Submitted to ISSAC '95

# A Toolkit for Algebra and Geometry

(Extended Abstract)

Ashutosh Rege *          John Canny*

Computer Science Division
University of California
Berkeley, CA, 94720

**Abstract**

We describe a toolkit for solving a certain class of problems in algebra and geometry. Among other things, the toolkit can solve systems of multivariate polynomial equations and inequalities over the reals and complex numbers. Many problems from various branches of engineering and science can be formulated using systems of polynomial equations and inequalities. However, there is no practical system around that can deal with large problems of this kind. Our toolkit takes advantage of specialized representations like straight-line programs to improve the efficiency of computations with polynomials. It implements several improvements in algorithms for sign determination and for symbolic-numeric computation and provides an efficient scheme for arithmetic. The early success we have had with this toolkit, as exemplified by some of the benchmarks, indicates that it should provide an efficient general-purpose approach to solving large problems from a variety of domains.

## 1 Introduction

In this paper, we describe a toolkit for solving a certain class of problems in computational algebra and geometry. The kinds of problems we are interested in include :

- Problems in algebraic geometry such as point location in varieties, computing resultants, geometric theorem proving.
- Problems in geometric modelling such as geometric constraint satisfaction.
- Solving systems of polynomial equations and inequalities over both algebraically closed and real closed fields.
- Efficient use of infinitesimals for problems in geometry.

There were several motivating factors for the development of this toolkit :

*E-mail: {rege,jfc}@cs.Berkeley.edu. Fax: +1-510-642-5775. Supported by a David and Lucile Packard Foundation Fellowship and by NSF Grant FD93-20588

- Improved algorithms and data-structures for polynomial computations allow us to perform fundamental operations more efficiently. Our toolkit implements an efficient scheme to perform basic operations like computing, for example, Sylvester resultants, fast. Our approach is based on using *straight-line programs* to represent all intermediate polynomial computations and using interpolation to recover the polynomials as required. Additionally, we implement efficient *mixed arithmetic* to perform arithmetic operations.

- We make use of recent improvements in symbolic-numeric algorithms for higher-level computations such as sign determination and for computations involving semi-algebraic sets [5].

- An important impetus for developing this toolkit was the absence of a convenient platform in order to implement these ideas to obtain an efficient system. A computer algebra system such as Maple should be ideal for the implementation of such algorithms. However, such an approach results in several drawbacks. One is the fact that we use specialized data-structures like straight-line programs for performing a lot of our polynomial computations. Secondly we perform all field arithmetic operations using a mixed approach where a number is represented as a pair - an integer modulo a prime and a floating-point approximation. Implementing relatively low-level data structures such as these is not very convenient in systems like Maple. Further, we wanted to implement several algorithms, such as interpolation of straight-line programs to be space efficient, requiring us to control the amount of memory allocated for such computations. Consequently we have implemented our toolkit in C. It should be emphasized that the toolkit is not intended to be an alternative to standard computer algebra systems. Rather it is meant to complement such systems by providing a specialized set of routines to perform a certain class of operations. In fact our toolkit can interface easily with computer algebra systems like Maple or Mathematica. This allows the user to use the power of both the computer algebra system and the toolkit as and when required.

The fundamental approach we use in our toolkit is the reduction of multivariate polynomial computations to computations with univariate polynomials. Though the toolkit can perform multivariate polynomial computations if required, one instead represents a multivariate polynomial as a univariate polynomial with coefficients which are straight-line program representations of the coefficient polynomials. Additionally, our toolkit implements algorithms using a resultant-based approach to reduce a given problem defined by multivariate polynomials to an equivalent problem over univariate polynomials.

The paper describes these concepts in more detail and is organized as follows : Section 2 describes the theoretical underpinnings and the high-level algorithms we implement in the toolkit. We then give an overview of the various modules of the toolkit in Section 3. Section 4 describes the straight-line program approach we implement in our toolkit to make polynomial computations more efficient. Section 5 shows how to use straight-line programs to solve problems which require symbolic perturbation of the input polynomials by infinitesimals. In Section 6 we describe some of the arithmetic issues involved in our toolkit and we conclude with a discussion of applications of the toolkit and future directions in Section 7.

