A Toolkit for Algebra and Geometry

by

Ashutosh Gajanan Rege

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor John Canny, Chair
Professor David Forsyth
Professor Bernd Sturmfels

1996

The dissertation of Ashutosh Gajanan Rege is approved:

Chair Date

Date

Date

University of California at Berkeley

1996

A Toolkit for Algebra and Geometry

by
Ashutosh Gajanan Rege

Abstract

A Toolkit for Algebra and Geometry

by

Ashutosh Gajanan Rege
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor John Canny, Chair

This thesis deals with the efficient solution of systems of polynomial equations and inequalities, with particular emphasis on solutions over the reals. We describe the algorithms and data structures which form the basis of an implemented library of routines - a toolkit - for solving a variety of problems involving systems of multivariate polynomial equations and inequalities. The problems we are interested in solving can be broadly categorized as counting, enumerating, and answering queries about the common (real) solutions of such systems. We describe the application of the toolkit to problems in different areas including robotics, geometric modeling and geometric theorem proving. These problems raise interesting questions in their own right and we provide algorithmic solutions, of independent interest, to some of them.

The two fundamental components of our toolkit are the resultant and the straight-line program. We use resultant-based algorithms for a variety of symbolic computations in the toolkit including reduction of a multivariate system to a univariate one via elimination, symbolic encoding of the common roots of a system of polynomial equations and determining the signs of such a system at the common roots of another. We describe the use of the straight-line program approach to reduce the symbolic complexity of such algorithms.

The utility of these two ideas, i.e., resultant-based computations and straight-line programs is demonstrated in several problems: We demonstrate an efficient way to correctly count the number of real roots of a given system of equations. In geometric modeling, we describe the algorithms for the problem of implicit point location and predicates for geometry.

We also provide an algorithm for efficiently solving triangular systems such as those arising in geometric theorem proving. We give necessary and sufficient conditions for theorem proving over algebraically closed fields as well as over the reals. These algorithms and conditions are combined to provide an efficient geometric theorem prover which we have successfully applied to various nontrivial theorems.

Professor John Canny
Dissertation Committee Chair

Contents

Chapter 1

Introduction

The primary focus of this thesis is the efficient and practicable solution of systems of multivariate polynomial equations and inequalities with particular emphasis on solutions over the *reals*. The language of polynomial equations and inequalities affords description of a large variety of problems arising in disparate areas ranging from robotics and vision to economics. Once we have translated the particular problem under investigation into such a system of equations, our interest then becomes the determination of the solutions to this system efficiently without sacrificing correctness. The motivation for the work described in this thesis comes from two sources

- The development of improved algorithms and data-structures for computations involving polynomials.
- The absence of a convenient and widely available programming platform for implementing these algorithms.

This thesis describes the algorithmic basis and implementation of a library of routines - a toolkit - which provides a basic set of data-structures and efficient implementation of the afore-mentioned algorithms. The toolkit is intended to be used either as a stand-alone platform for developing applications based on these algorithms or as a supplemental library of routines which can be used as part of a larger application.

In the subsequent sections of this chapter we will address several issues which arise from the above discussion: First we consider a few example application problems from different domains to give us an idea of the kinds of problems we are interested in solving. It turns out that these problems can be categorized into various classes of problems involving

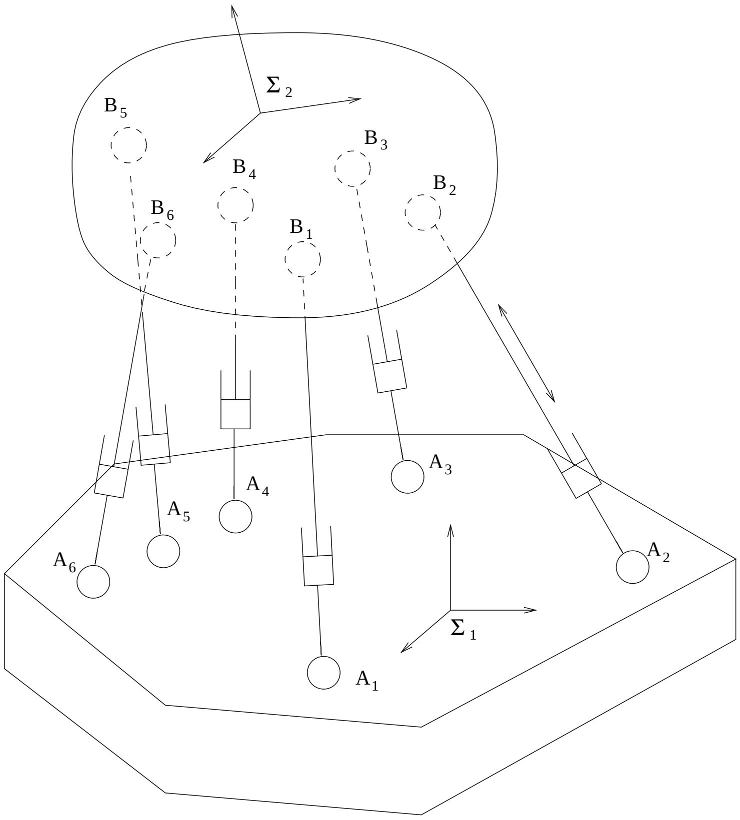

Figure 1.1: Stewart Platform

systems of polynomial equations or inequalities. We briefly review the state of the art with respect to the algorithms that can be used to solve problems within each class. We then give an overview of the particular techniques, data-structures and algorithms which will enable us to obtain these solutions, presumably in an efficient manner. And finally, we will consider the issue of what kind of role the toolkit would play in the context of current developmental platforms and software.

1.1 Example Problems

We begin with a first look at some problems drawn from different domains to give us a flavor of the kinds of problems we expect to be able to solve with the toolkit.

1.1.1 Forward Kinematics for the Stewart Platform

Figure 1.1 depicts the forward kinematics problem for the Stewart platform. The Stewart platform [Ste65] is a six-degree-of-freedom parallel manipulator comprising a base platform, a support platform (or *end-effector*) and six chains (or legs) connected in parallel between the base and the support. Each chain comprises a prismatic (i.e., sliding) joint with a universal ball-joint connection to the base and to the top platform - the prismatic joints serving as the independent inputs. The Stewart platform finds use in automotive and flight simulators, positioning tables for assembly, and in general, for applications requiring linkages with high structural stiffness.

The *forward kinematics problem for the Stewart platform* can then be stated as: given the values of the six prismatic joint lengths, compute the position and orientation of the end-effector (i.e. the top platform). Thus we have a well-defined problem with 6 input variables (corresponding to the lengths of the chains) and 6 output variables (corresponding to the position and orientation of the top platform). We can express the distance constraint for each leg of the Stewart platform as an equation as follows: Let Σ_1 and Σ_2 be the coordinate systems fixed to the base and top platforms respectively. Let A_i and B_i $i = 1, \ldots, 6$ be the points on the base and top platforms where the universal ball joints are located. Let $\mathbf{X}_{A_i}$ and $\mathbf{X}_{B_i}$ be the position vectors of the points A_i and B_i in the coordinate systems Σ_1 and Σ_2 respectively. Finally, let $\mathbf{p}$ be the position vector of Σ_2 in Σ_1 and let R be the 3 x 3 orthogonal change of basis matrix describing the orientation of Σ_2 in Σ_1. Then for the i^{th} leg we can write:

$$(\mathbf{p} + R\mathbf{X}_{B_i} - \mathbf{X}_{A_i}) \cdot (\mathbf{p} + R\mathbf{X}_{B_i} - \mathbf{X}_{A_i}) = {d_i}^2 \tag{1.1}$$

where d_i is the length of the i^{th} leg. For a given configuration, the values of the quantities, the *parameters*, $\mathbf{X}_{A_i}$, $\mathbf{X}_{B_i}$ and d_i are known. And the problem is to determine the corresponding values of $\mathbf{p}$ and R. The matrix R, being an orientation matrix, is defined by three scalar variables. Thus we get a system of 6 equations in 6 unknowns. We can write R as (see [Cra86]):

$$R = \begin{pmatrix} \cos\alpha\cos\beta & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma \\ \sin\alpha\cos\beta & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma \\ -\sin\beta & \cos\alpha\sin\gamma & \cos\beta\cos\gamma \end{pmatrix} \tag{1.2}$$

We can substitute (1.2) into (1.1) to get a system of 6 equations in the 6 unknowns, $\mathbf{p} = (x_1, x_2, x_3)$, α, β and γ. Unfortunately, this does not give us a system of *polynomial* equations. To convert them into polynomial form, we set $\sin\alpha = x_4$, $\cos\alpha = x_5$, $\sin\beta = x_6$, $\cos\beta = x_7$, $\sin\gamma = x_8$ and $\cos\gamma = x_9$ and introduce the additional constraints:

$$x_4{}^2 + x_5{}^2 = 1 \tag{1.3}$$

$$x_6{}^2 + x_7{}^2 = 1 \tag{1.4}$$

$$x_8{}^2 + x_9{}^2 = 1 \tag{1.5}$$

Combining these with (1.1) and (1.2), we get a system of 9 polynomial equations in 9 variables. Our primary objective is, of course, to solve this system of equations. In other words, we would like an algorithm which when given these equations, and input values for the parameters, will output the sets of values of the x_i's which satisfy the equations. It turns out, however, that in order to be able to do that, we first need to determine upper and lower bounds on the number of (finite) roots of this system. Thus, solution of the Stewart problem entails the solution of two problems: determining bounds on the number of roots of the system, and then providing an algorithm to determine these roots, in particular over the reals.

1.1.2 Implicit Point Location and Predicates for Geometry

Consider the equations for three spheres in $\mathbb{R}^3$ given below.

$$\begin{gathered} x_1{}^2 + x_2{}^2 + x_3{}^2 - 1 = 0 \\ (2x_1 - 1)^2 + 4x_2{}^2 + 4x_3{}^2 - 4 = 0 \\ 4x_1{}^2 + (2x_2 - 1)^2 + 4x_3{}^2 - 4 = 0 \end{gathered} \tag{1.6}$$

A simple first question we would like answered is whether the common intersection points in $\mathbb{R}^3$ of the above spheres lie inside, outside, or on the set given by the intersection of two other spheres:

$$\begin{gathered} (4x_1 - 3)^2 + (4x_2 - 3)^2 + (4x_3 - 3)^2 - 16 \leq 0 \\ (5x_1 + 1)^2 + (5x_2 - 1)^2 + (5x_3 - 3)^2 - 25 \leq 0 \end{gathered} \tag{1.7}$$

Simple enough as the problem sounds, it raises several important issues. First we note that the problem can be generalized to locating the intersection points precisely with respect to the two spheres - that is, we can imagine a subdivision of 3-space by the two spheres into 0, 1, 2 and 3-dimensional components and the question would be to determine in precisely which component each intersection point lies. The critical issue then is the question of determining the coordinates of the common intersection points. Clearly, a naive numerical approximation approach to solving equations 1.6 will not suffice since a numerical approximation can easily fail to give the correct answer in cases where the intersection points lie very close to the spheres 1.7. Further any numerical scheme would have to be *provably* correct, that is it should either provide a numerical error analysis of its computation or prove that the number of bits it used was sufficient for correctness. Another issue raised by this problem is that of efficiency - typically one would use an algorithm for solving such a problem as a subroutine for another algorithm. For example, in a CSG (constructive solid geometry) modeler, one might create a large model made up of various objects such as spheres, surface patches etc. In this case, queries of the above kind might occur very frequently and it would be important to solve several hundred or thousands of such queries in a matter of seconds.

Related to the point location problem is that of *efficient predicates for geometry*: Typical examples are the *InCircle* and *InSphere* predicates. In the former case, one is given (the cartesian coordinates of) three points a, b and c in the plane which determine a circle and a fourth point d which has to be located with respect to the circle. $InCircle(a, b, c, d)$ will return $+1$ if d lies inside the oriented circle abc, -1 if it lies outside, and a 0 if it lies on the circle. The *InSphere* predicate is defined analogously for five points. It can be shown easily that such predicates can be reduced to a determinant computation. The algorithmic issue then is the fast (and correct) determination of the sign of the determinant. Such problems can be generalized in two directions: (i) Determining predicates for more complicated objects such as surfaces of higher degree and (ii) Determining predicates for the case where the points and surfaces are defined implicitly, e.g., by polynomial equations. Even relatively elementary problems such as the *InCircle* and *InSphere* predicates pose non-trivial numerical problems. A more general version would have to solve essentially two problems: providing a predicate (i.e. a checkable condition or program which will provide the correct answer) and providing a numerically coherent and efficient approach to the checking process.

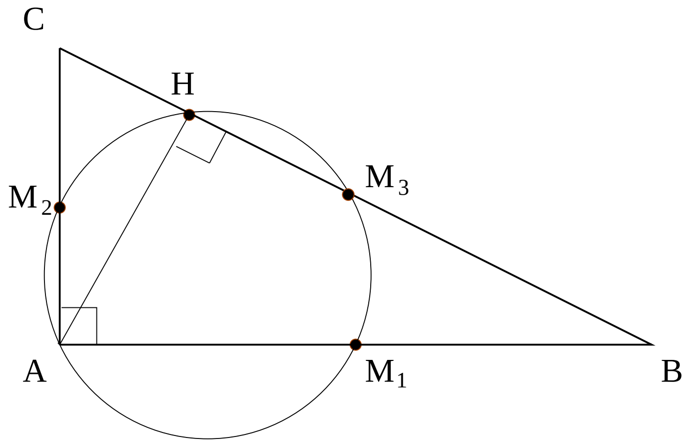

Figure 1.2: Circle Theorem of Apollonius

1.1.3 Constraint-based Drawing and Geometric Theorem Proving

In this section, we consider a different kind of problem - that of automated geometric reasoning. This sort of problem arises in artificial intelligence and in geometric modeling where one can prove or disprove conjectures about the relationship between geometric objects. In *constraint-based drawing* one is given a variety of geometric objects such as line segments, triangles, circles, spheres etc. and a partial list of pairwise constraints between them. The reasoning system is required to determine whether the given constraints are feasible, i.e. whether they correspond to one or more physically realizable configurations in two or three dimensional (Euclidean) space and if so, to determine the configurations.

A related (and harder) problem is that of *geometric theorem proving.* As an example of this kind of problem consider the following theorem statement illustrated by Figure 1.2:

Theorem 1.1.1 (The Circle Theorem of Apollonius). *Let ABC be a right triangle in the plane, with right angle at A. The midpoints of the three sides and the foot of the altitude from A to to the segment BC all lie on one circle. (See Figure 1.2.)*

Our reasoning system is required to "prove" this theorem. That is to output "true" or "false" for the conclusion (that the four points lie on a circle) given the hypotheses (the triangle is a right-triangle, the point H lies on the altitude etc.) We can view this problem as a constraint-based drawing problem as follows: the hypotheses correspond to constraints on the given geometrical objects, i.e., the constraints are that ABC has to be a right triangle, with right angle at A, that the point H in Figure 1.2 lies on the segment CD and the angle it subtends is a right angle and so on. The conclusion, that the points H, M_1, M_2 and M_3 lie on a circle, corresponds to another constraint on the configuration. All of these constraints can be expressed as polynomial equations as will be seen in Chapter 5. A *given* configuration, i.e, one in which the dimensions of the triangle are specified, will give a particular set of equations corresponding to these constraints. Thus, a particular instance of the theorem gives rise to a specific constraint-based drawing problem. However, we are required to determine the validity of the configuration for *all* right-triangles. Thus we need to *parametrize* the entire family of right-triangles and determine the validity of the above conclusion for the entire family. Further we would like to eliminate from consideration *degenerate* right triangles such as those with one or both of the right sides with length zero. In essence, therefore, we are required to solve a *family* of constraint-based problems and the addition of the parameters indexing members of the family adds to the complexity of the problem.

1.2 Recapping the Problems

From the above discussion we can see that the problems fall into a few broad categories all involving systems of polynomial equations and inequalities.

- **Counting** the number of (real) roots.
- **Enumerating** the (real) roots.
- **Symbolically encoding** the (real) roots.
- **Answering logical questions** about the (real) roots.

We will now provide a brief review of the main results in each category. A fundamental result from classical algebraic geometry is *Bezout's theorem* which essentially states that the number of common solutions of a system of polynomial equations is bounded from above by the total degree of the system (i.e. the product of the degrees of the polynomials in the system). More precisely we have (cf. [Sha74],[BR90]): if $p_1, \ldots, p_n$ are homogeneous polynomials of degree $d_1, \ldots, d_n$ respectively then if all the common solutions to the p_i's are isolated, the total number of solutions (counting multiplicities) is $d_1 \cdots d_n$. In practice, however, Bezout's bound turns out to be too loose; considerable improvement can be obtained on Bezout's bound by using the Bernstein-Kushnirenko-Khovanskii (BKK) bound ([Ber75], [Kus75], [Kho78]) which takes into account inherent sparseness in a given (generic) system of polynomial equations (see [Stu91] for an overview of sparse elimination theory). Several systems have proved themselves amenable to sparse elimination theory and tighter bounds for systems such as those defined by the cyclic n-roots problem and the "motion from point matches" problem in vision have been obtained [ER94].

The case of the reals is harder and not many general results exist on tight bounds for the number of real roots of a given polynomial system. If the real solutions are isolated then the Bezout bound can be used to get an upper bound on the number of real solutions. However, since the number of real solutions is in general much smaller than the total number of solutions, the Bezout bound can be very weak. An algorithm based on Hermite's method is given in [PRS92] for counting the number of real roots of zero-dimensional systems. This algorithm is based on computing a monomial basis (usually via a Gröbner basis computation) and setting up a multiplication table for the coordinate algebra. An alternate approach based on multivariate symmetric functions is given in [Ped91]. A similar approach is outlined in [ER94] where a monomial basis for the coordinate algebra of a zero-dimensional variety is obtained via the sparse resultant formulation [CE93]. The problem of counting and enumerating roots can then be reduced to a matrix eigenvalue problem [AS88], [MC92].

The standard method for encoding the roots of a system of polynomial equations is to compute a (generalized) primitive element polynomial for the system. Typically one computes (a multiple of) the resultant as outlined in [Can88] as the primitive element polynomial and a set of rational functions which encode the roots symbolically. A more efficient approach using the same general idea was given by Renegar in [Ren89b], [Ren92] and we will discuss this algorithm in more detail in Chapter 2. An alternate approach, based on much the same idea, goes by the name of the "Generalized Shape Lemma" [ABRW94]. The main difference from Renegar's approach is that the primitive element is obtained using a Gröbner basis computation rather than by resultant.

The general setting for the problem of answering logical questions about polynomial equations and inequalities is the *first order theory of the reals* [Tar48]. The language of the first order theory of the reals is made up of formulae of the form

$$Q_1 x_1 \ldots Q_n x_n F(x_1, \ldots, x_n) \tag{1.8}$$

where the Q_i's are either universal or existential quantifiers, F is a boolean formula of atomic predicates of the form $p_j(x_1, \ldots, x_n) \geq 0$ or $q_j(x_1, \ldots, x_n) > 0$. This theory has been shown to be decidable [Tar48] in time doubly exponential in the size of the input [Col75]. A useful restriction of the first order theory is the *existential theory of the reals* where only existential quantifiers are allowed. This has been shown to be decidable in polynomial space and single exponential time [Can88], [Ren89a]. Most of the work on real algebraic algorithms makes use of a sign-determination lemma due to Ben-Or, Kozen and Reif [BOKR86] which provides an algorithm for determining the signs of a set of polynomials at the roots on another. Since most logical questions such as those posed in section 1.1.2 can be rephrased in terms of signs of polynomials, this algorithm, and variants thereof [Can91a], can be used to solve a large class of such problems.

1.3 Thesis Motivation and Overview

Several factors motivated the development of the toolkit described in this thesis. The first is the development of a collection of techniques which improve the efficiency of solving systems of equations and inequalities. These include algorithms for sign determination [Can91a] and techniques for integrating symbolic and numeric computations which we will describe in Chapter 4. Additionally, we make use of the straight-line program data structure to represent computations involving polynomials and rational functions. Straight-line programs are used in the toolkit in two fundamental ways: We use straight-line programs to perform basic operations involving polynomial or rational functions. Coupled with interpolation, this gives us an efficient scheme to perform basic computations with polynomials such as computing pseudoremainders or Sylvester resultants. Straight-line programs further allow us to compute with systems of polynomials symbolically perturbed by infinitesimals. Both these uses in the toolkit are described in Chapter 3.

Another 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 the straight-line program data structure mentioned above, for performing a lot of our polynomial computations. Secondly we perform 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. Finally, the development of a serious single-purpose application is nigh impossible with a computer algebra system designed for general use. Consequently we have implemented our toolkit in C++.

It should be emphasized though that the toolkit is not necessarily intended to be an alternative to standard computer algebra systems. One can use it in conjunction with such systems with the toolkit 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. We intend the users of the toolkit to come from a broad spectrum ranging from mathematicians interested in the solution of specific computational problems related to algebraic geometry to engineers and computer scientists who would like to develop application programs using the toolkit as the developmental platform. A casual user might interested in solving a specific problem, i.e. some question related to a specific set of polynomial equations and/or inequalities. Such a user would probably use the Maple interface routines to call routines from the toolkit. On the other hand, someone working in an area like robotics might be interested in solving several problems (e.g., counting and enumerating the roots for a class of polynomial equations related to a particular robotic system) and would like to use the routines in the toolkit as procedures in the context of a broader application program.

The remainder of this thesis is organized as follows: In Chapter 2 we survey the basic concepts and algorithms we use in the toolkit, including resultants, algorithms for encoding roots and for sign determination. Chapter 3 deals with the fundamental data structure in the toolkit - namely the straight-line program. The rest of the thesis deals with the application of the toolkit to the solution of various problems. Though these applications demonstrate the utility of the toolkit, they pose questions in their own right and we consider those too within each chapter. In Chapter 4 we describe the application of the toolkit to the problem of implicit point location and predicates for geometry described in section 1.1.2. Chapter 5 deals with the geometric theorem proving problem. We conclude with an overview of the future directions that this research should and will likely take.

Chapter 2

Fundamental Algorithms and Toolkit Overview

This chapter deals with the basic concepts and algorithms which form the theoretical basis of the toolkit. The primary concept used in the toolkit is that of the *resultant* of a set of polynomials. As we shall see, the resultant allows us to encode information about the given system of polynomials in a compact form. Various questions about the system including the total number of roots, the number of real roots, the actual values of the roots can be recovered from the resultant.

The starting point for theory of resultants is perhaps the *Sylvester resultant* of two polynomials in one variable. The Sylvester resultant gives a (necessary and sufficient) condition on the coefficients of the two polynomials for the existence of a solution common to both. More generally, we have the *homogeneous resultant* of n homogeneous polynomials $f_1, \ldots, f_n$ in n variables. Letting c_{ij} be the j^{th} coefficient of the i^{th} polynomial, the resultant is a polynomial in the c_{ij}, where we treat the coefficients c_{ij} as indeterminates. The resultant vanishes identically when the system has a nonzero solution in $\overline{K}^n$, where $\overline{K}$ is the algebraic closure of the base field K. Further, the resultant is irreducible and is homogeneous in the coefficients of each f_i with degree

$$\prod_{j=1, j\neq i}^{n} d_j \tag{2.1}$$

where d_j is the degree of f_j.

