathcal{S} \cap F$ is regular, for F a z-facet (resp. x, y-facet) of B.*

We will say that $\mathcal{S}$ is *regular* in B if it is regular in B for the direction x, y or z. Here again, if a point $p \in \mathcal{S}$ is singular, then any line through this point is tangent to $\mathcal{S}$ at p. Thus a surface in B which is regular is also smooth.

PROPOSITION 2.6. *[1] If $\mathcal{S}$ is regular in B, then its topology is uniquely determined by its intersection with the edges of B.*

As in the 2D case, simple tests of regularity can be derived from the representation of f in the Bernstein basis.

PROPOSITION 2.7. *Let (u, v, w) be any permutation of (x, y, z). Suppose that the coefficients of $\partial_u f$ in the Bernstein basis of B have the same sign $\in \{-1, 1\}$ and that the coefficients of $\partial_v f$ or $\partial_w f$ on the u-facets of B are also of the same sign $\in \{-1, 1\}$. Then $\mathcal{C}$ is regular in B.*

This criterion implies that in the valid cells, the derivative of f in one direction is of constant sign and on the two faces transverse to this direction, another derivative is of constant sign. This may be difficult to obtain, when a point of the surface where two derivatives vanish is on (or near) the boundary of the cell. A situation where $\partial_u f \neq 0$ but where both derivative $\partial_v f$, $\partial_w f$ are not of constant sign on a u-facet F of B, can be handled by applying recursively the 2D algorithm of the facet F.

For handling the singularities of an algebraic surface $f(x, y, z) = 0$, we exploit the properties of the polar variety $\mathfrak{C}_z(\mathcal{S})$ defined by $f(x, y, z) = 0, \partial_z f(x, y, z) = 0$. For that purpose, we apply the implicit space curve algorithm of Section 2.3.

ALGORITHM 2.6. TOPOLOGY OF A SURFACE

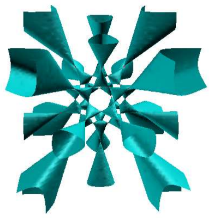
INPUT: a surface $\mathcal{S}$ defined by a squarefree equation $f(x, y, z) = 0$, a domain $B_0 = [a_0, b_0] \times [a_1, b_1] \times [a_2, b_2] \subset \mathbb{R}^3$ and $\epsilon > 0$.

- Compute generators $g_1, \ldots, g_k$ of the ideal $I(\mathfrak{C}_z(\mathcal{S})) = (f, \partial_z f) : J(f, \partial_z f)$.
- Compute the Bernstein coefficients of the f and g_i in the Bernstein basis of $B := B_0$.
- If $\mathcal{S}$ is regular in B, compute its topological structure.
- Else if the polar variety $\mathfrak{C}_z(\mathcal{S})$ is regular and connected in B, compute the topological structure of $\mathcal{S} \cap \partial B$ by Algorithm 2.3 on each facet of B
- Else if $|B| > \epsilon$, subdivide the box B and proceed recursively on each subdomain.
- Otherwise find a point p in $\overset{\circ}{B}$, compute the topological structure of $\mathcal{S} \cap \partial B$ by Algorithm 2.3 and its link over p.

OUTPUT: a set of points, arcs and patches and adjacency relations describing the topology of $\mathcal{S} \cap B_0$ arcs connecting them.

As in the previous case, for $\epsilon > 0$ small enough, the output of this algorithm is topologically equivalent to $\mathcal{S} \cap B$.

Example.



Here is the Barth sextic surface whose polar variety has been computed in section 2.3. This surface of degree 6 has the maximum number of isolated singularities for this degree, that is 65. These singular points are also singular points of its polar variety.

3. The design of the library. As illustrated in previous sections, various internal representations (eg. dense Bernstein basis or sparse representations for polynomials) for the abstract data-types (eg. vectors, polynomials) are required, together with algorithm specializations on these representations. The library SYNAPS makes it possible to define parametrized but efficient data structures for fundamental algebraic objects such as vectors, matrices, monomials and polynomials ... which can easily be used in the construction of more elaborated algorithms.