## 2 Theoretical Background for the Toolkit

In this section we describe the theory and algorithms behind some of the modules of the toolkit. One of the primary functions of the toolkit is solving systems of equations and inequalities over both the reals and complexes. Our approach to these problems is to reduce a multivariate problem to a univariate one via the use of *resultants*. Consider a system of $n+1$ polynomials in $n$ variables, $x_1, \ldots, x_n$. If we treat the coefficients of these polynomials as variables, the *resultant* of this system is a polynomial in these coefficients which gives us a condition for the solvability of the system. The system of polynomials has a common solution if and only if the resultant is identically zero. Examples of resultants include the classical Sylvester and Dixon resultants for 2 polynomials in 1 variable and 3 polynomials in 2 variables respectively. More generally, one is interested in determining the resultant of polynomials with an arbitrary number of variables. The classical approach to this was to use Macaulay's construction of the resultant as a quotient of two determinants [12]. However, this approach does not take advantage of the sparseness of the input system. Instead one can use the construction given in [2] to obtain a multiple of the resultant whose size depends on how sparse the input system is. This approach has been used successfully to obtain solutions of systems of equations among other things [2], [8].

In the rest of this section we discuss show how to use a resultant-based approach to encode the roots of the system using univariate rational (i.e. quotient of two polynomials) functions. We can then apply this technique to solve systems of inequalities over the reals.

### 2.1 Encoding roots of systems of polynomials

The function of this module of the toolkit is to take a system of $n$ polynomials in $n$ variables

$$f_1(x_1, \ldots, x_n) = \cdots = f_n(x_1, \ldots, x_n) = 0$$

which has solutions $\xi^{(i)} \in \mathbb{C}^n$, for $i = 1, \ldots, N$, and produce from it a univariate polynomial $p(s)$ and rational functions $r_1(s), \ldots, r_n(s)$. The pair $(p, r)$ is a symbolic description of the roots $\{\xi^{(i)}\}$ in the following sense: If the roots of $p(s) = 0$ are $\alpha^{(i)} \in \mathbb{C}$, for $i = 1, \ldots, N$, then

$$\xi^{(i)} = r(\alpha^{(i)})$$

Put another way, $r(s)$ is a parametric curve in $\mathbb{C}^n$ which passes through all the roots $\xi^{(i)}$, $i = 1, \ldots, N$. The values of $s$ at which it arrives at a root are precisely the roots of $p(s) = 0$.

There are several ways to construct $(p, r)$ from $f$. One method uses polynomial GCDs and was described in [3]. A more efficient method based on differentiation was given by Renegar [14], and that is the one we use in the toolkit. Compared to the cost of computing the polynomial $p(s)$, Renegar's method computes all the $r_i(s)$'s in only $n$ times as many operations.

To start, we add to the system $f = 0$, a linear polynomial

$$s - l(x) = 0, \qquad \text{where } l(x) = l_0 + l_1 x_1 + \cdots l_n x_n$$

and compute the resultant $R(s, l)$. Suppose that $l$ is specialized to a random linear polynomial $L$, then with probability one, the values $l(\xi^{(i)})$ will be distinct for distinct $\xi^{(i)}$. The roots of the resultant $R(s, l = L)$ will be $s = L(\xi^{(i)})$, and will also be distinct. We set $p(s) = R(s, l = L)$ and we are halfway to our goal.

We obtain the functions $r_i$ by differentiating $R(s, l)$. We define $r_i$ as

$$r_i(s) = \left. \frac{\left(\frac{dR(s,l)}{dl_i}\right)}{\left(\frac{dR(s,l)}{dl_0}\right)} \right|_{l=L}$$

The reader can verify that the $r_i$'s have the correct values at $s = \alpha^{(j)}$ by computing the derivatives of $R(s,l)$ in its factored form $R(s,l) = \prod_{j=1}^{N}(s - l(\xi^{(j)}))$.

