EXACT POLYNOMIAL SYSTEM SOLVING
FOR
ROBUST GEOMETRIC COMPUTATION

A Dissertation

by

KOJI OUCHI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2006

Major Subject: Computer Science

EXACT POLYNOMIAL SYSTEM SOLVING FOR ROBUST GEOMETRIC COMPUTATION

A Dissertation

by

KOJI OUCHI

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2006

Major Subject: Computer Science

ABSTRACT

Exact Polynomial System Solving for

Robust Geometric Computation. (December 2006)

Koji Ouchi, B.S., University of Tokyo;

M.S., New York University

Co–Chairs of Advisory Committee: Dr. Donald Friesen
Dr. John Keyser

I describe an exact method for computing roots of a system of multivariate polynomials with rational coefficients, called the rational univariate reduction. This method enables performance of exact algebraic computation of coordinates of the roots of polynomials. In computational geometry, curves, surfaces and points are described as polynomials and their intersections. Thus, exact computation of the roots of polynomials allows the development and implementation of robust geometric algorithms. I describe applications in robust geometric modeling. In particular, I show a new method, called numerical perturbation scheme, that can be used successfully to detect and handle degenerate configurations appearing in boundary evaluation problems. I develop a derandomized version of the algorithm for computing the rational univariate reduction for a square system of multivariate polynomials and a new algorithm for a non-square system. I show how to perform exact computation over algebraic points obtained by the rational univariate reduction. I give a formal description of numerical perturbation scheme and its implementation.

To My best friend, Ana Erendira Flores Mendoza

ACKNOWLEDGMENTS

First of all, I thank all the people who assisted in my pursuit of this research.

I thank Prof. John Keyser, the co-chair of the dissertation committee. Most of the geometric work in this research was supervised by him. He made many contributions to the robustness issues of geometric computation, and my goal was to attack some of the problems he has had in his mind. I really appreciate him because he keeps his door always open and allows me many opportunities to discuss various topics in my research.

I thank Prof. Donald Friesen, the co-chair of the dissertation committee. He has been coaching me for years. He directs me whenever I am uncertain about how to carry my research. Without his help, I would not have been able to choose the research topic of this dissertation.

I thank Prof. J. Maurice Rojas, a member of the dissertation committee. Most of the algebraic work in this research is based on his earlier work. I still remember that I was so excited when I first read his paper about polynomial system solving.

I thank Prof. Jianer Chen, a member of the dissertation committee. Besides his advice as a researcher and teacher in theoretical computer science, he keeps encouraging me both in research and in my personal life.

Although I do not list their names here, I again thank all the people who helped me pursue my research. Those people include the faculty and staff of the school and friends.

This work was funded, in part, by NSF/DARPA CARGO award DMS-0138446 and NSF ITR Award CCR-0220047. The Bradley Fighting Vehicle was provided courtesy of the Army Research Lab.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Geometric computation suffers from robustness problems [100] [51]. A robustness problem refers to the tendency of well-designed algorithms to fail in practice due to invalid assumptions.

For example, many program designers assume that any real number arithmetic operation can be done exactly in constant time. This is invalid. In fact, most real numbers cannot be represented exactly on existing computers, and real numbers represented in fixed-precision arithmetic usually include round-off errors. Furthermore, exact arithmetic operations cannot be done in constant time.

Another example is an assumption that every geometric object is in general position, meaning that small changes in the input will not change the topological nature of the output. The situation where this assumption is violated is called a degeneracy. The general position assumption is often invalid in practice. There are many degeneracies occurring in the real world. Say a human touches the wall of a building. Then, the surface of the human intersects with the wall tangentially. This is a degeneracy, but a very normal situation in our life.

Robustness problems must be resolved, because naive implementations of algorithms with invalid assumptions can end up with the wrong results or, even worse, catastrophic crashes.

In order to make geometric computation robust, exact computation must be used. The use of exact computation resolves the problems due to numerical inaccuracy. The use of exact computation also helps us to detect degeneracies.

This dissertation follows the style of *IEEE Transactions on Automatic Control.*

The research described in this dissertation is motivated by the demands of the exact computations that enable geometric computation to be robust. Geometry is the study of points and curves. Curves can be expressed exactly by polynomials. A point can be specified as an intersection of curves. Thus, the problem of representing a point exactly reduces to the problem of solving a system of polynomial equations exactly.

Solving a given system of polynomial equations is one of the most fundamental problems in computer science. Most of my effort has been dedicated to the development of algorithms for solving a given system of polynomial equations exactly. There are various methods for solving a system of polynomial equations. However, most of them are designed to find approximations for solutions of a given system of polynomial equations. The problem becomes significantly more challenging when exactness is required.

I solve a system of polynomial equations using the Rational Univariate Reduction (RUR). The method uses successive application of relatively simple linear algebra subroutines. It can be implemented to produce exact representation of common roots of a given system of polynomials.

Points in the RUR can be readily adapted to exact computation. I propose a relatively simple implementation of geometric predicates that enables us to perform exact computation over algebraic points and curves.

Finally, as an application of exact geometric computation, I will deal with another major source of robustness problems, degeneracies. I will develop a method for detecting degeneracies and propose a new approach to remove degeneracies.

1.1 Geometric Computation

Computational geometry is the study of algorithms for solving geometric problems.

The objects dealt with in computational geometry are sets of points in a vector space over a field $\mathbb{K}$. We assume that a coordinate system is introduced to the space so that every point can be represented as a tuple of numbers belonging to $\mathbb{K}$, a certain type of metric is defined, and the topology induced from the metric is endowed to the space. A typical example of such a space is the real Euclidean space $\mathbb{R}^d$.

These sets of points are not necessarily finite, but they must be finitely specifiable so that they are encoded as a string of finite length in algorithms. Thus, in addition to sets of finitely many individual points, computational geometry also deals with curves, surfaces, portions of curves, portions of surfaces, and solid objects. For example, a line is an object in computational geometry because it can be specified by two distinct points. Note that a line can also be specified via a string of finite length as the zero set of a linear polynomial.

There are two major sources of robustness problems in computational geometry:

(1) numerically inaccurate geometric data appearing during computation, and

(2) topologically degenerate configurations of (input, intermediate, or output) geometric objects.

The use of exact computation resolves the robustness problems due to numerical inaccuracy.

In geometric computations, slight modification of objects in degenerate position can result in configurations that are significantly different topologically. Accumulated numerical errors may cause predicates to be evaluated incorrectly, and thus, an execution of the program goes into the wrong branch. Hence, exact computation is necessary to handle degeneracies.

In any case, in order for geometric computation to be robust, exact computation must be used.

1.2 Exact Computation for Algebraic Points and Curves

In this dissertation, we assume that all the geometric objects are algebraic or semi-algebraic sets over a field $\mathbb{K}$ so that they are exactly represented on the Turing machine. We mainly consider the case where $\mathbb{K}$ is a field $\mathbb{Q}$ of rational numbers, but most theories and algorithms can naturally be adapted to the other types of algebraic numbers such as algebraic numbers over some finite field.

An algebraic curve is implicitly described as the zero set of some polynomial with coefficients belonging to $\mathbb{K}$. Some algebraic curves may also be represented parametrically as a tuple of rational functions with coefficients belonging to $\mathbb{K}$.

An algebraic point is defined to be an intersection of algebraic curves. The coordinates of an algebraic point are algebraic numbers (over $\mathbb{K}$). An algebraic point is also specified by giving all its coordinates.

I would like to develop a technique for exact computation of algebraic points and curves. Here, by exact computation, we mean that the numeric computations associated with geometric objects are computed to enough precision that topological decisions about the objects are made correctly. This is different from exact arithmetic in the naive sense, where all the numerical data will be computed to full precision, which is usually costly and often infeasible.

An exact computation technique developed in some geometric algorithm usually applies only to the data structures appearing in the algorithm. As a result, it can be quite difficult to design robust geometric algorithms by integrating exact geometric subroutines.

For example, suppose that we determine the topology of a given real planar curve. The algorithm returns a set of monotone pieces of the curve. Often, a piece of a curve is specified by its endpoints, which are represented as a so-called “insulating box,” which is defined to be a box of arbitrary size containing one and only one point of interest. This representation of points is perfectly robust for describing the topology of the curve. However, if the determination of the topology of a curve is a part of some other process, e.g. a geometric solid modeling process, and the endpoints of the pieces of the curve will be passed to some other subroutines, then these insulating boxes might no longer be appropriate. In fact, if a point is represented by only an insulating box then the query *whether or not* a point lies on a curve might not be answerable.

In this dissertation, I propose the use of the *Rational Univariate Representation* (RUR) for algebraic points. In the RUR, an n-dimensional algebraic point x is specified as $n+1$ univariate polynomials $h, h_1, \dots, h_n$ so that

$$x = \left(h_1\left(\theta\right), \dots, h_n\left(\theta\right)\right) \tag{1.1}$$

for some root θ of h. I shall show an algorithm for computing the RUR for a given system of polynomial equations. I extend the so-called root bound approach to exact sign determination of algebraic numbers [23] [7]. Consequently, we can tell whether or not a given point in the RUR lies on a given curve.

In particular, the coordinates of real algebraic points are real algebraic numbers that are possibly irrational. A real algebraic number is specified as the *unique* root of some polynomial with rational coefficients in some interval on $\mathbb{R}$. The endpoints of this interval can be chosen to be rational numbers. Thus, in general, a real algebraic point can be represented as an n-dimensional hypercube. The corners of this hypercube can be set to have rational coordinates.

1.3 Degeneracies

I will apply the developed method for exact computation over algebraic points and curves to the degeneracy detection / removal problem in the robust boundary evaluation of solid objects. Handling degeneracies is a well-known problem in geometric computation.

Many geometric algorithms are designed and implemented under the "general position assumption." Stated roughly, this assumption says that small changes in the input will not change the nature of the output. More specifically, small changes in the input can change the numerical (or geometric) aspects of the output, but not the combinatorial (or topological) aspects. A violation of this assumption is referred to as a degeneracy.

More formally, degeneracies are defined in terms of a description of program flow. As a program is executed, branching decisions are made on the basis of various *predicates*, where the program branches depending on whether the predicate evaluates to negative, 0, or positive [78]. Situations that lead to a predicate being evaluated to 0 are considered degenerate. An infinitesimally tiny perturbation in the data could change the evaluation of the predicate to be either positive or negative, and thus change program flow.

As a simple example, consider a collision-detection test on two spheres touching at a single point. A degeneracy will be encountered since a small change in the position of a sphere will make the spheres either interpenetrate or separate.

Degenerate data is common in many real-world geometric applications. Sometimes these problems are unintentional, for example due to round-off error in computation, low sampling frequency, or poor training of a designer. Other times the problems may be intentional, for example a designer placing two spheres in contact.

If programs correctly account for all predicates, i.e. handle every case where a predicate evaluates to 0 consistently and ensure that there is no error in the evaluation of the predicates, they are considered robust. Regardless of the source, robust geometric computation must be able to deal with degeneracies.

Our goal is to handle degeneracies cleanly so that a program on a degenerate input will not fail.

1.4 Objective

The objective of this dissertation is the following:

> The Rational Univariate Representation (RUR) effectively supports exact computation over algebraic points and curves. This enables robust geometric computation, in particular, degeneracy handling.

The objective is proved by:

(1) developing an exact representation of an algebraic point based on the Rational Univariate Representation (RUR),

(2) developing methods to support exact computation over algebraic numbers, points and curves,

(3) applying these methods in order to detect degeneracies appearing in boundary evaluation of solid objects, and

(4) developing an exact numeral perturbation scheme for handling degeneracies appearing in boundary evaluation of solid objects.

1.5 Results

The primary result of this research is a comprehensive description and implementation of algorithms for exact representation of algebraic points and exact manipulation of algebraic points and curves. The developed technique is applied to degeneracy detection / removal problems appearing in the boundary evaluation process of geometric modeling.

The subproblems explored in this dissertation are as follows:

The Rational Univariate Reduction (RUR) is used for computing an exact representation for algebraic points. An algorithm for computing the RUR for the zero set of a given system of multivariate polynomials is described. The algorithm refines the ones presented in [81] [83] [60]. The algorithm is derandomized in a sense that random choices are made only if the probability that making a successful choice is 1 and the wrong choices are detectable. As a consequence of derandomization, a variation of the algorithm for computing real roots of a system becomes much simpler. The subalgorithm for computing the RUR for the zero set of an overconstrained system is new.

The existing root-bound approach to sign determination of real algebraic numbers [57] [7] is extended to sign determination of the real and imaginary parts of complex algebraic numbers. Together with the RUR, an algorithm for exact manipulation of algebraic points and curves is established.

The developed exact computation technique for algebraic points and curves is applied to the degeneracy detection problem appearing in boundary evaluation of solid objects. Degeneracy detection is done by checking for topologically irregular interaction between objects. Because the RUR can be computed even for degenerate intersections, it is appropriate to use the RUR for this application.

An exact numerical perturbation scheme for removing degeneracies is described. It is shown that, comparing to the symbolic perturbation scheme, an exact numerical perturbation scheme potentially works better in terms of arithmetic complexity.

These algorithms are implemented. The implementation is built on top of ESOLID [58] and MAPC [59]. MAPC is a library that manipulates exact computation for two-dimensional real algebraic points and curves. MAPC may not be able to answer some queries such as whether or not a point lies on a curve. ESOLID is a robust geometric modeling system. ESOLID performs exact boundary evaluation on curved solids. ESOLID assumes that all the solids are in general position. The exactly implemented RUR is the library which computes the exact RUR for the zero set of a system of polynomials with rational coefficients exactly. Together with the root bound approach to sign determination of algebraic numbers, exact primitive geometric predicates are implemented, and, I can add routines for detecting / handling degeneracies. See Figure 1.

The implementation is optimized for relatively small n. The implementation is exact; all the rational coefficients of the univariate polynomials forming the RUR will be computed to full precision. I show some experimental results. Tested systems include examples of a degenerate system and an overdetermined system as well as some problems picked from real world industry.

1.6 Overview of Chapters

The organization of the rest of this dissertation is as follows:

Chapter II describes the background materials relevant to this research.

Chapter III describes algorithms for computing the Rational Univariate Reduction (RUR) of a given system of polynomials.

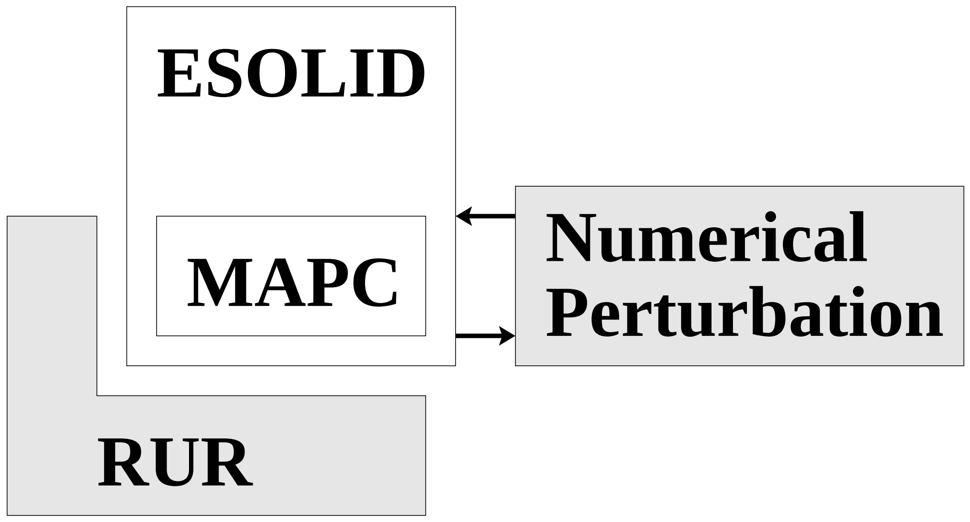


Fig. 1. Libraries implemented. ESOLID is the existing library that performs exact boundary evaluation of given solid objects. MAPC is the portion of ESOLID that provides exact manipulation of algebraic points and curves. The exactly implemented RUR computes the exact RUR for a given system of multivariate polynomials. I also implement the library that performs numerical perturbation over input solids to ESOLID.