We pay special attention to genericity in designing structures for which effectiveness can be maintained. Thanks to the parametrization of the code using *templates* and to the control of their instantiations using *traits* and *template expressions* [14], they offer generic programming without losing effectiveness. We need to combine generic implementations, which allow to reuse code on different types of data representation, with specialized implementations tuned to specific critical operations on some of these data structures. This is typically the case if we want to use 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 should coexist with generic ones on matrices. In order to optimize the implementation, while avoiding rewriting several times the same code, we need to consider hierarchical classes of data-structures and a way to implement specializations for each level in this hierarchy. In this section, we describe the design of the library, which allows such a combination. Since, it is aimed to be used in dedicated applications, we should also take care of techniques to embed the library functionalities into an external application. In this section, we also describe the method, that we adopt to build dynamic modules, combining generic and special code through a transparent plugin mechanism. This approach is illustrated by the connection with the geometric modeler AXEL, which uses SYNAPS library for geometric computation on algebraic curves and surfaces.

3.1. The View Hierarchy. A mathematical object can have different logic representations. Moreover, the fact that a data structure represents a particular mathematical object is a meaningful information in generic programming in order to dispatch algorithms. Template functions can be defined for a formal type parameter T, assuming a semantically and syntactically well defined interface. This interface is called a *concept*. When an instantiation on T of a generic algorithm makes sense, we say that T is a *representation* of the *concept* assumed by the algorithm.

In order to dispatch a function call to the right generic version, people have to write some type declarations associated to T, specifying what kind of *concept* it implements. This can be done using *traits*, which are types wearing explicit information about types. In our development, we choose another solution, based on a *view* mechanism.

The idea behind a *view* is to reify a *concept* in order to dispatch algorithms by concepts using function overloading: when a data type `T` implements a *concept* `C`, we say that it *can be seen* as a `C`. A *view* is then defined as a generic type parametrized by a formal type `T` that is assumed to implement the *concept* associated to the *view*. As an example, `Polynomial<T>` is a *view* associated to the `Polynomial` *concept*, reifying the information that `T` implements the required interface for the `Polynomial` *concept*. Let's assume we write a generic function `f` expecting a type implementing the `Polynomial` *concept*, written in the namespace `POLYNOMIAL`:

```
namespace POLYNOMIAL {
    // generic implementation of f assuming the Polynomial concept
    template<typename Polynomial> void f(const Polynomial & p) { ... ; g(p) ; ... }
} ;
```

and the following function:

```
template<class PolynomialType> struct Polynomial {} ;

template<class PolynomialType> void f(const Polynomial<PolynomialType> & p) {
    using namespace POLYNOMIAL ; f((const PolynomialType&)p) ;
}
```

Then, by specifying that a data type T inherits from `Polynomial<T>`, the preceding code ensures that the generic function `POLYNOMIAL::f` will be used when there is no version of `f` written for T. A polynomial having dense representation can allow a better implementation of `f`, in this case, we can say that it implements the `DensePolynomial` *concept* which is a *sub-concept* of `Polynomial`.

To reify the idea of *sub-concept*, we define the `DensePolynomial<T>` *view* as inheriting from `Polynomial<T>`. This way, a type `T` inheriting from `DensePolynomial<T>`, will have an implementation of `f` corresponding to the first one available, *seeing* the type successively as a `T`, a `DensePolynomial<T>` and a `Polynomial<T>`. This mechanism also allows to specialize a function for a given representation (eg. representation in the Bernstein basis `bezier::rep1d<C>`), by providing the function

```
namespace bezier { template<class C> void f(const bezier::rep1d<C> & p) ; }
```

3.2. Interfacing with an interactive environment. The preceding *view* mechanism is in fact a way to associate functions to objects considering a hierarchy of *concepts*. It is actually closely linked to another kind of design by virtual classes, which we used to make the library collaborate with the modeler AXEL. In this framework, the external tool defines a virtual hierarchy of objects and each interface defines a set of member functions inheriting one from another, and, corresponds to a *concept*, each inheritance relationship being seen as a *sub-concept* declaration. We describe here the procedure used to link the static view hierarchy with the dynamic virtual hierarchy.