In our implementation, it is particularly easy to compute the derivatives. All the algorithms in the symbolic toolkit use an SLP (straight-line program) representation of arithmetic (see section 4 below). The polynomial $p(s) = R(s, l = L)$ is computed explicitly. That is, we compute SLPs for each of the coefficients $p_0, \ldots, p_m$ of $p(s)$. Because they depend on the coefficients of $l$, they can be written $p_i(l)$. Then we compute the derivative of the SLP for $p_i(l)$ with respect to some $l_j$. For each node $v$ of $p_i(l)$, we add a new node representing the derivative of $v$ with respect to $l_j$. On average, this roughly doubles the size of the SLP (see section 4). Computing all the derivatives of $R(s,l)$ increases the size of the SLP by a factor of $n+2$.

This method works even if the system $f = 0$ has roots with multiplicity $> 1$. When this happens, $p(s)$ will have repeated factors for all choices of $l$. The partial derivatives above will all be zero. But we can remove the repeated factors by first choosing a random $l$, setting $p(s) = R(s,l)$ and then defining

$$\hat{p}(s) = p(s)/GCD(p(s), \frac{dp}{ds})$$

This time, we define $r_i = \frac{d\hat{p}}{dl_i} / \frac{d\hat{p}}{dl_0}$. The partial derivatives are again computed using SLPs, and they will be non-zero for almost all choices of $l$.

We have applied this approach, implemented in our toolkit, to efficiently solve systems of triangular polynomial equations such as those arising in the problem of geometric theorem proving [13]. In this case, we use the Sylvester resultant iteratively to eliminate variables one at a time to give us an efficient theorem prover (see Section 7).

## 2.2 A Polynomial Inequality Solver

This section describes the algorithm we plan to implement for solving systems of polynomial inequalities. As shown in figure 1, the primary function of the inequality solver is to determine solvability of systems of polynomial inequalities over the real numbers. A numerical approximation to the solution, if any, can be easily extracted. The solver can also compute optimal solutions to systems of inequalities, given a polynomial objective function.

The input is a predicate that specifies a *semi-algebraic set*. The predicate can be a general boolean combination, rather than simply a conjunction, of polynomial inequalities:

**Definition 2.1** *A semi-algebraic set $S$ is the set of points in $\mathbb{R}^n$ satisfying a predicate of the form $B(A_1, \ldots, A_k)$ where $B : \{0,1\}^k \rightarrow \{0,1\}$ is a boolean function and each $A_i$ is an atomic formula of one of the following types:*

$$(f_i = 0),\ (f_i \neq 0),\ (f_i > 0), (f_i < 0),\ (f_i \geq 0),\ (f_i \leq 0) \tag{1}$$

*with each $f_i$ a polynomial in $x_1, \ldots, x_n$ with rational (for our computational purposes) coefficients.*

Semi-algebraic (SA) sets are a versatile class that includes the forms of most familar objects in $\mathbb{R}^3$, such as cones, cylinders, spheres, and combinations of these. SA sets in 3D are slightly more general than CSG (constructive solid geometry) models, for example, SA sets include sets of mixed dimension, and sets which are not topologically closed. Both of these are forbidden in CSG.