Chapter IV describes algorithms for exact computation over algebraic numbers, points and curves and the way to apply those algorithms in order to detect degeneracies appearing in boundary evaluation of solid objects.

Chapter V describes a method for removing degeneracies.

Chapter VI describes the implementation of the algorithms described in this dissertation. Performance on examples is also presented.

Chapter VII concludes the dissertation and discusses future directions of research related to this work.

CHAPTER II

BACKGROUND

This chapter provides the background material relevant to this research. In Section 2.1, definitions and preliminary results for this research are reviewed. In Section 2.2, the previous work relevant to this research is listed.

2.1 Definitions and Preliminary Results

In this section, definitions and preliminary results for this research are reviewed.

In Section 2.1.1, the Rational Univariate Reduction (RUR) for the zero set of a system of multivariate polynomials is introduced.

In Section 2.1.2, the root bound approach to sign determination of real algebraic numbers is explained.

In Section 2.1.3, degeneracies appearing in geometric computation are described and several methods for removing them are discussed. Furthermore, boundary evaluation of solid objects and degeneracies appearing in it are described. Also, the library ESOLID that performs boundary evaluation of solid objects [58] is introduced.

2.1.1 Rational Univariate Reduction

Let $\mathbb{K}$ be a field. Write $\overline{\mathbb{K}}$ for the algebraic closure of $\mathbb{K}$ and $\mathbb{K}^*$ for $\mathbb{K} \setminus \{0\}$.

Consider a *square* system of n polynomials $f_1, \ldots, f_n$ in n variables with coefficients in $\mathbb{K}$. It is known [81] [86] [3] that there exists a finite set Z' which contains all the isolated common roots of the input system (in $\overline{\mathbb{K}}^n$) such that

$$Z' = \left\{ \left(h_1(\theta), \ldots, h_n(\theta)\right) \in \overline{\mathbb{K}}^n \mid \theta \in \overline{\mathbb{K}} \text{ with } h(\theta) = 0 \right\}. \tag{2.1}$$

That is, the i-th coordinate of every point in Z' is represented as some univariate polynomial h_i with coefficients in $\mathbb{K}$ evaluated at a root θ ($\in \overline{\mathbb{K}}$) of some other univariate polynomial h with coefficients in $\mathbb{K}$. This reduction of a set of multivariate polynomials to the set of uni-variate polynomials is called the *Rational Univariate Reduction (RUR)* and this representation of the zero set of the system is called the *Rational Univariate Representation (RUR)*.

Write M for the cardinality of the finite set of Z'. Generally, the quantity M is the number of the common roots of the input system. More precisely, if the input system is zero-dimensional, i.e., the input system has only finitely many common roots, and all the roots are toric, i.e., all the roots are in $(\overline{\mathbb{K}}^*)^n$ then Z' is the zero set of the input system and M matches the number of distinct roots of the input system.

The RUR can be derived from the toric perturbation [81], which is a generalization of the "toric" u-resultant. In this section, the preliminary facts about toric resultants (Section 2.1.1.1) and toric perturbations (Section 2.1.1.2) are described.

2.1.1.1 Toric Resultants

Let f be a polynomial in n variables $X_1, \ldots, X_n$ with coefficients in $\mathbb{K}$. Define the *support* of f to be the finite set A of exponents of all the monomials appearing in f with non-zero coefficients. Thus, A is some non-empty finite set of integer points in $\mathbb{R}^n$, and

$$f = \sum_{a \in A} c_a X^a, \qquad c_a \in \mathbb{K}^*$$

where $X^a = X_1^{a_1} \cdots X_n^{a_n}$ for $a = (a_1, \ldots, a_n)$.

Fix $n+1$ non-empty sets $A_0, A_1, \ldots, A_n$ of integer points in $\mathbb{R}^n$. A system of $n+1$ polynomials $f_0, f_1, \ldots, f_n$ in n variables $X_1, \ldots, X_n$ with supports $A_0, A_1, \ldots, A_n$ is specified via coefficient vectors $\boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_n$ where

$$\boldsymbol{c}_i = (c_{ia} \in \mathbb{K}^* \mid a \in A_i) \quad \text{such that} \quad f_i = \sum_{a \in A_i} c_{ia} X^a.$$

For $i = 0, 1, \ldots, n$, write MV_{-i} for the mixed-volume of the convex hulls of $A_0, A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$ [94] [17] [95]. Recall that all these mixed volumes are non-negative, i.e., $\mathrm{MV}_{-i} \geq 0$ for $i = 0, 1, \ldots, n$ [17]. Assume at least one of these mixed volumes $\mathrm{MV}_{-0}, \mathrm{MV}_{-1}, \ldots, \mathrm{MV}_{-n}$ is strictly positive, i.e., $\sum_{i=0}^{n} \mathrm{MV}_{-i} > 0$. Then, there exists a unique (up to sign) irreducible polynomial

$$\mathrm{TRes}_{A_0, A_1, \ldots, A_n}(\boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_n) \quad \in \quad \mathbb{Z}[\boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_n],$$

called the *toric resultant* or the *sparse resultant* for the system which has the following property:

$$\text{the system } (f_0, f_1, \ldots, f_n) \text{ has a common root in } (\overline{\mathbb{K}}^*)^n$$
$$\Longrightarrow \mathrm{TRes}_{A_0, A_1, \ldots, A_n}(\boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_n) = 0.$$

The toric resultant is also written as $\mathrm{TRes}(f_0, f_1, \ldots, f_n)$.

Several algorithms for computing the toric resultant for a given system of $n+1$ polynomials in n variables with supports $A_0, A_1, \ldots, A_n$ have been proposed [32] [14]. These algorithms construct a square matrix N, called the *toric resultant matrix* or the *Newton matrix*, whose determinant is some non-trivial multiple of the toric resultant. The non-zero entries of every row of N are the coefficients $\boldsymbol{c}_i$ of some input polynomial f_i. It follows that $\det N$ is a homogeneous polynomial in each coefficient vector $\boldsymbol{c}_i$ [32] [14], and thus, the total degree of $\det N$ with respect to each coefficient vector $\boldsymbol{c}_i$, $\deg_{\boldsymbol{c}_i}(\det N)$, is well-defined. These quantities $\deg_{\boldsymbol{c}_i}(\det N)$ are bounded in terms of the mixed-volumes. More precisely, it is known [32] [14] that

$$\deg_{\boldsymbol{c}_0}(\det N) = \mathrm{MV}_{-0}, \tag{2.2}$$

$$\deg_{\boldsymbol{c}_i}(\det N) \geq \mathrm{MV}_{-i}, \qquad i = 1, \ldots, n. \tag{2.3}$$

Note that the equality (2.2) always holds, while, in (2.3), the equalities hold only when $\det N$ is the toric resultant without any extraneous factor [76].

2.1.1.2 Toric Perturbations

Consider a *square* system of n polynomials $f_1, \ldots, f_n \in \mathbb{K}[X_1, \ldots, X_n]$ with supports $A_1, \ldots, A_n$. Assume the mixed volume MV_{-0} of the convex hulls of $A_1, \ldots, A_n$ is strictly positive, i.e., $\mathrm{MV}_{-0} > 0$.

Let $A_0 = \{\boldsymbol{o}, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\}$ where $\boldsymbol{o}$ is the origin and $\boldsymbol{b}_i$ is the i-th standard basis vector in $\mathbb{R}^n$. Also, let $f_0 = u_0 + u_1 X_1 + \cdots + u_n X_n$ where $\boldsymbol{u} = (u_0, u_1, \ldots, u_n)$ is a vector of parameters. Choose n polynomials $f_1^*, \ldots, f_n^* \in \mathbb{K}[X_1, \ldots, X_n]$ with supports contained in $A_1, \ldots, A_n$, that have only finitely many common roots in $(\overline{\mathbb{K}}^*)^n$. Define the *toric Generalized Characteristic Polynomial* $\mathrm{TGCP}(s, \boldsymbol{u})$ for the system $(f_1, \ldots, f_n)$ to be the toric resultant for the perturbed system $(f_0, f_1 - sf_1^*, \ldots, f_n - sf_n^*)$:

$$\mathrm{TGCP}(s, \boldsymbol{u}) = \mathrm{TRes}(f_0, f_1 - sf_1^*, \ldots, f_n - sf_n^*) \quad \in \quad \mathbb{K}[s][\boldsymbol{u}].$$

Also, define a *toric perturbation* $\mathrm{TPert}(\boldsymbol{u})$ for the system $(f_1, \ldots, f_n)$ to be the non-zero coefficient of the lowest degree term in $\mathrm{TGCP}(s, \boldsymbol{u})$ regarded as a polynomial in variable s.

Theorem 2.1. Rojas ([81] Main Theorem 2.4)

$\mathrm{TPert}(\boldsymbol{u})$ *is well-defined, i.e., for a given square system of polynomials* $f_1, \ldots, f_n$ *in* n *variables with coefficients in* $\mathbb{K}$, *making a suitable choice of polynomials*

$f_1^*, \ldots, f_n^*$, $\mathrm{TGCP}(s, \boldsymbol{u})$ *(regarded as a polynomial in s) always has a non-zero coefficient.* $\mathrm{TPert}(\boldsymbol{u})$ *is a homogeneous polynomial in parameters $u_0, u_1, \ldots, u_n$ with coefficients in $\mathbb{K}$ which has the following properties:*

(1) If $(\zeta_1, \ldots, \zeta_n) \in (\overline{\mathbb{K}}^*)^n$ is an isolated common root of the input system $(f_1, \ldots, f_n)$ then $u_0 + u_1\zeta_1 + \cdots + u_n\zeta_n$ is a linear factor of $\mathrm{TPert}(\boldsymbol{u})$.

(2) $\mathrm{TPert}(\boldsymbol{u})$ *completely splits into linear factors over $\overline{\mathbb{K}}$. Letting Z be the zero set of the system (which might be infinite), for every irreducible component W of $Z \cap (\overline{\mathbb{K}}^*)^n$, there is at least one factor of* $\mathrm{TPert}(\boldsymbol{u})$ *corresponding to a point $(\zeta_1, \ldots, \zeta_n) \in W$.*

Immediately from (2.2) and (2.3) together with the definitions in the above

Corollary 2.2.

$$\deg_{u_0} \mathrm{TPert}(\boldsymbol{u}) = \mathrm{MV}_{-0}, \tag{2.4}$$

$$\deg_s \mathrm{TGCP}(s, \boldsymbol{u}) = \sum_{i=1}^{n} \mathrm{MV}_{-i} \le \dim N - \mathrm{MV}_{-0}. \tag{2.5}$$

Remark 2.3. *The assumption that $\mathrm{MV}_{-0} > 0$ can be removed by paying only a reasonable amount of extra computation. In the event that $\mathrm{MV}_{-0} = 0$, we can add $\boldsymbol{O}(n)$ points to the supports $A_1, \ldots, A_n$ so that MV_{-0} is strictly positive. Those points can be chosen deterministically [82] or at random. Because of efficiency, I will use a randomized method, and will not talk about this in detail in this dissertation.*

Remark 2.4. *There is a deterministic method to choose polynomials $f_1^*, \ldots, f_n^*$ in Theorem 2.1 [82]. However, because of efficiency, we will use a randomized method, and will not talk about this in detail in this dissertation.*

2.1.2 Exact Computation over Algebraic Numbers

A complex number α is said to be *algebraic* if it is a root of a non-zero polynomial with rational coefficients. In this section, I explain a method for determining the sign of any given real algebraic number α.

2.1.2.1 Root Bounds

Let α be an algebraic number. A positive real number r such that

$$\alpha \neq 0 \quad \Leftrightarrow \quad |\alpha| \geq r. \tag{2.6}$$

is called a *root bound* or a *root separation bound* for α [69] [101]. Having a root bound r for α, the query whether or not $\alpha = 0$ is answered correctly by computing an approximation $\widetilde{\alpha}$ of α such that $|\widetilde{\alpha} - \alpha| < \frac{r}{2}$, namely,

$$\alpha = 0 \quad \Leftrightarrow \quad |\widetilde{\alpha}| < \frac{r}{2}. \tag{2.7}$$

If both α and $\widetilde{\alpha}$ are real then the sign of α is determined from the sign of $\widetilde{\alpha}$:

$$|\widetilde{\alpha}| \geq \frac{r}{2} \quad \Rightarrow \quad \alpha \cdot \widetilde{\alpha} > 0. \tag{2.8}$$

One of the root bounds was proposed by Mahler [68] and is described in detail by Mignotte [70]; define the Mahler measure (or simply the measure) $M(e)$ of a polynomial $e(T) = e_n \prod_{i=1}^{n} (T - \zeta_i) \in \mathbb{Z}[T]$ with $e_n \neq 0$ as

$$M(e) = |e_n| \prod_{i=1}^{n} \max\{1, |\zeta_i|\}. \tag{2.9}$$

Moreover, define the *degree* $\deg \alpha$ and *measure* $M(\alpha)$ of an algebraic number α to be the degree and measure of a minimal polynomial for α over $\mathbb{Z}$. Since, over $\mathbb{Z}$, a minimal polynomial for an algebraic number is uniquely determined up to a sign, the degree and measure of α are well-defined. Then

$$\frac{1}{M(\alpha)} \leq |\alpha| \leq M(\alpha). \tag{2.10}$$

In general, it is difficult to compute the measure $M(\alpha)$ of α explicitly. Instead, some computable upper bound of $M(\alpha)$ is used.

If α and β are algebraic numbers then $\alpha \pm \beta$, $\alpha\beta$ and $1/\alpha$ are all algebraic numbers. Furthermore, upper bounds for the measure of those numbers can be computed from the degree and measure of α and β:

Proposition 2.5. Mignotte [70]

Let α and β with $\alpha\beta \neq 0$ be algebraic numbers. Then

(1) $M(\alpha \pm \beta) \leq 2^{\deg\alpha \deg\beta} M(\alpha)^{\deg\beta} M(\beta)^{\deg\alpha}$.

(2) $M(\alpha\beta) \leq M(\alpha)^{\deg\beta} M(\beta)^{\deg\alpha}$.

(3) $M\left(\frac{1}{\alpha}\right) = M(\alpha)$.

(4) $M\left(\alpha^{1/k}\right) \leq M(\alpha)$ *for every positive interger k.*

Several root bounds have been proposed [7] [92] [9] [10] [66] [77]. They are tighter than the Mahler-Mignotte bound, but, in this dissertation, arguments are carried based mainly on the Mahler-Mignotte bound for flexibility.