The homogeneous resultant provides a condition for the solvability of homogeneous systems. However, it can vanish identically when no affine solution exists due to a component at infinity. The algorithmic question then is to construct this polynomial for a given system of equations. For the case of two polynomials in one unknown, the Sylvester formulation of the resultant (See, e.g. [Mis93] for a review of the Sylvester resultant and its properties) gives a determinantal condition on the existence of a common solution. The Dixon resultant [Dix08] gives a similar condition for the case of three polynomials in two unknowns. A more general construction is due to Macaulay [Mac21] who gave the formulation of the resultant of n homogeneous polynomials in n variables as the quotient of a determinant by one of its minors. The drawback of the Macaulay formulation (and formulations based on it, such as the u -resultant [vdW50] [Laz81]) is that it is infeasible to construct the resultant for even low degree input polynomials. For more details on these formulations see the excellent review by Kapur and Lakshman [KL92]. More recently, attention has focused on taking the inherent sparseness of the given system of equations into account. The foundations of *sparse elimination theory* were laid in the work of Gelfand, Kapranov and Zelevinsky [GKZ90] and Sturmfels [Stu91]. Canny and Emiris [CE93] provide an algorithm for constructing the so-called sparse resultant which, in general, gives a more compact condition than the Macaulay or other formulations.

To summarize then, we have in general a range of resultant formulations available to us. All of these formulations rely on the construction of a matrix which essentially encodes the properties of the given system of equations. It turns out, however, that it is not, in general, possible to provide *Sylvester type* formulae for the resultant (i.e. where the determinant of the matrix equals the resultant exactly). And most “resultants” obtained are really multiples of the actual resultant or projection operators rather than the actual resultant. The type of resultant we use will depend on the particular problem in hand. In Chapter 5, we will show that the Sylvester resultant is sufficient for solving triangular systems of equations. In other cases, such as the Stewart platform discussed in section 1.1.1 using the sparse resultant gives us a more compact formulation than can be obtained from other methods [Emi94] [ER94].

2.1 Encoding the Roots of Systems of Polynomials

A fundamental algorithm in the toolkit is that for *symbolically* encoding the roots of a given system of polynomials. The basic idea is to reduce the original multivariate problem to a univariate one by computing a (generalized) primitive element as follows:

Consider a system of n polynomials in n variables

$$f_1(x_1, \ldots, x_n) \;=\; \cdots \;=\; f_n(x_1, \ldots, x_n) \;=\; 0 \qquad (2.2)$$

Let the solutions to this system be denoted by $\xi^{(i)} \in \mathbb{C}^n$, for $i = 1, \ldots, N$. We will produce a univariate polynomial $p(s)$ and rational functions $(r_1(s), \ldots, r_n(s)) = r(s)$ such that the pair

$$(p(s), r(s)) \qquad (2.3)$$

is a symbolic description of the roots $\{\xi^{(i)}\}$ in the following sense: Let the roots of $p(s) = 0$ be $\alpha^{(j)}$, for $j = 1, \ldots, N$, then

$$\xi^{(i)} = r(\alpha^{(i)}) \qquad (2.4)$$

Thus, one could (conceptually) recover the roots of the original system 2.2 by solving for the roots of the univariate polynomial $p(s)$ and evaluating the n-tuple of rational functions $r(s)$ at these roots. More importantly, we get a *symbolic* encoding of the roots and we can use these (as we shall see) to answer logical queries involving the *signs* of other polynomials at the common solutions of 2.2.

2.1.1 Algorithm for encoding roots

There are several approaches to obtaining the encoding $(p(s), r(s)$ of equation 2.3. One method is to use polynomial greatest common divisors as outlined in [Can88]. An alternate approach based on differentiation was given by Renegar in [Ren89b], [Ren92]. This algorithm is more efficient in practice and is particularly suited for use with the straight-line program described in Chapter 3.

Algorithm 2.1.1 (Renegar's Algorithm)
Input: *Polynomials* $f_1, \ldots, f_n \in \mathbb{C}[x_1, \ldots, x_n]$.
Output: $(p(s), r(s))$, *satisfying 2.3 and 2.4.*

1. *Compute the resultant* $R(s, l_0, \ldots, l_n) =$ *(Resultant)*$(f_1, \ldots, f_n, g)$ *where*

$$g(s, x_1, \ldots, x_n) = s - (l_0 + l_1 x_1 + \cdots + l_n x_n) = s - l(x_1, \ldots, x_n) \qquad (2.5)$$

2. *Compute the rational functions* $r_i(s)$*:*

$$r_i(s) = \left. \frac{\left(\frac{dR(s,l)}{dl_i}\right)}{\left(\frac{dR(s,l)}{dl_0}\right)} \right|_{l=L} \tag{2.6}$$

where L *is a random linear polynomial, i.e. a specialization of* l*.*

3. *Set* $p(s) = R(s, l = L)$*.* $\square$

When we specialize l to L, the values $l(\xi^{(i)})$ will be distinct for distinct $\xi^{(i)}$ with probability one. Similarly the roots of the resultant $p(s) = R(s, l = L)$ will also be distinct. We can write $R(s, l)$ in its factored form

$$R(s, l) = \prod_{j=1}^{N} (s - l(\xi^{(j)})) \tag{2.7}$$

Taking derivatives as required for computing the rational functions it can easily be verified the r_i's have the correct values at $s = \alpha^{(j)}$.

We will use Renegar's algorithm, and variations thereof, as a basic routine in algorithms for solving several application problems in the toolkit including the point location problem and geometric theorem proving described in Chapter 4 and Chapter 5 respectively. Those chapters will also explore the use of straight-line programs in Renegar's algorithm, an approach which enables us to solve problems of high degree relatively efficiently.

2.2 Sign Determination

In several applications such as the point location problems described in Chapter 1 one can reduce various algorithmic questions to determining the signs of a set of univariate polynomials $q_1(s), \ldots, q_k(s)$ at the real roots of another polynomial $p(s)$. A fundamental result in computational real algebraic geometry is that due to Ben-Or, Kozen and Reif [BOKR86]: The input is the polynomial p and q_i's and the output is m sign sequences $\sigma \in \{+, -, 0\}^k$ where m is the number of real roots of $p(s)$. For each real root α_j of $p(s)$ a sign sequence σ_j is returned satisfying $\sigma_{ji} = \mathrm{sign}(q_i(\alpha_j))$.

Definition 2.2.1 *If* $f(s)$ *and* $g(s)$ *are two polynomials, the* **Sturm sequence** *of* f *and* g *is defined as the sequence* $r_0, r_1, \ldots, r_k$ *where* $r_0(s) = f(s)$*,* $r_1(s) = g(s)$ *and for* $i > 0$*,*

$$r_{i+1}(s) = q_i(s) r_i(s) - r_{i-1}(s) \tag{2.8}$$

where q_i is the quotient obtained by division of r_{i-1} by r_i. The **sign alternations** *of the Sturm sequence of f and g at a real value x is defined to be the number of changes in sign in the sequence $r_0(x)$, $r_1(x)$, $\ldots$, $r_k(x)$ and is denoted by* $\mathrm{SA}(f,g,x)$. *The quantity* $\mathrm{SA}(f,g,+\infty) - \mathrm{SA}(f,g,-\infty)$ *is denoted* $\mathrm{SC}(f,g)$.

Denote by c_g the set $\{x|f(x) = 0 \text{ and } g(x) > 0\}$ and by $\bar{c_g}$ the set $\{x|f(x) = 0 \text{ and } g(x) < 0\}$, where g is relatively prime to f. We then have,

Theorem 2.2.2 (Sturm's Theorem) *The number of real roots of $f(s)$ in the interval (a,b) equals* $\mathrm{SA}(f,g,a) - \mathrm{SA}(f,g,b)$. *Specifically, the number of real roots of $f(s)$ equals* $\mathrm{SC}(f,f')$, *i.e.,* $\mathrm{SC}(f,f') = |c_g| + |\bar{c_g}|$.

Theorem 2.2.3 (Tarski[Tar48]) $\mathrm{SC}(f,f'g) = |c_g| - |\bar{c_g}|$.

The following identity follows easily from the above:

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} |c_g| \\ |\bar{c_g}| \end{bmatrix} = \begin{bmatrix} \mathrm{SC}(f,f') \\ \mathrm{SC}(f,f'g) \end{bmatrix} \tag{2.9}$$

Thus for the case of one polynomial g we need to solve (2.9) to determine the signs of g at the roots of f. For the case of two polynomials g_1, g_2, we have similarly:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} |c_{g_1} \cap c_{g_2}| \\ |\bar{c_{g_1}} \cap c_{g_2}| \\ |c_{g_1} \cap \bar{c_{g_2}}| \\ |\bar{c_{g_1}} \cap \bar{c_{g_2}}| \end{bmatrix} = \begin{bmatrix} \mathrm{SC}(f,f') \\ \mathrm{SC}(f,f'g_1) \\ \mathrm{SC}(f,f'g_2) \\ \mathrm{SC}(f,f'g_1g_2) \end{bmatrix} \tag{2.10}$$

If we denote by A_1 the 2×2 matrix in (2.9) and by A_2 the 4×4 matrix in (2.10) then $A_2 = A_1 \otimes A_1$ where $\otimes$ denotes the Kronecker tensor product. In general for n polynomials, g_i, one can generate the matrix A_n analogously. Ben-Or, Kozen and Reif [BOKR86] give a method for solving such systems in time polynomial in n. An improvement on the BKR algorithm was given in [Can91a] and [Can93] where the matrix rank tests of the BKR algorithm were eliminated. This algorithm is the one we shall use in the toolkit.

2.3 An Overview of the Toolkit

The toolkit can be viewed as made up of several modules ranging from low-level arithmetic modules implementing a variety of arithmetics to high-level application code such as that for geometric theorem proving and point location. The toolkit implements a basic computer algebra system which allows the representation and manipulation of polynomials. The user can write application code using the basic polynomial routines much in the same way as with a general-purpose computer algebra system. Starting from the lowest level and moving upward, here's an outline of the modules of the of the toolkit:

Arithmetic. The toolkit provides various arithmetics that the user can use depending on the application. These include integer arithmetic modulo a prime, fixed precision floating-point, arbitrary precision integer and floating-point arithmetics (currently implemented using LEDA's "bignum" arithmetic [NU]), rational arithmetic, (implemented as pairs of integers), adaptive floating point arithmetic (see section 4.5.1 for a description) among others. Additionally, the toolkit provides "mixed" arithmetic where a number is represented as a pair - the integer value of the number modulo a prime, and a floating-point approximation. This approach is discussed further in section 4.5.2. The toolkit also allows the mixing of some combinations of pairs of arithmetics depending on whether the result is well-defined.

Straight-line Programs. The straight-line program (SLP) forms a key data structure in the toolkit and we provide a wide range of utilities for straight-line programs. These include SLP arithmetic, evaluation, interpolation, differentiation (both symbolic and otherwise), detecting singularities, and computing the sign of an SLP over a real ordered field. Chapter 3 discusses all of these operations in detail.

Polynomials. Polynomials are represented in the toolkit using the sparse recursive representation [GCL92]. That is, the sparse recursive polynomial data structure defines it as a polynomial in the leading variable with coefficients which are sparse recursive polynomials in the rest of the variables. Various algorithms for polynomials are provides including polynomial arithmetic, pseudoremainders etc. The algorithms are oblivious to the underlying ring or field of coefficients. A user can therefore easily add to the basic set of algorithms.

Special-purpose Routines. These are high-level routines for various computations such as those for computing resultants, Sturm sequences, encoding roots, sign determination etc.

Chapter 3

Straight-line Programs

3.1 Introduction

The representation of algebraic objects such as multivariate polynomials and rational functions is one of the important issues in the design of any computer algebra system. One of the standard representations for a polynomial is the *sparse recursive* representation: A polynomial $p \in A[x_1, \ldots, x_n]$ (where A is a ring) is represented recursively as a polynomial in x_1 with coefficients which are (sparse recursive) polynomials in $A[x_2, \ldots, x_n]$. Computations involving such standard representations often lead to intermediate expression swell, especially in algorithms using resultant-based techniques. One way to combat this problem is to allow the sharing of common subexpressions. Straight-line programs, or SLP's, allow us the compact representation for multivariate polynomial or rational functions. Informally, a straight-line program is a directed acyclic graph, which has two kinds of nodes: leaf nodes (i.e. nodes with in-degree zero) and operator nodes which represent arithmetic operations and which have in-degree at most two. Figure 3.1 shows an example of an SLP that computes $\epsilon_2\epsilon_3 - \epsilon_1\epsilon_2$.

Straight-line programs provide a convenient implementation of the principle of "deferred" or "lazy" evaluation. That is, instead of performing a given computation, one stores the computation in an SLP. When a particular question needs to be answered, one can query the SLP to determine the solution. Straight-line programs comprise perhaps the most important data structure in the toolkit, enabling us to perform several time-intensive computations relatively efficiently. 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 for a large class of problems. Instead, we represent multivariate polynomials as univariate polynomials in the leading variable with coefficients which are straight-line program representations of the multivariate coefficient polynomials in the rest of the variables. Inspite of providing an indirect representation for a polynomial or rational function, SLP's can be queried efficiently for answers to various questions. For example, one can compute the derivative of a rational function (as required by Renegar's algorithm for root encoding described in Chapter 2) by computing a derivative SLP of the SLP representing that function. Another area where SLP's are useful is where one wants to perform various computations on polynomials which have been *symbolically* perturbed by the addition of infinitesimals. This approach can be used in various algorithms in which the assumption is that the semi-algebraic sets corresponding to the input polynomials are in generic position [HRS90], [Can91b]. One provides a scheme to perturb the input with infinitesimals without changing properties such as connectivity. Typically, then, one is interested in determining the signs of the leading coefficients of various polynomials generated in the course of the computation. Again, normal computations with infinitesimals would be rather expensive, with the cost growing exponentially with the number of infinitesimals added - with SLP's one simply treats the infinitesimals as part of the SLP coefficients of the polynomials reducing the computations to constant time. Information about the signs of the coefficients can be recovered by computing the signs of their corresponding SLP's in the limit as the infinitesimals go to zero - as described in section 3.5.

Of course, SLP's do not provide a panacea for all such problems. For one, they make the recovery of information from the computation more difficult than in the normal case. For another, for some computations, the memory required to store the SLP's can grow fast with the size of the problem and the user will have to tradeoff the time required for the computation with the memory required. In the rest of the chapter we will consider various algorithms and implementation issues related to computations with straight-line programs.

3.2 Previous Work

Straight-line programs were introduced by Strassen [Str72] as a model for algebraic computation. Since then a considerable amount of work has been devoted to understanding the algebraic complexity of various problems in terms of straight-line programs. The prob-

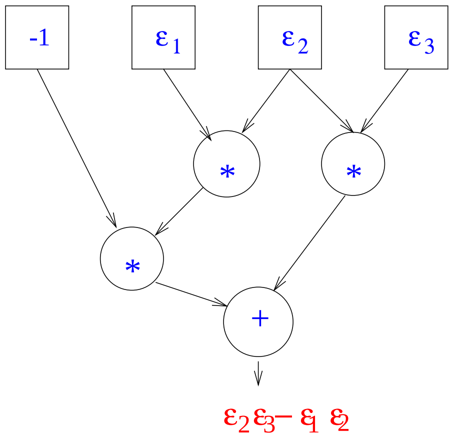

Figure 3.1: Straight-line program example

lem of equivalence of polynomials is considered in [HS80], [IM83]. Various problems such as greatest common divisors and factorization are considered in [Kal88], [Kal89]. The straight-line program view was extended to multivariate elimination theory in [GH93], [GJML95], [GJM+95]. Most of the above work was theoretical in nature, attempting to classify various algebraic problems with respect to their complexity as measured in the sizes of the SLP's etc. An implementation of a system for creating and manipulating SLP's is described in [FIKY88].

3.3 Straight-line Program Basics

There are two standard definitions of straight-line programs which lead to two distinct approaches for their implementation.

Definition 3.3.1 *(see [Str72], [vzG87]) Let A be a ring (resp. field) and $x_1, \ldots, x_n$ indeterminates over A. A* **straight-line program** *is sequence $P = (P_1, \ldots, P_r)$ of arithmetic operations where*

1. *Either $P_i \in A \cup \{x_1, \ldots, x_n\}$. These are called the* **inputs**.
2. *Or $P_i = (\omega_i, j_i, k_i)$, where $\omega_i \in \{+, -, *\}$ (resp. $\omega_i \in \{+, -, *, /\ \}$ for A, a field) is a binary operation and j_i, k_i are numbers of previous instructions.*

The **result sequence** *$(u_1, \ldots, u_r)$ is defined by $u_i = P_i$ in case 1 and by $u_i = u_{j_i}\omega_i u_{k_i}$ in case 2. We stipulate that no division by the rational function zero occurs. The program is* *said to* compute *the rational function* $u_r \in A(x_1, \ldots, x_n)$

The above definition can be thought of as the *program-theoretic* definition of an SLP. One is given a sequence of instructions which do not loop or jump (i.e. a "straight-line" program) which define the function that is computed. This model is useful when one is given actual code for computing a function. One can treat the code as an SLP and perform various operations with it.

The other definition is more useful when building an SLP for a function from the bottom up and is the one we shall use as the basis for our implementation of the SLP data structure.

Definition 3.3.2 *A* **straight-line program** *over a ring* A *(resp. field) with indeterminates* $x_1, \ldots, x_n$ *is a directed acyclic graph with node set* V *and directed edges* E *satisfying the following conditions:*

1. *Each node is labeled as one of two types: a* **leaf** *node or an* **operator** *or* **interior** *node.*

2. *The set of operators is* $\{+, -, *\}$, *(resp.* $\{+, -, *, /\ \}$ *).*

3. *A leaf has in-degree zero. All interior nodes have in-degree two.*

4. *Every leaf node* w *is assigned an element from* $A \cup \{x1, \ldots, xn\}$.

As before, an SLP computes a rational function in $A(x_1, \ldots, x_n)$.

The above definition then gives us a directed acyclic graph representation which has nodes which compute various rational functions and gives us a recursive representation of a rational function. That is, one takes an SLP as given by Definition 3.3.1 and converts it into one given by Definition 3.3.2 by going over each program statement. (It should be noticed that in practice, we define an SLP going *down* from the interior node which computes the given function. That is, the edges are directed down towards the operands of a given operator nodes. This is necessary since the fan-out (or out-degree) at a node in Definition 3.3.2 is unbounded - a node can be the operand for several nodes, whereas it can have at most two operands as inputs.)

From the above definitions, we can see that the set of all SLP's over $x_1, \ldots, x_n$ is virtually in 1-1 correspondence with the set of rational functions $A(x_1, \ldots, x_n)$. (The exception being the cases where two SLP's represent the same rational function.)

3.3.1 Computing with SLP's

The two basic operations for computing with SLP's are *creation* and *evaluation*. The creation phase involves creating SLP nodes from the given sequence of arithmetic operations - for a given operation on two intermediate variables a and b, one simply creates a node representing that operation with children $SLP(a)$ and $SLP(b)$. This 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.

Evaluation of an SLP proceeds in a recursive top-down fashion. At each node of the SLP we have a counter which keeps track of the last time the particular node was evaluated. The algorithm for evaluation then proceeds as follows:

Algorithm 3.3.1 (Simple SLP Evaluation)

Input: An SLP node v representing $f \in A(x_1, \ldots, x_n)$, $a_1, \ldots, a_n \in A$.
Output: $f(a_1, \ldots, n)$.

1. Increment global counter C by 1.
2. Return Evaluate$(v, a_1, \ldots, a_n)$

Procedure: Evaluate$(g, a_1, \ldots, a_n)$

1. If counter(g) is equal to C return $value(g)$.
2. Else if $op(g) \in \{+, -, *\}$, set

$$l_{val} = Evaluate(lchild(g), a_1, \ldots, a_n)$$

$$r_{val} = Evaluate(rchild(g), a_1, \ldots, a_n)$$

$$value(g) = l_{val}\ op(g)\ r_{val}$$

Set counter$(g) = C$.

3. Else if $op(g) = /$ set

$$l_{val} = Evaluate(lchild(g), a_1, \ldots, a_n)$$

$$r_{val} = Evaluate(rchild(g), a_1, \ldots, a_n)$$

If $r_{val} \neq 0$, set $value(g) = l_{val}\ op(g)\ r_{val}$ and counter$(g) = C$. else if $r_{val} = 0$, return "ERROR"

4. Else if g is a numerical leaf node, return $value(g)$

5. Else if g is a variable leaf node x_i, return a_i

One problem with the algorithm is that when the value of a divisor is 0 at the given points a_i, the algorithm terminates without providing the desired evaluation (if it exists) - this can be taken care of by taking derivatives, as described in section 3.4, and applying l'Hôpital's rule.

The basic algorithm for evaluating an SLP also generalizes to various other operations. For example, to compute the degree (with respect to some variable) of a polynomial represented by an SLP, one can easily modify the above algorithm replacing the value of an SLP node by the degree.

3.3.2 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 in section 3.1. Since SLP's essentially form a black box for the polynomial, we can 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 [Zip93]. 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. Interpolation is now straight-forward and can be done in time proportional to $\mathcal{O}(d^2)$ and space $\mathcal{O}(d)$ [Zip93]. 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 $\mathcal{O}(d)$.

Table 3.1 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

Table 3.1: Benchmarks for selected SLP computations (Sun Sparc 10).

A Simple Example

We now consider a simple 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 [Col67],[BT71]. 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. Table 3.3.2 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	-

Table 3.2: Time for computing the resultant of two bivariate polynomials (Sun Sparc 10).

3.4 Derivatives of Straight-Line Programs

Taking derivatives of straight-line programs is an important operation in the toolkit required primarily in two cases:

- In Renegar's algorithm (section 2.1.1), we need to compute the derivatives of the resultant with respect to the variables l_i in the linear form introduced for computing the resultant. Instead of treating these variables as symbolic variables, we will treat them as SLP variables. The resultant obtained then will be a an SLP in the variable s and the l_i. To get the rational function which encode the roots of the system, we have to take derivatives of this SLP with respect to each of the l_i.
- When computing with infinitesimals, we again need to take derivatives of various SLP's with respect to the infinitesimals as described in secton 3.5.

In the simplest case, one is interested in computing the (first) derivative SLP of an SLP with respect to one variable. This can be done easily by modifying algorithm 3.3.1. At each node for which the derivative has not been computed, instead of evaluation, we compute the derivative using the usual rules for differentiation. For example at a division node, $h = f/g$, we have $h' = (f'/g) + h * (g'/g)$. Thus we create the 4 additional nodes corresponding to the 4 new operations to get the SLP for h'. Division being the worst case, the derivative SLP will have size at most 4 times that of the original SLP.

The more interesting problem is when we are given a straight-line program computing $f \in A(x_1, \ldots, x_n)$ and we need to compute all derivatives

$$\partial_{x_1} f, \ldots, \partial_{x_n} f \tag{3.1}$$

This problem was considered by Baur and Strassen ([BS83]) who show that if one is given a multivariate rational function by an SLP (as given by Definition 3.3.1) then one can simultaneously compute all first order partial derivatives with only a constant growth in the length of the computation. As observed in [KS91] the Baur-Strassen construction can produce an SLP of depth as large as the number of instructions in the original SLP. A modified construction is provided in [KS91] which increases the length by a factor of no more than 4. This construction is easy to implement with the SLP model given in Definition 3.3.1. However, it is relatively complicated to implement their scheme in the graph-theoretic model of an SLP. This is because in the first form, the SLP is built from the bottom up, with each successive statement built from the earlier ones. This allows an easier implementation of the chain rule for differentiation. In the second case, the SLP is given in a top down fashion and applying the chain rule becomes harder to implement. We use an alternate scheme which allows us to compute the derivative SLP of an SLP when given in the graph-theoretic form and with the same worst-case penalty as in [KS91].