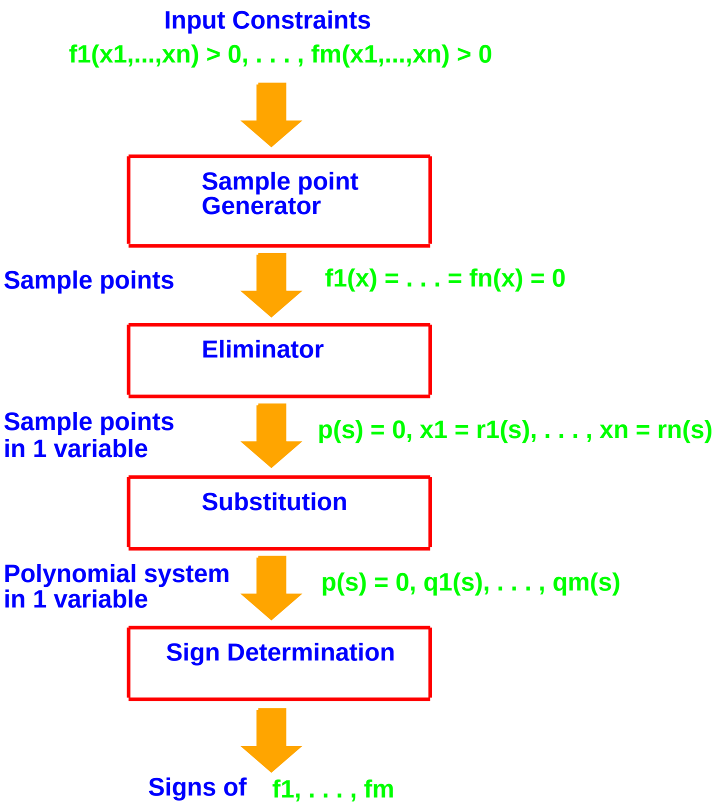


Figure 1: Symbolic subsystem modules

SA sets are defined in any dimension, and can also be used to represent the set of legal (e.g. obstacle-avoiding) configurations of a mechanical system, such as a robot, or the set of camera transformations consistent with some geometric constraints, or the set of positions at a given distance from a surface to guide a milling machine. The defining formulae for SA sets are the real analogue of SAT formulae, and they provide the ability to search over a space of real values for a solution satisfying some property.

The goal of the inequality solver then, is to decide if a semi-algebraic set $S$ is non-empty, and to produce a sample point if it is. We first choose a generic linear map $\pi : \mathbb{R}^n \rightarrow \mathbb{R}$ by selecting $n$ random integers $\pi_1, \ldots, \pi_n$. Then the algorithm proceeds as follows:

- **Sample point enumeration:** To determine if the predicate is satisfiable, the solver attems to find a witness point in the semi-algebraic set the predicate defines. This is done by enumerating the critical points of the generic map $\pi$ on the common zeros of subsets of the input polynomials. To do this, we first define a polynomial

$$g \;=\; \sum_{l=1}^{j} f_{i_l}^2$$

and solve the system

$$g = \mu \qquad \frac{\partial g}{\partial x_2'} = \cdots = \frac{\partial g}{\partial x_n'} = 0 \tag{2}$$

"in the limit" as $\mu \rightarrow 0$. Thus, we enumerate all subsets of $j \leq n$ polynomials $\{f_{i_1}, \ldots, f_{i_j}\}$. Do this in order of increasing $j$, so that "easy" witness points will be found early.

- **Elimination:** For each subset, construct a representation of the critical points of $\pi$ as a polynomial $p(s)$ and rational functions $r_1(s), \ldots, r_n(s)$ as described in Section 2.1.

- **Substitution:** Substitute $r_i(s)$ for $x_i$ in the other polynomials, giving $q_i(s) = f_i(r(s))$ for $i = 1, \ldots, k$.

- **Sign Determination:** Determine the signs of the $q_i(s)$ at roots of $p(s) = 0$ using the algorithm of [5]. Substitute these signs into the formula $B$ to check if the corresponding critical point lies in $S$. If yes, $S$ is non-empty, so return "true".

- Continue until no more sample points, and then return "false".

# 3 Overview of the Toolkit

The toolkit is made up of several modules ranging from low-level arithmetic modules implementing operations over a variety of fields to high-level modules such as those for encoding roots, sign determination etc. as detailed below. Starting from the lowest level and moving upward, here's an outline of the modules of the toolkit that have been implemented or will be implemented in the near future :

- **Field Arithmetic Modules**. This level implements various fields that the user might be interested in doing computation over. The fields that have already been implemented include integers modulo a prime, floating-point, mixed arithmetic (see Section 6), and straight-line programs (see Section 4). Additionally, we expect to implement arbitrary precision integer arithmetic and floating-point interval arithmetic very soon. The user can add a new field very easily by providing code for performing the basic field operations since the higher-level polynomial routines are oblivious to the type of field the polynomial computations are being done over.

- **Straight-line Program Module**. Though SLP's constitute a field themselves, and are treated as such in the modular hierarchy of the toolkit, they require various specialized operations which are implemented in this module. These include routines to take derivatives of SLP's, evaluate SLP's, check singularity and interpolate SLP's as described in Section 4.