2.1.2.2 Root Bound Approach to Exact Computation over Real Algebraic Numbers

The root-bound approach [23] [92] [8] has been used for exact sign determination of a real algebraic number of the form $e(\xi_1, \ldots, \xi_n)$ where ξ_i is a real root of a univariate polynomial with rational coefficients and e is an algebraic expression involving $\pm, *, /$

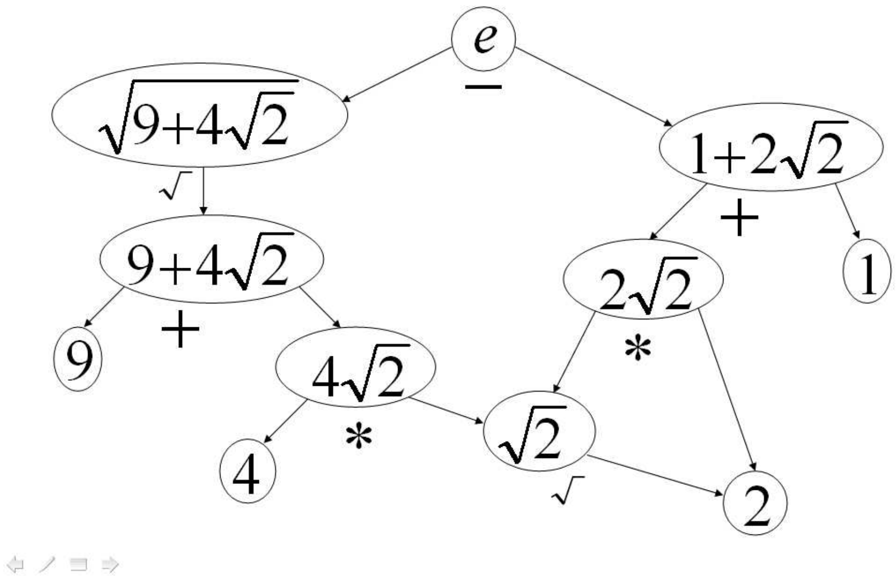


Fig. 2. The DAG for a real algebraic number $e = \sqrt{9+4\sqrt{2}} - \left(1+2\sqrt{2}\right)$.

and $\sqrt[k]{}$ in the existing libraries LEDA [69] * [10] † and CORE [57] ‡. In these libraries, an algebraic number e is represented as a Directed Acyclic Graph (DAG). Every internal node of the DAG is labeled by unary or binary operators. Every leaf of the DAG is labeled by a rational number or a real algebraic number specified as some real root of some univariate polynomial with rational coefficients. Every node f of e maintains a root bound for the real algebraic number represented as the subgraph rooted at f as well as an approximation to a certain precision.

For example, a real algebraic number $e = \sqrt{9+4\sqrt{2}} - \left(1+2\sqrt{2}\right)$ is represented

*http://www.algorithmic-solutions.com/enleda.htm

†http://www.mpi-sb.mpg.de/projects/EXACUS/leda_extension/

‡http://www.cs.nyu.edu/exact/

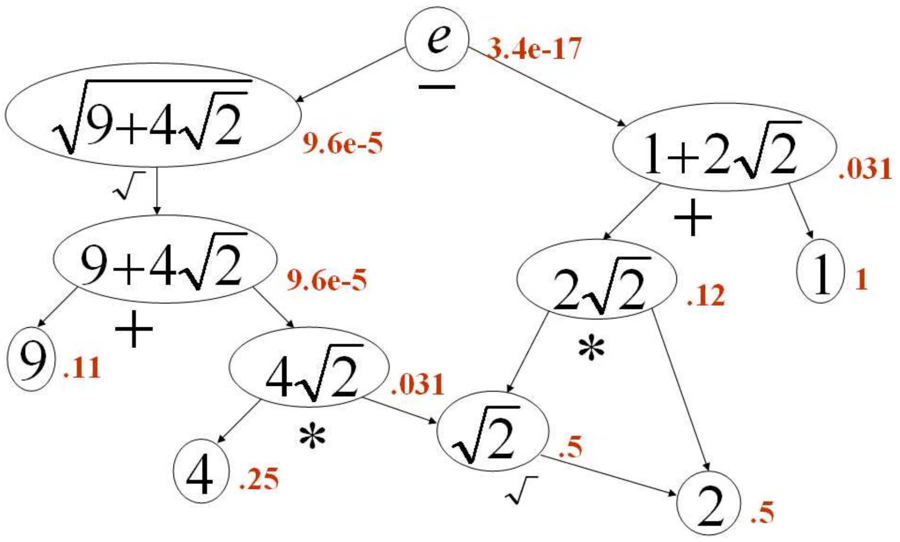


Fig. 3. A root bound for every node of the DAG is recursively computed.

as the DAG shown in Figure 2. The root of the DAG is labeled by the minus operator and has two children which represent $\sqrt{9+4\sqrt{2}}$ and $1+2\sqrt{2}$.

Like the Mahler-Mignotte bound, the root bounds implemented in the libraries LEDA and CORE are constructable, i.e., the root bound for every node of a DAG is calculated as soon as the node (or equivalently, the sub-DAG rooted at the node) is constructed. If e is a real algebraic number of the form $f \circ g$ where f and g are real algebraic numbers and $\circ$ is some operator, then a root bound for e is recursively computed from the root bounds for f and g, using the rules which are similar to, but better than the rules stated in Proposition 2.5 [9] [10] [66] [77].

For example, the root bound of every node of the DAG for the real algebraic number $e = \sqrt{9+4\sqrt{2}} - \left(1+2\sqrt{2}\right)$ is computed as follows; The root bound for

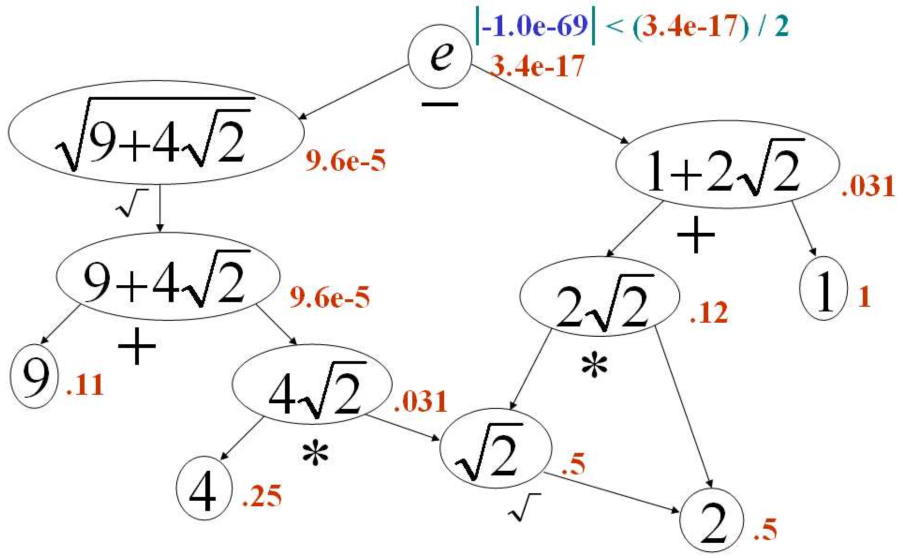


Fig. 4. The root bound approach to exact sign determination of a real algebraic number e. An approximation of e to the absolute precision smaller than the half of the root bound is computed. By (2.6), $e = 0$ is concluded.

the leaf labeled by a rational number 2 is 1/2. (By definition, the root bound for a rational number x can be $|x|$.) The same quantity can be used for the root bound for the node labeled by $\sqrt{2}$ (see the statement (4) in Proposition 2.5). In this way, the root bound for every node of the DAG is computed while the DAG is recursively constructed from its leaves to the root. See Figure 3.

The sign of e is exactly determined by computing $\widetilde{e}$ to enough precision so that (2.7) or (2.8) guarantees the sign of e.

Precision-driven computation [23] [8] is used in order to compute an approximation $\widetilde{e}$ for e to any prescribed precision p. It is a recursive process.

Suppose e is a real algebraic number of the form $f \circ g$, where f and g are real algebraic numbers and $\circ$ is some operator. Precision-driven computation is applied to e as follows; First, precisions q and r to which f and g will be approximated are calculated. Next, approximations $\widetilde{f}$ and $\widetilde{g}$ for f and g to precision q and r, respectively, are computed. Finally, $\widetilde{f} \circ \widetilde{g}$ is computed to obtain $\widetilde{e}$.

If e is a rational number, then the exact value has already been known. If e is a real algebraic number specified as some real root of some univariate polynomial with rational coefficients then an approximation to e to any prescribed precision is computed by using Sturm's method.

Hence, during precision-driven computation, the DAG is traversed twice: first, from the root to the leaves, calculating precisions, and then, from the leaves to the root, computing approximations.

In Figure 4, an approximation of e to the absolute precision smaller than the half of the root bound is computed. Since the approximation is smaller than the half of the root bound, by (2.6), $e = 0$ is concluded.

The argument in this section is summarized to the following proposition:

Proposition 2.6. *Let $e\left(X_1, \ldots, X_m\right)$ be a rational function with rational coefficients. Also, let $\xi_1, \ldots, \xi_m$ be real algebraic numbers, each of which is specified as a real root of some univariate polynomial with rational coefficients. Assume that we are able to compute an approximation for each of $\xi_1, \ldots, \xi_m$ to any prescribed precision. Then, the sign of the real algebraic number $e\left(\xi_1, \ldots, \xi_m\right)$ is determined exactly via the root bound approach.*

The statement in Proposition 2.6 will be extended later in Section 4.1.

2.1.3 Degeneracy and Perturbations

An input to an algorithm is said to be in general position when a minor perturbation of the input will not change the branching decisions made in a computation of the algorithm. Many geometric algorithms are designed under the assumption that the input is in general position. In practice, though, an input is not in general position. An input that is not in general position is said to be degenerate. Degeneracy occurs because of numerical error or because of the input itself. Unexpected degeneracies can cause executions of algorithms to fail, or worse, crash.

This section reviews degeneracies.

In Section 2.1.3.1, degeneracies appearing in geometric computation are described.

In Section 2.1.3.2, several methods for removing degeneracies, in particular, perturbation schemes and discussed.

In Section 2.1.3.3, boundary evaluation of solid objects are discussed.

In Section 2.1.3.4, degeneracies appearing in the process are described.

In Section 2.1.3.5, the library ESOLID that performs boundary evaluation of solid objects [58] is introduced.

More formal and detailed descriptions about degeneracies and perturbations will be given in Chapter V.

2.1.3.1 Degeneracies

Degeneracies are defined in terms of computations of algorithms. In a computation of an algorithm on some input, branching decisions are made based on the sign of the results of predicates. An input that leads to some predicate being evaluated to 0 is said to be *degenerate*. If a computation can evaluate all the predicates exactly and handle every case where a predicate evaluates to 0 consistently then the computation is said to be *robust*. Degeneracies are one of the major sources that make computations not robust.

More formally, degeneracy is defined as follows [90] [91].

Consider an algorithm A and let F be a function from some input space $\mathcal{I}$ to some output space $\mathcal{O}$ that is computed by A. We assume that some topology is introduced to both $\mathcal{I}$ and $\mathcal{O}$.

A computation of an algorithm for a function $F : \mathcal{I} \rightarrow \mathcal{O}$ is modeled by a ternary tree T called an *extended algebraic decision tree* [78]. In this model, $F(x)$ is computed by a traversal of the tree T from the root to one of the leaves. Each interior node v of T is associated with the predicate $f_v : \mathcal{I} \rightarrow \mathbb{R}$ and its (three) branches are labeled by -1, 0 and 1, respectively. At each internal node v of T, $s_v(x) = \operatorname{sgn}(f_v(x))$ is evaluated and the branch labeled by $s_v(x)$ is taken. Each leaf v of T is associated with the result function $g_v : \mathcal{I} \rightarrow \mathcal{O}$. When a leaf v of T is reached, $g_v(x)$ is evaluated and returned.

An input $x \in \mathcal{I}$ is said to be *degenerate* (for F) if there exists an internal node v of T such that $s_v(x) = \operatorname{sgn}(f_v(x))$ becomes 0, that is, there exists some predicate that evaluates to 0 at x.

Degeneracies can be classified into the following categories [61]:

(1) *Input* degeneracies are those that affect the output. An example is a set of points on the convex hull, three or more of which are collinear, as an input to the problem of computing the vertices of the convex hull of a given set of points.

(2) *Unpredictable* degeneracies are those due solely to arbitrary choices a program makes. Those degeneracies will not affect the output. An example is a set of points on the convex hull, three or more of which are collinear, as an input to

the problem of computing the area of the convex hull of a given set of points. Another example is a set of points, three or more of which are collinear and not all of these collinear points lie on the convex hull of the input set, as an input to the problem of computing the vertices of the convex hull of a given set of points.

Degeneracies may be produced within a program (rather than the input to the program) [61]. *Intentional* degeneracies are those that the program later relies on (i.e., it assumes that they are true at some later stage in the computation). An example is a program that produces the midpoint of two points, then relies on the fact that it is a midpoint later.

In this dissertation, I will not directly address intentional degeneracies. Thus, unless specified otherwise, all future references to degeneracies refer to input and unpredictable degeneracies.

2.1.3.2 Handling Degeneracies

There are two main approaches for handling degeneracies: special cases, and perturbation.

One approach is to treat degeneracies as special cases; whenever a program meets the situation where some predicate evaluates to zero, the special routine is called.

In order to take this special cases approach, a program designer must determine all potential degeneracies, detect them, and deal with them. These requirements are difficult or worse, impossible. Still, special cases are the most commonly used method [102].

A general approach for handling degeneracies is a perturbation scheme. The idea is to modify the input so that no predicate evaluates to zero.

More formally, a perturbation is defined as follows [90] [91]:

For every point $x \in \mathcal{I}$, a *perturbation* of x is a curve $\pi_x : [0, \infty) \rightarrow \mathcal{O}$ staring at x (i.e., $\pi_x(0) = x$).

For every function $F : \mathcal{I} \rightarrow \mathcal{O}$, a *perturbation scheme* Π assigns a perturbation π_x to every input $x \in \mathcal{I}$.

Given a function $F : \mathcal{I} \rightarrow \mathcal{O}$ and a perturbation scheme Π, define a *perturbed function* of F to be the function $\overline{F}^{\Pi} : \mathcal{I} \rightarrow \mathcal{O}$ such that

$$\overline{F}^{\Pi}(x) = \lim_{\epsilon \rightarrow 0+} F(\pi_x(\epsilon)), \qquad \forall x \in \mathcal{I}. \tag{2.11}$$

There are several arguments against the validity of perturbations.

The first argument is that we solve the perturbed problem but not the original problem. The output $\overline{F}^{\Pi}(x)$ of the perturbed computation is not actually the output $F(x)$ of the original problem. Thus, in order to obtain the solution of the original problem, the output $\overline{F}^{\Pi}(x)$ of the perturbed computation must be post-processed. Such post-processing is difficult or impossible.

If F is continuous at x then $\overline{F}^{\Pi}(x) = F(x)$. If F is continuous at all inputs then the output of the perturbed computation is exactly the same as the original. If F is not continuous at x then there is little or nothing we can say about the relation between $\overline{F}^{\Pi}(x)$ and $F(x)$.

In this sense, input degeneracies correspond to discontinuity in discontinuous functions. Both input degeneracies and unpredictable degeneracies occur in continuous functions while degeneracies for continuous functions are unpredictable.

Another argument is that the implementation of perturbation methods runs much slower. In order to ensure no predicate evaluates to zero, the sign of the result of the evaluation of the predicate must be computed exactly, and thus, some sort of exact computation is required. Also, an infinitesimal amount ϵ has to be dealt with somehow symbolically which is costly either at run-time or in design / analysis of predicates.

Despite all these arguments, a perturbation method is the one of the best known approaches for handling degeneracies consistently. A well-chosen perturbation method eliminates all potential degeneracies. Algorithms can be implemented straightforwardly. Perturbation methods have proved successful for several problems.