The interfaces `I` are implemented as classes with virtual functions. In order to automatically construct a class which implements the set of functions for the interface `I` and the representation `R`, we define a wrapper class `W<I,R>`.

To choose, at compilation time (see section 3.1), the most specialized function for a given data type, we define a view class `V<I,R>`, where `R` is a representation implementing the *concept* associated to the interface `I`.

Suppose that we have defined a function α, which associates to a type `T`, its interface class `I` (implemented by the traits class `interfaceof<T>`), and, that we also have defined β(`T`) operating on types, which computes the base class of `T` (accessible as `T::base_t`). Then for a given representation type `R`, the following inheritance relationship of classes defined by induction as:

$$\texttt{R} \to \texttt{W<}\alpha\texttt{(R),R>}; \texttt{W<I,R>} \to \texttt{V<I,R>}; \texttt{V<I,R>} \to \texttt{W<}\beta\texttt{(I),R>}; \texttt{V<}\emptyset\texttt{,R>} \to \alpha\texttt{(R)}$$

yields the following inheritance chain:

$$\texttt{R} \to \texttt{W<I}_0\texttt{,R>} \to \texttt{V<I}_0\texttt{,R>} \to \texttt{W<I}_1\texttt{,R>} \to \ldots \to I_*,$$

where I_0 is the interface of R and $I_1, \ldots, I_*$, the upper level of classes in the hierarchy. This allows us to define the specialized implementations of this hierarchy level by level and to automatically choose most optimized functions.

Let's assume that we have defined two `length` generic functions for the `ParametricCurve` and `BSplineCurve` *concepts*, that is, for the types `V<IParametricCurve,R>` and `V<IBSplineCurve,R>`:

```
template<class C, class R> C length(const V<IParametricCurve,R> & c, const C & eps)
{
    using namespace PARAMETRICCURVE ; return length(c.rep(), eps) ;
}

template<class C, class R> C length(const V<IBSplineCurve,R> & c, const C & eps)
{
    using namespace RATIONALCURVE ; return length(c.rep(), eps) ;
}
```

We define now the implementation of the interface, using the wrapper class associated to the `IParametricCurve`:

```
template<class R> struct W<IParametricCurve,R> : V<IParametricCurve,R>
{
    double length(double eps) const { return length(rep(), eps) ; }
} ;
```

In order to insert a specific representation of rational curves `MPolDseRationalCurve<C>` in the hierarchy, we use the following construction which specifies its interface and its implementation:

```
template<class C> struct interfaceof< MPolDseRationalCurve<C> >
{
    typedef IRationalCurve T ;
} ;
```

```
template<class C> struct MPolDseRationalCurve
: W< IRationalCurve,MPolDseRationalCurve<C> >
{
    typedef IRationalSurface base_t ;
    ...
} ;
```

This technique allows us to easily embed the library into an external interactive environment such as the geometric modeler AXEL, as we illustrate it now. This yields a plugin `ShapePlugin`, which will be compiled separately and loaded dynamically into the modeler. To build this plugin, we first furnish a factory from a list of types:

```
typedef type::gentlist<
    MPolDseRationalCurve<double>,
    MPolDseRationalSurface<double>,
    ...
>::T TypeList ;

void ShapePlugin::init(void)
{
    factory = new WShapeFactory<TypeList> ;
}
```

This factory allows to maintain a map from the interfaces of external tools (as in AXEL) to SYNAPS interfaces, which is enriched each time an object is created:

```
void ShapePlugin::create(QRationalCurve * c)
{
    IShape * shape = factory->interface("IRationalCurve") ;
    IRationalCurve * rc ;
    IParametricCurve * pc ;
    rc = dynamic_cast<IRationalCurve *>(shape) ;
    pc = dynamic_cast<IParametricCurve *>(shape) ;
    rc->setEquations(c->pw(),c->px(),c->py(),"t") ;
    pc->setRange(c->smin(),c->smax()) ;
    map.insert(c, shape) ;
}
```

The application forwards function calls over its objects to calls of functions in the plugin, following the same pattern. This approach allows to make code designed for different purposes, occasionally in different languages, coexist.