- **Polynomial Arithmetic Module**. Polynomials in the toolkit are represented using a sparse recursive structure. A polynomial therefore is defined over one variable with coefficients which are themselves polynomials. We use this representation in the toolkit since most of our algorithms involve reduction of the polynomial to a univariate one with coefficients which are defined over the field of straight-line programs as discussed in Section 4.

- **Polynomial Routines**. This module implements the some of the fundamental polynomial operations required in the higher-level algorithms we are interested in. These include computing Sylvester resultants, subresultant and Sturm-Habicht sequences and encoding roots of polynomial systems.

- **Sign Determination Module**. This module implements the algorithm of [5] to determine the sign of a set of univariate polynomials at the roots of another univariate polynomial.

- **Polynomial Inequality Solver** (Not implemented yet). This module will solve systems of polynomial inequalities over the reals as described in Section 2.2.

# 4 Straight-line Programs

A fundamental approach in the toolkit is the reduction of multivariate polynomial computations to univariate ones. Though our toolkit can perform explicit multivariate polynomial arithmetic, it is more efficient not to do so. Instead, we represent multivariate polynomials as univariate polynomials in the leading variable with coefficients which are straight-line program representations of the multivariate coefficients in the rest of the variables. As we shall see below, this allows us to perform several fundamental computations more efficiently. Further, straight-line programs enable us to conveniently implement the principle of "lazy or deferred evaluation", i.e. the idea that we perform actual computations only when we need to do so.

Straight-line programs have long been recognized as an efficient approach to solving problems involving multivariate (or even univariate) polynomials [11], [9]. A straight-line program (or SLP) can be represented as a directed acyclic graph. The interior nodes in the graph represent arithmetic operations such as addition. Each interior node has outdegree (at most) two with children which represent the operands (which could be interior nodes themselves). The terminal nodes have no children and represent the input to the program which could be numbers or variables. An SLP thus represents a rational computation over the variables and numbers which form its input. In other words, the set of all straight-line programs over a given field $K$ and over a given set of variables, say $x_1, \ldots, x_n$ is in 1-1 correspondence with the field of rational functions $K(x_1, \ldots, x_n)$.

## 4.1 Computing with SLP's

The toolkit performs a number of its multivariate polynomial computations by representing all multivariate polynomials in $K[x_1, \ldots, x_n]$ – as well as all intermediate (possibly rational) computations – as univariate polynomials in $K(x_2, \ldots, x_n)[x_1]$, i.e. as univariate polynomials defined over the field of SLP's. The space required to represent a multivariate polynomial as an SLP is linear in both the degree and number of variables, as well as in the number of terms. Note the difference between the recursive *polynomial* representation and the corresponding *straight-line program* representation. In the former, the representation for a polynomial is a sparse recursive structure (i.e. the coefficients are also polynomials). In the latter, a polynomial is represented as a straight-line program. The former structure allows recovering information like coefficients of a polynomial (and their coefficients) easily which is not the case with the SLP form.

Computing with SLP's is easy and takes constant time per operation. For example to multiply two polynomials (or rational functions) represented by two SLP's, we simply create a new node representing the operation multiply and set the two given SLP's as its children.

Another fundamental operation with SLP's is evaluation of the SLP's. When performing basic polynomial arithmetic over $K(x_2, \ldots, x_n)[x_1]$ we often need to check if the SLP coefficients of the intermediate polynomials are identical to zero. To do this we evaluate the SLP setting random values for the variables $x_2, \ldots, x_n$. Evaluation of an SLP involves a simple depth-first search of the SLP graph with each SLP node storing its value when it is computed. This takes time that is linear in the size of the SLP. It is possible that the SLP being evaluated at a certain value of the inputs could be singular. This could happen when the SLP represents a rational function of the type $f(x) \cdot g(x)/g(x)$, where the function $g$ evaluates to 0 at the given input values. To obtain the correct value of the SLP at the given values of the input variables, we use L'Hopital's rule, i.e. we take successive derivatives of the SLP till we get a non-singular SLP. The derivative SLP has size which is at most five times the size of the SLP, though on average it will be much smaller than that. We will discuss this in more detail in section 5 on computing with infinitesimals.