We first introduce a new type of SLP operator node called the "DIFF" or "∂" node. A DIFF node has only one input and the input edge must come from a leaf node of variable type. The algorithm for computing all derivatives then is the same as the algorithm for computing the derivative with respect to a single variable with the difference that when we encounter a leaf node of variable type, we introduce a ∂ node and make the leaf node its child. We can then use this SLP as essentially a *symbolic* derivative of the original SLP. The requirement however is that when we actually query this SLP, such as evaluating it at some point $(a_1, \ldots, a_n) \in A^n$, we have to specify the variable with respect to which the derivative was taken.

3.5 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 [Can91b], [HRS90]. 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 ϵ: the computations are now carried out over the extension field $A(\epsilon)$. Each number in this field is a rational function in ϵ. 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 ϵ 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.

3.5.1 Straight-line Programs for Infinitesimals

As discussed in section 3.3, every element in the infinitesimal extension field $A(\epsilon_1, \ldots, \epsilon_k)$ can be written as a straight-line program over $\epsilon_1, \ldots, \epsilon_k$. 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, ϵ. 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 four - at a division node. Thus the size of the derivative SLP will be at most four times the size of the original SLP. (In practice, for the kinds of problems we are interested in, such as computing resultants, it turns out to be around two times the size of the original.) We can now evaluate the derivative SLP at $\epsilon = 0$, to get the coefficient of ϵ in f. If this is non-zero, we're done as before, else we take successive derivatives with respect to ϵ. 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 3.5.1 *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.*

3.6 Application to Counting Real Solutions

We now describe an application of the concepts outlined in this chapter to the problem of counting the number of isolated real solutions of a system of two bivariate polynomials. That is, we are given two polynomials $f(x, y)$ and $g(x, y)$ (which describe varieties in $\mathbb{R}^2$) and we would like to count the number of common real solutions (i.e., the number of intersections in $\mathbb{R}^2$) to the system $f(x, y) = g(x, y) = 0$. We use the following straightforward algorithm (the notation $SLP(y)$ means y is treated as an SLP variable and all computations involving y are done using SLP's):

Algorithm 3.6.1 (Counting Real Solutions (Bivariate Case))
Input: *Polynomials $f(x, y)$ and $g(x, y)$ with $\mathcal{V}(f, g)$ 0-dimensional.*
Output: *The number of real common solutions to f and g.*

1. *Convert $f(x, y)$ and $g(x, y)$ to $f(x, SLP(y))$ and $g(x, SLP(y))$ respectively.*

2. *Compute Sylvester resultant $R(f, g, x) = SLP(R(y))$ (the resultant is obtained as an SLP in y).*

3. *Interpolate $SLP(R(y))$ using the Chinese remainder theorem [Knu81] to get $R(y)$.*

4. *Compute the Sturm sequence*

$$r_0(y) = R(y), r_1(y) = R'(y), r_2(y), \ldots, r_k(y)$$

   with numerical coefficients as SLP's.

5. *Interpolate the leading coefficients of the r_i's using the Chinese remainder theorem.*
   □

Degree f, g	Terms	Primes Used	Ave. Roots	APU time (s)	Posso Gb	HOM-PACK*
4	10	72	2.8	1.88	1.4	0.3
5	10	134	3.1	6.6	9.1	0.8
6	10	201	4.3	21.8	45.5	2.7
7	20	314	4.3	95.0	574.0	11.3
8	10	440	4.3	120.0	-	19.5
10	10	1200	6.0	1350.0	-	35.0

Table 3.3: Time for counting real solutions for two bivariate polynomials (Sun Sparc 10).

Complexity Analysis: Let d be an upper bound on the degrees of the input polynomials. The symbolic complexity is $\mathcal{O}(d^4)$ (using the same analysis as that in section 3.3.2). The numerical complexity is $\mathcal{O}(d^3)$ and can be seen as follows: assuming that the coefficients in the input polynomials are bounded by some number k, the computation of the resultant in step 2 will result in numbers of size k^{2d} (from the fact that we're basically computing a determinant of size $2d \times 2d$). Step 4 has a similar computation except the degree of the input polynomials is now d^2, therefore the numbers at the end of this computation will be of the order of $(k^{2d})^{2d^2} = k^{4d^3}$, and, thus, the bit complexity of the numbers will be $\mathcal{O}(d^3)$. The total complexity of the algorithm therefore is $\mathcal{O}(d^7)$.

3.6.1 Benchmarks for Counting Real Solutions

We will compare the implementation of our approach with two other methods - one based on Gröbner bases and Hermite's method [PRS92], the other on a homotopy-based algorithm [MSW89]. The former approach has been implemented as part of the "Gb" package [Fau94], [CF] and is indicated as such in the following table. The latter approach is based on using numerical continuation methods and is implemented in the Fortran package HOMPACK. The results of our toolkit are listed under the name "APU", the column "Primes Used" indicates the number of primes (of size roughly 10^6) used to interpolate the largest number using the Chinese remainder theorem.

As seen from the above table, our method outperforms the Gb implementation on the polynomials we attempted to solve the problem for. These polynomials were chosen randomly (and uniformly) using Maple's *randpoly* function with the restrictions that they be of the given degree etc. and with coefficients which were in the range -99 to $+99$. After degree 8, the Gb implementation consumed too much virtual memory (of the order of 150 Mb) and caused the machine to crash. HOMPACK, on the other hand, outperforms both APU and Gb. For the case of randomly chosen polynomial, it even returns the correct number of real roots. However, when the polynomials are selected to be close to degenerate cases, HOMPACK fails to give the correct number of roots (in some cases returning an even number of real roots even when the degree of the system was odd). Further, as far as we know, there is no error analysis provided by HOMPACK to determine when the number of real roots returned is unreliable.

3.7 Other Applications of Straight-line Programs

We have applied the concepts outlined in the previous sections to two other cases where straight-line programs make a significant difference. The first application is to the problem of implicit point location which is discussed in Chapter 4. Here the concept of SLP's is useful for the automatic generation of predicates for certain geometric problems. The second major application of SLP's is to the problem of geometric theorem proving which is discussed in Chapter 5.

Chapter 4

Implicit Point Location and Predicates for Geometry

4.1 Introduction

Point location problems are a fundamental area of interest in computational geometry [Ede87]. Problems of this kind arise in various contexts: In some applications one is given a subdivision of the Euclidean plane or 3-space and a query point, and one wishes to locate the query point with respect to the subdivision. In other applications, the problem is that of providing a geometric predicate which will answer logical questions such as whether a point (given by its cartesian coordinates) is located inside a given object, such as a sphere. Whatever the flavor of the problem, these predicates form core subroutines within the algorithms that use them and are called a large number of times for each run of the algorithm. Most of the work so far, however, has focused on problems involving lines, line segments, planes, polyhedra and other linear objects. Problems involving curves and surfaces tend to be computationally more difficult and the issues of numerical stability tend to get further compounded with such non-linear objects. In this chapter, we consider point location problems involving non-linear objects in two and three dimensions.

The abstract version of the problem we consider is this: we are given a 0 - dimensional variety V in $\mathbb{R}^2$ or $\mathbb{R}^3$, described by a set of polynomial equations. Additionally, we are given a semi-algebraic set which is described by conjunctions, disjunctions and negations of a set of polynomial equations. The problem is to determine whether the points in the variety V lie in the semi-algebraic set. (See section 1.1.2 for an example.)

Semi-algebraic sets provide a powerful language for expressing a wide variety of constraints. Most familiar objects in $\mathbb{R}^3$ such as cones, cylinders, spheres and combinations thereof constitute semi-algebraic sets. Algebraic and semi-algebraic sets are used for representing objects and constraints in geometric modeling, computer graphics, robotics and computer vision. Semi-algebraic sets form a super-set of CSG (constructive solid geometry) objects in geometric modeling. Once a CSG object is built, the most elementary query is to test whether a point is inside a solid, on its surface or outside it. Even if the point is given by its cartesian coordinates, locating it with respect to a solid is still a non-trivial task due to the difficulties arising when the point lies on the surface of the object [Hof89]. The more natural problem arises when the point is represented implicitly as the intersection of other geometric objects, where the complications are compounded. The two over-riding computational issues which come up when attempting to solve such problems are

- The inherent computational complexity of symbolic algorithms.
- Issues of numerical correctness and stability.

Symbolic algorithms for algebraic computation have been notoriously slow even on problems in low dimensions. Part of the computational complexity of these algorithms arises from the use of general-purpose methods, such as Gröbner bases, which though elegant and complete, are perhaps too powerful for use in solving certain kinds of problems. Our approach is based on encoding the common intersection points using resultants. The use of resultants allows us to reduce the point location problem to that of determining the sign of a collection of univariate polynomials at the roots of another univariate polynomial. We take the approach of trying to have our algorithm do the minimum amount of work it can by using the idea of straight-line programs (SLP's) as described in Chapter 3. The advantage of straight-line programs lies in the fact that they allow a convenient method do defer computation till such time as necessary. In problems such as point location, one is primarily interested in the *signs* of polynomials at the real roots of another. The use of SLP's allows us to minimize the computation required to determine these.

The other critical component of any algorithm for algebraic computation is that of numerical correctness. Fixed-precision floating point arithmetic has the problem that there are no simple techniques that guarantee the reliability of the result. One runs into fairly severe problems of numerical stability even when dealing with relatively uncomplicated computations such as those in planar computational geometry of line segments and polygons [FW93]. When dealing with higher degree objects such as spheres, the problem gets worse. Most algebraic computations result in polynomials with large coefficients, usually beyond the range of fixed-precision floating point. One solution is to use software multiprecision integer arithmetic. The problem with this approach is that in most applications, full precision is not required to obtain a correct solution and the computation is unnecessarily slowed down by several orders of magnitude. The alternate approach is to use fixed-precision floating point representation for the numbers. The primary problem with this approach (even assuming that we do not encounter underflow or overflow) is that the answers we get are not provably correct unless we provide an error analysis which shows that the number obtained at the end of a sequence of computations has a value which lies within certain bounds and whose sign can therefore be considered to be reliably correct.

We use the *adaptive floating point* approach [Pri91], [Pri92]. Informally, an adaptive floating point number is a representation of a number with an "adaptive" number of bits, i.e. the number of bits of a given number are computed till such point as we know the sign of the number correctly. The adaptive approach has the advantage that the amount of computation required will be the minimum required to determine the sign of a number correctly. Thus for "easy" computations, adaptive floating point will do the same amount of work as that done by fixed-precision floating point, whereas for "harder" cases, it will do compute sufficient number of bits to ensure correctness.

The main computational problem with adaptive floating point is that in the cases when the number is identically zero, the number of bits required to provably determine this fact can be very large. We propose a *mixed* approach to numerical computations - a number is represented as a pair, the integer value of the number modulo a prime and an adaptive floating-point approximation to the number. The former is used to detect whether the number is identically 0 (probabilistically), the latter to determine the sign of the number. This alleviates significantly the problem of deciding whether a number is identically zero - we first check its value modulo a large prime and only proceed with the adaptive expansion of bits if this value is not zero. In the case the value is 0 modulo a prime, we assume that the number is indeed zero. This, of course, makes our algorithm probabilistic. We note however that this is a completely different notion of probability from the case where we use just fixed-precision floating point numbers. In the mixed case, we can control the probability of error by the use of a large enough prime and by using multiple primes. In the latter case, the accuracy of the computation is strongly dependent on the actual numbers involved in the computation.

This chapter is organized as follows: Section 4.2 defines semi-algebraic sets and the problem we solve in terms of the same. Section 4.3 describes the overall algorithm that we use for point location. In Section 4.4 we describe the complete algorithm in more detail including the interaction between the symbolic portion of the algorithm and the SLP part. We conclude with a description of the implementation of our algorithm and benchmarks on some problems.

4.2 Problem Definition

We first introduce the notation and terminology that we will use in the rest of the paper.

Definition 4.2.1 *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 $B : \{0,1\}^k \to \{0,1\}$ is a boolean function and each A_i is an atomic formula of one of the following types:*

$$(f_i = 0),\ (f_i \neq 0),\ (f_i > 0), (f_i < 0),\ (f_i \geq 0),\ (f_i \leq 0) \tag{4.1}$$

with each f_i a polynomial in $x_1, \ldots, x_n$ with rational coefficients.

Let $f_1, \ldots, f_n$ be polynomials in $\mathbb{Q}[x_1, \ldots, x_n]$, and let $V = V(f_1, \ldots, f_n) \subset \mathbb{R}^n$ be the variety of $f_1, \ldots, f_n$. The problem can then be stated as:

Problem 4.2.2 *Given a set of polynomials $f_1, \ldots, f_n$ in $\mathbb{Q}[x_1, \ldots, x_n]$, and a semi-algebraic set $S \subset \mathbb{R}^n$ determine whether the variety V satisfies the set S.*

We will denote by $g_1, \ldots, g_m \in \mathbb{Q}[x_1, \ldots, x_n]$ the polynomials which define the semi-algebraic set S. The algorithm we will provide will in fact solve the slightly more general problem of determining the signs of the polynomials defining S at the points in V. Though the algorithm can work over arbitrary dimensions, our primary interest is to perform efficient point location in 2 and 3 dimensions.

4.3 Point Location in Semi-algebraic Sets

The algorithm for point location will proceed in three broad steps: first we encode the common roots of the f_i's symbolically via a resultant computation. This essentially reduces the problem to a univariate one. Next, we determine the signs of the polynomials g_j defining the semi-algebraic set at these common roots. And finally we test the whether the sign sequences so obtained satisfy the constraints of the semi-algebraic set. This last step is, of course, trivial once the sign sequences have been obtained. The next two sub-sections will discuss the first two steps of the algorithm in detail.

4.3.1 Encoding the roots of systems of polynomials

We are given 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 $i = 1, \ldots, N$, As discussed in section 2.1, we will produce an encoding of its roots from it as a univariate polynomial $p(s)$ and rational functions $r_1(s), \ldots, r_n(s)$ using Renegar's algorithm [Ren89b] described in section 2.1.1.

Computing the Resultant

Algorithm 2.1.1 presupposes a method for computing the resultant of $n+1$ polynomials in n variables. There are several approaches to doing this. These include using successive applications of the Sylvester resultant, Gröbner bases, characteristic set methods, all of which have the problem of inefficiency. For general multivariate systems, the two efficient methods for computing the resultant are the Dixon resultant and the "sparse" resultant. Since we are primarily interested in computing the resultant of low-dimensional systems we will use the Dixon method (see [KL92] and [KSY94] for more on the Dixon resultant formulation).

There are several considerations which need to be addressed when using resultants. First, for non-generic polynomials all resultant based methods do not guarantee that the polynomial obtained will provide a necessary or a sufficient condition for the solvability of the system of polynomials. Further, it is possible for the matrix obtained as the resultant matrix to be singular or non-square and even when the determinant is computable it may not be the actual resultant but rather a multiple of it.

As stated earlier, we will use the Dixon resultant formulation to compute the resultant of the given system. The non-genericity of the input polynomials can result in a matrix which is singular or non-square. In both these cases we can resort to using Gaussian elimination (using a fraction-free method like that of Bareiss [Bar68]) to obtain the determinant of a maximal non-singular minor. As reported in [KSY94] this approach seems to work in most cases. In Section 4.4 we will address the issue of extra factors introduced during the resultant computation.

As is clear from the description of Algorithm 2.1.1, the resultant will be computed for polynomials in $x_1, \ldots, x_n$ with coefficients in $s, l_0, \ldots, l_n$. Computing with symbolic coefficients is rather expensive. As outlined in Chapter 3, we will use straight-line programs and interpolations thereof to reduce the complexity of these computations to manageable levels.

4.3.2 Sign Determination

Once we have obtained an encoding $p(s), r(s)$ of the variety $V(f_1, \ldots, f_n)$, we can determine the signs of the g_j defining the semi-algebraic set S as follows:

First we substitute $x_i = r_i(s)$ in the g_j. This gives us a collection of univariate rational functions, $h_j(s)$, which correspond to the parametrization of the g_j by the roots of $f_1, \ldots, f_n$. Then the sign of g_j at the roots of of $f_1, \ldots, f_n$ is obtained by determining the sign of h_j at the roots of the resultant polynomial $p(s)$. This can be done efficiently by using the algorithm of [Can93].

4.4 The Complete Algorithm

We will now give a description of the complete algorithm based on the aforementioned ideas. The algorithm proceeds as follows:

Algorithm 4.4.1 *Point Location*

Input:

Polynomials $f_1, \ldots, f_n \in \mathbb{Q}[x_1, \ldots, x_n]$. Semi-algebraic set S described by polynomials $g_1, \ldots, g_m \in \mathbb{Q}[x_1, \ldots, x_n]$

Output:

True, if $V(f_1, \ldots, f_n)$ satisfies S; false, otherwise.

1. *Dixon matrix construction.* The Dixon matrix is constructed for the given polynomials $f_1, \ldots, f_n$ and $f_{n+1} = s - l(x) = s - l_0 - l_1 x_1 - \ldots - l_n x_n$. This involves computing the Dixon polynomial first and then recovering the coefficients of the various power products in $x_1, \ldots, x_n$ and $\alpha_1, \ldots, \alpha_n$. The result is a matrix D with entries which are polynomials in s and $l_0, \ldots, l_n$.

2. *Conversion to SLP form.* The entries in the matrix D are converted to straight-line programs in a straightforward manner. The entries of D can now be treated essentially as constants.

3. *Computation of the Dixon resultant.* If the Dixon matrix obtained in step 2 was square, we can compute its determinant using Bareiss' algorithm [Bar68]. If the determinant was not identically zero, we proceed to the next step. Else we have two cases: 1. The determinant is zero and 2. the matrix was not square. In both cases Bareiss' algorithm provides us with a Gaussian elimination scheme on D and we can return the determinant of a maximal non-singular minor as the last non-zero pivot. It is still possible, however, for the polynomial so obtained to be a multiple of the resultant rather than the exact resultant. We will take care of that possibility in a later step.

   All of the computation in this step is performed over SLP's and the complexity of this step is the same as the complexity of Bareiss algorithm with constant time for each arithmetic operation. The output of this step gives us us the resultant polynomial p as a straight-line program in $s, l_0 \ldots, l_n$.

4. *Computation of derivatives of the resultant.* As required by the algorithm for encoding roots, we now compute derivatives of the SLP for p with respect to $l_0, \ldots, l_n$. We can avoid having to compute multiple derivatives by simply using a symbolic node DIFF (as described in section 3.4) representing the differentiation operator applied to the input nodes of the SLP which represent variables. Thus the derivation operation needs to be applied only once. And we get a *symbolic* derivative SLP of p, denoted Dp.

5. *Interpolation of SLP's.* Given Dp we recover the derivatives of p w.r.t. each l_i as polynomials in s. Recall that we want to compute

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

That is we're interested in recovering the derivatives of $p(s, l_0, \ldots, l_n)$ *evaluated* at a random value L of l. We can therefore proceed as follows: For each variable l_i we interpolate the (symbolic derivative) SLP, Dp, at a fixed random value of l. This process involves evaluating Dp at different values of s. During the evaluation process we will encounter DIFF nodes which represent the symbolic derivatives of their child nodes (which represent input variables). If l_i is the variable w.r.t. which the derivative was taken, the value of a DIFF node is simply either 1 when the child node represents l_i or 0 when it represents $l_j, j \neq i$.

Thus after $n+1$ interpolation operations, one for each l_i, we recover the rational functions $r_i(s), i = 1, \ldots, n$.

The SLP for $p(s)$ is also interpolated to recover $p(s)$ as a polynomial. This completes the encoding of roots into the pair (p, r) as described in section 4.3.

6. *Reparametrizing the semi-algebraic set.* We now re-parametrize the set S by substituting the rational functions $r(s)$ in the g_j's describing S. That is we replace x_i by $r_i(s)$. Clearing denominators, this gives us a set of polynomials $h_j(s), j = 1, \ldots, m$, one for each g_j.

7. *Sign determination and consistency check w.r.t V* We now have a set of univariate polynomials $h_j(s)$ and the resultant polynomial $p(s)$. Recall from Section 2.1 that the roots of $p(s)$ give us the values of s corresponding to the points in the variety $V(f_1, \ldots, f_n)$. Thus we are interested in computing the signs of the $h_j(s)$ at the roots of $p(s)$ - these will be the signs of the g_j's at the common roots of the f_i's. We compute these signs using univariate sign determination algorithm of [Can93] which computes the signs of a set of polynomials at the roots of a given polynomial. These are output as as tuples $[sign(g_1), \ldots, sign(g_m)]$ for each root of $p(s)$, where $sign(g_j) \in \{0, +, -\}$.

   It could be the case, as mentioned earlier, that the polynomial $p(s)$ represents a proper multiple of the resultant rather than the resultant itself due to the addition of extraneous factors during the resultant computation. In order to take care of this possibility, we substitute $x_i = r_i(s), i = 1, \ldots, n$ in the original $f_1, \ldots, f_n$ to get the set of polynomials $f_1'(s), \ldots, f_n'(s)$. We now determine the sign tuples of $h_1(s), \ldots, h_m(s)$ *and* $f_1'(s), \ldots, f_n'(s)$ at the roots of $p(s)$. The correct sign tuples will have the signs of the $f_i'(s)$ as 0.

8. *Checking consistency w.r.t. the semi-algebraic set.* This step is straightforward: For each g_j we have its signs at the points of V. Thus every atomic clause describing the semi-algebraic set of the form $g_j \leq 0$ evaluates to either True or False. We combine the various True or False results for each g_j according to the boolean conjunctions or disjunctions to get the final result as True or False.

The computational complexity of the algorithm is dominated by the time required to interpolate the SLP's and the sign determination step. The time for SLP interpolation depends on the size of the SLP's produced. This quantity will be essentially equal to the complexity of the algorithm used for Gaussian elimination of the Dixon matrix which in turn will depend on the size of the Dixon matrix produced in the first step. If d is a bound on the degrees of each variable appearing in the input polynomials then the complexity of the algorithm in the general case will be $\mathcal{O}(d^n)$. However, the real question is how well the algorithm performs in practice, since every algorithm for this kind of problem would take at least exponential time in the worst case. In section 4.6 we examine the performance of the implementation of the algorithm for some problems.

4.5 Numerical Issues

In this section we address the most important issue involved in point location problems and robust predicates for geometrical problems, namely that of efficient and correct schemes for arithmetic.

Most geometric algorithms are described using exact arithmetic on real numbers, i.e., using the "real RAM" model. No such computer exists in real life, of course, and actual implementations of such algorithms will encounter and have to deal with problems involving the robustness and correctness of the arithmetic they use. A common approach is to use fixed precision floating-point arithmetic but there are no simple techniques that guarantee that the program will provide reliable answers. Usually the approach is to analyze the particular algorithm on hand for stability and correctness when using floating-point arithmetic [For92], [Mil88]. Such analyses do not lend themselves to generalization and tend to be difficult and algorithm-specific.