4. Conclusion. SYNAPS is an open source `c++` library for symbolic and numeric computations, that provides algebraic algorithms and data structures for manipulating polynomials, for solving polynomial equations and computing with real algebraic numbers. The wide range of the algebraic operations that are implemented in the library, as well as the design of it, based on the view mechanism, allows us to tune the algebraic operations in order to tackle difficult problems in non-linear computational geometry, such as the computation of the topology of real plane and space curves and surfaces. Last but not least, the design of the library permits the development of algebraic plugins that can be used as algebraic primitives to external software packages specialized to geometric modeling, such as AXEL.

SYNAPS is a continuous effort to combine symbolic and numeric techniques in algebraic computing under the generic programming paradigm. The library contains more than 200 000 lines of codes. In a short term, we plan to structure it into autonomous subpackages and to extend its capability to build dedicated plugins for external interactive tools, while improving the implementations for polynomials.

Acknowledgments: B. Mourrain, J.P. Pavone and J. Wintz are partially supported by the European projects ACS (Algorithms for Complex Shapes, IST FET Open 006413), AIM@Shape (IST Network of Excellence 506766) and the ANR project GECKO (Geometry and Complexity).

REFERENCES

[1] L. Alberti, G. Comte, and B. Mourrain. Meshing implicit algebraic surfaces: the smooth case. In L.L. Schumaker M. Maehlen, K. Morken (editors), *Mathematical Methods for Curves and Surfaces: Tromso'04*, pages 11–26. Nashboro, 2005.

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

[3] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry.* Springer-Verlag, Berlin, 2003. ISBN 3-540-00973-6.

[4] D. Bini. Numerical computation of polynomial zeros by means of aberth's method. *Numerical Algorithms*, 13, 1996.

[5] L. Busé, H. Khalil, and B. Mourrain. Resultant-based method for plane curves intersection problems. In *Proc. of the conf. Computer Algebra in Scientifc Computing*, volume 3718 of *LNCS*, pages 75–92. Springer, 2005.

[6] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*, 2nd edition. Undergraduate Texts in Mathematics. Springer, New York, YEAR= 1997.

[7] A. Dickenstein, M.J. Rojas, K.Rusekz and J. Shihx. Extremal real algebraic geometry and A-discriminants. Preprint, 2007.

[8] A. Eigenwillig, V. Sharma, and C. K. Yap. Almost tight recursion tree bounds for the descartes method. In *ISSAC '06: Proceedings of the 2006 International Symposium on Symbolic and Algebraic Computation*, pages 71–78. ACM Press. New York, NY, USA, 2006.

[9] I. Z. Emiris and E. P. Tsigaridas. Univariate polynomial real root isolation: Continued fractions revisited. In *14th Annual European Symposium on Algorithms*, volume 4168 of *LNCS*, pages 817-828. Springer, 2006.

[10] I. Z. Emiris, B. Mourrain, and E. P. Tsigaridas. Real algebraic numbers: Complexity analysis and experimentations. In P. Hertling, C.M. Hoffmann, W. Luther, N. Revol (editors), *Reliable Implementation of Real Number Algorithms: Theory and Practice*, *LNCS*, Springer-Verlag, 2007 (to appear).

[11] G. Farin. *Curves and surfaces for computer aided geometric design : a practical guide.* Comp. science and sci. computing. Acad. Press, 1990.