### 4.2 Interpolating Straight-line Programs

There are various computations which involve knowing the exact polynomial represented by an SLP. That is, we would like to represent the polynomial using the sparse recursive structure mentioned earlier. We use interpolation to recover a polynomial from its SLP form. Consider the case where we want to interpolate an SLP with one input variable (say $x$) into its polynomial form. We can do this easily by using any (univariate) interpolation scheme such as Vandermonde interpolation [15]. A depth-first search of the SLP will give the degree $d$ in $x$ of the polynomial represented by the SLP. To interpolate, we need the value of the polynomial at $d+1$ points where $d$ is the degree of the polynomial. This can be obtained with $d+1$ evaluations of the SLP. Interpolating is now straight-forward and can be done in time proportional to $\mathcal{O}(d^2)$ and space $\mathcal{O}(d)$ [15]. If N is the number of nodes in the SLP, the total time required is therefore $max(\mathcal{O}(d^2), \mathcal{O}(d \cdot N))$ and the space required is $\mathcal{O}(d)$.

The table below gives benchmarks for doing some of the SLP computations mentioned above on a Sun Sparc 10.

| Computation | Time in seconds |
|---|---|
| Creating $10^6$ SLP's | 4.8 |
| Evaluating SLP of size $10^5$ | 0.15 |
| Taking derivative of SLP of size $10^5$ | 1.7 |

### 4.3 An Example

We now give an example of the power of SLP's to make multivariate polynomial computation more efficient. We consider the problem of computing the Sylvester resultant of two dense bi-variate homogeneous polynomials both of degree $d$. The Sylvester resultant of two bivariate polynomials can be computed using a sub-resultant based algorithm such as that of Collins [7],[1]. We treat the input bivariate polynomials as polynomials in one variable say $x$ and compute the resultant with respect to it. The resultant will be a polynomial in the second variable, $y$, and will have degree $d^2$. Suppose we use simple algorithms for polynomial arithmetic operations in this computation. The coefficients generated in the resultant computation would be polynomials in $y$ of degree $\mathcal{O}(d^2)$ and the basic operations with these would require time proportional to $\mathcal{O}(d^4)$. There are $\mathcal{O}(d^2)$ such operations and the total complexity of using this approach would be $\mathcal{O}(d^6)$.

On the other hand, if we use straight-line programs, the complexity of computing the resultant would be the same as that for univariate problems - $\mathcal{O}(d^2)$. However, the resultant will be obtained as an SLP in $y$ of size $\mathcal{O}(d^2)$. To recover it as a polynomial in $y$, we use interpolation as explained above. The time complexity of interpolation therefore is $\mathcal{O}(d^4)$ and the space required is $\mathcal{O}(d^2)$ (the total complexities are also of the same respective orders). When $d$ is even as small as 20, the savings obtained from this approach can be substantial. The following table gives running times of the toolkit for computing the resultant using this approach as compared with

Maple – the times are for computing the resultant of two dense homogeneous polynomials defined over $K[x, y]$ where K is the set of integers mod a given prime. The computations were done on a Sun Sparc 10.

| Degree of input polynomials | Degree of resultant | Toolkit time (secs) | Maple time (secs) |
|---|---|---|---|
| 10 | 100 | 1.0 | 6.9 |
| 20 | 400 | 10.4 | 107.5 |
| 30 | 900 | 95.5 | 755.0 |
| 40 | 1600 | 265.0 | - |
| 50 | 2500 | 651.0 | - |

# 5 Computing with Infinitesimals

A standard approach to dealing with singularities in various algebraic and geometric problems defined by polynomials is to perturb the input polynomials with infinitesimals [4], [10]. This de-singularizes the problem, and preserves important properties like connectivity or non-emptiness of the input. Perturbations of this kind require computation over the base field augmented by infinitesimals. Consider the case where we introduce one infinitesimal $\epsilon$ : the computations are now carried out over the extension field $\mathbb{Q}(\epsilon)$. Each number is a rational function (i.e. a ratio of two polynomials) in $\epsilon$. Arithmetic operations in this field correspond to operations with rational functions. The sign of such an element is obtained by performing an exclusive-or on the sign of the numerator and denominator - the sign of a polynomial in $\epsilon$ being given by the sign of the lowest degree non-zero coefficient. Similarly if we introduce $k$ infinitesimals, $\epsilon_1, \ldots, \epsilon_k$, the sign is obtained by first ordering the infinitesimals $\epsilon_1 << \epsilon_2 << \ldots << \epsilon_k$ and then taking the sign of the lowest degree element under this lexicographic ordering.