An alternate approach is to use exact integer arithmetic. The immediate problem with this approach is that one often requires precision beyond the standard 32 bits provided by most computers. The solution to this problem then is to use multiprecision arithmetic implemented in software. This can be expensive and quite often unnecessarily so - in many problems one is interested in determining the *sign* of a particular number and multiprecision arithmetic, though guaranteed to be correct, would be an overkill. The other problem with using integer arithmetic is that some problems require division of numbers. If the divisions are not guaranteed to be exact one would have to use either rational arithmetic (which would increase the complexity of computation by roughly 2 to 3 times) or homogeneous integer coordinates if the problem is amenable to them. Alternately one might use algebraic numbers (i.e. those represented as roots of polynomials with integer coefficients).

A third approach is to use interval arithmetic. There are two problems associated with intervals: the first is that intervals tend to grow rather rapidly and eventually contain the number 0 which renders them useless for determining the sign of a number, the computation in that case will have to be repeated using finer bounds on the intervals. The other problem with interval arithmetic is that often one is required to implement the ends of the intervals as multiprecision numbers which would entail the use of (and the problems associated with) software multiprecision arithmetic.

Yet another approach is to use multiprecision *floating point*. If the divisions are exact and the input data can be represented by a finite number of bits, then multiprecision floating point is guaranteed to provide correct answers. The problem, of course, is that we might be required to use a large number of bits to represent certain numbers.

As explained in the introduction, the approach we adopt falls conceptually in beween the use of fixed-precision floating point and arbitrary precision floating point. The idea is to do the minimum amount of work required to determine the sign of a number correctly. Our emphasis on the signs of numbers is natural in light of the algorithm presented for point location in section 4.4. In the remainder of this section we outline the *adaptive* approach to floating point computation and extend it to the mixed approach we use in the toolkit.

4.5.1 Adaptive Precision Floating-Point Arithmetic

The adaptive floating point approach was pioneered by Priest [Pri92], [Pri91] and extended by Shewchuk [She96]. There are two standard approaches to storing numbers in implementations of arbitrary precision floating point arithmetic: In the *multiple-digit format*, an arbitrary precision floating point number is stored as a sequence of digits, of some large radix like 2^{32}, with a single exponent. The other approach is the *multiple-term format* where a number is expressed as a sum of ordinary floating-point numbers, each with its own significand and exponent. An addition involving two numbers like $2^{300} + 2^{-300}$ can be stored in two ordinary floating point numbers. In the multiple-digit format the the significand of the result would require 601 bits. The advantage of the multiple-digit approach is that it can compactly represent most numbers.

The adaptive approach is based on the multiple-term format: each number x is written as an *expansion* $x = x_n + \cdots + x_1$ where each x_i is called a *component* of x and is represented by a floating-point value with p-bit significand (where p is the number of bits to the significand allocated by the floating-point format of the given machine). An expansion is required to be ordered by magnitude with x_n the largest component and x_1 the smallest. Expansions are also required to be *nonoverlapping*: Two floating-point numbers x and y are nonoverlapping if the least significant bit of x is more significant than the most significant bit of y or vice versa. An expansion is nonoverlapping if all of its components are mutually nonoverlapping. Given a nonoverlapping expansion, it is easy to compute its sign - simply take the sign of the largest component. Similarly one can get an approximation to the magnitude of the expansion by taking the magnitude of the largest component. The basic idea behind adaptive floating point then is to successively compute the components of the expansion representing a number starting with the largest component. Once we have a nonzero largest component, we can stop the computation since we have the sign and an approximation to the magnitude. The issue then becomes devising algorithms for the various arithmetic operations which will take as input two expansions and compute the components of the resulting expansion in this successive fashion. These algorithms are provided in [Pri92] where the restrictions on the format of the floating point numbers are few (different radices and rounding modes are allowed), which makes the algorithms more general but somewhat slower. Shewchuk improves on these algorithms by assuming the IEEE 754 standard with radix 2 and exact rounding. We use Shewchuk's algorithms in most instances since the IEEE 754 standard has become widely accepted and implemented on most machines.

4.5.2 Mixed Adaptive Floating Point

The Priest-Shewchuk adaptive floating point approach works well when the result of a computation is a nonzero number. For numbers which evaluate exactly to zero, however, the adaptive algorithms will have to compute the expansion to the fullest extent before asserting provably that the number was identically zero. If we are willing to relax this requirement, one can fruitfully use a *mixed* approach as follows: We assume that every input datum can be represented by an integer and that divisions, if any, are exact. This implies that every number in our computation will be integral (this is certainly the case in our resultant based algorithms). The value of the number modulo a prime is therefore well-defined and in our computations we will store every number as a mixed number, i.e., as a pair of numbers - the integer value modulo a prime and an adaptive floating point number. If the finite field value of a mixed number is zero we will treat the number as zero. Thus in the cases where the actual value of a number is zero, we will avoid the long adaptive floating point computation at the expense of correctness with probability one. On the other hand, the only time we will compute every component of an expansion is when the number is provably nonzero.

4.6 Implementation and Benchmarks

For the numerical computation in the benchmarks below we use the big integer package from LEDA (Library of Efficient Data types and Algorithms) package of Näher and Mehlhorn [NU]. As mentioned in the Introduction, we use arbitrary precision arithmetic for the current version of the implemented algorithm. As of writing these not all of Priest's or Shewchuk's algorithms have been implemented (in particular those for subtraction and division) and we are currently working on implementing these to obtain an adaptive version of the predicates.

4.6.1 Application Problems

We have tried our implemented algorithm on the following set of problems which provide a flavor of the kinds of problems we expect to be able to solve efficiently. The table below gives the times required for these problems obtained on a Sun Sparc 10 with a preliminary implementation.

Problem 4.6.1 *Locating intersections of 2 circles among several circles.*

$$f_1 = (x_1 - 4)^2 + (x_2 - 4)^{2^2} - 5 = 0$$

$$f_2 = (x_1 - 3)^2 + (x_2 - 4)^{2^2} - 4 = 0$$

$$g_1 = (x_1 + 1)^2 + (x_2 + 2)^{2^2} - 100 = 0$$

$$g_2 = (x_1 - 1)^2 + (x_2 - 2)^{2^2} - 8 = 0$$

$$g_3 = (x_1 + 1)^2 + (x_2 - 5)^{2^2} - 16 = 0$$

$$g_4 = (x_1 + 6)^2 + (x_2 - 8)^{2^2} - 2 = 0$$

$$g_5 = (x_1 - 1)^2 + (x_2 + 8)^{2^2} - 200 = 0$$

The sign sequences produced for the g_i's at the two real roots of f_1, f_2 are: $\{-,-,+,+,-\}$ and $\{-,+,+,+,0\}$.

Problem 4.6.2 *Intersection of 3 spheres in a sphere*

$$f_1 = {x_1}^2 + {x_2}^2 + {x_3}^2 - 1 = 0$$

$$f_2 = (x_1 - 1/2)^2 + {x_2}^2 + {x_3}^2 - 1 = 0$$

$$f_3 = {x_1}^2 + (x_2 - 1/2)^2 + {x_3}^2 - 1 = 0$$

$$g_1 = {x_1}^2 + {x_2}^2 + {x_3}^2 - 4 \leq 0$$

The sign sequences produced for g_1 are both $\{-\}$ as is expected for containment.

Problem 4.6.3 *Same as 4.6.2 with extra g_i's.*

$$g_1 = {x_1}^2 + {x_2}^2 + {x_3}^2 - 4 \leq 0$$

$$g_2 = {x_1}^2 + {x_2}^2 + (x_3 - 3/4)^2 - 2 \leq 0$$

$$g_3 = (x_1 - 1/3)^2 + (x_2 + 1/3)^2 + {x_3}^2 - 1 \leq 0$$

The sign sequences produced for the g_i's at the two real roots of f_1, f_2, f_3 are: $\{-,+,+\}$ and $\{-,-,+\}$.

Problem 4.6.4 *Point Location in a Grooved Toroid ([Hof89],pp.197)*

$$f_1 = {x_1}^3 + {x_2}^3 - 6x_1x_2 = 0$$

$$f_2 = {x_1}^2 + {x_2}^2 - x_3 - 4 = 0$$

$$f_3 = 2x_1 + 2x_2 + 2x_3 - 7 = 0$$

$$g_1 = (12 + {x_1}^2 + {x_2}^2 + {x_3}^2)^2 - 64({x_1}^2 + {x_2}^2) \leq 0$$

$$g_2 = x_2 \leq 0$$

$$g_3 = (8 + {x_1}^2 + {x_2}^2 + {x_3}^2)^2 - 36({x_1}^2 + {x_2}^2) \leq 0$$

$$g_4 = (24 + {x_1}^2 + {x_2}^2 + {x_3}^2)^2 - 100({x_1}^2 + {x_2}^2) \leq 0$$

$$g_5 = {x_2}^3 + {x_3}^3 - 6x_2x_3 = 0$$

The semi-algebraic set S is given by:

$$((g_1 \wedge g_2) \wedge (\neg g_3) \wedge (\neg g_4)) \wedge g_5$$

(The object defined by S is discussed in [Hof89] where the problem of edge identification is considered. It is obtained by intersecting a torus with 2 other tori, the halfplane g_2 and the Cartesian cylinder g_5.)

Problem	Time	Degree of resultant
4.6.1	0.20s	2
4.6.2	0.12s	2
4.6.3	0.40s	2
4.6.4	38.5s	6

Chapter 5

Geometric Theorem Proving

This chapter deals with the application of the toolkit to the problem of geometric theorem proving. We provide an algorithm, of independent interest, for solving *triangular systems* of polynomial equations, i.e., systems in which each successive polynomial introduces a new variable. In addition to theorem proving, such systems arise in problems involving *constraint-based solving.* In such problems, one is given various geometrical objects such as circles, line segments, spheres etc. and pairwise constraints between them (e.g., " the circle should be tangent to the line segment at its midpoint"). The question then is to decide whether the constraints are feasible (i.e., whether they can be satisfied simultaneously) and if so, to determine the actual physical configuration(s) they realize.

We have had a good measure of success with our algorithm due to its ability to solve non-trivial problems of large dimension and degree, which were hitherto unsolvable by all of the standard approaches. Portions of our work described in this chapter appeared in [Reg95].

The overview of the chapter is as follows: We start with an example which illustrates the problem and motivates the rest of the material in the chapter. In section 5.2 we will provide the historical background and references to previous work. Section 5.4 puts the problem on a theoretical footing, providing the notation, terminology and the formulation of the problem. In section 5.5 we look at the celebrated algorithm for the problem credited to the work of Ritt and Wu. The Ritt-Wu algorithm has been the basis for the most successful theorem provers to date and we shall use that as a basis of comparison for our algorithm and its implementation. The remaining sections provide an overview of our algorithm (5.6), details of the algorithm for triangular systems and its complexity analysis (5.7), discussion

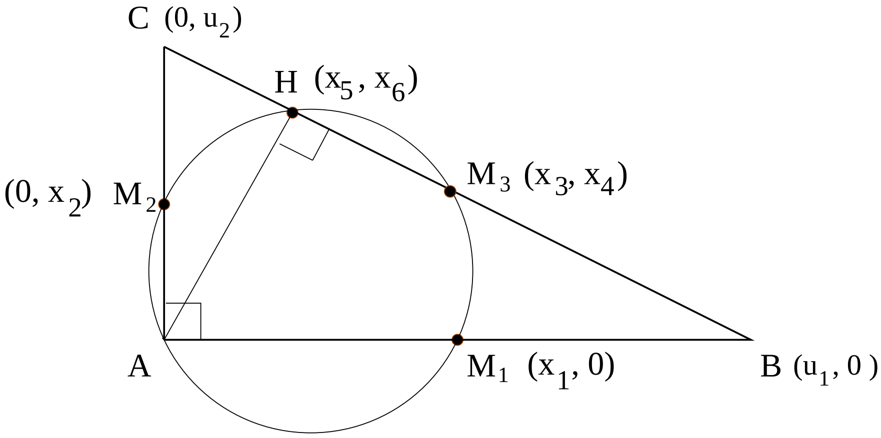

Figure 5.1: Circle Theorem of Apollonius

of the thorny issues of genericity and theorem proving over the reals (5.9) and benchmark times on some of the standard problems, including some previously unsolvable problems.

5.1 Example

The basic idea is to formulate the hypotheses and the conclusions of the theorem we are interested in proving as polynomial equations. One can then reason about the solutions to these equations and their relation to the validity of the conclusion. We consider the following standard example ([CLO92]):

Theorem 5.1.1 *(Circle Theorem of Apollonius). Let ABC be a right triangle in the plane, with right angle at A. The midpoints of the three sides and the foot of the altitude drawn from A to BC all lie on one circle. (See Figure 5.1.)*

Without loss of generality we can assume that the triangle is placed as shown in Figure 5.1, with $A = (0, 0)$, $B = (u_1, 0)$ and $C = (0, u_2)$, where u_1 and u_2 are are input parameters which can take on different values. Each combination of the parameter values will specify a particular configuration. We are, of course, interested in proving the theorem for all (valid) values of u_1 and u_2. Note that by selecting $C = (0, u_2)$, we have already incorporated one of the hypotheses of the theorem, namely that ABC is a right triangle. We further assign variables to various points on the diagram as shown in the figure. We can now convert the rest of the hypothesis to polynomial equations as follows:

M_1 midpoint of AB	$2x_1 - u_1 = 0$	(h1)
M_2 midpoint of AC	$2x_2 - u_2 = 0$	(h2)
M_3 midpoint of BC	$2x_3 - u_1 = 0$	(h3)
	$2x_4 - u_2 = 0$	(h4)
$AH \perp BC$	$u_1x_5 - u_2x_6 = 0$	(h5)
B, H, C collinear	$u_2x_5 + u_1x_6 - u_1u_2 = 0$	(h6)
$M_1O = M_2O$	$(x_1 - x_7)^2 + x_8^2 - x_7^2 - (x_8 - x_2)^2 = 0$	(h7)
$M_1O = M_3O$	$(x_1 - x_7)^2 + (0 - x_8)^2 - (x_3 - x_7)^2 - (x_4 - x_3)^2 = 0$	(h8)

Finally we convert the conclusion to a polynomial equation

$$HO = M_1O \quad (x_5 - x_7)^2 + (x_6 - x_8)^2 - (x_1 - x_7)^2 - x_8^2 = 0 \tag{5.1}$$

Several issues now arise:

The first question is what constitutes proving a theorem; in order to prove the theorem, we need to show that whenever the the hypotheses are true, the conclusion is also true. That is, whenever the system of equations (h1) - (h8) has a solution, so does 5.1. Thus we need the solution set of (h1) - (h8) to be contained in that for 5.1. However, it turns out we have more variables (ten) than equations (eight) and in general the solution set of (h1) - (h8) will not be a 0-dimensional variety. Thus any automated approach for proving the above theorem would have to deal with this issue. Further, even in this case, the truth or invalidity of the theorem would be proven only over an algebraically closed extension field. In practice, one is interested in showing that the set of *real* roots of (h1) - (h8) is contained in the set of real roots of 5.1. This, in general, poses a more complicated problem than the previous version.

Second, it would be unreasonable to expect the conclusion to be true when the input parameters u_1 and u_2 take on extreme values such as 0. Clearly the theorem as stated was supposed to be true for non-degenerate right-triangles and an automated procedure would have to characterize what precisely is meant by "non-degenerate". As we shall see in section 5.4, we can relax the requirements of what it means to "prove" a theorem in this setting to take into account such degenerate cases.

5.2 Background and Related Work

Earlier methods for geometric theorem proving were based on a synthetic geometric reasoning approach using natural deduction, forward and backward chaining and the like [Gel63]. Wu's seminal work ([Wu78], [Wu84]) recast the problem in terms of algebraic geometry and thereafter a significant amount of the recent work has focussed on an algebraic approach involving determining the feasibility of systems of polynomial equations [Cho84], [KS86], [Kap88]. The papers by Wu and others showed how a subclass of geometric theorems can be proved by expressing the hypotheses and conclusions as polynomial equations as described in section 5.1. A geometric theorem can then be proved by essentially determining the algebraic set corresponding to the hypotheses and checking whether the algebraic sets of the conclusions contain it. Different methods have been proposed for doing this - Wu's method involves reduction of the hypotheses system of polynomials to a triangular form (i.e. one in which each successive polynomial has at most one extra variable) and then using successive pseudodivision to determine the feasibility of the conclusion. Other work has focussed on using Gröbner bases to determine whether a system of polynomials (obtained from the original polynomials) has a common zero [KS86], [Kap88]. As noted in [Cho88] (pp. 88-89) and evinced by the some of the benchmarks [KS86], [Kap88], Gröbner basis methods for theorem proving have proved to be rather slow, in most cases requiring times 50-100 times more than the characteristic set methods [1].

A somewhat different approach is taken in [Wan93] and [Wan95]. Though the method is similar to the characteristic set approach, the requirements for the decomposition of a given hypotheses system into its irreducible components are relaxed. Both the Ritt-Wu approach and this one, however, rely on factorization over extension fields as we shall see in section 5.5.

Another technique based on algebraic formulation is the *distance geometry* method for proving theorems [Hav91]. Here the hypotheses constraints are formulated as polynomials in variables which represent *distances* rather than cartesian coordinates. The trade-off is that equations involving distances are reduced in degree at the expense of adding more variables than in the case of cartesian coordinates. As far as we know, no prover has been implemented based on this idea so it is not easy to judge the practical implications of this

[1] However, see the remarks made in sections 5.5.3 and 5.10 regarding the nature of most geometric theorems that are typically used as benchmarks.

approach.

A more recent approach is that based on the "area" method [CGZ94]. It is a combination of the synthetic and algebraic approaches. The algebraic component reflects the natural progression in geometric invariants from cartesian coordinates to distance geometry to the use of *areas* and *volumes* as the basic quantities. The advantage of this approach is that a proof provided by an automated prover will tend to be shorter and have greater geometric meaning in each step thereby allowing the proof to be comprehensible by humans.

It should be pointed out that in our approach, we ignore the issues of readability of the proofs and explanations based on them. Our goal is not to provide an interactive prover such as one that might be used in a classroom. Our interest in the problem comes primarily from the point of view of exploring algorithms for solving systems of polynomial equations efficiently. To that end, we're primarily interested in looking at non-trivial problems that arise in geometric theorem proving such as Morley's trisector theorem and the Thébault-Taylor theorem discussed in section 5.10.

5.3 Some Issues in Theorem Proving

As we shall see, Wu's original method is incomplete in the sense that the feasibility of the conclusions is determined over an algebraically closed field whereas geometric theorems are often assertions about real numbers. It is not possible to refute a conjecture over the reals using this approach and in general, confirmation of a theorem can not be obtained automatically from a proof over an algebraically closed field.

As we noted in section 5.1, another important consideration in geometric theorem proving is the notion of *genericity*. It is necessary to rule out *degenerate* cases such as non-distinct points or zero-radius circles. Wu's method and others generate a set of subsidiary polynomials representing such cases so that the conclusion polynomial vanishes at the common zeros of the hypotheses which are not zeros of the subsidiary polynomials.

From the above discussion, it seems important that a geometric theorem prover should satisfy the following criteria

1. It should work over the reals as well as the complex field.

2. It should handle degenerate cases.

3. It should be practicable.

5.4 Geometric Theorem Proving - Formulation of the Problem

This section provides the formulation we shall use of the questions of what constitutes a geometric theorem and what it means to prove it. Several formulations of these two questions have been proposed [Gel63], [Kap86],[KS86]. See the book by Chou [Cho88] for a discussion of the various formulations and the limitations of each. We will use the formulation used by Chou [Cho88] which seems to be the one that is the most satisfactory:

Definition 5.4.1 *By a* **geometric statement** *(S) we shall mean a collection of n polynomial equations*

$$h_1(y_1, \ldots, y_m) = \ldots = h_n(y_1, \ldots, y_m) = 0 \tag{5.2}$$

called the **hypotheses** *and a* **conclusion** $g(y_1, \ldots, y_m) = 0$ *where the* h_i *and* g *are in* $K[y_1, \ldots, y_m]$. *Here* K *is some field under consideration, usually* $\mathbb{Q}$.

Let $V = \mathcal{V}(h_1, \ldots, h_n)$ denote the set of common zeroes in E^m, where E is an extension of K Thus an instance of the theorem, i.e. a configuration which satisfies the given hypotheses, corresponds to an element of V and vice versa. Note, however, that V contains the so-called degenerate cases which happen to satisfy the hypotheses. In order to distinguish the degenerate cases from the nondegenerate ones, we use the variables $u_1, \ldots, u_d \equiv \mathbf{u}$, called the **parameters**, among the y_i to denote the nonzero coordinates of those points which can be chosen arbitrarily and we use the variables $x_1, \ldots, x_n \equiv \mathbf{x}$, to denote the rest of the y_i. These latter are called **dependent variables**.

If the extension field E is *algebraically closed* a basic fact of algebraic geometry is that every algebraic set has a unique decomposition as a union of irreducible algebraic sets such that no one contains another. Thus the set V can be expressed as a union,

$$V = \mathcal{V}(P_1) \cup \ldots \cup \mathcal{V}(P_s) \cup \mathcal{V}({P_1}^*) \cup \ldots \cup \mathcal{V}({P_t}^*) \tag{5.3}$$

where the P_i and ${P_k}^*$ are prime ideals and where the $V_i = \mathcal{V}(P_i)$ are all those (**nondegenerate**) components on which the u_j are algebraically independent and the ${V_k}^* = \mathcal{V}({P_k}^*)$ are those (**degenerate**) components on which they are dependent. The sets V_i are called components which are **general for u**.

If E is *not algebraically closed*, the decomposition is, in general, not an irreducible one as above. However, the u_j are algebraically dependent on the algebraic sets ${V_k}^*$ and the latter still correspond to the degenerate cases. We will still call the V_i components **general for u**. (Note, however, that in this case any V_i can be reducible or empty.)

Definition 5.4.2 *A geometric statement (S) is* **generally true** *if g vanishes on all nondegenerate components of V. Else it is* **not generally true**. *A geometric statement is* **generally false** *if g vanishes on none of the nondegenerate components of V.*

5.5 The Ritt-Wu Algorithm

In this section, we provide a brief description of the algorithm based on the work of Ritt and Wu. We will also consider a prover implemented by Chou and derived essentially from this algorithm [Cho88]. Among all the approaches that have been used and implemented, Chou's prover seems to be most complete and efficient and we will use this prover as the basis of comparison for our prover. This section will therefore examine the Ritt-Wu algorithm and the techniques used by Chou's prover so we can better understand its strengths and limitations.

In 1978, Wu introduced an elegant method for automated theorem proving for algebraic geometry [Wu78]. The basis of his work was a triangulation procedure which was implicit in the work of Ritt [Rit32], [Rit50] and which was given in detail by Wu. Wu's approach was implemented with various modifications and improvements by Chou in his prover [Cho84], [Cho88]. The following is based on the book by Chou [Cho88] with some notational changes. Readers familiar with this work can skip this section.