2.1.3.3 Boundary Evaluation

There are two major representations used by solid modeling systems: Constructive Solid Geometry (CSG) and boundary representation.

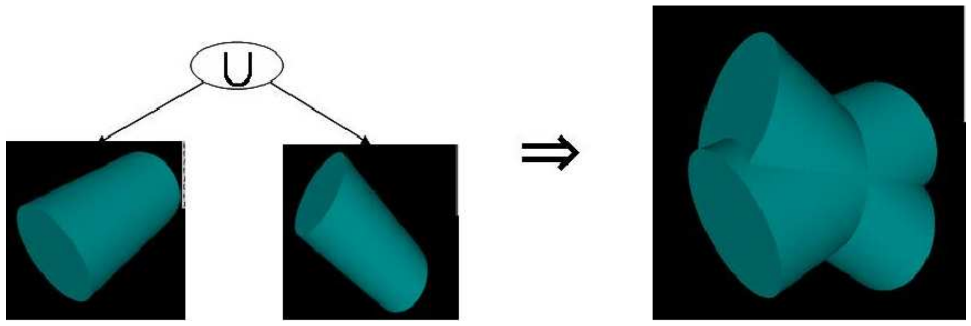


Fig. 5. A solid object in a CSG model.

A CSG model is a Boolean combination of primitive solid objects such as boxes, cylinders (or generalized cones), spheres (or generalized ellipsoids) and tori (in $\mathbb{R}^3$). A CSG representation is usually stored in a binary tree, where each internal node is associated with one Boolean operation (union, intersection, or difference) applied to its two children nodes, and each leaf is associated with a primitive. See an example in Figure 5.

A boundary representation consists of geometric and topological information about the boundary of the solid object.

Boundary evaluation refers to the process of determining the boundary representation of a solid object produced as the result of a Boolean combination of primitives given in a CSG model. Boundary evaluation is a key operation in computer aided design. Achieving accuracy and robustness with reasonable efficiency remains a challenge in boundary evaluation.

A face of a solid object is defined by a surface and boundary edges. A surface is usually represented as a parametric patch described as a tuple of polynomials or rational functions with rational coefficients. A parametric patch is a map from a 2-dimensional domain of parameters, called the patch domain, into the 3-dimensional space. The implicit form for the surface is often known, or else can be determined.

Intersections of these surfaces form the edges of the solid objects. Inside the 3-dimensional space, these curves are sometimes represented as algebraic plane curves. In the patch domain, these curves represented and defined by the intersections of two surfaces are known as either *trimming curves* if they are specified in the input, or *intersection curves* if they arise during boundary evaluation. Intersection curves that are output become trimming curves when input to the next boundary evaluation operation.

Intersections of three or more surfaces form vertices of the solid objects. Such vertices may be represented in the 3-dimensional space as the common solution to three or more trivariate equations, and in the parametric domain of the patches as the common solution of two or more bivariate equations, or a combination of these. The coordinates of these vertices are thus tuples of real algebraic numbers.

Boundary evaluation involves several stages, but the most fundamental operations are finding intersections, that is finding solutions to systems of polynomials. The accuracy, efficiency, and robustness of the entire boundary evaluation operation is usually a direct result of the accuracy, efficiency, and robustness of the computations used to find and work with these algebraic numbers. Determining the signs of algebraic expressions evaluated at algebraic numbers thus becomes a key to performing the entire computation.

2.1.3.4 Degeneracies in Boundary Evaluation

Degeneracies are a major obstacle in boundary evaluation.

Degeneracies appearing in boundary evaluation of solid objects have been enumerated in terms of the ways that surfaces, curves, and points can interact [61]. These are listed in Table I.

When solid objects are in general position, they can interact with each other in only two ways:

(1) Two surfaces can meet transversely along a set of curves.

(2) A surface and a curve can meet transversely at a set of points.

No other interaction is allowed if the solid objects are in general position. Taking the contra positive, degeneracies can occur in one of the following two situations:

1. Two objects interact that should not. An example is an interaction between a surface and a point.

2. An interaction between two objects that could interact in a non-degenerate way (two surfaces, or a surface and a curve) but the interaction is actually not transverse. An example is a curve meeting a surface tangentially at a point, instead of the curve passing through the surface at that point.

The (non-degenerate) intersection of two surfaces defines a curve. The (non-degenerate) intersection of three surfaces defines a point. Four or more surfaces do not meet, generically. Substituting two or three surfaces to a curve or a point in the above enumeration, the entries of Table I are filled.

Table I. Possible degeneracies in boundary evaluation. The order of each degenerate intersection involved is in bold: points (**0**), curves (**1**), surfaces (**2**).

	Surface	Curve	Point
Surface	**2** Surfaces overlap **1** Surfaces are tangent along a curve **0** Surfaces are tangent at a point	**1** A curve lies on a surface **0** A curve is tangent to a surface at a point	**0** A point lies on a surface
Curve		**1** Curves overlap **0** Curves intersect	**0** A point lies on a curve
Point			**0** Points coincide

Among all ten degenerate intersections listed in Table I, eight are considered as the interactions between objects that cannot interact in non-degenerate way:

(1) Two surfaces meet but not along a curve.

 (a) Surfaces overlap.

 (b) Surfaces are tangent at a point.

(2) Three or more surfaces meet along a curve but not at a point.

 (a) A curve lies on a surface.

 (b) Curves overlap.

(3) Four or more surfaces meet.

 (a) A point lies on a surface.

(b) Curves intersect.

(c) A point lies on a curve.

(d) Points coincide.

The other two degenerate intersections listed in Table I ("surfaces are tangent along a curve" and "a curve is tangent to a surface at a point") are interactions between objects that could interact in non-degenerate ways but the interactions are tangential instead of transverse.

Note that tangential intersection can occur between objects that should not interact in non-degenerate ways.

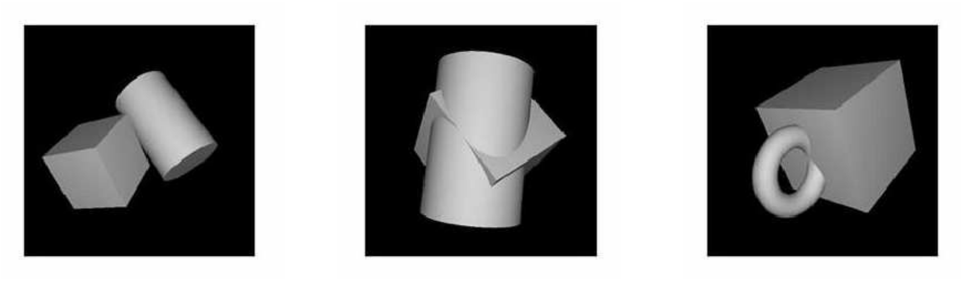


Fig. 6. Examples of degeneracies in boundary evaluation.

Some examples of degeneracies appearing in boundary evaluation are shown in Figure 6. More examples and complete descriptions are found in [61]. I will describe how to detect degeneracies appearing in boundary evaluation in Chapter IV and how to remove these degeneracies in Chapter V.

2.1.3.5 ESOLID

ESOLID is a geometric solid modeling system that performs exact boundary evaluation of a given CSG model [58]. ESOLID uses exact representations for geometric objects and exact computations in order to guarantee accuracy and eliminate robustness problems due to numerical error (e.g. roundoff error and its propagation). ESOLID performs geometric computation mainly on the 2-dimensional patch domain using MAPC, the library for exact Manipulation of Algebraic Points and Curves [59]. Though significantly less efficient than an equivalent floating-point routine, it runs at "reasonable" speed—at most 1-2 orders of magnitude slower than an inexact approach on real-world data. Unfortunately, ESOLID is designed to work only for objects in general position. Also, for efficiency, MAPC assumes curves are non-singular and ignores some singular intersections such as tangential intersections.

In MAPC, an algebraic point (whose coordinates are real algebraic numbers) is represented as a rectangle that contains one and only one intersection of a pair of algebraic plane curves. I introduce the algorithm for finding such a point. More formally, the problem is stated as follows; given a pair of algebraic plane curves $f(S,T) = 0$ and $g(S,T) = 0$ and a region $[s_1, s_2] \times [t_1, t_2]$, find rectangles in the region each of which contains one and only one intersection of those curves. The procedure of the algorithm consists of several stages.

First, compute univariate polynomials $s(S) = \mathrm{SRes}_T(f,g)$ and $t(T) = \mathrm{SRes}_S(f,g)$ where $\mathrm{SRes}_X(f,g)$ is the Sylvester resultant for polynomials f and g both regarded as univariate polynomial in variable X. The S and T coordinates of the intersections of $f = 0$ and $g = 0$ are given by the roots of s and t, respectively.

Next, isolate the roots of s within the interval $[s_1, s_2]$, and isolate the roots of t within the interval $[t_1, t_2]$ by using Sturm's method. Each of these roots are represented either by a rational number or an interval whose endpoints are rational numbers. Let s and t have m and n roots within the intervals $[s_1, s_2]$ and $[t_1, t_2]$, respectively. Form nm boxes that may contain the intersections of $f = 0$ and $g = 0$.

Then, perform a series of tests to determine which of these boxes contain the intersections of $f = 0$ and $g = 0$ and which of these do not. The assumption that curves are non-singular is used in this stage, and thus, some singular intersections such as tangential ones may not be found. (See Figure 7.)

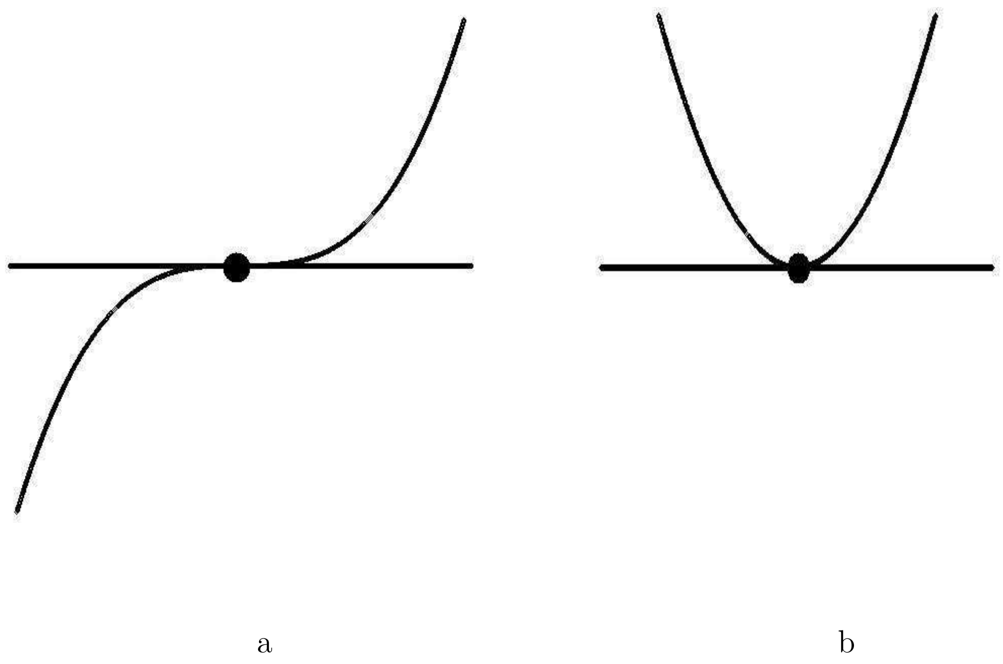


Fig. 7. Singular intersections of curves can and cannot be found by MAPC.
a. An example of a singular intersection of curves that can be found by MAPC.
b. An example of a singular intersection of curves that cannot be found by MAPC.

Note that the rectangle representing an algebraic point can be shrunken into any size.

MAPC may not be able to answer the query whether or not an algebraic point lies on a curve. If the point DOES NOT lies on the curve then, after shrinking the rectangle finitely many times, the rectangle will become small enough so that the curve will not intersect it. On the other hand, if the point DOES lie on the curve then, however many times it is shrunken, the curve always touches the rectangle.

ESOLID assumes that every object is in general position. The algorithms implemented in ESOLID do not have portions that take care of degenerate inputs (See Theorem 5.24 in Chapter V). When an input is degenerate, ESOLID may return the incorrect result, or worse, crash.

Overcoming these limitations has been a major motivation of my research.

2.2 Previous Work

This section reviews some other previous work relevant to this research.

2.2.1 Robust Geometric Computation

The need for robustness in geometric algorithms has been advocated for decades. Robustness issues have been an active area of work in computational geometry for a long time, now. Much of the need for robustness was highlighted by [52] [23] [100].

Exact computation [50] [57] [8] as a method for eliminating numerical errors has been addressed. Much of the earliest work focused on polyhedrons, with only limited work on curved objects [96] [102] [4] [36].

The research presented in this dissertation builds on top of earlier work on exact geometric solid modeling [59] [58].

More general work supporting exact computation includes the development of the LEDA [69] and CORE [57] libraries, along with the more general algorithms supported by CGAL [34]. The Effective Computational Geometry for Curves and Surfaces (ECG) project is currently developing a large set of exact geometric algorithms [8] [9] [21] [74].

2.2.2 Rational Univariate Reduction

The RUR for a system of polynomials has been known for more than a century. The RUR was first seen in [64], but the RUR has been used in computer algebra only recently [42] [81] [86] [3].

If an input system is of dimension zero then the RUR can be computed via the "multiplication table method" [86] [44] [3]. An extension of this method finds all the isolated real roots as well as at least one point from every *real* positive-dimensional component [2] [22]. A standard implementation of the method requires reduction of the input polynomials into some normal form via the Gröbner basis. This implementation has the disadvantage that the worst case time complexity is exponential time. The Gröbner basis is discontinuous with respect to changes in the coefficients of the input polynomials [73].

A Gröbner-free algorithm to compute the RUR for a zero-dimensional system has been proposed [40]. Recent work even handles systems with multiple roots [65]. The complexity analysis of this algorithm is considered in [55].

2.2.2.1 Toric Resultants

The *toric resultant* (or the *sparse resultant*) for a system of $n + 1$ polynomials with indeterminate coefficients in n variables is a polynomial with integer coefficients in these indeterminates (as variables) that vanishes iff the system has a common root on some toric variety over an algebraic closure of the field to which the coefficients of the polynomials belong [13] [94] [17] [14] [95] [15] [30]. The toric resultant is expressed as a divisor of the determinant of some square matrix, called the *toric resultant matrix* or the *Newton matrix* [13] [94] [27] [14] [95] [30] [84]. The *mixed-subdivision based algorithm* [13] [14] [30] is historically the first practical algorithm that constructs the resultant matrix, but the size of the matrix constructed is often too large. Another version of this algorithm first constructs a small matrix and incrementally constructs larger matrices until one that works is found [32]. Several efforts have been made to construct smaller resultant matrices [18] [29] [54] [72] [62] [63].

The resultant-based method for solving a system of polynomial equations fails if the zero set of the input system has some positive dimensional components. In order to solve such systems, a perturbation technique is used. The Generalized Characteristic Polynomial (GCP) by [12] can be used to express solutions to dense homogeneous square systems with degeneracies. The toric perturbation [81] [83] is defined as a particular coefficient of the toric GCP [80] [81] [83]. The toric perturbation works even if an input square system has some multiple roots at the point at infinity. A potentially more efficient perturbation technique that finds expectedly fewer monomials has been proposed by [19] [20].

The toric resultant-based method can be modified so that it finds some set containing all the affine roots of a square system [85] [67] [82] [81] [83].