However, computing with explicit rational functions is very expensive, especially when we introduce a large number of infinitesimals, $\epsilon_1, \ldots, \epsilon_k$. More important, in algorithms of this kind, one often wants to know only the *sign* of a particular number given by a rational function. In the toolkit, we use straight-line programs instead of explicit rational computations. This results in an efficient scheme to determine the signs of elements from infinitesimal extensions.

## 5.1 Straight-line Programs for Infinitesimals

As discussed in section 4, straight-line programs over $\epsilon_1, \ldots, \epsilon_k$ are in 1-1 correspondence with the infinitesimal extension field $\mathbb{Q}(\epsilon_1, \ldots, \epsilon_k)$. Thus we can use SLP's to represent elements in the extension field.

We are typically interested in determining the sign of an element, say $f$, in this field. Given an SLP representing the element, we can easily determine its sign as follows : Consider the case where we have only one infinitesimal, $\epsilon$. The sign of the rational function $f(\epsilon)$ is given by the sign of the lowest degree term. To determine this sign we evaluate the SLP at $\epsilon = 0$. If $f(0)$ is non-zero, then the sign of $f$ is the sign of $f(0)$ otherwise we construct an SLP for the derivative $\frac{df(\epsilon)}{d\epsilon}$. Constructing the derivative SLP is straightforward - we create an SLP representation of the operations required for differentiation at each node and proceed in a depth-first fashion. The maximum number of operations required is five - at a division node. Thus the size of the derivative SLP will be at most five times the size of the original SLP. (On average it should be more like two times the size of the original.) We can now evaluate the derivative SLP at $\epsilon = 0$, to get the coefficient of $\epsilon$ in $f$. If this is non-zero, we're done as before, else we take successive derivatives with respect to $\epsilon$. The *total* program size to compute the $k^{th}$ derivative is $\binom{k+2}{2}$ times the original.

This process can be easily generalized to the case of more than one infinitesimal. It can then be shown

**Proposition 5.1** *Let $f(\epsilon_1, \ldots, \epsilon_m)$ be a polynomial, represented as an SLP with L nodes. If $\epsilon_1 \prec \cdots \prec \epsilon_m$ are infinitesimals, and if the lexicographically first term in $f$ is $c\epsilon^{k_1} \cdots \epsilon^{k_m}$, then an SLP for this term can be constructed having size $\leq Lk_1^2 \cdots k_m^2$, in the same number of steps.*

# 6 Arithmetic in the Toolkit

The toolkit combines two standard approaches to implementing arithmetic in computer algebra systems. In this *mixed arithmetic* approach, we represent a (possibly very large) integer number as a mixed number, i.e a structure with two values - one is the value of the integer modulo a prime and the other is the floating-point approximation of that number. Tests for equality of two mixed numbers are done using the finite field part. For determining the relative magnitude of the two numbers, we first check for equality and then use the floating-point portion. This implies that most of the computations in the toolkit are probabilistic but the probability of error is very small depending on the size of the prime chosen. The advantage of this approach is that in a lot of the algorithms we use, equality comparisons are more frequent than determining relative magnitudes. However, sometimes (for example, when doing sign determination over the reals) we do need correct signs for the mixed numbers. For large degree problems, it is possible that the accuracy of the floating-point computation may not be enough. In that case, we can extend the mixed approach by replacing the floating-point part by an interval of two floats. This feature has not been implemented yet in the toolkit but will be added in the very near future.