5.5.1 Pseudodivision and Successive Pseudodivision

Consider two polynomials f and $g \in A[z]$, where A is a commutative ring (e.g., $K[y_1, \ldots, y_m]$) and where $f = a_s z^s + \cdots + a_0$ and $g = b_t z^t + \cdots + b_0$. Then by the process of pseudodivision (see e.g., [Knu81]) we have,

$$b_t{}^u f = qg + r, \quad u \leq s - t + 1, \quad deg(r, z) < deg(g, z) \tag{5.4}$$

The remainder r is called the **pseudoremainder** of f on division by g with respect to z and denoted $prem(f, g, z)$.

Definition 5.5.1 *A* **triangular form** *is a collection of polynomials f_i of the form*

$$\begin{aligned} &f_1(u_1, \ldots, u_d, x_1) \\ &f_2(u_1, \ldots, u_d, x_1, x_2) \\ &\ldots \\ &f_r(u_1, \ldots, u_d, x_1, x_2, \ldots, x_r) \end{aligned} \tag{5.5}$$

A triangular form can be obtained from the original hypotheses polynomials very easily by applying the following naive triangulation algorithm (cf. [Cho88], [CLO92]).

Algorithm 5.5.1 (Naive Triangulation)
Input: *Set of polynomials $H = \{h_1, \ldots, h_n\}$, $h_i \in K[\mathbf{x}, \mathbf{u}]$*
Output: *Ordered set of polynomials $T = \{f_1, \ldots, f_n\}$ satisfying (5.5).*
Procedure: *NaiveTriangulate(H)*

1. *If $|H| = 0$ return. Else if $|H| = 1$ return H.*
2. *Let i be the highest number such that there exists a polynomial in H with non-zero degree in x_i Compute S, the set of all polynomials in H which have non-zero degree in x_i.*
3. *If $|S| = 1$, compute T = NaiveTriangulate($H - S$). Append S to T and return T*
4. *Else let $H' = H - S$ and reduce S by the repeating the following steps till only one polynomial in S has non-zero degree in x_i.*
   (a) *Pick $f \in S$ with the smallest degree in x_i.*
   (b) *For all $g \in S - \{f\}$, replace g by $\mathrm{prem}(g, f, x_i)$*
5. *S now contains only one polynomial f with non-zero degree in x_i. Compute T = NaiveTriangulate($H' \cup S - \{f\}$). Append f to T and return T.* □

The importance of this algorithm lies in the fact that it turns out that a large number of geometric theorems can be proved by using the above algorithm. Let g be a polynomial, then we can apply *successive pseudodivision* to g with respect to a triangular form (as above) as follows. Letting $g = R_n$, the remainder R_0 is computed by taking successive pseudoremainders: at the i^{th} step ($i = n \ldots 0$) the remainder computed is

$prem(R_{i+1}, f_{i+1}, x_{i+1}) = R_i$. It can then be easily shown that there exist polynomials Q_j such that

$$I_1^{s_1} \cdots I_r^{s_r} g = Q_1 f_1 + \cdots + Q_r f_r + R_0 \tag{5.6}$$

Here the I_j's are the leading coefficients of the f_j's. We denote by $prem(g, f_1, \ldots, f_r)$ the remainder R_0 obtained by the above process.

It can be easily verified that if $R_0 = 0$ and if $\mathbf{z}$ is a common root of the f_j's then:

$$I_1(\mathbf{z}) \neq 0, \ldots, I_r(\mathbf{z}) \neq 0 \Rightarrow g(\mathbf{z}) = 0 \tag{5.7}$$

This simple observation is the basis of most geometric theorem proving algorithms based on algebraic methods. If we can arrange to have our hypotheses polynomials to be in triangular form, then we can at least prove (though not refute) the conclusion (subject to the additional constraints imposed in equation 5.7) if the pseudoremainder obtained by successive pseudodivision is identically zero.

The situations corresponding $I_k = 0$, $(1 \leq k \leq r)$, usually turn out to have a geometric meaning - they often correspond to the "degenerate" cases and it is entirely reasonable to exclude them from consideration.

Thus the goal is to convert our hypotheses into triangular form and this question (and related issues) is addressed in the next subsection.

5.5.2 Characteristic Sets

The fundamental object in the Ritt-Wu algorithm is the *characteristic set* of a set of polynomials $f_1, \ldots, f_n \in K[x_1, \ldots, x_m]$. Essentially, a characteristic set of a set of polynomials is a triangular form describing the common roots of the given set of polynomials. To define a characteristic set we first need the following definitions and terminology:

Let us fix an ordering, without loss of generality, on the variables x_i as: $x_1 \prec \cdots \prec x_m$.

Definition 5.5.2 *The* **class** *of a polynomial $f \in K[y_1, \ldots, y_m]$ is the smallest integer c such that $f \in K[y_1, \ldots, y_c]$ and we write $Class(f) = c$. If no such c exists we write $Class(f) = 0$. The variable y_c is called the* **leading variable** *of f, denoted $lv(f)$. We define the* **leading coefficient** *of f as the leading coefficient treating f as a polynomial in y_c and denote it $lc(f)$. Similarly the degree of f in x_c is called the* **leading degree** *of f and denoted $ld(f)$ (or $ldeg(f)$).*

Definition 5.5.3 *A polynomial g is said to be* **reduced with respect to** *a polynomial f if $degree(g, y_c) < degree(f, y_c)$ where $c = Class(f) > 0$.*

Definition 5.5.4 *The* ordered *set of polynomials $f_1, \ldots, f_n$ is called a* **chain** *and denoted by $< f_1, \ldots, f_n >$.*

Definition 5.5.5 *A chain $C =< f_1, \ldots, f_r >$ is called an* **ascending chain** *if*

1. *Either $r = 1$ and $f_1 \neq 0$.*

2. *Or $r > 1$ and $0 < Class(f_1) < \cdots < Class(f_r)$. And f_j is reduced w.r.t. f_i for $i < j$.*

(Note that every ascending chain is a triangular form, with a suitable renaming of the variables if necessary. We will henceforth assume that every ascending chain is written as such.)

We can now impose a partial order on the set of all ascending chains as follows: If f and g are polynomials, then $f \prec g$ if

1. Either $Class(f) < Class(g)$

2. Or $Class(f) = Class(g)$ and $ldeg(f) < ldeg(g)$.

If neither $f \prec g$ nor $g \prec f$ we say that f and g are of the same *rank*.

Definition 5.5.6 *Let $C_1 = < f_1, \ldots, f_r >$ and $C_2 = < g_1, \ldots, g_s >$ be two ascending chains. We define $C_1 \prec C_2$ if there exists t such that $t \leq min(r, s)$ and*

1. *Either f_i and g_i are of the same rank for $i < t$ and $f_t < g_t$*

2. *Or $s < r$ and f_i and g_i are of the same rank for $i < s$.*

Though the above definition is hard to parse, it is in fact the obvious partial order that one would get on extending the order from polynomials to ascending chains. That is, if we have two ascending chains of equal length we compare the polynomials index-by-index till we get an f_i which satisfies $f_i \prec g_i$. If we have two chains of unequal length with the second one being shorter, and the polynomials having the same rank, then we declare the longer one to be the *lesser* in the partial order. We note that the above ordering is a well-ordering on the set of all ascending chaing.

Definition 5.5.7 *Let S be a nonempty set of polynomials. A minimal ascending chain (w.r.t. $\prec$) in the set of all chains formed from polynomials in S is called a* **characteristic set** *of S. (Note that since $\prec$ is a well-ordering, every nonempty set S will have a characteristic set.)*

Characteristic sets of an ideal characterize the ideal in a manner similar to (but weaker than) the (Gröbner) bases of the ideal as demonstrated in the following proposition.

Proposition 5.5.8 *Let $C = < f_1, \ldots, f_r >$ be a characteristic set of an ideal $I \subset K[y_1, \ldots, y_m]$ and $g \in K[y_1, \ldots, y_m]$.*

1. *C is a triangular form.*
2. *If $g \in I$ then $\mathrm{prem}(g, f_1, \ldots, f_r) = 0$.*
3. *If I is prime and $\mathrm{prem}(g, f_1, \ldots, f_r) = 0$ then $g \in I$.*

In practice, one computes what is called an *extended characteristic set* (see [Cho88]). The basic Ritt-Wu algorithm is then obvious:

Algorithm 5.5.2 (Basic Ritt-Wu Algorithm)

Input: *Hypotheses $H = \{h_1, \ldots, h_n\}$ and conclusion g*
Output: *"True", if the conclusion is generally true.*

1. *Compute a characteristic set $C =< f_1, \ldots, f_r >$ of (H).*
2. *Compute $R = \mathrm{prem}(g, f_1, \ldots, f_r)$ by successive pseudodivision.*
3. *If $R = 0$ return "True" else return "Unknown".* □

We note of course that the basic algorithm above cannot refute a conclusion. In order to be able to do that, the characteristic set obtained needs to have a stronger property namely *irreducibility.*

Definition 5.5.9 *Let $C =< f_1, \ldots, f_r >$ be an ascending chain in the form*

$$\begin{array}{l} f_1(u_1, \ldots, u_d, x_1) \\ f_2(u_1, \ldots, u_d, x_1, x_2) \\ \ldots \\ f_r(u_1, \ldots, u_d, x_1, x_2, \ldots, x_r) \end{array}$$

C is called **irreducible** *if each f_i is irreducible in the polynomial ring $K(u_1, \ldots, u_d)[x_1, \ldots, x_i]/(f_1, \ldots, f_{i-1})$.*

The importance of irreducible characteristic sets lies in the fact that $prem(g, f_1, \ldots, f_r) = 0$ now also becomes a sufficient condition for the conclusion g to be generally true over an algebraically closed field. Thus, the basic Ritt-Wu algorithm can now be modified as follows

Algorithm 5.5.3 (Ritt-Wu Algorithm)

Input: *Hypotheses $H = \{h_1, \ldots, h_n\}$ and conclusion g.*
Output: *"True", if the conclusion is generally true. "False" if the conclusion if false over an algebraically closed extension.*

1. *Compute an (extended) characteristic set $C =< f_1, \ldots, f_r >$ of (H).*
2. *If C is reducible, decompose C into two strictly lower characteristic sets and repeat.*
3. *If C is irreducible, record C.*
   At the end of the above procedure we get a collection $C_1, \ldots, C_k$ of irreducible characteristic sets. We compute the psuedoremainder of R w.r.t. each of these as before.
   □

The algorithm will terminate (from the well-ordering principle) and will return a collection of irreducible characteristic sets, each of which will correspond to a (full-dimensional) irreducible component of the original variety of the hypotheses. In effect, the algorithm "solves" the system of equations given by the hypotheses. In the next subsection we will consider the implications of this algorithm to (the practical problem of) theorem proving.

5.5.3 Issues Raised by the Ritt-Wu Algorithm

We will now consider the various issues that are raised by the Ritt-Wu algorithm and their implications for the problem of actually implementing a theorem prover. As stated earlier, we will use the prover due to Chou [Cho88] as the reference point since it is based on the Ritt-Wu algorithm and has proven to be the most successful implementation.

We first consider the case of theorem proving over an algebraically closed field. (The case of the reals will be considered in section 5.9.) As seen above, we have to obtain a sufficient and a necessary condition for us to declare a conclusion valid or invalid respectively. A sufficiency condition is not hard to obtain. If we have a triangular form $< f_1, \ldots, f_r >$ describing the variety $V = \mathcal{V}(h_1, \ldots, h_n)$, then as shown earlier, $prem(g, f_1, \ldots, f_r) = 0$ is sufficient to state that the conclusion g is true given the hypotheses. In general, however, the hypotheses will not be given in triangular form and we will have to convert them into such. A triangular system can be obtained easily from the original system of hypotheses polynomials by using pseudoremainders to eliminate variables as in Algorithm 5.5.1.

The main problem with the naive triangulation procedure is that the variety of the triangular system obtained at the conclusion of the procedure could be strictly larger than the one we started out with due to the process of taking pseudoremainders. If it turns out that the conclusion polynomial g vanishes on this new larger variety, then the validity of the theorem holds on the smaller variety *a fortiori*. It could be the case, however, that g does not vanish on the larger variety. As it turns out, for a good majority of the theorems that are used as benchmarks for provers, the naive triangulation process is sufficient - the systems are nearly triangular to begin with and extra zeros are rarely added. Since the theorem prover is essentially *checking* a theorem known to be true, $prem(g, f_1, \ldots, f_r)$ equals 0 and the task is finished. Clearly, the performance of a theorem proving algorithm on such a theorem would not prove to be a fair measure of the efficiency of the algorithm. More interesting are the cases where the naive triangulation adds zeros or the conjectured conclusion is not true, that is the case when $prem(g, f_1, \ldots, f_r) \neq 0$ where the f_i's are obtained by naive triangulation.

This problem is resolved in the Ritt-Wu algorithm by testing if the triangular system obtained above is irreducible that is, whether each f_i is irreducible in the polynomial ring $K(u_1, \ldots, u_n)[x_1, \ldots, x_i]/(f_1, \ldots, f_{i-1})$. We have the following result (see e.g., [Cho88]):

We first define

$$P = \{ \ g \mid g \in K[u, x] \text{ and } prem(g, f_1, \ldots, f_n) = 0\} \tag{5.8}$$

Theorem 5.5.10 (Necessary Condition) *If $F = < f_1, \ldots, f_r >$ is an irreducible system,* $\mathrm{prem}(g, f_1, \ldots, f_r) = 0$ *is a necessary condition for (S) to be generally true if one of the following is satisfied*

1. *$\mathcal{V}(P)$ is of degree d.*

2. *E is an algebraically closed field.*

3. *The system $f_1, \ldots, f_n$ has a generic point in E.*

Testing for irreducibility. The first question that arises is testing for irreducibility. If it turns out that $deg(f_i, x_i) = 1$ for all i, then the triangular system is trivially irreducible and real generic points trivially exist. In this case, $prem(g, f_1, \ldots, f_n) = 0$ gives a necessary and sufficient condition to determine whether the theorem is generally true or not. Several of the simpler theorems in the collection solved by Chou's prover fall in this category ([Cho88], p. 56). If the degree of a polynomial is not 1 in its primary variable, Chou's prover uses factoring over extension fields to check for irreducibility. This approach, however, has been implemented only for *quadratic* extension fields. If the degree of some f_i in x_i is greater than 2, Chou's prover can not handle this case (unless $i = 1$ in which case only factoring over $\mathbb{Q}$ is required). Further, even in the case of quadratic polynomials, the prover works in general only over *algebraically closed fields*. For the prover to work over the *reals*, the existence of a real generic point needs to be shown as required by the sufficiency condition given above. As far as we know, a mechanical approach to this has not been implemented in Chou's prover ([Cho88], p. 56).

Reducible systems. Next, we come to the question of what happens when the triangular system is reducible. If this happens, in general Chou's prover uses Ritt's decomposition algorithm to obtain all irreducible components of the original variety. This also involves factoring over field extensions and is computationally expensive for high degrees. Again, this has been implemented only for quadratic cases.

Successive pseudodivision Finally, we address the question of successive pseudodivision. As shown in the earlier section, we have the following remainder formula

$$I_1^{s_1} \cdots I_n^{s_n} g = Q_1 f_1 + \cdots + Q_n f_n + R_0$$

If the remainder $R_0 \neq 0$ then we can conclude that the theorem is not generally true since the right side would be non-zero at any instance of the theorem implying $g \neq 0$ at that instance. If, however, the remainder is zero, g could still be non-zero unless $I_j \neq 0$ for all j. These then become the additional subsidiary hypotheses under which the theorem is true. It could be true however that $prem(I_j, f_1, \ldots, f_{j-1}) = 0$ in which the hypotheses would become inconsistent. To avoid this situation, in general, one has to use the complete method in Chou's prover which involves triangulation using the Ritt decomposition algorithm.

To summarize, Chou's prover, though efficient and applicable in the case of a large class of geometric theorems, has the following limitations:

- Requires factoring over extension fields.
- Applicable in general to systems of degree quadratic or less.
- Does not work in all cases over $\mathbb{Q}$.
- Uses significant amount of computational resources for some problems of degree 2 ([Cho88], p. 69).

5.6 Overview of the Theorem Prover

We will give a schematic overview of our theorem prover in this section. The notation used in this and subsequent sections is the same as introduced earlier. The prover takes as input a set of hypotheses polynomials $h_1, \ldots, h_n$ and a conjecture or conclusion polynomial g in $\mathbb{Q}[u_1, \ldots, u_m, x_1, \ldots, x_n]$. Here, as before, the x_i's represent the dependent variables and the u_j's, the independent variables. In what follows, we will abuse notation and write the hypotheses as polynomials over the x_i's only. It returns "true" or "false" according to whether the theorem is generally true or not. The modules of the prover can be summarized as follows:

Algorithm 5.6.1 (Theorem Proving)

1. **Triangulation** The hypotheses are first triangulated using naive triangulation (Algorithm 5.5.1) to obtain the polynomials

$$f_1(x_1), f_2(x_1, x_2), \ldots, f_n(x_1, \ldots, x_n)$$

2. **Cheap Test** Compute $R = prem(g, f_1, \ldots, f_n)$. If $R \equiv 0$, return True if the theorem was to be proved over $\mathbb{C}$. Else continue with the remaining steps.

3. **Random instantiation of parameters** The parameters u_i are set to random values to get the f_i as polynomials in $x_1, \ldots, x_n$ only. (See the implications of this step for the reals in Section 5.9).

4. **Root representation for triangular systems** This module takes as input the triangular system

$$f_1(x_1) = f_2(x_1, x_2) = \ldots = f_n(x_1, \ldots, x_n) = 0$$

and returns a *symbolic* representation of the roots of the system: the output is a univariate polynomial $p(s)$ and rational functions $r_1(s), \ldots, r_n(s)$. If the roots of $p(s) = 0$ are $\alpha^{(i)} \in \mathbb{C}$, for $i = 1, \ldots, N$, then

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

One can think of $r(s)$ as 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 passes through a solution of $f_1 = f2 = \ldots = f_n$ are precisely the roots of $p(s) = 0$. The fact that we have a symbolic representation of the roots allows us to determine correctly the feasibility of the conjecture polynomial given the hypotheses polynomials.

5. **Substitution** We put $x_i = r_i(s)$, $1 \leq i \leq n$, in the conjecture polynomial g. This reduces g to a univariate polynomial $g(s)$.

6. **Factoring** We check if the primitive element polynomial $p(s)$ can be factored over $\mathbb{Q}$. If so, we have $p(s) = p_1(s) \cdots p_t(s)$ and we perform the remainder of the algorithms with each $p_j(s)$ individually.

7. **Sign Determination** This module determines the sign, i.e. +, - or 0, of the instantiated conjecture $g(s)$ at the real roots of $p(s) = 0$ using Sturm sequences.

8. **Theorem proving or refutation:** The conjecture should vanish at all of the common real roots of the hypotheses polynomials. Thus the sign determination module should return 0 for the sign of $g(s)$ at every root. If not, there exists a root where the conjecture does not hold and is therefore false OR the triangulation process introduced extra roots. Section 5.8 discusses what to do in that case. $\square$

5.7 An Algorithm for Solving Triangular Systems

This section describes the basic algorithm for solving triangular systems used in our geometric theorem prover. Section 5.7.1 sketches the algorithm from a theoretical viewpoint, section 5.7.2 goes into more details of the algorithm especially the use of straight-line programs. We conclude with section 5.7.3 which gives a complexity analysis of the algorithm.

It should be noted that our algorithm for triangular systems is *probabilistic* in two different ways. First, the random instantiation of parameters in the third step has the effect of making our algorithm probabilistic in the sense that the theorem will be proven to be true or false, over $\mathbb{C}$, with probability 1. That is, we will be able to determine whether the theorem is generically true or false. It is possible to eliminate this step by not performing the random instantiation. Instead one treats the input triangular system as a set of polynomials in the x_i with coefficients which are straight-line programs in the u_j. The computations are then carried over $\mathbb{Q}(\mathbf{u})$ rather than over $\mathbb{Q}$. The algorithm then will have the same complexity (due to the use of SLP's) till we get to the factorization step. At this point we will have to interpolate the polynomial $p \in \mathbb{Q}[s](\mathbf{u})$ and proceed with the rest of the algorithm as before.

The second cause for the probabilistic nature of our algorithm is the fact that we use straight-line programs for representing the rational functions which make up the coefficients of the polynomials generated in the algorithm. Most algorithms for polynomials require us to verify that the leading coefficient is not identically zero and this is done by giving random values to the variables in the corresponding SLP and evaluating the SLP at those values. It could turn out, of course, that we treat as identically zero an SLP coefficient which was not so though the probability of this happening is low.

5.7.1 Theoretical Algorithm

Our algorithm is based on an approach for computing (p, r) for more general systems due to Renegar [Ren89b]. We use his basic method but take advantage of certain properties of triangular systems.

Algorithm 5.7.1 (Triangular Systems)

Input: *The input to the algorithm is a system of triangular polynomial equations in* $\mathbb{Q}(x_1, \ldots, x_n)$,

$$f_1(x_1) = f_2(x_1, x_2) = \cdots = f_n(x_1, \ldots, x_n) = 0$$

Output: *Let* $\xi^{(i)} \in \mathbb{C}^n$ *denote the solutions to the given system. The algorithm returns a* symbolic *representation of the roots of the system i.e. the output is a univariate polynomial* $p(s)$ *and rational functions* $r_1(s), \ldots, r_n(s)$. *If the roots of* $p(s) = 0$ *are* $\alpha^{(i)} \in \mathbb{C}$, *for* $i = 1, \ldots, N$, *then* $\xi^{(i)} = r(\alpha^{(i)})$ *for* $i = 1, \ldots, N$.