The algorithm for computing the RUR of a given system of polynomials with rational coefficients described in this paper refines the versions in [82] and [60]. I give a step-by-step description together with an exception handler and a new algorithm for overdetermined systems.

2.2.3 Root Bound

The root-bound approach to exact sign determination for *real* algebraic numbers has been implemented in the libraries LEDA [11] [69] and CORE [57]. Several improvements on root-bounds have also been reported [7] [9] [10] [66] [77].

The sign of the real algebraic number (given as roots of a univariate polynomial with rational coefficients) can be determined algorithmically (e.g. Sturm's method).

2.2.4 MAPC and ESOLID

MAPC [59] is a library that manipulates exact computation for two-dimensional real algebraic points and curves. ESOLID [61] [58] is a robust geometric modeler built on top of MAPC. ESOLID is currently the only system we know of that supports exact boundary evaluation for solids with curved surfaces. For efficiency, MAPC assumes curves are non-singular and ignores some singular intersections such as tangential intersections. Thus, ESOLID works only when provided solids that are in general position. Addressing this deficiency has been a goal of my research.

An example from boundary evaluation is when intersection curves contain singularities (e.g. self-intersections, isolated point components, cusps). There are some approaches that can deal with these curves [88] [53] [43] [98] [26]. However, methods such as that in [59] fail in such situations, while the exact RUR approach is perfectly capable of finding, e.g. the intersection between two curves at a self-intersection of one of the curves.

2.2.5 Degeneracies

There have been a variety of methods previously proposed for handling with degeneracies or perturbations.

Handling with degeneracies via special-case code has been the predominant approach used. Examples can be seen in [50]. For curved objects, the issues of degeneracies become more complicated. A great deal of effort has focused just on handling the intersections of quartic surfaces, such as that of Farouki et al. [35] and Geismann et al. [39].

Perturbation schemes have arisen as a more general way of dealing with degeneracies. There are several different types of perturbations.

The earliest symbolic perturbation scheme was probably that of Edelsbrunner and Mucke [25]. Emiris and Canny varied the perturbation used to a simpler one, and applied it to a wider variety of cases [31] [33]. Yap provided an even more generalized perturbation approach [99]. Seidel provides a summary of these techniques, along with a critique of the general perturbation scheme itself [90] [91]. For solid modeling applications, Fortune describes the use of symbolic perturbation for linear solids [36].

When numerical perturbation has been applied in the past, e.g. as in Sugihara's work [96], it is usually done in the context of fixed precision computation. That is, data is perturbed point-wise across a grid made up of representable points in fixed-precision space. This significantly limits the types and amounts of perturbation that can be applied; though it works for several cases, it is not as general in application as the approach proposed in this dissertation. In contrast to these earlier methods, the use of exact computation allows for a wider range of perturbation amounts. In fact, as will be shown below, the new approach yields the full generality that could be obtained through symbolic perturbation.

Recently, a different type of perturbation approach has been proposed by Song et al. [93]. In this approach, input data (specifically, the parametric surfaces) are modified in such a way as to meet a particular constraint (specifically, intersecting on a certain curve). That is, the input is perturbed in a very specific way to ensure a specific outcome. In contrast, our method modifies the input data, but in a more random fashion, rather than insisting on achieving a particular result.

There are also several perturbation schemes that rely on fixed-precision computation [45] [37] [48] [49] [47] [79] [46]. In some of these approaches (e.g. [48]), geometric objects are fattened (e.g. points become circles) and some adjustments have to be made to the geometric algorithms. Funke et al. [38] described controlled perturbation for Delaunay triangulations. Although they use floating point arithmetic rather than exact arithmetic, the idea of controlled perturbation is very similar to a "backward stable" operation as the scheme proposed in this dissertation.

CHAPTER III

RATIONAL UNIVARIATE REDUCTION

This chapter describes new algorithms for computing the Rational Univariate Reduction (RUR) of a given system of multivariate polynomials. In Section 3.1, the algorithms are described. In Section 3.2, some examples are shown. In Section 3.3, a worst-case asymptotic arithmetic complexity of the algorithm is analyzed.

3.1 Algorithm

In this section, we will describe an algorithm for computing the RUR. After discussing the derandomized process for computing the RUR for the toric zero set (the zero set in $(\overline{\mathbb{K}}^*)^n$) of a square system (Section 3.1.1), we will discuss extending this to the RUR for the affine zero set (the zero set in $\overline{\mathbb{K}}^n$) of a non-square system (Section 3.1.2). We will also discuss handling the cases when $\mathbb{K} = \mathbb{Q}$ or $\mathbb{K} = \mathbb{R}$ and all we want to find are the real roots (Section 3.1.3).

In the rest of this chapter, we will assume that the characteristic of the field $\mathbb{K}$ is 0 or sufficiently large. An upper bound for the characteristic of $\mathbb{K}$ (when it is not zero) will be given later in Section 3.1.1.

Algorithms described in this section are as follows (Figure 8):

Algorithm **RUR_toric_square** computes the RUR for the toric zero set (the zero set in $(\overline{\mathbb{K}}^*)^n$) of a square system of polynomials $f_1, \ldots, f_n$ in n variables with rational coefficients.

Algorithm **RUR_square** computes the RUR for the affine zero set (the zero set in $\overline{\mathbb{K}}^n$) of a square system of polynomials $f_1, \ldots, f_n$ in n variables with rational coefficients. It internally calls algorithm **RUR_toric_square**.

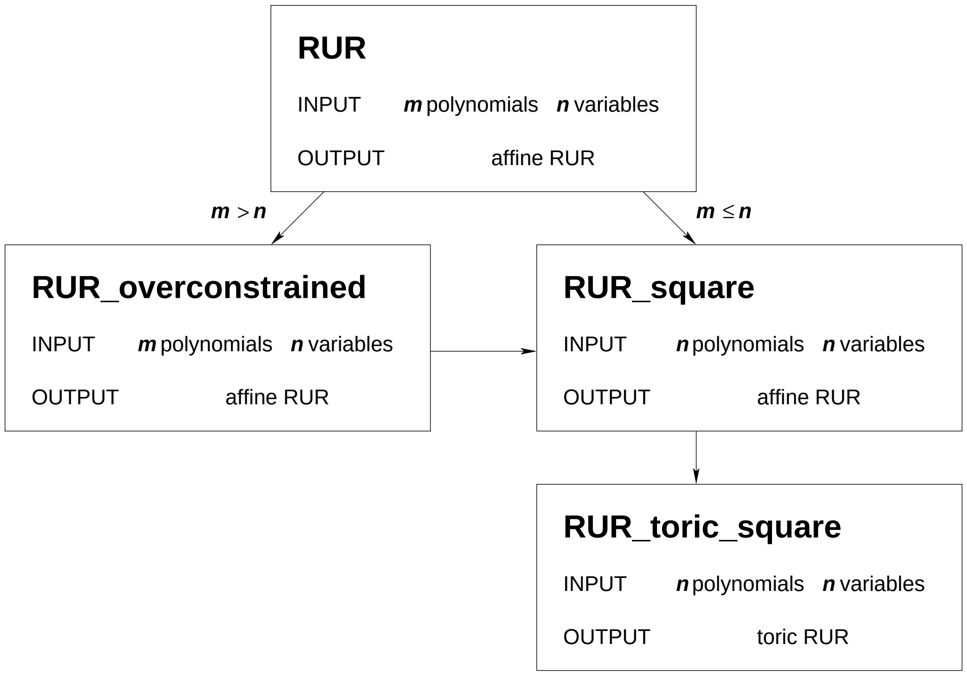


Fig. 8. Algorithms described in Section 3.1.

Algorithm **RUR_overconstrained** computes the RUR for the affine zero set of a system of polynomials $f_1, \ldots, f_m$ in n variables with rational coefficients when $m > n$. It internally calls Algorithm **RUR_square**.

Algorithm **RUR** computes the RUR for a system of polynomials $f_1, \ldots, f_m$ in n variables with rational coefficients. It internally calls Algorithm **RUR_overconstrained** or Algorithm **RUR_square**.

3.1.1 Toric RUR for Square Systems

Consider a *square* system of n polynomials $f_1, \ldots, f_n$ in n variables with coefficients in $\mathbb{K}$. Let Z be the zero set of the system. Assume that the mixed volume MV_{-0} of the convex hulls of supports $A_1, \ldots, A_n$ of $f_1, \ldots, f_n$ is strictly positive. It follows from Theorem 2.1 that, with a suitable choice of polynomials $f_1^*, \ldots, f_n^*$, there exists a finite subset Z' of $Z \cap (\overline{\mathbb{K}}^*)^n$ such that

- Z' contains all the isolated common roots of the input system in $(\overline{\mathbb{K}}^*)^n$ as well as at least one point from every irreducible component of $Z \cap (\overline{\mathbb{K}}^*)^n$, and
- the univariate polynomial $h(T)$ in the RUR for Z' is derived from $\mathrm{TPert}(\boldsymbol{u}) = \mathrm{TPert}(u_0, u_1, \ldots, u_n)$ by setting u_0 to a variable T and specializing parameters $u_1, \ldots, u_n$ to some appropriate values in $\overline{\mathbb{K}}$.

The other univariate polynomials $h_1, \ldots, h_n$ in the RUR are also derived from toric perturbations. Recall that M is defined to be the cardinality of Z' (Section 2.1.1). Note that if the input system has only finitely many roots then $Z' = Z \cap (\overline{\mathbb{K}}^*)^n$, and thus, the input system has M distinct common roots (in $(\overline{\mathbb{K}}^*)^n$). Auxiliary polynomials $f_1^*, \ldots, f_n^*$ can be chosen deterministically [81], though the process is costly. In practice, polynomials with random coefficients are used. The probability that random polynomials work suitably is 1 and unsuitable choices are detectable. (See steps **11** and **12** below in Section 3.1.1.1.) Thus, we will describe a version of the algorithm in which choices of auxiliary polynomials remain randomized. The conditions for an appropriate specialization of parameters $u_1, \ldots, u_n$ will be clarified later. We will see that parameters $u_1, \ldots, u_n$ are specialized appropriately to some integer values.

We give an algorithm for computing the RUR for Z' (Section 3.1.1.1). Step-by-step details are given immediately afterward (Section 3.1.1.2).

The algorithm computes the RUR only when the mixed volume MV_{-0} for the convex hulls of supports $A_1, \ldots, A_n$ of polynomials $f_1, \ldots, f_n$ is strictly positive. If MV_{-0} turns out to be 0 then the algorithm adds some points to $A_1, \ldots, A_n$ so that MV_{-0} becomes strictly positive. See Remark 2.3 and steps from **3** through **6** below (Section 3.1.1.1).

3.1.1.1 Toric RUR for Square Systems: Algorithm

This section gives the pseudo code of Algorithm **RUR_toric_square**.

Algorithm RUR_toric_square

Input: $f_1, \ldots, f_n \in \mathbb{K}[X_1, \ldots, X_n]$ with supports $A_1, \ldots, A_n \subseteq \mathbb{Z}^n$.

Output: $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for some finite set Z' which contains all the isolated common roots of the input system in $(\overline{\mathbb{K}}^*)^n$ as well as at least one point from every irreducible component of the zero set of the input system in $(\overline{\mathbb{K}}^*)^n$.

```
1: $A_0 \leftarrow \{\boldsymbol{o}, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_n\}$ where $\boldsymbol{o}$ is the origin and $\boldsymbol{b}_i$ is the $i$-th standard basis vector
   in $\mathbb{R}^n$
2: compute $\mathrm{MV}_{-0}$ of the convex hulls of $A_1, \ldots, A_n$
3: if $\mathrm{MV}_{-0} = 0$ then
4:     for $i := 1, \ldots, n$ do:
5:         choose a point $a \in \mathbb{Z}^n$ with random coordinates and $A_i \leftarrow$
           $\{a\} \cup A_i$
6:     go to 2
7: compute $\mathrm{MV}_{-1}, \ldots, \mathrm{MV}_{-n}$ and construct the toric resultant matrix $N$ for a system
   of polynomials with supports $A_0, A_1, \ldots, A_n$
8: $(u_0, u_1, u_2, \ldots, u_n) \leftarrow (1, 0, 0, \ldots, 0)$
9: choose polynomials $f_1^*, \ldots, f_n^*$ in variables $X_1, \ldots, X_n$ with random coefficients in
   $\mathbb{K}$ and supports contained in $A_1, \ldots, A_n$
```

```
10: set $d$ to be a non-negative integer such that $s^d$ is the lowest degree term with
    non-zero coefficient in $\mathrm{TGCP}(s, \boldsymbol{u})$ (regarded as a polynomial in $s$)
11: if $\mathrm{TGCP}(s, \boldsymbol{u})$ is identically zero then
12:     go to 9
13: $u \leftarrow 0$ and $\overline{M} \leftarrow 0$
14: $(u_1, u_2, \ldots, u_n) \leftarrow (1, u, \ldots, u^{n-1})$
15: compute $p(T) := \mathrm{TPert}(T, u_1, \ldots, u_n)$ where $\mathrm{TPert}(\boldsymbol{u})$ is the coefficient of the
    term $s^d$ in $\mathrm{TGCP}(s, \boldsymbol{u})$ (regarded as a polynomial in $s$)
16: compute the square-free part $q(T)$ of $p(T)$
17: if $\deg_T p(T) = \deg_T q(T)$ then    /* if $p(T)$ is square-free then */
18:     $M \leftarrow \deg_T p(T)$
19: else    /* if $p(T)$ is not square-free then */
20:     if $u \leq n\binom{\mathrm{MV}_{-0}}{2}$ then
21:         if $\overline{M} < \deg_T q(T)$ then
22:             $\overline{M} \leftarrow \deg_T q(T)$
23:         increment $u$ and go to 14
24:     else    /* if $u > n\binom{\mathrm{MV}_{-0}}{2}$ then */
25:         $M \leftarrow \overline{M}$
26: $u \leftarrow 0$
```

```
27: $(u_1, u_2, \ldots, u_n) \leftarrow (1, u, \ldots, u^{n-1})$
28: compute $p(T) := \mathrm{TPert}(T, u_1, \ldots, u_n)$
29: compute the square-free part $q(T)$ of $p(T)$
30: if $\deg_T q(T) < M$ then
31:     increment $u$ and go to **27**
32: for $i := 1, \ldots, n$ do:
33:     compute $p_i^{\pm}(t) := \mathrm{TPert}(t, u_1, \ldots, u_{i-1}, u_i \pm 1, u_{i+1}, \ldots, u_n)$ *
34:     compute the square-free part $q_i^{\pm}(t)$ of $p_i^{\pm}(t)$
35:     if $\deg_t q_i^{-}(t) < M$ or $\deg_t q_i^{+}(t) < M$ then
36:         increment $u$ and go to **27**
37: $h(T) \leftarrow q(T)$
38: for $i := 1, \ldots, n$ do:
39:     compute the greatest common divisor $g(t)$ of $q_i^{-}(t)$ and $q_i^{+}(2T - t)$
        (regarded as a polynomial in $t$)
40:     compute $h_i(T) := -T - \frac{g_0(T)}{g_1(T)} \bmod h(T)$ where $g_0(T)$ and $g_1(T)$ are
        the constant term and the linear coefficient of $g(t)$, respectively, i.e.,
        $g(t) = g_0(T) + g_1(T)\,t$
```

The algorithm differs from the prior version [81] in the following:

*Step **33** does not make sense when the characteristic of $\mathbb{K}$ is 2, but we assume that the characteristic of $\mathbb{K}$ is 0 or sufficiently large.