Even though we advocate the use of the above mixed approach, the toolkit allows easy introduction of any field that the user might be interested in doing computations over. The user has to only provide the C code for performing field operations and define a new type. Currently the toolkit implements arithmetic mod a prime, floating-point, extended floating-point (where a float is represented as a pair of numbers - a floating-point mantissa and an integer exponent), and mixed arithmetic. Additionally we intend to add arbitrary precision integer arithmetic and floating-point interval arithmetic.

# 7 Applications of the Toolkit and Future Work

We have applied this toolkit to the problem of geometric theorem proving. In [13] we describe an efficient algorithm for solving triangular systems of polynomial equations. We use this algorithm as part of our theorem prover for problems in geometry. The prover was implemented using our toolkit routines and determines the validity of geometric conjectures over the reals as well as complex numbers. The following table gives a comparison of the times obtained with our toolkit-based prover as compared to times for the prover outlined in [6]. The times for our prover were obtained on a Sun Sparc 10 as compared to a Symbolics 3600 for Chou's prover.

| Geometric Theorem | Chou's Prover | This Prover |
|---|---|---|
| Pascal Conic-1 | 14.43s | 0.38s |
| Pappus | 1.52s | 0.06 |
| Pappus Dual | 1.45s | 0.08 |
| AMS-1 | 4.05s | 0.23s |
| Simson | - | 0.37s |

Our primary goal at present is to complete the development of the toolkit in the very near future by implementing the polynomial inequality solver. We then intend to apply our toolkit to a variety of problems such as fast query algorithms for point location problems in geometric modelling, adapting the algorithm of [13] to geometric constraint satisfaction and solving connectivity problems for varieties.

# 8 Toolkit Availability

Version 1.0 of the toolkit is available by anonymous ftp from **robotics.eecs.berkeley.edu**. Read the file /pub/rege/toolkit/README to get further instructions.

# 9 Bibliography

[1] W.S. Brown and J.F. Traub. On Euclid's algorithm and the theory of subresultants. *Journal of the ACM*, 18:505–514, 1971.

[2] J. Canny and I. Emiris. An efficient algorithm for the sparse mixed resultant. In G. Cohen, T. Mora, and O. Moreno, editors, *Proc. 10th Intern. Symp. on Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, pages 89–104. Springer Verlag, May 1993. Lect. Notes in Comp. Science 263.

[3] J.F. Canny. Some algebraic and geometric computations in PSPACE. In *ACM Symposium on Theory of Computing*, pages 460–467, 1988.

[4] J.F. Canny. Computing roadmaps of general semi-algebraic sets. In *AAECC-91*, 1991. New Orleans.

[5] John Canny. Improved algorithms for sign-determination and existential quantifier elimination. *Computer Journal*, 1993. Special Issue on Quantifier Elimination.

[6] S.-C. Chou. Proving elementary geometry theorems using Wu's algorithm. In W.W. Bledsoe and D.W. Loveland, editors, *Theorem Proving : After 25 Years*. American Mathematical Society, 1984.

[7] G. Collins. Subresultants and reduced polynomial remainder sequences. *Journal of ACM*, 14:128–142, 1967.

[8] I. Emiris and A. Rege. Monomial bases and polynomial system solving. In *International Symposium on Symbolic and Algebraic Computation*, pages 114–122, 1994. Oxford, England.

[9] T. S. Freeman, G. Imirzian, E. Kaltofen, and L. Yagati. Dagwood: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software*, 14(3):218–240, 1988.

[10] J. Heintz, M.F. Roy, and P. Solerno. Complexité du principe de Tarski-Seidenberg. *Bull. Soc. Math. France*, 118:101–126, 1990.

[11] Erich Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *Journal of the ACM*, 35(1):231–264, 1988.

[12] F.S. Macaulay. On some formulae in elimination. *Proceedings of London Mathematical Society*, 33(1):3–27, May 1902.

[13] A. Rege. A complete and practical algorithm for geometric theorem proving. Submitted to the *ACM Symposium on Computational Geometry*, Vancouver, 1995.

[14] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals, parts I, II and III. Technical Report 852,855,856, Cornell University, Operations Research Dept., 1989.

[15] R. Zippel. *Effective polynomial computation.* Kluwer Academic publishers, Massachusetts, 1993.