Step 1: *For the first step, the algorithm sets* $x_1 = (s_1 - l_0)/l_1$ *in* f_1 *to get* $R = p_1(s_1)$, *i.e.* $p_0 \equiv 0$.*)*

i^{th} **step:** *At the* i^{th} *step add the polynomial* f_i *and eliminate the extra variable* x_i *introduced by the polynomial to get a rational univariate representation* $r_1(s_i), \ldots, r_n(s_i)$ *of the roots of the first i polynomials. More precisely, at the start of the* i^{th} *step, the algorithm has computed*

$$r_1(s_{i-1}), \ldots, r_{i-1}(s_{i-1}) \text{ and } p_{i-1}(s_{i-1}) \tag{5.9}$$

1. *Introduce* $f_i(x_1, \ldots, x_i)$. *Set* $x_1 = r_1, \ldots, x_{i-1} = r_{i-1}$ *in* f_i.
2. *For the new variable,* x_i, *set*
$$x_i = \frac{s_i - l(\mathbf{x})}{l_i} = \frac{s_i - l_0 - l_1 x_1 - \cdots - l_{i-1} x_{i-1}}{l_i} \tag{5.10}$$
where $x_1, \ldots, x_{i-1}$ *are instantiated to* $r_1, \ldots, r_{i-1}$ *respectively.*
3. *We obtain from* f_i *a rational function* g_i *in* s_{i-1} *and* s_i. *We denote by* g_{in} *the numerator polynomial of* g_i *and by* g_{id} *the denominator.*
4. *Compute the (univariate) Sylvester resultants* $R(g_{in}, p_{i-1})$ *and* $R(g_{id}, p_{i-1})$ *with respect to the variable* s_{i-1}.
5. *Set*
$$R = \frac{R(g_{in}, p_{i-1})}{R(g_{id}, p_{i-1})} \tag{5.11}$$
6. *Set* $p_i(s_i) = R$.
7. *Compute* r_j *as*
$$r_j(s_i) = \left. \frac{\left(\frac{dR}{dl_j}\right)}{\left(\frac{dR}{dl_0}\right)} \right|_{l=L} \quad j = 1, \ldots, i \tag{5.12}$$
where L *is a random linear polynomial.* □

In order to prove the algorithm works correctly we will first show that equation 5.11 is an exact division:

Lemma 5.7.1 *$R(g_{id}, p_{i-1})$ divides $R(g_{in}, p_{i-1})$ exactly.*

Proof The proof is based on the following well-known property of resultants - the resultant of $n+1$ polynomials f_i, $(i = 1, \ldots, n+1)$, with the first n polynomials having common roots $\xi^{(j)}$, $j = 1, \ldots, N$, is given by $\prod_{j=1}^{N} f_{n+1}(\xi^{(j)})$. In particular if $f_{n+1} = s - l_0 - l_1 x_1 - \cdots - l_n x_n = s - l(\mathbf{x})$, then the resultant of the f_i's is $R(s,l) = \prod_{j=1}^{N}(s - l(\xi^{(j)}))$. Further, let $r_{in}(s,l) = (dR/dl_i)$ and $r_d(s,l) = (dR/dl_0)$ and let $\xi^{(j,k)}$ denote the k^{th} coordinate of the j^{th} root. Then we have

$$r_{in}(s,l) = \frac{dR}{dl_i} = \sum_{j=1}^{N}(s - l(\xi^{(1)}))\cdots$$

$$(s - l(\xi^{(j-1)}))(-\xi^{(j,i)})(s - l(\xi^{(j+1)}))\cdots(s - l(\xi^{(N)}))$$

It follows from the above property that the resultant of R and $r_{in}(s,l)$ w.r.t. s is (by evaluating $r_{in}(s,l)$ at the roots of R)

$$\prod_{j=1}^{N}(l(\xi^{(j)}) - l(\xi^{(1)}))\cdots(l(\xi^{(j)}) - l(\xi^{(j-1)}))(-\xi^{(j,i)})$$

$$(l(\xi^{(j)}) - l(\xi^{(j+1)}))\cdots(l(\xi^{(j)}) - l(\xi^{(N)}))$$

Similarly the resultant of R and $r_d(s,l)$ can be shown to be

$$\prod_{j=1}^{N}(l(\xi^{(j)}) - l(\xi^{(1)}))\cdots(l(\xi^{(j)}) - l(\xi^{(j-1)}))(-1)$$

$$(l(\xi^{(j)}) - l(\xi^{(j+1)}))\cdots(l(\xi^{(j)}) - l(\xi^{(N)}))$$

Thus we have

$$\mathit{Resultant}(R, r_d) \mid \mathit{Resultant}(R, r_{in}) \tag{5.13}$$

To use these results in our algorithm, we note that the two step process, at the i^{th} step, of instantiation of f_i with the rational functions and computing the resultants $R(g_{in}, p_{i-1})$ and $R(g_{id}, p_{i-1})$ corresponds exactly to computing the resultant of $f_1, \ldots, f_i$ and $s - l_0 - l_1 x_1 - \cdots - l_i x_i$, since the input system of equations was triangular. Finally, to complete the proof, we apply the same approach as we did above for showing (5.13) to $R(g_{in}, p_{i-1})$ and $R(g_{id}, p_{i-1})$. $\square$

To see that the algorithm works correctly, we note that from Lemma 5.7.1, equation 5.13 is an exact division. If the polynomial $l(\mathbf{x}) = l_0 + l_1 x_1 + \cdots l_{i-1} x_{i-1}$ is specialized to a random linear polynomial L, then with probability one, the values $l(\xi^{(j)})$ will be distinct for distinct $\xi^{(j)}$. The roots of the resultant R will also be distinct. It can be verified that the r_j's have the correct values at $s_i = \alpha^{(k)}$. At termination the algorithm eliminates all the variables $x_1, \ldots, x_n$ and returns rational functions $r_j(s)$ in the variable $s = s_n$ and the polynomial $p(s) = p_n(s_n)$. We therefore have:

Theorem 5.7.2 *Algorithm 5.7.1 returns a symbolic representation* $(p(s), r(s))$ *of the roots of the triangular system* $f_1, \ldots, f_n$ *with commons solutions* $\xi^{(i)} \in \mathbb{C}$. *Here* $p(s)$ *is a univariate polynomial and* $r(s) = (r_1(s), \ldots, r_n(s)$ *are rational functions such that if* $\alpha^{(i)} \in \mathbb{C}$, $i = 1, \ldots, N$ *are roots of* $p(s) = 0$ *then* $\xi^{(i)} = r(\alpha^{(i)})$, $i = 1, \ldots, N$.

5.7.2 Algorithm Details

We now give a more complete description of the algorithm based on the theoretical version given in section 5.7.1. The algorithm performs all its computations representing the polynomials involved as *univariate* polynomials in the *primary variable* with coefficients which are rational functions in the remaining variables. These coefficients are represented by *straight-line programs*

As mentioned in 5.7.1, the first step of the algorithm corresponds to simply computing the resultant of f_1 and $s - l_0 - l_1 x_1$ by setting $x_1 = (s - l_0)/l_1$ in f_1 and then taking derivatives of this resultant w.r.t l_0 and l_1 to get the rational function representation for x_1. The i^{th} step of the algorithm proceeds as follows:

1. Instantiation At the start of the i^{th} step the algorithm the algorithm has computed

$$r_1(s_{i-1}), \ldots, r_{i-1}(s_{i-1}) \text{ and } p_{i-1}(s_{i-1})$$

These are represented as quotients of polynomials in s_{i-1}, the primary variable for the i^{th} iteration (the coefficients of these polynomials are simply numbers as will be seen at the last step of the loop). The algorithm sets

$$x_i = \frac{s_i - l_0 - l_1 x_1 - \cdots - l_{i-1} x_{i-1}}{l_i}$$

with $x_1, \ldots, x_{i-1}$ instantiated to the rational functions r_j. Thus, all of these rational functions including the one for x_i are represented as quotients of univariate polynomials in s_{i-1} whose coefficients are rational functions in s_i and the l variables and represented by SLP's.

2. Substitution The rational functions are substituted in the input polynomial f_i. We therefore get f_i as a rational function which is a quotient of polynomials in s_{i-1}.

3. Resultant computation. The resultants of the numerator and the denominator of the instantiated f_i with the previous resultant p_{i-1} are now computed w.r.t s_{i-1}. These two computations are identical to univariate computations since the cost of doing SLP arithmetic is constant time per operation. The output of this step will be the two resultants. However, since the resultant is a polynomial in the coefficients of the two input polynomials, the resultants so obtained will be both SLP's in s_i and the l_k's.

4. Division We divide the two resultants in SLP form by simply creating a divide node with the two SLP's as children. Call this resultant R.

5. Differentiation To compute the rational functions r_j required for the next step, we need to compute the derivatives of R w.r.t. the l_k's. We do this by differentiating the SLP corresponding to R with respect to each variable as described in section 3.4. We now have a collection of SLP's in s_i and the l_k's.

6. Random instantiation We set each l_k to a random value, to get the random hyperplane corresponding to the L. Our collection of SLP's is now defined only over s_i.

7. Interpolation For the start of the next iteration, we need the resultant $R = p_i$ and the numerators and denominator of the rational functions r_j as *polynomials* in s_i. To do this we interpolate the SLP's representing them into polynomials over s_i. This can be done with straight-forward univariate polynomial interpolation. The degree of the polynomial an SLP represents is obtained by a single depth-first traversal of the SLP. The value of the polynomial at $s_i = q \in \mathbb{Q}$ is obtained by simply evaluating the SLP with s_i set to q, an operation also requiring a depth-first traversal of the SLP. At the end of this process we have the p_i and the numerators and denominator of the r_j's as polynomials in s_i as required by the first step.

5.7.3 Complexity Analysis

We now analyze the time and space requirements of the algorithm in 5.7.2. We will look at the final loop of the algorithm (when x_n is eliminated) since that step dominates the time and space complexity. Note that the space complexity corresponds to the number of operations we will perform over coefficients of univariate polynomials, since instead of actually performing these operations we store them in SLP's.

Let d_i be the total degree of f_i. Then it is straightforward to show that the **degree of the final resultant** $p_n(s_n)$ will be $D = d_1 \cdots d_n$. As explained in 5.7.2, Steps 1 and 2 in the algorithm (instantiation and substitution) have the same complexity as substitution with univariate polynomials. The polynomials we substitute with (i.e. the numerators and denominator of the r_j's) have degree $d_1 \cdots d_{n-1}$ in s_{n-1}. The time and space complexity of these steps is therefore $\mathcal{O}((d_1 \cdots d_n)^2)$. The resultant computation now takes as input two polynomials with degree $\leq D$ and the time and space required is therefore $\mathcal{O}((d_1 \cdots d_n)^2)$. The next step, differentiation takes as input the resultant SLP obtained in the previous step which has size $\mathcal{O}((d_1 \cdots d_n)^2)$. The worst case while differentiating arises when we encounter a division node which gives rise to 5 SLP nodes and we have to take derivatives w.r.t $n+1$ variables. Thus the time required is $\mathcal{O}(5(n+1)(d_1 \cdots d_n)^2)$ and space required is $\mathcal{O}(5(d_1 \cdots d_n)^2)$ (there are $n+1$ derivatives we need to take but we can by perform the differentiation and interpolation w.r.t. a given l_k one after the other and then re-use the space for the next l_k).

Finally we interpolate these SLP's. Each SLP has to be evaluated a number of times equal to the degree of the polynomial (in s_n) it represents. This degree being D, it follows that the time spent in evaluating an SLP's is $\mathcal{O}((d_1 \cdots d_n)^3)$ and since there are $n+1$ SLP's, the total time for interpolation is $\mathcal{O}(n(d_1 \cdots d_n)^3)$ The space required for univariate interpolation is linear in the degree of the input using any standard interpolation algorithm ([Zip93]) and we can re-use the space from the previous interpolation as explained before. Hence the space required for interpolation is $\mathcal{O}((d_1 \cdots d_n))$. We have n such iterations of the algorithm, however the space required in one iteration can be re-used in the other since once the SLP's have been interpolated we no longer need the space to represent them. From this analysis it follows that:

Theorem 5.7.3 *The (probabilistic) time complexity of the above algorithm for determining the root representation of a triangular system is $\mathcal{O}(n^2(d_1 \ldots d_n)^3)$ where d_i is the total degree of f_i. i.e. $\mathcal{O}(n^2 d^{\mathcal{O}(n)})$ where d is an upper bound on each d_i. The space required is $\mathcal{O}((d_1 \ldots d_n)^2)$ or $\mathcal{O}(d^{\mathcal{O}(n)})$*

In contrast the worst-case complexity of computing characteristic sets as required by the original algorithm of Ritt with subsequent modifications by Wu is non-elementary. However, it was shown in [GM91] that the degree of the polynomials in a characteristic set for a zero-dimensional ideal $I \subseteq \mathbb{Q}[x_1, \ldots, x_n]$ is $\mathcal{O}(d^{\mathcal{O}(n)})$ and the complexity of computing these characteristic sets is $\mathcal{O}(n^{3.376}d^{\mathcal{O}(n^3)})$. (The complexity of the algorithm used by Chou's prover, as far as we know, is not known). Thus, at even relatively small degrees, $d \geq 4$, the computation required by the modified Wu's method can be rather expensive. On the other hand, our algorithm is probabilistic as mentioned in the introduction to section 5.7.

5.8 Necessary and Sufficient Conditions for the Complex Case

Once we have a symbolic representation (p, r) of the roots of the triangular system of hypotheses, it is relatively straightforward to determine whether the theorem is true or not over $\mathbb{C}$.

Theorem 5.8.1 (Necessary and Sufficient Condition over $\mathbb{C}$) *Let (S) be a geometric statement with hypotheses $h_1, \ldots, h_n$ and conclusion c, all in $\mathbb{Q}[u_1, \ldots, u_d, x_1, \ldots, x_n]$. Let $f_1, \ldots, f_n \in \mathbb{Q}[x_1, \ldots, x_n]$ be the corresponding triangular system obtained by randomly setting $u_i = v_i$ for $i = 1, \ldots, d$ and applying Algorithm 5.5.1. Let $(p(s), r(s))$ be the symbolic representation obtained by applying Algorithm 5.7.1 to $f_1, \ldots, f_n$. Define*

$$g(s) = \text{Numerator}(c(u_1 = v_1, \ldots, u_d = v_d, x_1 = r_1(s), \ldots, x_n = r_n(s))) \tag{5.14}$$

Let the total degree of $f_1, \ldots, f_n$ equal the total degree of $h_1, \ldots, h_n$ in $x_1, \ldots, x_n$. Then (S) is generally true with probability 1 if and only if $\text{prem}(g(s), p(s)) = 0$

Proof The condition that the total degree of the the system $f_1, \ldots, f_n$ equals that of the hypotheses in $x_1, \ldots, x_n$ ensures that no extra roots were added during the naive triangulation. Thus $(p(s), r(s))$ represents the roots of the hypotheses (with substitution $u_i = v_i$, $i = 1, \ldots, d$) in the sense of Theorem 5.7.2. It follows then that the substitution (5.14) gives us a polynomial $g(s)$ which satisfies the following property: if $\alpha^{(i)}$ is a root of $p(s)$, with corresponding root $\xi^{(i)}$ of $f_1, \ldots, f_n$, then $g(\alpha^{(i)}) = c(\xi^{(i)})$. We note that $prem(g(s), p(s)) = 0$ iff $p(s)$ divides $g(s)$ over $\mathbb{Q}$ iff g vanishes on all the roots $\alpha^{(i)}$ of $p(s)$, i.e., iff $c(\xi^{(i)}) = g(\alpha^{(i)}) = 0$. Since the instantiation $u_1 = v_1, \ldots, u_d = v_d$ was random, it follows that $prem(g(s), p(s) = 0$ iff (S) is generally true with probability 1. $\square$

The above theorem, therefore, gives us the criterion for determining the truth or falsehood of the conjecture made in our geometric statement (S) - we instantiate the conclusion polynomial $c \in \mathbb{Q}[u_1, \ldots, u_d, x_1, \ldots, x_n]$ with $u_i = v_i$ selected randomly and $x_j = r_j(s)$ obtained by applying algorithm 5.7.1 to get a polynomial $g(s)$, and then check whether $prem(g(s), p(s)) = 0$. It can happen, however, that our conjecture was true for a strict subset of the variety corresponding to the hypotheses. This can happen, for example as in the case of Morley's Trisector Theorem (section 5.10.2), when the hypotheses are formulated to include cases where the theorem is not true. We can check such situations using the following criterion:

Corollary 5.8.2 (Conditions for Subvarieties of the Hypothesis Variety) *With the same assumptions and notation as in Theorem 5.8.1, let $p(s) = p_1(s) \cdots p_k(s)$ where the p_i's are the irreducible factors of p. Then each factor $p_i(s)$, $i = 1, \ldots, k$ corresponds to an irreducible subvariety V_i of the original variety and (S) is generally true over V_i with probability 1 if and only if* $\mathrm{prem}(g(s), p_i(s)) = 0$, $i = 1, \ldots, k$.

Proof The proof follows directly from Theorem 5.8.1 and the fact that the resultant of a reducible variety is reducible and factors into irreducible factors, one for each irreducible component. (See, e.g., [vdW50], pp. 13-15.) □

We could have the problem discussed earlier of having introduced extra roots in the triangulation procedure, that is, we could add extra components to the original hypothesis variety. In that case, the variety represented by the triangular system $f_1, \ldots, f_n$ is necessarily reducible and it follows that the resultant polynomial $p(s)$ will also be reducible. To check for this eventuality, we instantiate the original hypotheses polynomials h_j with the rational functions $r_i(s)$ as we did for c to get a corresponding collection of polynomials $g_1(s), \ldots, g_n(s)$. We compute $prem(g_i(s), p_j(s))$ for each factor $p_j(s)$ of $p(s)$. From Corollary 5.8.2 it follows that p can be written as the product of two polynomials $q_1(s)$ and $q_2(s)$ such that $prem(g_i(s), q_1(s)) = 0$ for all i, $i = 1, \ldots, n$ and $prem(g_i(s), q_2(s)) \neq 0$ for some i, $i = 1, \ldots, n$. That is the factor $q_1(s)$ of $p(s)$ corresponds to the original variety and $q_2(s)$ corresponds to the extra component added by the triangulation procedure.

5.9 Necessary and Sufficient Conditions for the Reals

We now turn our attention to the implications of the fact that we are actually supposed to determine the feasibility of conjectures based on hypotheses defined in $K[u_1, \ldots, u_d, x_1, \ldots, x_n]$ rather than just $K[x_1, \ldots, x_n]$. In the latter case, the algorithms given above work in a straightforward manner – one uses simple triangulation followed by the algorithm for triangular systems and then that for sign determination. In the former case, things are not quite as nice. In algorithm 5.6.1 we eliminate the u_i's in step 3 by setting them to random values. This approach works when computing over $\mathbb{C}$ since with probability 1 we will not run into a degenerate situation. However, when the field is not algebraically closed, the situation becomes complicated. Consider, for example, the polynomial equation $x^2 + u^2 - 1 = 0$. For values of u such that $-1 \leq u \leq 1$, the equation has real solutions for x. However for other values of u there are no real solutions. If u were a parameter and the equation was an input hypothesis polynomial, random instantiation for u could lead to a wrong conclusion about the validity of the conjecture over the reals.

Obtaining a sufficient condition for the reals is relatively easy:

Theorem 5.9.1 (Sufficient Condition for the Reals) *With the assumptions and notation of Theorem 5.8.1, define $r(s) = p(s)/(gcd(p(s), g(s))$. If the number of real roots of $r(s)$ equals zero then the statement (S) is generally true with probability 1.*

Proof The roots of r(s) are by assumption all strictly complex which implies that $\mathcal{V}_{\mathbb{R}}(p) \subset \mathcal{V}_{\mathbb{R}}(g)$ (where $\mathcal{V}_{\mathbb{R}}$ denotes the real variety of the particular polynomial). This in turn implies that if $\alpha^{(i)} \in \mathbb{R}$ is a root of $p(s)$ then it is a root of $g(s)$ and hence $g(\alpha^{(i)}) = c(\xi^{(i)}) = 0$. Further since $\xi^{(i)} = (r_1(\alpha^{(i)}), \ldots, r_n(\alpha^{(i)}))$, where $r_j(s) \in \mathbb{Q}(s)$, $j = 1, \ldots, n$, it follows that $\xi^{(i)} \in \mathbb{R}^n$. Thus $\mathcal{V}_{\mathbb{R}}(h_1, \ldots, h_n) = \mathcal{V}_{\mathbb{R}}(f_1, \ldots, f_n) \subset \mathcal{V}_{\mathbb{R}}(c)$ with probability 1. □

Counting the number of real roots of the univariate polynomial $r(s)$ can be done by computing Sturm sequences as described in section 2.2 [2]

For theorem proving over the reals, if $r(s)$ has no real roots then the theorem is true. However, if $r(s)$ does have real roots we need to check if those roots correspond to degenerate conditions or not, i.e. whether the u_j's are algebraically dependent or not on the real components of the variety of r.

In order to disprove conjectures over the reals, we need a *necessary* condition which a system of hypotheses equations and conclusion must satisfy. If the necessary condition is not satisfied then the theorem is not true over the reals. To motivate the method that we use to obtain the necessary condition, we use the example given in the introductory part of this section, namely $r(s, u) = p(s)/(gcd(p(s), g(s)) = s^2 + u^2 - 1$. This defines the unit

[2] Note that we can rephrase the necessary and sufficient conditions for $\mathbb{C}$ in terms of $r(s)$: the statement (S) is generally true with probability 1 iff $r(s)$ is identically 1.

circle in the plane. Given such an r we want to determine whether there exists a connected component of the u-line for which $r(s,u)$ has real solutions. If that is the case then we can select u independently over this connected component and have real solutions for r, that is, u is algebraically *independent* over this connected component. This means that r has a real component on which u is not degenerate and the theorem would be false over the reals since this component of r was not contained in the (real) variety of the conclusion. The condition on r can be obtained by projecting $\mathcal{V}(r)$ down to the u-line and examining the behavior of the projected set $W \subset \mathbb{R}$. Let W^c denote the complement of W in $\mathbb{R}$. We note that the connected components of W^c are necessarily open sets in $\mathbb{R}$. Any point v in one of these open sets corresponds to a selection of the parameter u in $r(s,u)$. We observe that for a given v, $r(s,v) = s^2 + v^2 - 1$ can be solved for s and has a certain number of solutions which remain constant for any v within a connected component. In other words, we look at the fiber over v as v is varied within a component. Thus, each connected component can be represented by a "witness" point v, which tells us the behavior of the variety $\mathcal{V}(r)$ over that component. If the variety has a real zero over a particular witness point then we know that all values in the component corresponding to the witness point give rise to real zeros and u would be algebraically independent in that component implying that the theorem was false. If no connected component of W^c has a real zero in the fiber of its witness point then all the real solutions of r correspond to degenerate conditions on u. Thus all we need is to provide a method for determining a witness point in each component and checking for the number of real zeros its corresponds to. Such a method is outlined in Algorithm 5.9.1 given below. Before we can use the algorithm, though, we need to obtain the polynomial r as a function of both the variable s and the parameters $\mathbf{u}$. We do that as follows:

We run the same algorithms as before (i.e. triangulation and triangular system solution) but instead of random instantiation for the u_j's we do our computation over $K(u_1, \ldots, u_d)$. (The coefficients of the various polynomials will be rational functions in $\mathbf{u}$ and are represented by straight-line-programs.) At the end of this process we obtain $r \in K[\mathbf{u}, s]$ instead of $K[s]$. We then apply the following algorithm:

Algorithm 5.9.1 (Necessary Condition)

1. Compute the Sylvester resultant $R(\mathbf{u})$ of $r(s,\mathbf{u})$ and $dr(s,\mathbf{u})/ds$ with respect to s using a Sturm sequence algorithm to get a remainder sequence,

$$p_1(s,\mathbf{u}), \ldots, p_k(s,\mathbf{u}).$$

2. Compute $R'(\mathbf{u}) = (R+\epsilon)(R-\epsilon)$ where ϵ in an infintesimal variable. R' represents the complement of the projection of $\mathcal{V}(r)$.

3. Choose a random linear map $\pi : \mathbb{R}^d_{|W^c} \rightarrow \mathbb{R}$ and find all the critical points of this map as follows: Let $u'_2, \ldots, u'_d$ be a coordinate basis for the kernel of π which is a $d-1$ dimensional linear space. Solve the following system in the limit as $\epsilon \rightarrow 0$:

$$R' = 0 \qquad \frac{\partial R}{\partial u'_2} = \cdots = \frac{\partial R}{\partial u'_d} = 0 \tag{5.15}$$

using the algorithm described in [Can91b]. The result of this step is a polynomial $H(t)$ and rational functions $(h_1(t), \ldots, h_d(t))$ describing "witness points" in the connected components of $\mathcal{V}_{\mathbb{R}}(r)$.