- The loop from step **3** through step **6** handles the input system when MV_{-0} is 0.
- Step **10** uses a new criteria (see Proposition 3.7) to determine a non-negative integer d such that s^d is the lowest degree term with non-zero coefficient in $\mathrm{TGCP}\,(s, \boldsymbol{u})$.
- The specialization of $\boldsymbol{u}$ is derandomized:

  The loop from step **13** through step **25** finds an appropriate specialization of parameters $u_1, \ldots, u_n$ for computing the cardinality M of Z' (counting without multiplicity).

  Steps **26** through **36** find an appropriate specialization of parameters $u_1, \ldots, u_n$ for computing all the univariate polynomials h and $h_1, \ldots, h_n$ in the RUR.

3.1.1.2 Toric RUR for Square Systems: Description

This section gives step-by-step details of Algorithm **RUR_toric_square**.

Step **2** computes MV_{-0}.

The loop from step **3** through step **6** adds points to $A_1, \ldots, A_n$ so that MV_{-0} is assured to be strictly positive. Those points are chosen randomly or deterministically. For efficiency, we use a randomized method.

Step **7** computes $\mathrm{MV}_{-1}, \ldots, \mathrm{MV}_{-n}$ and constructs the toric resultant matrix N for a system of $n+1$ polynomials with supports $A_0, A_1, \ldots, A_n$. Entries of matrix N remain undetermined, and will be specialized to some values later at steps **10**, **15**, **29** and **33**. Step **7** needs to be performed once and only once for any square system of n polynomials in n variables with given supports $A_1, \ldots, A_n$.

The loop from step **8** through step **12** determines a non-negative integer d such that $\mathrm{TPert}\,(\boldsymbol{u})$ is the coefficient of the term s^d in $\mathrm{TGCP}\,(s, \boldsymbol{u})$.

We will show, in Proposition 3.7, that if s^d is the lowest degree term with non-zero coefficient in $\mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0)$ then $\mathrm{TPert}\,(\boldsymbol{u})$ is the coefficient of the term s^d in $\mathrm{TGCP}\,(s, \boldsymbol{u})$. Thus, parameters $u_0, u_1, \ldots, u_n$ are specialized to $1, 0, \ldots, 0$ at step **8** and fixed throughout the loop.

Step **9** chooses auxiliary polynomials $f_1^*, \ldots, f_n^*$. While randomly chosen f_i^*'s could turn out not to be suitable, this is almost never the case, and is detected at step **11** if they are. As mentioned earlier, there is a deterministic method for choosing suitable auxiliary polynomials [81], but the method is costly, and suffers from expression swell. Thus, we stick with the randomized method.

Step **10** determines a non-negative integer d such that s^d is the lowest degree term with non-zero coefficient in $\mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0)$. From (2.5), $\deg_s \mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0) = \sum_{i=1}^{n} \mathrm{MV}_{-i}$, the right hand side of which can easily be calculated from the quantities computed at step **7**, and thus, all the coefficients of $\mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0)$ can be computed via interpolation. More precisely, choose $\sum_{i=1}^{n} \mathrm{MV}_{-i} + 1$ many values for s, specialize the entries of N with the coefficients of f_0 (which is the constant polynomial 1 here), $f_1 - sf_1^*, \ldots, f_n - sf_n^*$, evaluate $\mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0)$, and interpolate $\mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0)$ from these values.

Recall that the determinant of the resultant matrix N is some non-trivial multiple of the toric resultant. In order to calculate the explicit value of $\mathrm{TGCP}\,(s, \boldsymbol{u})$ at fixed s and $\boldsymbol{u}$, the contribution of the extraneous factor must be eliminated. An elimination of the extraneous factor is done by another level of interpolation through the values of the determinant of N whose entries are specialized in several ways. One such method is called the division method [14], which applies to both cases — when the characteristic of $\mathbb{K}$ is 0 or positive.

If the characteristic of $\mathbb{K}$ is 0 then d can be determined by scanning the coefficients of some non-trivial multiple of $\mathrm{TGCP}\,(s, 1, 0, 0, \ldots, 0)$ instead of the coefficients of $\mathrm{TGCP}\,(s,1,0,0,\ldots,0)$ without any extraneous factor. From (2.5), $\deg_s \mathrm{TGCP}\,(s,1,0,0,\ldots,0)$ is known to be bounded from above by $\dim N - \mathrm{MV}_{-0}$, and thus, some non-trivial multiple of $\mathrm{TGCP}\,(s,1,0,0,\ldots,0)$ can be computed via interpolation from the values of $\det N$. More precisely, choose $\dim N - \mathrm{MV}_{-0} + 1$ many values for s, specialize the entries of N with the coefficients of f_0 (which is the constant polynomial 1 here), $f_1 - sf_1^*, \ldots, f_n - sf_n^*$, evaluate $\det N$, and interpolate.

Step **11** checks whether or not the randomly chosen f_i^*'s at step **9** are suitable.

The loop from step **13** through step **25** finds an appropriate specialization of parameters $u_1, \ldots, u_n$ for computing the cardinality M of Z' (counting without multiplicity).

Step **14** specializes parameters $u_1, \ldots, u_n$ to some integer values.

Step **15** computes $p\,(T) := \mathrm{TPert}\,(T, u_1, \ldots, u_n)$ introducing a variable T. From (2.4), $\deg_T \mathrm{TPert}\,(T, u_1, \ldots, u_n) = \mathrm{MV}_{-0}$, the right hand side of which has been computed at step **7**, and thus, $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$ can be computed via interpolation: choose $\mathrm{MV}_{-0} + 1$ many values for u_0, evaluate the coefficient $\mathrm{TPert}\,(\boldsymbol{u}) = \mathrm{TPert}\,(u_0, u_1, \ldots, u_n)$ of the term s^d in $\mathrm{TGCP}\,(s, \boldsymbol{u})$, and interpolate $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$ from these values. From (2.5), $\deg_s \mathrm{TGCP}\,(s, \boldsymbol{u})$ is known, and thus, the coefficient of the term s^d in $\mathrm{TGCP}\,(s, \boldsymbol{u})$ can be computed via another level of interpolation: choose $\sum_{i=1}^{n} \mathrm{MV}_{-i} + 1$ many values for s, specialize the entries of N with the coefficients of $f_0, f_1 - sf_1^*, \ldots, f_n - sf_n^*$, calculate the explicit values of $\mathrm{TGCP}\,(s, \boldsymbol{u})$, and interpolate $\mathrm{TGCP}\,(s, \boldsymbol{u})$ from these values. Note that calculation of the explicit values of $\mathrm{TGCP}\,(s, \boldsymbol{u})$ requires an elimination of the extraneous factor from $\det N$.

The above paragraph holds when the characteristic of $\mathbb{K}$ is 0 or positive. If the characteristic of $\mathbb{K}$ is 0 then $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$ can be computed via interpolation from the values of the coefficient of the term s^d in some non-trivial multiple of

TGCP $(s, \boldsymbol{u})$ instead of the values of the coefficient of the term s^d in TGCP $(s, \boldsymbol{u})$ without any extraneous factor. This is possible because, by (2.4), the contributions of the extraneous factor are independent of $\boldsymbol{u}$, in particular u_0, and will be canceled out during interpolation. More precisely, choose $\mathrm{MV}_{-0} + 1$ many values for u_0, evaluate the coefficient of the term s^d in some non-trivial multiple of TGCP $(s, \boldsymbol{u})$, and interpolate TPert $(T, u_1, \ldots, u_n)$ from these values. The coefficient of the term s^d in some non-trivial multiple of TGCP $(s, \boldsymbol{u})$ is computed via another level of interpolation: choose $\dim N - \mathrm{MV}_{-0} + 1$ many values for s, specialize the entries of N with the coefficients of $f_0, f_1 - sf_1^*, \ldots, f_n - sf_n^*$, evaluate $\det N$, and interpolate.

Step **16** computes the square-free part $q(T)$ of $p(T) :=$ TPert $(T, u_1, \ldots, u_n)$ by dividing $p(T)$ by the greatest common divisor of $p(T)$ and its derivative $p'(T)$ found using the Euclidean algorithm.

Steps **17** through **25** find M. If, for some specialization of parameters $u_1, \ldots, u_n$, $p(T) :=$ TPert $(T, u_1, \ldots, u_n)$ is square-free then $M = \deg_T p(T)$ and the computation immediately exits from the loop. On the contrary, if $p(T)$ remains non-square-free for all $n\binom{\mathrm{MV}_{-0}}{2} + 1$ many specializations of parameters $u_1, \ldots, u_n$ then M is set to be the maximum degree of the square-free part $q(T)$ of $p(T)$. The correctness of this part of the algorithm will be shown later.

Steps **26** through **36** find an appropriate specialization of parameters $u_1, \ldots, u_n$ for computing all the univariate polynomials h and $h_1, \ldots, h_n$ in the RUR.

Step **27** specializes parameters $u_1, \ldots, u_n$ to some integer values.

Steps **28** and **29** are the same as steps **15** and **16**, respectively. In the loop from step **13** through step **25**, we have already tried several specializations of parameters $u_1, \ldots, u_n$, and have found at least one appropriate specialization (possibly more if step **25** has been reached) for computing M and possibly some inappropriate ones. For those specializations of parameters $u_1, \ldots, u_n$ that have been tried in the previous loop, steps **28** and **29** do not need to be performed. If a specialization of parameters $u_1, \ldots, u_n$ has been found inappropriate then the computation immediately goes back to step **27**. On the other hand, if a specialization of parameters $u_1, \ldots, u_n$ has been found appropriate for computing M, which means that it is appropriate for computing h, then the computation jumps to step **32** and checks whether or not it is also appropriate for computing $h_1, \ldots, h_n$.

Similar to step **15**, at step **33**, $p_i^{\pm}(t)$ are computed via interpolations, and similar to step **16**, at step **34**, $q_i^{\pm}(t) := p_i^{\pm}(t) \,/\, \gcd\left(p_i^{\pm}(t), \left(p_i^{\pm}\right)'(t)\right)$.

Step **37** determines the univariate polynomial h in the RUR, and the loop from step **38** through step **40** determines the univariate polynomials $h_1, \ldots, h_n$ in the RUR.

Step **39** computes the greatest common divisor $g(t)$ of $q_i^-(t)$ and $q_i^+(2T - t)$ (as a polynomial in t). We will show that whenever parameters $u_1, \ldots, u_n$ are specialized appropriately, $g(t)$ is linear. In this case, the ratio of the coefficients of $g(t)$ matches the ratio of the coefficients of the *first subresultant* for $q_i^-(t)$ and $q_i^+(2T - t)$ [12] [41]; if $g(t) = g_0 + g_1 t$ and $r_0 + r_1 t$ is the first subresultant for $q_i^-(t)$ and $q_i^+(2T - t)$, then $\frac{g_0}{g_1} = \frac{r_1}{r_0}$. Note that r_0 and r_1 are actually polynomials in T. By definition, r_0 and r_1 are the determinants of some submatrices of the Sylvester matrix for $q_i^-(t)$ and $q_i^+(2T - t)$, and the degree of r_0 and r_1 in T are both known to be $M(M-1)$. Thus, they can be computed via interpolation.

Step **40** determines the univariate polynomials $h_1(T), \ldots, h_n(T)$ in the RUR. The computation at this step involves the (extended) Euclidean algorithm and arithmetic operations over the ring of univariate polynomials with coefficients in $\mathbb{K}$.

3.1.1.3 Toric RUR for Square Systems: Proof for Correctness

We give the proof for the correctness of Algorithm **RUR_toric_square**.

The following proposition completes the proof of the correctness of the loop from step **8** though step **12** of the algorithm.

Proposition 3.7. *If s^d is the lowest degree term with non-zero coefficient in* $\mathrm{TGCP}\left(s,1,0,0,\ldots,0\right)$ *then* $\mathrm{TPert}\left(\boldsymbol{u}\right)$ *is the coefficient of the term s^d in* $\mathrm{TGCP}\left(s,\boldsymbol{u}\right)$.

Proof. Suppose otherwise. Then, there exists a non-negative integer $e < d$ such that $\mathrm{TPert}\left(\boldsymbol{u}\right)$ is the non-zero coefficient of the term s^e in $\mathrm{TGCP}\left(s,\boldsymbol{u}\right)$. (We do not have to consider the case $e > d$. If $e > d$ then, for any $\boldsymbol{u}$, the coefficient of the term s^d in $\mathrm{TGCP}\left(s,\boldsymbol{u}\right)$ is identically zero, and in particular, the coefficient of the term s^d in $\mathrm{TGCP}\left(s,1,0,\ldots,0\right)$ is zero.) By Theorem 2.1, $\mathrm{TPert}\left(\boldsymbol{u}\right)$ splits into (not necessarily distinct) linear factors:

$$\mathrm{TPert}\left(\boldsymbol{u}\right) = c\prod_{j=1}^{M}\left(u_0 + \sum_{l=1}^{n} u_l \zeta_l^{(j)}\right)^{\mu^{(j)}} \tag{3.1}$$

where c is a non-zero constant belonging to $\mathbb{K}$ and $\mu^{(1)},\ldots,\mu^{(M)}$ are positive integers. Thus, $\mathrm{TPert}\left(1,0,0,\ldots,0\right) \neq 0$. This is a contradiction, since the coefficient $\mathrm{TPert}\left(1,0,0,\ldots,0\right)$ of the term s^e in $\mathrm{TGCP}\left(s,1,0,\ldots,0\right)$ is not identically zero but $e < d$. □

In the rest of this section, we describe the conditions for an appropriate specialization of parameters $u_1,\ldots,u_n$, the existence of appropriate specializations and the remaining proofs of the correctness of the algorithm.

We use the famous concept of separating polynomials and their properties. For more details, see textbooks like [3].

A polynomial f in n variables with coefficients in a field $\mathbb{L}$ is said to *separate* two distinct points α and β in $\mathbb{L}^n$ if $f\left(\alpha\right) \neq f\left(\beta\right)$. A polynomial f in n variables with coefficients in $\mathbb{L}$ is said to *separate* a finite subset A of $\mathbb{L}^n$ if f separates every pair of two distinct points in A.

Lemma 3.8. *Let $\mathbb{L}$ be a field of characteristic* 0 *or a finite field of characteristic at least $n+1$. Furthermore, let α and β be two distinct points in $\mathbb{L}^{n+1}$. Then, at least one of the linear polynomials in $n+1$ variables $X_0, X_1, \ldots, X_n$ with integer coefficients*

$$v_u = X_0 + uX_1 + \cdots + u^n X_n, \qquad u = 0, 1, \ldots, n, \tag{3.2}$$

separates α and β.

We describe the conditions for an appropriate specialization of parameters $u_1, \ldots, u_n$.