[12] M. S. Floater. On zero curves of bivariate polynomials. *Journal Advances in Computational Mathematics*, 5(1):399–415, 1996.

[13] J.C. Faugère and F. Rouillier. *FGB/RS Gröbner basis computation and real root isolation*, `http://fgbrs.lip6.fr/salsa/Software/`

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431, 1993.

[15] T. Gunji, S. Kim, M. Kojima, A. Takeda, K. Fujisawa and T. Mizutani. *PHoM – a Polyhedral Homotopy Continuation Method for Polynomial Systems*, Research Report B-386, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, Meguro, Tokyo 152-8552, Japan, December 2002,

[16] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A CORE library for robust numeric and geometric computation. In *15th ACM Symp. on Computational Geometry*, 1999.

[17] C. Liang, B. Mourrain, and J.P. Pavone. Subdivision methods for 2d and 3d implicit curves. In *Computational Methods for Algebraic Spline Surfaces.* Springer-Verlag, 2006. To appear.

[18] T. Lickteig and M.-F. Roy. Sylvester-habicht sequences and fast cauchy index computations. *J. of Symbolic Computation*, 31:315–341, 2001.

[19] H. Lombardi, M.-F. Roy, and M. Safey El Din. New structure theorems for subresultants. *J. of Symbolic Computation*, 29:663–690, 2000. Special Issue Symbolic Computation in Algebra, Analysis, and Geometry.

[20] V.J. Milenkovic and E. Sacks. *An approximate arrangement algorithm for semi-algebraic curves.* Proceedings of the 22th Annual Symposium on Computational Geometry, ACM, pages 237–245, June 2006.

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

[23] B. Mourrain. *Pythagore's dilemma, Symbolic-Numeric Computation and the Border Basis Method*, pages 223–243. Mathematics and Visualisation. Birkhäuser, 2006.

[24] B. Mourrain and J.-P. Pavone. Subdivision methods for solving polynomial equations. Technical Report 5658, INRIA Sophia-Antipolis, 2005.

[25] B. Mourrain and Ph. Trébuchet. Solving projective complete intersection faster. In C. Traverso, editor, *Proc. Intern. Symp. on Symbolic and Algebraic Computation*, pages 231–238. New-York, ACM Press., 2000.

[26] B. Mourrain and Ph. Trébuchet. Generalised normal forms and polynomial system solving. In M. Kauers, editor, *Proc. Intern. Symp. on Symbolic and Algebraic Computation*, pages 253–260. New-York, ACM Press., 2005.

[27] T. Ogita, S. Oishi. *Fast Inclusion of Interval Matrix Multiplication.* Reliable Computing 11(3): 191-205 (2005)

[28] J.P. Pavone. *Auto-intersection de surfaces pamatrées réelles.* PhD thesis, Université de Nice Sophia-Antipolis, 2004.

[29] F. Rouillier and P. Zimmermann. Efficient isolation of a polynomial real roots. *Journal of Computational and Applied Mathematics*, 162(1):33–50, 2003.

[30] S. Rump. *Computational error bounds for multiple or nearly multiple eigenvalues*, Linear Algebra and its Applications, 324 (2001), pp. 209–226.

[31] E. C. Sherbrooke and N. M. Patrikalakis. Computation of the solutions of nonlinear polynomial systems. *Comput. Aided Geom. Design*, 10(5):379–405, 1993.

[32] H. J. Stetter. *Numerical polynomial algebra.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2004.

[33] Ph. Trébuchet. *Vers une résolution stable et rapide des équations algébriques.* PhD thesis, Université Pierre et Marie Curie, 2002.

[34] J. Verschelde. *Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation.* ACM Transactions on Mathematical Software 25(2): 251-276, 1999

[35] J. von zur Gathen and J. Gerhard. *Modern computer algebra.* Cambridge University Press, New York, 1999.

[36] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Oxford University Press, 2000. ftp:/Preliminary/cs.nyu.edu/pub/local/yap/algebra-bk.html.