4. Set $u_i = h_i(t)$ in the leading coefficients of the Sturm sequence $p_1, \ldots, p_k$ computed in step 1. That is, set $u_i = h_i(t)$ in

$$\text{lcoeff}(p_1(s)), \ldots, \text{lcoeff}(p_k(s))$$

5. Apply the sign determination algorithm of [Can91a] to determine the sign sequences for the lcoeff(p_i) at the roots of $H(t)$. For each root α of $H(t)$ we get a sign sequence $\sigma_\alpha \in \{+,-,0\}^k$.

6. Determine the number of real roots for each sign sequence $\sigma_\alpha = \{\sigma_{\alpha_1}, \ldots, \sigma_{\alpha_k}\}$ by computing the sign changes corresponding to α as $SC(\alpha)$ equal to the number of sign alternations in

$$\{\sigma_{\alpha_1}(-1)^{deg(p_1(s))}, \ldots, \sigma_{\alpha_k}(-1)^{deg(p_k(s))}\}$$

minus the number of sign alternations in

$$\sigma_\alpha = \{\sigma_{\alpha_1}, \ldots, \sigma_{\alpha_k}\}$$

7. If one of the roots of $H(t)$ gives rise to a value of $SC(\alpha) > 0$, then $\mathcal{V}_{\mathbb{R}}(r(s, \mathbf{u}))$ has a non-trivial component on which the u_i's are independent and the theorem is false.

□

It can be shown, by an analysis similar to that in section 5.7.3, that the worst case complexity of the above procedure is $\mathcal{O}(d^{\mathcal{O}(n^2)})$. This algorithm has *not* been implemented in our prover yet. From the above discussion we have:

Theorem 5.9.2 (Necessary Condition for the Reals) *Using the notation above, if* $r(s, \mathbf{u}) = p(s, \mathbf{u})/(gcd(p(s, \mathbf{u}), g(s, \mathbf{u})))$ *is not identically* 1 *and has generic real roots then the statement (S) is generally false.*

We can also further conclude from the above discussion that if r is not identically 1 and has *no* generic real roots, the theorem is true (in the case the real components all correspond to degenerate situations). To summarize then:

Theorem 5.9.3 (Necessary and Sufficient Condition for the Reals) *With the assumptions and notation of Theorem 5.8.1, define* $r(s, \mathbf{u}) = p(s, \mathbf{u})/(gcd(p(s, \mathbf{u}), g(s, \mathbf{u}))$. *The statement (S) is generally true iff either* r *is identically 1 or* r *is not identically 1 and has no generic real roots.*

5.10 Comparison with Chou's Prover

We will consider two examples, Brianchon's Theorem and the Thébault-Taylor Theorem which illustrate the differences between the two approaches. Though it might seem somewhat tedious to go through the set of equations defining each problem, it is a rather instructive exercise in understanding the difference between Chou's prover and ours. The reader who wishes to avoid reading through the actual examples might want to skip to the summary at the end of the section.

5.10.1 Brianchon's Theorem

This is the dual of the famous Pascal's Theorem ([Cox69]).

Theorem 5.10.1 (Pascal) *If a hexagon* $ABCDEF$ *is inscribed in a conic, the points of intersection of pairs of opposite sides, i.e.,* (EF, BC), (CD, FA) *and* (AB, CD) *are collinear.*

We consider the dual of this theorem, the proof of which was given by Brianchon, for the case of the circle:

Theorem 5.10.2 (Brianchon) *If a hexagon* $A_1B_1C_1D_1E_1F_1$ *is circumscribed about a circle, its three diagonals, i.e.* A_1D_1, B_1E_1 *and* C_1F_1 *are concurrent. (See Figure 5.2.)*

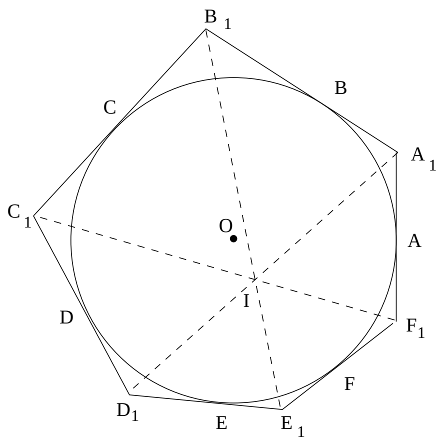

Figure 5.2: Brianchon's Theorem

Let O be the center of the circumscribed circle and A, B, C, D, E and F be points at which the perpendiculars from O meet the hexagon as shown. We can choose O to be the origin $(0, 0)$ without loss of generality. And we can choose point $A = (u_1, 0)$ arbitrarily. Let I be the intersection point of line segments B_1E_1 and A_1D_1. The **conclusion** is: points C_1, F_1 and I are collinear.

Now we let $B = (x_1, u_2)$, $C = (x_2, u_3)$, $D = (x_3, u_4)$, $E = (x_4, u_5)$, $F = (x_5, u_6)$, $A_1 = (u_1, x_6)$, $B_1 = (x_8, x_7)$, $C_1 = (x_{10}, x_9)$, $D_1 = (x_{12}, x_{11})$, $E_1 = (x_{14}, x13)$, $F_1 = (u_1, x_{15})$ and $I = (x_{17}, x_{16})$.

The hypotheses can be converted to polynomial equations as follows.

$$
\begin{aligned}
BO \equiv OA: &\quad {x_1}^2 + {u_2}^2 - {u_1}^2 = 0 \\
CO \equiv OA: &\quad {x_2}^2 + {u_3}^2 - {u_1}^2 = 0 \\
DO \equiv OA: &\quad {x_3}^2 + {u_4}^2 - {u_1}^2 = 0 \\
EO \equiv OA: &\quad {x_4}^2 + {u_5}^2 - {u_1}^2 = 0 \\
FO \equiv OA: &\quad {x_5}^2 + {u_6}^2 - {u_1}^2 = 0 \\
A_1B \perp BO: &\quad (u_1 - x_1)(x_1 - 0) + (x_6 - u_2)(u_2 - 0) = 0 \\
B_1C \perp CO: &\quad (x_8 - x_2)(x_2 - 0) + (x_7 - u_3)(u_3 - 0) = 0 \\
B_1B \perp BO: &\quad (x_8 - x_1)(x_1 - 0) + (x_7 - u_2)(u_2 - 0) = 0 \\
C_1C \perp CO: &\quad (x_{10} - x_2)(x_2 - 0) + (x_9 - u_3)(u_3 - 0) = 0 \\
C_1D \perp DO: &\quad (x_{10} - x_3)(x_3 - 0) + (x_9 - u_4)(u_4 - 0) = 0
\end{aligned}
$$

$$
\begin{aligned}
D_1D \perp DO: &\quad (x_{12}-x_3)(x_3-0)+(x_{11}-u_4)(u_4-0)=0\\
D_1E \perp EO: &\quad (x_{12}-x_4)(x_4-0)+(x_{11}-u_5)(u_5-0)=0\\
E_1E \perp EO: &\quad (x_{14}-x_4)(x_4-0)+(x_{13}-u_5)(u_5-0)=0\\
E_1F \perp FO: &\quad (x_{14}-x_5)(x_5-0)+(x_{13}-u_6)(u_6-0)=0\\
F_1F \perp FO: &\quad (u_1-x_5)(x_5-0)+(x_{15}-u_6)(u_6-0)=0\\
I\ on\ B_1E_1: &\quad (x_{13}-x_7)(x_{17}-x_8)-(x_7-x_{16})(x_8-x_{14})=0\\
I\ on\ A_1D_1: &\quad (x_{11}-x_6)(x_{17}-u_1)-(x_6-x_{16})(u_1-x_{12})=0
\end{aligned}
\tag{5.16}
$$

The conclusion becomes

$$
C_1, F_1, I\ collinear: \quad (x_{10}-u_1)(x_{15}-x_{16})-(x_9-x_{15})(u_1-x_{17}) \tag{5.17}
$$

Both Chou's prover and our algorithm will first apply the naive triangulation procedure to the above system to get a triangular system:

$$
\begin{aligned}
&f_1 = {x_1}^2+{u_2}^2-{u_1}^2=0\\
&f_2 = {x_2}^2+{u_3}^2-{u_1}^2=0\\
&f_3 = {x_3}^2+{u_4}^2-{u_1}^2=0\\
&f_4 = {x_4}^2+{u_5}^2-{u_1}^2=0\\
&f_5 = {x_5}^2+{u_6}^2-{u_1}^2=0\\
&f_6 = (u_1-x_1)(x_1-0)+(x_6-u_2)(u_2-0)=0\\
&f_7 = x_2{x_1}^2+(u_3x_7-{u_3}^2-{x_2}^2)x_1+x_2u_2(u_2-x_7)=0\\
&f_8 = (x_8-x_1)(x_1-0)+(x_7-u_2)(u_2-0)=0\\
&f_9 = x_3{x_2}^2+(u_4x_9-{u_4}^2-{x_3}^2)x_2+x_3u_3(u_3-x_9)=0\\
&f_{10} = (x_{10}-x_3)(x_3-0)+(x_9-u_4)(u_4-0)=0\\
&f_{11} = x_4{x_3}^2+(u_4x_{11}-{u_4}^2-{x_3}^2)x_4+x_3u_5(u_5-x_{11})=0\\
&f_{12} = (x_{12}-x_3)(x_3-0)+(x_{11}-u_4)(u_4-0)=0\\
&f_{13} = x_4{x_5}^2+(u_5x_{13}-{u_5}^2-{x_4}^2)x_5+x_4u_6(u_6-x_{13})=0\\
&f_{14} = (x_{14}-x_5)(x_5-0)+(x_{13}-u_6)(u_6-0)=0\\
&f_{15} = (u_1-x_5)(x_5-0)+(x_{15}-u_6)(u_6-0)=0\\
&f_{16} = ((u_1-x_{12})x_{13}+(x_{12}-u_1)x_7+(x_{11}-x_6)x_{14}+(x_6-x_{11})x_8)x_{16}+\\
&((x_{11}-x_6)x_8+x_6x_{12}-u_1x_{11})x_{13}+((x_6-x_{11})x_{14}-x_6x_{12}+u_1x_{11})x_7=0\\
&f_{17} = (x_{11}-x_6)(x_{17}-u_1)-(x_6-x_{16})(u_1-x_{12})=0
\end{aligned}
$$

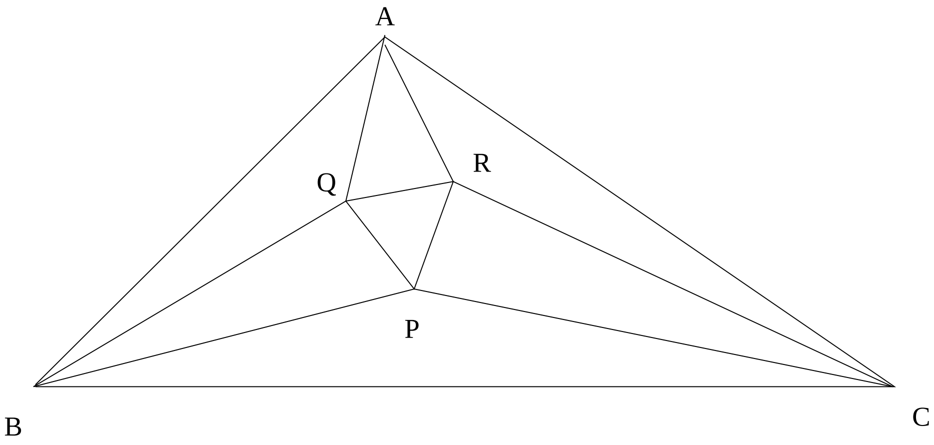

Figure 5.3: Morley's Trisector Theorem

Both systems of equations have 32 solutions for a given selection of values for the parameters u_i ; these solutions correspond to selecting where the midpoints of the sides of the hexagon will go (one of those, A, is fixed on the x-axis, but the other 5 are free to be placed on either side of the y-axis). One would then imagine that determining the validity of the conclusion would require the solution of this rather large system. However, it turns out that $prem(g, f_1, \ldots, f_{17}) = 0$, which proves the theorem. Thus the time taken for proving Brianchon's theorem for both provers is inconsequential, of the order of less than 1 second. It turns out that an overwhelming majority of the theorems proved by Chou's prover are of this form, that is, where the naive triangulation process is sufficient. The core component of the Ritt-Wu algorithm, namely deciding whether a triangular system is reducible and if so, factoring it, is never called. Obviously, proving such a theorem is no test for a prover.

5.10.2 Morley's Trisector Theorem

This theorem is one of the few theorems dealt with in the literature which forms a non-trivial test of the ability of a theorem prover. Interestingly enough, it turns out that one can reformulate the theorem as shown by Wu [Wu86] and it can be reduced to a system of equations which can be solved using naive triangulation. The more interesting problem arises when the theorem is formulated without using Wu's trick.

Theorem 5.10.3 (Morley's Trisector Theorem) *The neighboring trisectors of the three angles of a triangle intersect to form twenty-seven triangles in all, of which eighteen* *are equilateral. (See Figure 5.3.)*

The straightforward formulation is obtained from the following hypotheses:

$$\angle ABC = 3\angle PBC\ (h_1),\ \angle ACB = 3\angle PCB\ (h_2),\ \angle CAB = 3\angle RAB\ (h_3)$$

$$\angle ABR = \angle PBC\ (h_4),\ \angle ACQ = \angle PCB\ (h_5),\ \angle BAR = \angle QAC\ (h_6).$$

The conclusion to be proved is given by

$$PQ = PR,\ PQ = QR.$$

We assign the coordinates as follows, $A = (x_2, x_1)$, $B = (u_1, 0)$, $C = (u_2, 0)$, $P = (0, 1), Q = (x_6, x_5)$ and $R = (x_4, x_3)$. We can translate the hypotheses into polynomial equations by using the tangent equations for the angles (cf. [Cho88], pp. 40).

$$h_1 = (u_1 - x_2) - 3x_1u_1 - 3{u_1}^2(u_1 - x_2) + x_1{u_1}^3 \tag{5.18}$$

$$h_2 = (u_2 - x_2) - 3x_1u_2 - 3{u_2}^2(u_2 - x_2) + x_1{u_2}^3 \tag{5.19}$$

$$h_3 = \frac{(bc^3 - 3ac^2d - 3cd^2b + ad^3)}{({u_1}^2 - 2u_1x_2 + {x_2}^2 + {x_1}^2)} \tag{5.20}$$

where,

$$\begin{aligned} a &= (x_1u_1 - x_1u_2) \\ b &= {x_1}^2 + (x_2 - u_1)(x_2 - u_2) \\ c &= x_2x_3 - x_1x_4 + x_1u_1 - x_3u_1 \\ d &= x_1(x_1 - x_3) + (x_2 - u_1)(x_2 - x_4) \end{aligned}$$

$$h_4 = u_1(x_3(x_2 - u_1) - x_1(x_4 - u_1)) - x_1x_3 - (x_2 - u_1)(x_4 - u_1) \tag{5.21}$$

$$h_5 = u_2(x_5(x_2 - u_2) - x_1(x_6 - u_2)) - x_1x_5 - (x_2 - u_2)(x_6 - u_2) \tag{5.22}$$

$$h_6 = a_2d_2 - b_2c_2 \tag{5.23}$$

where,

$$\begin{aligned} a_2 &= (x_2 - u_1)(x_1 - x_3) - x_1(x_2 - x_4) \\ b_2 &= (x_2 - x_4)(x_2 - u_1) + x_1(x_1 - x_3) \\ c_2 &= x_1(x_2 - x_6) - (x_1 - x_5)(x_2 - u_2) \\ d_2 &= x_1(x_1 - x_5) + (x_2 - x_6)(x_2 - u_2) \end{aligned}$$

The conclusions are given by:

$$c_1 = {x_6}^2 + (x_5 - 1)^2 - {x_4}^2 - (x_3 - 1)^2 \tag{5.24}$$

and

$$c_2 = {x_6}^2 + (x_5 - 1)^2 - (x_6 - x_4)^2 - (x_5 - x_3)^2 \tag{5.25}$$

Thus we get a system of 6 equations each with the degree of the leading variable 1 except for the constraint $\angle CAB = 3\angle RAB$ which gives us an equation in x_4 of degree 3. Thus our system of equations should generically give rise to 3 solutions for given values of u_1, u_2. On application of the naive triangulation algorithm, we get a triangular set of equations $f_1, \ldots, f_6$. It turns out the pseudo-division does not give as a zero remainder and thus we have a non-trivial theorem at hand. This arises from the fact that the above hypotheses include the 9 triangles which are not equilateral. Clearly we have two components - corresponding to the two sets of triangles. We now apply our full algorithm to get a resultant polynomial $p(s)$ of degree 3 in s, which can be factored into two factors of degrees 1 and 2, one for each component. Each factor encodes the corresponding sub-variety and the theorem turns out to be true with respect to the degree 2 factor but not with respect to the degree 1 factor as expected.

We note that in Chou's prover the formulation used is that due to Wu mentioned earlier (i.e. one in which the bad case is eliminated by reformulating the theorem) and this system of equations does not require factoring.

5.10.3 Thébault-Taylor Theorem

We now come to a more interesting theorem. V. Thébault made this conjecture in 1938 and it was finally proved by K.B. Taylor in 1983 [Tay83].

Theorem 5.10.4 (Thébault-Taylor) *Let ABC be a triangle, with M a point on side BC. Let O and I be the centers of the circumscribed circle (O) and inscribed circle (I) respectively of ABC. Let w_1 be the center of the circle (w_1) which is tangent to (O) and to the two sides of angle AMB. Let w_2 be the center of the circle (w_2) which is tangent to (O) and to the two sides of angle AMC. Then the straight line joining w_1 and w_2 passes through I. (See Figure 5.4.)*

Following Chou ([Cho88], pp. 66-67), we will rephrase the theorem as follows:

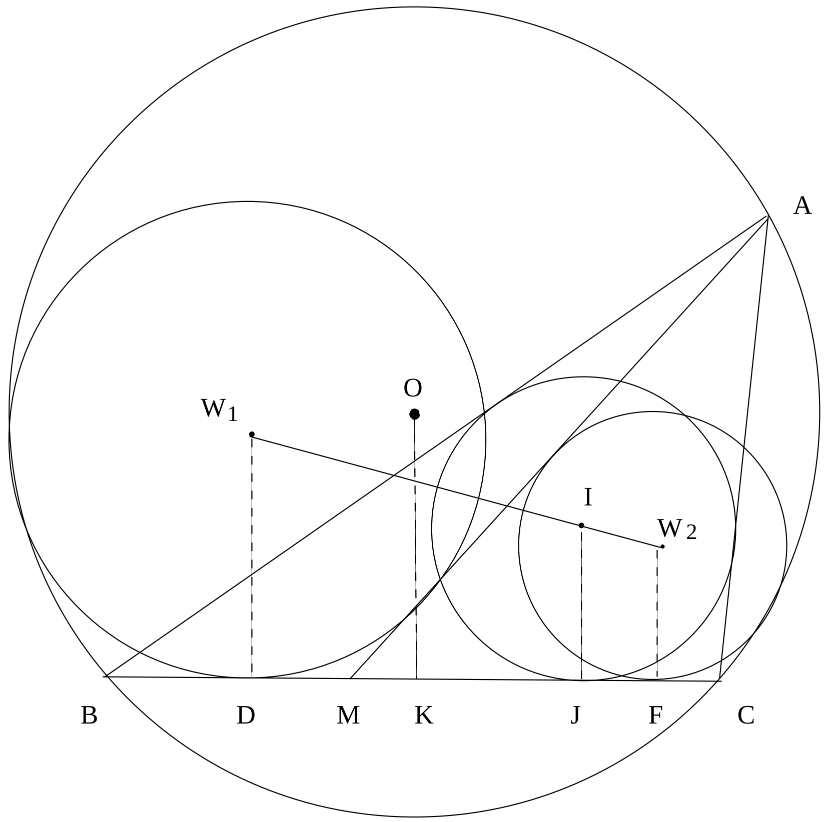

Figure 5.4: Thébault-Taylor Theorem

Theorem 5.10.5 (Thébault-Taylor 2nd version) *Let ABC be a triangle, with M a point on side BC. Let O and I be the centers of the circumscribed circle (O) and inscribed circle (I) respectively of ABC. Let w_1 be the center of the circle (w_1) which is tangent to (O) and to the two sides of angle AMB. Let w_2 be a point on line w_1I such that $w_2M \perp w_1M$. Then the circle (w_2) centered at w_2 is tangent to the circle (O).*

The equations for this theorem can be set up in a manner similar to that for Brianchon's theorem: Let D, F and K be the projections of w_1, w_2 and O respectively on the line BC. We make the following assignments: $B = (0, 0)$, $M = (u_1, 0)$, $A = (u_2, u_3)$, $w_1 = (x_2, x_3)$, $O = (1, x_1)$, $D = (x_2, 0)$, $K = (1, 0)$, $C = (x_4, 0)$, $I = (x_5, x_6)$, $w_2 = (x_7, x_8)$ and $F = (x_7, 0)$.
The hypotheses can be converted into polynomial equations as follows:

$$
\begin{array}{lll}
OB \equiv OA: & h_1 = & -2u_3x_1 + u_3{}^2 + u_2{}^2 - 2u_2 = 0 \\
Mw_1 \text{ bisects } \angle AMB: & h_2 = & ((2u_1u_2 - 2u_1{}^2)x_2 - 2u_1{}^2u_2 + 2u_1{}^3)x_3 + \\
 & & u_1u_3x_3{}^2 - u_1u_3x_2{}^2 + 2u_1{}^2u_3x_2 - u_1{}^3u_3 = 0 \\
Circle\ (w_1)\ tgt\ to\ circle\ (O): & h_3 = & 4x_3{}^2 + (4x_1x_2{}^2 - 8x_1x_2)x_3 \\
 & & -x_2{}^4 + 4x_2{}^3 - 4x_2{}^2 = 0 \\
K\ midpoint\ of\ BC: & h_4 = & x_4 - 2 = 0 \\
IC\ bisects\ \angle ACB: & h_5 = & ((-2x_4{}^2 = 2u_2x_4)x_5 + 2x_4{}^3 - 2u_2x_4{}^2)x_6 + \\
 & & u_3x_4x_6{}^2 + -u_3x_4x_5{}^2 + 2u_3x_4{}^2x_5 - u_3x_4{}^3 = 0 \\
IB\ bisects\ \angle ABC: & h_6 = & -u_3x_4x_6{}^2 - 2u_2x_4x_5x_6 + u_3x_4x_5{}^2 = 0 \\
w_2\ is\ on\ w_1I: & h_7 = & (-x_5 + x_2)x_8 + (x_6 - x_3)x_7 - x_2x_6 + x_3x_5 = 0 \\
w_2M \perp w_1I: & h_8 = & x_3x_8 + (x_2 - u_1)x_7 - u_1x_2 + u_1{}^2 = 0
\end{array}
\tag{5.26}
$$

And the conclusion that circle (w_2) is tangent to (O) is given by

$$
g = 4x_8{}^2 + (4x_1x_7{}^2 - 8x_1x_7)x_8 - x_7{}^4 = 4x_7{}^3 - 4x_7{}^2 = 0 \tag{5.27}
$$

As before, both provers first perform a naive triangulation. The resulting triangular system is rather long and we will not write it down here. The key observation though is that the hypothesis equation h_3 has degree 4 in the variable x_2. Eliminating x_2 from h_2 by taking a pseudoremainder results in an equation of degree 8 in x_2. Similarly eliminating x_6 from h_5 results in an equation of degree 4 in x_5. Thus we again have a system with 32 solutions. However, the triangular system now obtained is *not* reducible. This does not pose a problem for our prover. We simply proceed to the next step which is to eliminate the variables $x_1, \ldots, x_8$ in succession to get a resultant of degree 32 and rational functions encoding the roots of the system. Again the resultant polynomial $p(s)$ can be factored into four components of degree 8 each. The validity of the conclusion is then checked as described in section 5.8 and it turns out that the conclusion is true with respect to only one of the components. The time taken is about 45 minutes on a Sun Sparc 10.