Recall that Z' is some finite subset of the zero set of the input system of polynomials with coefficients in $\mathbb{K}$ such that the univariate polynomial $h(T)$ in the RUR for Z' is derived from TPert $(\boldsymbol{u})$ by setting u_0 to a variable T and specializing parameters $u_1, \ldots, u_n$ to some appropriate values in $\overline{\mathbb{K}}$. Let M be the cardinality of Z' so that

$$Z' = \left\{ \left(\zeta_1^{(1)}, \ldots, \zeta_n^{(1)}\right), \ldots, \left(\zeta_1^{(M)}, \ldots, \zeta_n^{(M)}\right) \right\}. \tag{3.3}$$

By Bernstein's theorem [5]

$$M \leq \mathrm{MV}_{-0} = \deg_T \mathrm{TPert}\left(T, u_1, \ldots, u_n\right). \tag{3.4}$$

We say that parameters $u_1, \ldots, u_n$ are specialized appropriately if the linear polynomials in n variables

$$u_1 X_1 + \cdots + u_n X_n \tag{3.5}$$

and

$$\begin{gathered} u_1 X_1 + \cdots + u_{i-1} X_{i-1} + \left(u_i \pm 1\right) X_i + u_{i+1} X_{i+1} + \cdots + u_n X_n, \\ i = 1, \ldots, n \end{gathered} \tag{3.6}$$

separate Z', or equivalently, the following conditions are satisfied:

$$j \neq k \Rightarrow \sum_{l=1}^{n} u_l \zeta_l^{(j)} \neq \sum_{l=1}^{n} u_l \zeta_l^{(k)} \tag{3.7}$$

and

$$j \neq k \Rightarrow \sum_{l=1}^{n} u_l \zeta_l^{(j)} \pm \zeta_i^{(j)} \neq \sum_{l=1}^{n} u_l \zeta_l^{(k)} \pm \zeta_i^{(k)}, \qquad i = 1, \ldots, n. \tag{3.8}$$

We show that there always exists an appropriate specialization of parameters $u_1, \ldots, u_n$.

Proposition 3.9. *At least one of the n-tuples in*

$$\left\{ (u_1, \ldots, u_n) = (u, \ldots, u^n) \;\middle|\; u = 0, 1, \ldots, n \binom{\mathrm{MV}_{-0}}{2} \right\}$$

satisfies (3.7).

Proof. Let u be a non-negative integer and $v_u = \sum_{i=0}^{n} u^i X_i$ as in (3.2). Define

$$Y' = \left\{ \left(0, \zeta_1^{(j)}, \ldots, \zeta_n^{(j)}\right) \;\middle|\; j = 1, \ldots, M \right\}. \tag{3.9}$$

Condition (3.7) holds if there exists a non-negative integer u such that v_u separates $\binom{M}{2}$ pairs of distinct points in Y'. Now, apply Lemma 3.8 and (3.4). $\square$

Proposition 3.10. *At least one of the n-tuples in*

$$\left\{ (u_1, \ldots, u_n) = (u, \ldots, u^n) \;\middle|\; u = 0, 1, \ldots, (2n+1)\, n \binom{M}{2} \right\}$$

satisfies (3.7) and (3.8).

Proof. Let u be a non-negative integer and $v_u = \sum_{i=0}^{n} u^i X_i$ as in (3.2). Define Y' as in (3.9) and define

$$Y'^{\pm}_i = \left\{ \left(\pm\zeta_i^{(j)}, \zeta_1^{(j)}, \zeta_2^{(j)}, \ldots, \zeta_n^{(j)}\right) \;\middle|\; j = 1, \ldots, M \right\}, \qquad i = 1, \ldots, n.$$

Note that the cardinality of each ${Y'}_i^{\pm}$ is M. Conditions (3.7) and (3.8) hold if there exists a non-negative integer u such that v_u separates $\binom{M}{2}$ pairs of distinct points in Y', $\binom{M}{2}$ pairs of distinct points in ${Y'}_i^{+}$ for $i = 1, \dots, n$, and $\binom{M}{2}$ pairs of distinct points in ${Y'}_i^{-}$ for $i = 1, \dots, n$, in total, $(2n+1)\binom{M}{2}$ pairs of distinct points. Now, apply Lemma 3.8. $\square$

Remark 3.11. *All the parameters $u_1, \dots, u_n$ are specialized appropriately to some integers.* [†]

We complete the proof of the correctness of the algorithm.

First, we show that condition (3.7) holds iff, at step **18** or step **25**, M is correctly set.

By Theorem 2.1, TPert $(T, u_1, \dots, u_n)$ splits into (not necessarily distinct) linear factors:

$$\mathrm{TPert}\,(T, u_1, \dots, u_n) = c \prod_{j=1}^{M} \left(T + \sum_{l=1}^{n} u_l \zeta_l^{(j)} \right)^{\mu^{(j)}} \tag{3.10}$$

where $c \in \mathbb{K}^*$ and $\mu^{(1)}, \dots, \mu^{(M)}$ are some positive integers.

Suppose TPert $(u_1, \dots, u_n)$ is not square-free. The possible situations are

(1) $\mu^{(j)} \geq 2$ for some j, and/or

(2) parameters $u_1, \dots, u_n$ are specialized inappropriately so that condition (3.7) does not hold.

If $\mu^{(1)} = \cdots = \mu^{(M)} = 1$ then, by Proposition 3.9, within finitely many attempts, a specialization of parameters $u_1, \dots, u_n$ satisfying condition (3.7) will be found eventually to compute TPert $(T, u_1, \dots, u_n)$ which becomes square-free. Thus, the computation reaches step **25** only if $\mu^{(j)} \geq 2$ for some j. In this case, $n\binom{\mathrm{MV}_{-0}}{2} + 1$ many

[†] *Since we assume that the characteristic of $\mathbb{K}$ is 0 or sufficiently large, $\mathbb{K}$ always contains integers.*

specializations of parameters $u_1, \ldots, u_n$ are tried. Again, by Proposition 3.9, at least one of them must satisfy condition (3.7). Whenever a specialization of parameters $u_1, \ldots, u_n$ satisfying condition (3.7) is used, $\deg_T q\,(T)$ at step **21**, which precisely matches the number of distinct linear factors of $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$, is maximized, since, with any inappropriate specialization of parameters $u_1, \ldots, u_n$ breaching condition (3.7), $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$ must have fewer distinct linear factors.

Next, we show that conditions (3.8) hold iff, at step **30**, $\deg_T q\,(T) = M$ and simultaneously, at step **35**, $\deg_t q_i^{\pm}\,(t) = M$ for $i = 1, \ldots, n$.

By (3.10), $\mathrm{TPert}\,(t, u_1, \ldots, u_{i-1}, u_i \pm 1, u_{i+1}, \ldots, u_n)$ splits into (not necessarily distinct) linear factors:

$$\begin{aligned} &\mathrm{TPert}\,(t, u_1, \ldots, u_{i-1}, u_i \pm 1, u_{i+1}, \ldots, u_n) \\ &\qquad = c \prod_{j=1}^{M} \left(t + \sum_{l=1}^{n} u_l \zeta_l^{(j)} \pm \zeta_i^{(j)} \right)^{\mu^{(j)}}, \quad i = 1, \ldots, n. \end{aligned} \tag{3.11}$$

If conditions (3.8) hold then every $q_i^{\pm}\,(t)$ computed at step **34** is a product of M distinct linear factors:

$$q_i^{\pm}\,(t) = c \prod_{j=1}^{M} \left(t + \sum_{l=1}^{n} u_l \zeta_l^{(j)} \pm \zeta_i^{(j)} \right), \qquad i = 1, \ldots, n. \tag{3.12}$$

Thus, $\deg_t q_i^{\pm}\,(t) = M$ for $i = 1, \ldots, n$.

On the other hand, if not all conditions (3.8) hold then, for some i, $q_i^{-}\,(t)$ or $q_i^{+}\,(t)$ has strictly fewer than M distinct linear factors. Thus, $\deg_t q_i^{-}\,(t) < M$ or $\deg_t q_i^{+}\,(t) < M$ for some i.

Furthermore, by Proposition 3.10, an appropriate specialization of parameters $u_1, \ldots, u_n$ satisfying conditions (3.7) and (3.8) will be found eventually, which results in $\deg_T q\,(T) = \deg_t q_i^{\pm}\,(t) = M$ for $i = 1, \ldots, n$. This shows that the computation eventually exits from the loop from step **26** through step **36**.

Finally, we show that the greatest common divisor of $q_i^-(t)$ and $q_i^+(2T-t)$ is linear.

It follows that, provided condition (3.7) holds, at step **37**, $h(T)$ is a product of M distinct linear factors:

$$h(T) = c\prod_{j=1}^{M}\left(T + \sum_{l=1}^{n} u_l\zeta_l^{(j)}\right), \tag{3.13}$$

and thus, $h(T)$ has precisely M distinct roots $\theta^{(1)}, \dots, \theta^{(M)}$ in $\overline{\mathbb{K}}$ where

$$\theta^{(j)} = -\sum_{l=1}^{n} u_l\zeta_l^{(j)}, \qquad j = 1, \dots, M. \tag{3.14}$$

Substituting (3.13) into (3.12), we see that, provided (3.7) and (3.8) hold,

$$\begin{aligned} q_i^-(t) &= c\prod_{j=1}^{M}\left(t - \left(\theta^{(j)} + \zeta_i^{(j)}\right)\right), \\ q_i^+(2T-t) &= c\prod_{j=1}^{M}\left(2T - t - \left(\theta^{(j)} - \zeta_i^{(j)}\right)\right) \end{aligned} \qquad i = 1, \dots, n. \tag{3.15}$$

Then

$$q_i^-(t) = q_i^+\left(2\theta^{(j)} - t\right) = 0 \Leftrightarrow t = \theta^{(j)} + \zeta_i^{(j)}, \qquad \begin{array}{l} i = 1, \dots, n, \\ j = 1, \dots, M. \end{array} \tag{3.16}$$

In fact, any root of $q_i^-(t)$ is of the form $\theta^{(k)} + \zeta_i^{(k)}$ for some k. But, if $k \neq j$ then $q_i^+\left(2\theta^{(j)} - t\right)$ is not square-free.

It now follows from (3.16) that, for every $h_i(T)$ computed at step **40**, $h_i\left(\theta^{(j)}\right) = \zeta_i^{(j)}$ for $j = 1, \dots, M$.

3.1.2 RUR

Recall that Algorithm **RUR_toric_square** computes the RUR for the toric zero set (the zero set in $(\overline{\mathbb{K}}^*)^n$) of a square system. We would like to develop an algorithm for computing the RUR for the affine zero set (the zero set in $\overline{\mathbb{K}}^n$) for a system which is not necessarily square.

3.1.2.1 RUR for Square Systems

Consider a square system of polynomials $f_1, \ldots, f_n$ in n variables with coefficients in $\mathbb{K}$. Let Z be the zero set of the system. Write $A_1, \ldots, A_n$ for the supports of $f_1, \ldots, f_n$, respectively. It is known [67] [81] [83] that if, in Algorithm **RUR_toric_square**, instead of $A_1, \ldots, A_n$, the sets $\{\boldsymbol{o}\} \cup A_1, \ldots, \{\boldsymbol{o}\} \cup A_n$ are used then the algorithm computes the RUR for some finite set $\overline{Z}'$ that contains all the isolated common roots of the input system (in $\overline{\mathbb{K}}^n$) as well as at least one point from every irreducible component of $Z (\subseteq \overline{\mathbb{K}}^n)$. Note that $\overline{Z}'$ may contain some extraneous points which do not belong to Z.

Algorithm RUR_square

Input: $f_1, \ldots, f_n \in \mathbb{K}[X_1, \ldots, X_n]$.

Output: $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for some finite set $\overline{Z}'$ which contains all the isolated common roots of the input system as well as at least one point from every irreducible component of the zero set of the input system.

```
1: for $i := 1, \ldots, n$ do:
2:     set $A_i$ to be the support of $f_i$
3:     $A_i \leftarrow \{\boldsymbol{o}\} \cup A_i$ where $\boldsymbol{o}$ is the origin in $\mathbb{R}^n$
4: call Algorithm **RUR_toric_square** on the input $f_1, \ldots, f_n$ and $A_1, \ldots, A_n$ to compute $h, h_1, \ldots, h_n \in \mathbb{K}[T]$
```

3.1.2.2 RUR for Overdetermined Systems

Throughout this section, let $\mathbb{L}$ be an algebraically closed field containing $\mathbb{K}$.

Let $f_1, \ldots, f_m$ be polynomials in n variables with coefficients in $\mathbb{L}$. Write $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$ for the zero set of polynomials $f_1, \ldots, f_m$ in $\mathbb{L}^n$:

$$\mathbf{Z}^{(n)}(f_1, \ldots, f_m) =$$
$$\{(z_1, \ldots, z_n) \in \mathbb{L}^n \mid f_1(z_1, \ldots, z_n) = \cdots = f_m(z_1, \ldots, z_n) = 0\}.$$

We will use the following facts. The proofs are found in textbooks of algebraic geometry, e.g., [16].

- Let f_1 and f_2 be polynomials in n variables with coefficients in $\mathbb{L}$. Then

  $$\mathbf{Z}^{(n)}(f_1, f_2) = \mathbf{Z}^{(n)}(f_1) \cap \mathbf{Z}^{(n)}(f_2). \tag{3.17}$$

  and

  $$\mathbf{Z}^{(n)}(f_1 \cdot f_2) = \mathbf{Z}^{(n)}(f_1) \cup \mathbf{Z}^{(n)}(f_2). \tag{3.18}$$

- An algebraic set Z in $\mathbb{L}^n$ is written as a finite union of irreducible algebraic sets in $\mathbb{L}^n$:

  $$Z = V_1 \cup \cdots \cup V_l \tag{3.19}$$

  where $V_1, \ldots, V_l$ are irreducible algebraic sets in $\mathbb{L}^n$. A decomposition (3.19) of Z is said to be minimal if $V_i \not\subseteq V_j$ for $i \neq j$. A minimal decomposition of Z always exists and is unique up to the order in which $V_1, \ldots, V_l$ are written.

- Let Z_1 and Z_2 be irreducible algebraic sets in $\mathbb{L}^m$ and $\mathbb{L}^n$, respectively. Then, $Z_1 \times Z_2$ is an irreducible algebraic set in $\mathbb{L}^{m+n}$.

In the rest of this paragraph, assume $m > n$.

Let $f_1, \ldots, f_m$ be polynomials in n variables $X_1, \ldots, X_n$ with coefficients in $\mathbb{L}$. Introducing $m-n$ variables $X_{n+1}, \ldots, X_m$, $f_1, \ldots, f_m$ are seen as polynomials in m variables $X_1, \ldots, X_n, X_{n+1}, \ldots, X_m$ with coefficients in $\mathbb{L}$. Thus, $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ is well-defined and

$$\begin{aligned}&\mathbf{Z}^{(m)}(f_1, \ldots, f_m) = \\ &\left\{ (z_1, \ldots, z_n, z_{n+1}, \ldots, z_m) \in \mathbb{L}^m \;\middle|\; \begin{array}{l} (z_1, \ldots, z_n) \in \mathbf{Z}^{(n)}(f_1, \ldots, f_m), \\ (z_{n+1}, \ldots, z_m) \in \mathbb{L}^{m-n} \end{array} \right\}.\end{aligned}$$

Let Π denote the projection from $\mathbb{L}^m$ onto $\mathbb{L}^n$ which ignores the last $m-n$ coordinates:

$$\Pi : \mathbb{L}^m \ni (z_1, \ldots, z_n, z_{n+1}, \ldots, z_m) \mapsto (z_1, \ldots, z_n) \in \mathbb{L}^n.$$

Proposition 3.12. *Let $f_1, \ldots, f_m$ be polynomials in n variables $X_1, \ldots, X_n$ with coefficients in $\mathbb{L}$. Furthermore, let*

$$\mathbf{Z}^{(n)}(f_1, \ldots, f_m) = V_1 \cup \cdots \cup V_l \tag{3.20}$$

be the unique minimal decomposition of $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^n$. For $k = 1, \ldots, l$, define

$$\begin{aligned} W_k &= V_k \times \mathbb{L}^{m-n} \\ &= \left\{ (z_1, \ldots, z_n, z_{n+1}, \ldots, z_m) \in \mathbb{L}^m \;\middle|\; \begin{array}{l} (z_1, \ldots, z_n) \in V_k, \\ (z_{n+1}, \ldots, z_m) \in \mathbb{L}^{m-n} \end{array} \right\}. \end{aligned}$$

Then

$$\mathbf{Z}^{(m)}(f_1, \ldots, f_m) = W_1 \cup \cdots \cup W_l \tag{3.21}$$

is the unique minimal decomposition of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^m$.