Chou's prover cannot deal with this problem. It will require factoring a degree 8 equation as well as a degree 4 equation over an extension field. As described in [Cho88] (pp. 67-69), Chou's approach is to first reformulate the problem to get a system of hypotheses equations which are at most quadratic in degree. This allows the factoring of the hypotheses into 4 systems of equations each of degree 8, corresponding to 4 general components $C_1, \ldots, C_4$ of the original system . Despite this step, the prover is unable, due to computational limitations, to prove the theorem. Instead, Chou uses *the proof of Thébault's conjecture by Taylor* to prove a generalization of the Thébault-Taylor's theorem. This requires proving that one of the components satisfies $prem(g, C_i) = 0$. This subproblem itself requires a considerable amount of time and memory. To quote Chou (op. cit, pp. 69):

> The total CPU time (on a Symbolics 3600) for computing $prem(g, C_1) = 0$ was about 44 hours, 99 polynomials of more than 100,000 terms. The largest remainder produced has 674,927 ($= maxt$) terms.

We note that the above example has not been solved by any theorem proving algorithm we know of. Wang [Wan95] reports the solution of this problem using a different formulation which reduces the problem to a degree 8 system rather than the original degree 32 system. Though, in general, it is a good idea to sidestep the complexity of a problem by reformulation, the process is nontrivial and requires coming up with clever re-statements of the original theorem.

The above example shows rather explicitly some of the ideas that our prover exploits strongly to be able to prove such theorems:

- The resultant approach does not require factoring over extension fields. This is not an issue in the cases where the theorems can be proved using naive triangulation. However, when naive triangulation fails, it becomes critical, especially in the case of theorems involving polynomials of high degree. The only factoring we require is that of the resultant polynomial itself and the degree of this polynomial is the product of the degrees of the polynomials in the triangular form.

- We note in passing that high degree polynomials are usually brought about by constraints involving two circles (such as tangency). Most of the "standard" theorems do not involve such constraints. If a prover were to be extended to more complicated objects and constraints such as those in three dimensions, our approach will scale in a straightforward manner with the degree of the objects.

- Our prover avoids requiring considerable CPU time by the use of straight-line programs. The penalty is that we will require a large amount of memory to store the computations in SLP form. In the case of the Thébault-Taylor theorem we required about 20 megabytes of memory.

5.11 Some benchmarks

The table below gives some benchmarks on some standard problems. All the benchmarks were run on a Sun Sparc 10 machine. These benchmarks were obtained from a first implementation and we expect to reduce the running time by using various optimizations but the initial benchmarks seem to be rather promising.

Theorem	Vars	Deg	Time (s) (RR)	Space SLP's	Time (Proof)
Pappus	7	1	0.03	70	0.03
Pappus Dual	7	1	0.08	75	0.08
AMS-1	13	1	0.23	90	0.1
Pascal Conic-1	15	1	0.38	100	0.2
Simson	11	2	0.31	180	0.2
Steiner	17	8	19.95	3320	1.7
Kirkman	17	8	20.40	3330	1.5
Brianchon	17	32	1623.60	51500	3.5
Morley	6	3	4.0	4500	4.5
Thebault	8	32	45min	10^6	50min

Explanation of benchmarks : The first column in the table gives the name of the theorem under consideration. Statements of these theorems can be obtained from [Cho88]. The second and third columns give the number of variables and the total degree of the system in the variables $x_1, \ldots, x_n$. The fourth column gives the time required to compute the *root representation* of the system using Algorithm 5.7.1. The computations in the algorithm were done modulo a large prime and as such are faster than would be obtained by running the algorithm with arbitrary precision arithmetic. The space is the number of SLP's required. The proof time is the time required to prove the theorem. As noted before, most theorems fall in the category of irreducible statements and a simple application of the naive triangulation (Algorithm 5.5.1) with *no instantiation* for the parameters u_i's suffices. The times shown are for that case for all the theorems above except for Morley's and the Thebault-Taylor theorem - in those cases, naive triangulation does not work and we have to use the full algorithm with instantiation for the parameters. (Note, for example, the difference in the times for proving Brianchon's theorem with naive triangulation as compared to obtaining the root representation for the same.)

Chapter 6

Conclusion and Future Work

This thesis attempts to set a framework for solving problems involving multivariate polynomial equations and inequalities. Our particular emphasis has been on obtaining solutions over the reals. This area of what might be termed "computational algebraic geometry" is not as developed as the case of algebraically closed fields. We provide a computational framework of algorithms and data structures to solve a variety of problems over the reals. These algorithms and data structures have been implemented in our toolkit and we present their application to several problems. The problems we study in this thesis pose interesting questions in their own right and we have provided efficient solutions to some of them.

Our primary algebraic tool is the resultant. Resultant based methods are fast gaining popularity as they tend to be faster than standard algebraic approaches like Gröbner bases for certain problems [MC92]. Their matrix formulation makes them particularly well suited to taking advantage of efficient and time-tested eigenvalue methods such as those implemented in standard linear algebra packages like LAPACK. One of the main contributions of this thesis is the demonstration of the utility of the straight-line program data structure. The SLP forms the cornerstone of our implementation and we show how to use it in conjunction with various symbolic algorithms which would otherwise be computationally rather severe.

We have applied the algorithms and data structures in the toolkit to various problems: In Chapter 3 we considered the problem of counting real roots for bivariate polynomials correctly and efficiently. We showed how a simple use of SLP's allows us to reduce the complexity of the problem. In Chapter 4 the problem of implicit point location and predicates for geometry was considered. We give an algorithm based on Renegar's algorithm [Ren89b] to solve this problem. We use the SLP approach again, this time to allow us a clear partition between the symbolic and numeric components of the algorithm - the SLP can be used to obtain a permanent symbolic encoding of a particular predicate and allows the use of different numerical methods to solve a particular instance of the problem. We also propose the use of mixed adaptive floating point arithmetic which allows us to take advantage of the speed of finite field computations coupled with the accuracy of the adaptive floating point method.

In Chapter 5 we considered the geometric theorem proving problem, a problem defining an important research area in its own right. We provided a resultant based algorithm for solving triangular systems such as those found in geometric theorem proving systems. We gave necessary and sufficient conditions for proving geometric theorems over both the complexes and the reals. Our method compares favorably to existing methods, as is borne out by the ability of our implemented prover to handle nontrivial theorems of large degree such as the Thébault-Taylor theorem which were beyond the ability of any prover to date.

6.1 Future Directions

There are several immediate problems which arise from the work described in this thesis. We have shown how to compute the number of real roots for polynomial systems defined by two equations in two variables using Sylvester resultants. The next step is to use the sparse resultant formulation [CE93] to compute the real roots of systems with more variables. In [Emi94] and [ER94], this formulation was applied successfully to count (or give upper bounds for) the total number of isolated roots (both real and complex) for large systems such as those defined by the Stewart platform described in Chapter 1. Systems such as these often have variables which describe physical quantities, such as joint angles or leg length, related to the system. These quantities are constrained to take on real values and though computing the total number of solutions is a good starting point, an even better solution to such problems would be the determination of bounds on the number of real roots and the use of these bounds for computing (an approximation to) the real roots. The logical extension of such problems, therefore, is to compute the number of real roots for such systems.

The implicit point location problem raises various numerical questions which have yet to be answered satisfactorily. We have proposed the mixed adaptive floating point and the imperative here is to test the feasibility of such an approach. The numerical aspect of algebraic computations is often ignored in the literature with its focus on symbolic algorithms. Arithmetic, in practice, unfortunately, does not incur a constant time cost per step and many seemingly efficient symbolic algorithms get slowed down considerably when implemented.

The geometric theorem proving problem poses problems ranging from purely theoretical questions to very practical problems involving issues of memory management when dealing with straight-line programs. On the theoretical front, there are various questions involving parametrization of real varieties. A second look at the problem definition tells us that the theorem proving problem lies in a broader class of problems: in the hypotheses equations, we are essentially given an underconstrained set of polynomial equations which define a variety which has dimension greater than zero. The "solution set" of such a system will then be parametrized by a subset of the variables (namely the parameters). Over the reals then the question becomes that of obtaining the description of the real components of this variety. The conclusion, in turn, also defines a nonzero-dimensional variety. And the broader question then is determining when a variety is contained in another.

The geometric theorem proving problem was shown to be related to the problem of constraint-based systems. Though a simpler problem (due to the absence of the parameters) it raises computational issues in its own right. There has been a fair amount of work done on developing constraint-based solvers (see [Ald88], [BFH+95], [Owe91] for a sampling). However, most of the work has focused on constraint-solvers for relatively simple objects such as line segments, points etc. We expect that the techniques that we use to solve the geometric theorem proving problem will enable us to solve problems involving more complicated objects such as curves and surfaces.

In conclusion, we hope this thesis will provide additional impetus to the study of various computational issues raised by problems involving nonlinear algebraic systems. Despite their evident ubiquity, some of these questions seem to have been ignored in the burgeoning area of "computational science", where the focus has been on solving linear and differential systems. This has perhaps been due to the added complexity when dealing with nonlinear systems and perhaps due to the lack of an efficient computational framework for attempting to solve such problems. It is our hope that this thesis fills some parts of this void.

Bibliography

[ABRW94] M. Alonso, E. Becker, M.-F. Roy, and T. Wörmann. Zeros, Multiplicities and Idempotents for Zero Dimensional Systems. In *Proceedings, MEGA-94*, 1994.

[Ald88] B. Aldefeld. Variations of Geometries based on a Geometric-Reasoning Method. *Computer Aided Design*, 20(3):117–126, 1988.

[AS88] 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., Intern. Series of Numerical Math., 86*, pages 12–30. Birkhäuser Verlag, Basel, 1988.

[Bar68] E.H. Bareiss. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Math. Comp.*, 22(103):565–578, 1968.

[Ber75] D. N. Bernstein. The number of roots of a system of equations. *(Translated from) Funktsional'nyi Analiz i Ego Prilozheniya*, 9(3):1–4, Jul-Sep 1975.

[BFH+95] W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A Geometric Constraint Solver. *Computer Aided Design*, 27(6):487–501, 1995.

[BOKR86] M. Ben-Or, D. Kozen, and J. Reif. The Complexity of Elementary Algebra and Geometry. *J. Comp. and Sys. Sci.*, 32:251–264, 1986.

[BR90] R. Benedetti and J.-J. Risler. *Real Algebraic and Semi-Algebraic Sets*. Hermann, Paris, 1990.

[BS83] W. Baur and V. Strassen. The Complexity of Partial Derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[BT71] W.S. Brown and J.F. Traub. On Euclid's Algorithm and the Theory of Subresultants. *Journal of the ACM*, 18:505–514, 1971.

[Can88] J.F. Canny. Some Algebraic and Geometric Computations in PSPACE. In *ACM Symposium on Theory of Computing*, pages 460–467, 1988.

[Can91a] J.F. Canny. An Improved Sign Determination Algorithm. In *AAECC-91*, 1991. New Orleans.

[Can91b] J.F. Canny. Computing Roadmaps of General Semi-Algebraic Sets. In *AAECC-91*, 1991. New Orleans.

[Can93] John Canny. Improved Algorithms for Sign-Determination and Existential Quantifier Elimination. *Computer Journal*, 1993. Special Issue on Quantifier Elimination.

[CE93] John Canny and Ioannis 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.

[CF] C. Chauvin and J.-C. Faugère. Basic user's manual of faugère's gb package. Manuscript available at http://posso.ibp.fr.

[CGZ94] S.-C. Chou, X.-S. Gao, and J.-Z. Zhang. *Machine Proofs in Geometry - Automated Production of Readable Proofs for Geometry Theorems*. World Scientifc, Singapore, 1994.

[Cho84] 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.

[Cho88] S.-C. Chou. *Mechanical Geometry Theorem Proving*. Mathematics and its applications. D. Reidel, Holland, 1988.

[CLO92] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, 1992.

[Col67] G. Collins. Subresultants and Reduced Polynomial Remainder Sequences. *Journal of ACM*, 14:128–142, 1967.

[Col75] G. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Springer LNCS*, 33:135–183, 1975.

[Cox69] H.S.M. Coxeter. *Introduction to Geometry*. J. Wiley and Sons, Inc., New York, 1969.

[Cra86] J. Craig. *Introduction to Robotics*. Addison-Wesley Publishing Co., 1986.

[Dix08] A.L. Dixon. The eliminant of three quantics in two independent variables. *Proceedings of London Mathematical Society*, 6:49–69, 209–236, 1908.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Number 10 in EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1987.

[Emi94] I. Emiris. Sparse Elimination and Applications in Kinematics. Ph.D. Thesis, University of Calfifornia, Berkeley, December 1994.

[ER94] 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.

[Fau94] J.-C. Faugère. Résolution des Systèmes d'Equations Algébriques. Ph.D. Thesis, Université Paris 6, 1994.

[FIKY88] 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.

[For92] S. Fortune. Numerical Stability of Algorithms for Delaunay Triangulations. In *ACM Symposium on Computationl Geometry*, pages 83–92, 1992.

[FW93] S. Fortune and C. Van Wyk. Efficient Exact Arithmetic for Computational Geometry. In *ACM Symposium on Computationl Geometry*, pages 163–172, 1993.

[GCL92] K. Geddes, S. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, Boston, MA, 1992.

[Gel63] H. Gelernter. Realization of a Geometry Theorem Proving Machine. In E.A. Feigenbaum and J.E. Feldman, editors, *Computers and Thought*. McGraw-Hill, New York, 1963.

[GH93] M. Giusti and J. Heintz. La détermination des points isolés et de la dimension d'une variété algébrique peut se faire en temps polynomial. In D. Eisenbud and L. Robbiano, editors, *Computational Algebraic Geoemetry and Commutative Algebra - Proce edings of the Cortona Conference*, volume 34 of *Symposia Matematica*. Cambridge University Press, 1993.

[GJM+95] M. Giusti, J.Heintz, J. Morais, J. Morgenstern, and L.Pardo. Straight-line Programs in Geometric Elimination Theory. Manuscript, 1995.

[GJML95] M. Giusti, J.Heintz, J. Morais, and L.Pardo. When Polynomial Equation Systems Can Be "Solved" Fast ? In *AAECC-11*. Springer LNCS 948, 1995.

[GKZ90] I. M. Gel'fand, M. M. Kapranov, and A. V. Zelevinsky. Discriminants of Polynomials in Several Variables and Triangulations of Newton Polytopes. *(Translated from) Algebara i Analiz*, 2:1–62, 1990. (English Translation in) *Leningrad Math. J.* 2 (1991).

[GM91] G. Gallo and B. Mishra. Efficient Algorithms and Bounds for Wu-Ritt Characteristic Sets. In T. Mora and C. Traverso, editors, *Effective Methods in Algebraic Geometry*, volume 94 of *Progress in Mathematics*, pages 119–142. Birkhäuser, 1991.

[Hav91] T. Havel. Some Examples of the use of Distances as Coordinates for Euclidean Geometry. *Journal of Symbolic Computation*, 11:579–593, 1991.

[Hof89] C.M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, San Mateo, California, 1989.

[HRS90] J. Heintz, M.F. Roy, and P. Solerno. Complexité du Principe de Tarski-Seidenberg. *Bull. Soc. Math. France*, 118:101–126, 1990.

[HS80] J. Heintz and C.P. Schnorr. Testing Polynomials which are Easy to Compute. In *ACM Symposium on Theory of Computing*, 1980.

[IM83] O. Ibarra and S. Moran. Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs. *JACM*, 30(1):217–228, 1983.

[Kal88] Erich Kaltofen. Greatest Common Divisors of Polynomials Given by Straight-Line programs. *Journal of the ACM*, 35(1):231–264, 1988.

[Kal89] Erich Kaltofen. Factorization of Polynomials Given by Straight-Line Programs. In S. Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, pages 375–412. JAI Press, Greenwhich, Connecticut, 1989.

[Kap86] D. Kapur. Geometry Theorem Proving using Hilbert's Nullstellensatz. In *Symposium on Symbolic and Algebraic Computation*, pages 202–208, 1986. Waterloo, Ontario.

[Kap88] D. Kapur. A Refutational Approach to Geometry Theorem Proving. In D. Kapur and J.L. Mundy, editors, *Geometric Reasoning.* MIT Press, 1988.

[Kho78] A.G. Khovanskii. Newton polyhedra and the genus of complete intersections. *(Translated from) Funktsional'nyi Analiz i Ego Prilozheniya*, 12(1):51–61, Jan-Mar 1978.

[KL92] D. Kapur and Y. Lakshman. Elimination Methods: An Introduction. *Journal of Symbolic Computation*, 11, 1992.

[Knu81] D. E. Knuth. *The Art of Computer Programming, vol. II - Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 1981.

[KS86] B. Kutzler and S. Stifter. On the Application of Buchberger's Algorithm to Automated Geometry Theorem Proving. *Journal of Symbolic Computation*, 2:409–420, 1986.

[KS91] E. Kaltofen and M. Singer. Size Efficient Parallel Algebraic Circuits for Partial Derivatives. In *Proceedings of the IV International Conference on Computer Algebra in Physical Research*, pages 133–145, 1991. Singapore.

[KSY94] D. Kapur, T. Saxena, and L. Yang. Algebraic and Geometric Reasoning using Dixon Resultants. In *International Symposium on Symbolic and Algebraic Computation*, pages 99–107, New York, N.Y., 1994. ACM Press.

[Kus75] A.G. Kushnirenko. The Newton polyhedron and the number of solutions of a system of k equations in k unknowns. *(Translated from) Uspekhi Mat. Nauk.*, 30:266–267, 1975.

[Laz81] D. Lazard. Résolution des Systèmes d'Equations Algébriques. *Theoretical Computer Science*, 15:77–110, 1981.

[Mac21] F.S. Macaulay. Note on the resultant of a number of polynomials of the same degree. *Proceedings of London Mathematical Society*, pages 14–21, June 1921.

[MC92] D. Manocha and J. Canny. Multipolynomial Resultants and Linear Algebra. In *International Symposium on Symbolic and Algebraic Computation*, pages 96–102, Berkeley, July 1992.

[Mil88] V. Milenkovic. Verifiable Implementations of Geometric Algorithms using Finite-Precision Arithmetic. *Artificial Intelligence*, 37:377–401, 1988.

[Mis93] Bud Mishra. *Computer Algebra.* Springer–Verlag, New York, NY 10010, 1993.

[MSW89] A.P. Morgan, A.J. Sommese, and L.T. Watson. Finding all isolated solutions to polynomial systems using hompack. *ACM Transactions on Mathematical Software*, 15:93–122, 1989.

[NU] S. Näher and C. Uhrig. The LEDA user manual. MPI, Germany.

[Owe91] J. C. Owen. Algebraic solution for geometry from dimensional constraints. In J. Rossignac and J. Turner, editors, *Symp. on Solid Modeling Foundations and CAD/CAM Applications*, pages 397–407. ACM Press, 1991.

[Ped91] P. Pedersen. Counting Real Zeros. Thesis. Courant Institute, New York University, 1991.

[Pri91] D. Priest. Algorithms for Arbitrary Precision Floating Point Arithmetic. In *Tenth Symposium on Computer Arithmetic*, pages 132–143, 1991. Los Alamitos, CA.

[Pri92] D. Priest. On Properties of Floting Point Arithmetic : Numerical Stability and the Cost of Accurate Computations. Ph.D. thesis, Department of Mathematics, University of California at Berkeley, Berkeley, CA, 1992.

[PRS92] P. Pedersen, M.-F. Roy, and A. Szpirglas. Counting Real Zeros in the Multivariate Case. In *Proc. MEGA-92*, 1992. appearing in 'Computational Algebraic Geometry', edited by F. Eyssette, and A. Galligo, Birkhauser.

[Reg95] A. Rege. A Complete and Practical Algorithm for Geometric Theorem Proving. In *ACM Symposium on Computational Geometry*, pages 277–286, 1995. Vancouver, B.C.

[Ren89a] J. Renegar. A Faster PSPACE Algorithm for Deciding the Existential Theory of the Reals. In *IEEE Symposium on Foundations of Computer Science*, 1989.

[Ren89b] 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.

[Ren92] J. Renegar. On the computational complexity of the first-order theory of the reals, parts I, II, III. *Journal of Symbolic Computation*, 13(3):255–352, 1992.

[Rit32] J.F. Ritt. *Differential Equations from the Algebraic Standpoint.* Number 14 in AMS Colloquium publications. American Mathematical Society, New York, 1932.

[Rit50] J.F. Ritt. *Differential Algebra.* Number 33 in AMS Colloquium publications. American Mathematical Society, New York, 1950.

[Sha74] I.R. Shafarevich. *Basic Algebra Geometry.* Springer Verlag, 1974.

[She96] J.R. Shewchuk. Robust Adaptive Floating-Point Geometric Predicates. In *ACM Symposium on Computationl Geometry*, 1996. To appear.

[Ste65] D. Stewart. A platform with six degrees of freedom. *Proc. Inst. Mech. Eng.*, 180:371–386, 1965.

[Str72] V. Strassen. Berechnung und Programm I. *Acta Informatica*, 1:320–335, 1972. In German.

[Stu91] Bernd Sturmfels. Sparse Elimination Theory. In D. Eisenbud and L. Robbiano, editors, *Proc. Computat. Algebraic Geom. and Commut. Algebra*, Cortona, Italy, June 1991. Cambridge Univ. Press.

[Tar48] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, Berkeley, 1948.

[Tay83] K.B. Taylor. Three Circles with Collinear Centers. *American Mathematical Monthly*, 90:486–487, 1983.

[vdW50] B.L. van der Waerden. *Modern Algebra*, volume 2. Ungar Publishing Co., New York, 3rd edition, 1950.

[vzG87] J. von zur Gathen. Feasible Arithmetic Computations : Valiant's Hypothesis. *JSC*, 4:137–172, 1987.

[Wan93] D. Wang. An Elimination Method for Polynomial Systems. *Journal of Symbolic Computation*, 16:83–114, 1993.

[Wan95] D. Wang. Elimination Procedures for Mechanical Theorem Proving in Geometry. *Annals of Mathematics and Artificial Intelligence*, 13:1–24, 1995.

[Wu78] W. Wu. On the Decision Problem and Mechanization of Theorem Proving in Elementary Geometry. *Sci. Sinica*, 21:150–172, 1978.

[Wu84] W. Wu. Some Recent Advances in Mechanical Theorem Proving of Geometries. In W.W. Bledsoe and D.W. Loveland, editors, *Theorem Proving : After 25 Years.* American Mathematical Society, 1984.

[Wu86] W. Wu. Basic Principles of Mechanical Theorem Proving in Elementary Geometries. *Journal of Automated Reasoning*, 2:221–252, 1986.

[Zip93] R. Zippel. *Effective Polynomial Computation.* Kluwer Academic publishers, Massachusetts, 1993.