Proof. It is easy to see that the equality (3.21) holds. It remains to be shown that (3.21) is a decomposition of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ into a union of irreducible algebraic sets in $\mathbb{L}^m$ and is actually the unique minimal decomposition.

For $k = 1, \ldots, l$, a product W_k of an irreducible algebraic set V_k in $\mathbb{L}^n$ and an irreducible algebraic set $\mathbb{L}^{m-n}$ is an irreducible algebraic set in $\mathbb{L}^m$.

Because of the minimality of the decomposition (3.20) of $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$, $V_i \not\subseteq V_j$ for $i \neq j$, i.e., there exists a point $(z_1, \ldots, z_n) \in V_i \setminus V_j$. Then, for every $m-n$ tuple $(z_{n+1}, \ldots, z_m) \in \mathbb{L}^{m-n}$, $(z_1, \ldots, z_n, z_{n+1}, \ldots, z_m)$ is a point in $W_i \setminus W_j$, i.e., $W_i \not\subseteq W_j$ for $i \neq j$. Thus, (3.21) is a minimal decomposition of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^m$, and its uniqueness follows from the minimality. $\square$

Corollary 3.13. *Let $f_1, \ldots, f_m$ be polynomials in n variables $X_1, \ldots, X_n$ with coefficients in $\mathbb{L}$. Furthermore, let*

$$\mathbf{Z}^{(m)}(f_1, \ldots, f_m) = W_1 \cup \cdots \cup W_l \tag{3.22}$$

be the unique minimal decomposition of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^m$. Then

$$\mathbf{Z}^{(n)}(f_1, \ldots, f_m) = \Pi(W_1) \cup \cdots \cup \Pi(W_l) \tag{3.23}$$

is the unique minimal decomposition of $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^n$.

Proof. Let $\mathbf{Z}^{(n)}(f_1, \ldots, f_m) = V_1 \cup \cdots \cup V_{l'}$ be the unique minimal decomposition of $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^n$. By Proposition 3.12, $\mathbf{Z}^{(m)}(f_1, \ldots, f_m) = V_1 \times \mathbb{L}^{m-n} \cup \cdots \cup V_{l'} \times \mathbb{L}^{m-n}$ is the unique minimal decomposition of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\mathbb{L}^m$. Thus, $l' = l$ and, relabeling if necessary, for $k = 1, \ldots, l$, $W_k = V_k \times \mathbb{L}^{m-n}$ which implies $V_k = \Pi(W_k)$. $\square$

Consider a system of m polynomials $f_1, \ldots f_m$ in n variables $X_1, \ldots, X_n$ with coefficients in $\mathbb{K}$. Introducing $m-n$ variables $X_{n+1}, \ldots, X_m$, construct a square system of m polynomials $g_1, \ldots, g_m$ in m variables $X_1, \ldots, X_m$ with coefficients in $\mathbb{K}$:

$$\begin{aligned} & g_i(X_1, \ldots, X_m) \\ & = f_i(X_1, \ldots, X_n) \cdot (X_{n+1} - a_{n+1}) \cdot \cdots \cdot (X_m - a_m) \end{aligned} \qquad i = 1, \ldots, m, \tag{3.24}$$

where $a_{n+1} \ldots, a_m$ are some constants in $\mathbb{K}$.

Let

$$\mathbf{Z}^{(m)}(f_1, \ldots, f_m) = W_1 \cup \cdots \cup W_l \tag{3.25}$$

be the unique minimal decomposition of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ into a union of distinct irreducible algebraic sets in $\overline{\mathbb{K}}^m$.

Proposition 3.14. *Following the notations above,*

$$\begin{aligned} & \mathbf{Z}^{(m)}(g_1, \ldots, g_m) \\ & = W_1 \cup \cdots \cup W_l \cup \mathbf{Z}^{(m)}(X_{n+1} - a_{n+1}) \cup \cdots \cup \mathbf{Z}^{(m)}(X_m - a_m) \end{aligned} \tag{3.26}$$

is the unique minimal decomposition of $\mathbf{Z}^{(m)}(g_1, \ldots, g_m)$ into a union of distinct irreducible algebraic sets in $\overline{\mathbb{K}}^m$.

Proof. The equality (3.26) holds since

$$\begin{aligned}
\mathbf{Z}^{(m)}(g_1,\ldots,g_m) &= \bigcap_{i=1}^{m} \mathbf{Z}^{(m)}(g_i) \\
&= \bigcap_{i=1}^{m} \left(\mathbf{Z}^{(m)}(f_i) \cup \bigcup_{j=n+1}^{m} \mathbf{Z}^{(m)}(X_j - a_j) \right) \\
&= \bigcap_{i=1}^{m} \mathbf{Z}^{(m)}(f_i) \cup \bigcup_{j=n+1}^{m} \mathbf{Z}^{(m)}(X_j - a_j) \\
&= \mathbf{Z}^{(m)}(f_1,\ldots,f_m) \cup \bigcup_{j=n+1}^{m} \mathbf{Z}^{(m)}(X_j - a_j) \\
&= \bigcup_{k=1}^{l} W_k \cup \bigcup_{j=n+1}^{m} \mathbf{Z}^{(m)}(X_j - a_j)
\end{aligned}$$

where the first and the fourth equalities follow from (3.17), the second equality follows from (3.18) and the last equality follows from (3.25).

All the components appearing on the right hand side of (3.26) are irreducible. By assumption, $W_1, \ldots, W_l$ are all irreducible. Each $\mathbf{Z}^{(m)}(X_j - a_j)$ is the zero set of a linear polynomial $X_j - a_j$, and thus, is irreducible.

Furthermore, all the components appearing on the right hand side of (3.26) are distinct. By assumption, $W_i \not\subseteq W_j$ for $i \neq j$. Any pair of W_k and $\mathbf{Z}^{(m)}(X_j - a_j)$ are distinct because W_k contains a point whose j-th coordinate is not a_j. If $i \neq j$ then $\mathbf{Z}^{(m)}(X_i - a_i) \not\subseteq \mathbf{Z}^{(m)}(X_j - a_j)$ since the former contains a point whose j-th coordinate is not a_j. Hence, the decomposition (3.26) is minimal, and its uniqueness follows from the minimality. $\square$

Note that all the irreducible components of the unique minimal decomposition of $\mathbf{Z}^{(m)}(g_1,\ldots,g_m)$ are of positive dimension; each W_k contains a copy of $\overline{\mathbb{K}}^{m-n}$ and each $\mathbf{Z}^{(m)}(X_j - a_j)$ contains a copy of $\overline{\mathbb{K}}^{m-1}$.

Suppose Algorithm **RUR_square** will be applied to the square system of polynomials $g_1, \ldots, g_m$ in m variables with coefficients in $\mathbb{K}$ and univariate polynomials h and $h_1, \ldots, h_m$ with coefficients in $\mathbb{K}$ are returned. These polynomials form the RUR for some finite set $\overline{Y}'$ that contains at least one point from every irreducible component of $\mathbf{Z}^{(m)}(g_1, \ldots, g_m)$. By Proposition 3.14, $\overline{Y}'$ contains at least one point from each W_k: for $k = 1, \ldots, l$, there exists a root θ (in $\overline{\mathbb{K}}$) of h such that $W_k \ni (h_1(\theta), \ldots, h_m(\theta))$. By Proposition 3.12 and Corollary 3.13, there is a bijective correspondence between irreducible components $V_1, \ldots, V_l$ of $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$ and irreducible components $W_1, \ldots, W_l$ of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$, and $V_k = \Pi(W_k)$ for $k = 1, \ldots, l$. Thus, for $k = 1, \ldots, l$, there exists a root θ (in $\overline{\mathbb{K}}$) of h such that $V_k \ni (h_1(\theta), \ldots, h_n(\theta))$. Hence, the set

$$\overline{Z}' = \Pi\left(\overline{Y}'\right) = \left\{(h_1(\theta), \ldots, h_n(\theta)) \mid \theta \in \overline{\mathbb{K}} \text{ with } h(\theta) = 0\right\}$$

contains at least one point from every irreducible component of $\mathbf{Z}^{(n)}(f_1, \ldots, f_m)$. In particular, $\overline{Z}'$ contains all the isolated roots of the input system of polynomials $f_1, \ldots, f_m$ in n variables.

Algorithm RUR_overconstrained

Input: $f_1, \ldots, f_m \in \mathbb{K}[X_1, \ldots, X_n]$.

Output: $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for some finite set $\overline{Z}'$ which contains all the isolated common roots of the input system as well as at least one point from every irreducible component of the zero set of the input system.

```
1: for $i := 1, \ldots, m$ do:
2:     $g_i(X_1, \ldots, X_m) \leftarrow f_i(X_1, \ldots, X_n) \cdot (X_{n+1} - a_{n+1}) \cdot \cdots \cdot (X_m - a_m)$ where
       $a_{n+1}, \ldots, a_m$ are some constants in $\mathbb{K}$.
```

3: call Algorithm **RUR_square** on the input $g_1, \ldots, g_m \in \mathbb{K}[X_1, \ldots, X_m]$ to compute $h, h_1, \ldots, h_m \in \mathbb{K}[T]$ forming the RUR for some finite set $\overline{Y}'$ which contains at least one point from every irreducible component of the zero set (in $\overline{\mathbb{K}}^m$) of the system of polynomials $g_1, \ldots, g_m$

4: discard $h_{n+1}, \ldots, h_m$ to obtain $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for $\overline{Z}'$

One may suspect that the RUR for (some finite subset of) $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ may be computed via Algorithm **RUR_square** by simply treating polynomials $f_1, \ldots, f_m$ as polynomials in m variables $X_1, \ldots, X_m$ instead of generating $g_1, \ldots, g_m$. Unfortunately, this is not so. Recall that all irreducible components $W_1, \ldots, W_l$ of $\mathbf{Z}^{(m)}(f_1, \ldots, f_m)$ are of positive dimension. In order to compute the RUR for (some finite subset of) the zero set of positive dimension, the input system of polynomials $f_1, \ldots, f_m$ must be perturbed by auxiliary polynomials $f_1^*, \ldots, f_m^*$ with the conditions

(1) the support of f_i^* is contained in the support of f_i for $i = 1, \ldots, m$, and

(2) $f_1^*, \ldots, f_m^*$ have only finitely many common roots in $\overline{\mathbb{K}}^m$. (See step **9** of Algorithm **RUR_toric_square**.)

Suppose the condition (1) is satisfied. Since variables $X_{n+1}, \ldots, X_m$ do not appear in the supports of $f_1, \ldots, f_m$, they are not in the supports of $f_1^*, \ldots, f_m^*$. Then, the zero set of $f_1^*, \ldots, f_m^*$ never becomes finite; it always contains $\overline{\mathbb{K}}^{m-n}$. Thus, there is no system of polynomials $f_1^*, \ldots, f_m^*$ satisfying the above two conditions simultaneously, and step **9** of Algorithm **RUR_toric_square** always fails.

3.1.2.3 RUR for Underdetermined Systems

Consider a system of m polynomials $f_1, \ldots, f_m$ in n variables with coefficients in $\mathbb{K}$. Assume $m < n$. Then, we can construct a square system by adding $n - m$ copies of

f_m to the input system and use Algorithm **RUR_square**.

In general, the zero set of an underdetermined system has some positive dimensional components. Our algorithm cannot find them, but just picks up finitely many points on them. This is not very interesting, as the result is nearly meaningless in any real application.

3.1.2.4 Algorithm RUR

Putting the results from the previous sections together, given a system of m polynomials $f_1, \ldots, f_m$ in n variables with coefficients in $\mathbb{K}$, even though $m \neq n$, we can compute the RUR for some set $\overline{Z}'$ which contains all the isolated common roots of the input system in $\overline{\mathbb{K}}^n$ (rather than in $(\overline{\mathbb{K}}^*)^n$) as well as at least one point from every irreducible component of the zero set of the input system.

Algorithm RUR

Input: $f_1, \ldots, f_m \in \mathbb{K}[X_1, \ldots, X_n]$.

Output: $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for some finite set $\overline{Z}'$ which contains all the isolated common roots of the input system as well as at least one point from every irreducible component of the zero set of the input system.

```
1: if $m > n$ then
2:     call Algorithm RUR_overconstrained to compute $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for $\overline{Z}'$
3: else if $m = n$ then
4:     call Algorithm RUR_square to compute $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for $\overline{Z}'$
```

```
5: else    /* if $m < n$ then */
6:     $g_1 \leftarrow f_1,\ \ldots,\ g_m \leftarrow f_m,\ g_{m+1} \leftarrow f_m,\ \ldots,\ g_n \leftarrow f_m$
7:     call Algorithm **RUR_square** on the input $g_1, \ldots, g_n$ to compute
       $h, h_1, \ldots, h_n \in \mathbb{K}[T]$ forming the RUR for $\overline{Z}'$
```

3.1.3 Real Solving via RUR

Consider a square system of polynomials $f_1, \ldots, f_n$ in n variables with coefficients in $\mathbb{Q}$. Assume $\mathrm{MV}_{-0} > 0$. We have seen that we are able to compute the RUR for some set $\overline{Z}' \subseteq (\overline{\mathbb{C}}^*)^n$. The value of the i-th coordinate of a point in $\overline{Z}'$ can be obtained by evaluating the univariate polynomial h_i with coefficients in $\mathbb{Q}$ at some root θ of the univariate polynomial h with coefficients h. If $\theta \in \mathbb{R}$ then, obviously, $h_i(\theta) \in \mathbb{R}$. In this section, we will show that, under a certain condition, the converse is also true. Although the result in this section sounds trivial, it is very important from the computational view point.

Our goal is to develop an algorithm for solving a system of a multivariate polynomials with coefficients in $\mathbb{Q}$. Via the RUR, a multivariate polynomial system solving problem can be reduced to a univariate polynomial system solving problem. But, there is no algebraic method for computing the values of complex roots of the univariate polynomial h with coefficients in $\mathbb{Q}$ exactly. Thus, the RUR cannot be applied for exact solving a system of multivariate polynomials with coefficients in $\mathbb{Q}$. On the other hand, we are able to isolate all the real roots of h. Furthermore, we are also able to approximate the value of a real root of h to any given precision. Together with the results in this section, we are able to perform the exact computation over the *real* algebraic numbers given as coordinates of common roots of a system of multivariate polynomials with coefficients in $\mathbb{Q}$.

Consider a square system of n polynomials $f_1, \ldots, f_n$ in n variables with rational coefficients. Let Z be the zero set of the input system (in $\mathbb{C}^n$). Suppose that Algorithm **RUR_square** is called on the input $f_1, \ldots, f_n$ and returns the univariate polynomials h and $h_1, \ldots, h_n$ with rational coefficients forming the RUR for some finite set $\overline{Z}'$. The set $\overline{Z}'$ contains all the isolated common roots of the input system.

Proposition 3.15. *For any root θ of h,*

$$\theta \in \mathbb{R} \quad \Leftrightarrow \quad (h_1(\theta), \ldots, h_n(\theta)) \in \mathbb{R}^n. \tag{3.27}$$

Proof. The necessary condition is obvious. Thus, we only need to show the sufficient condition.

Rewriting (3.14), $\theta = -\sum_{l=1}^{n} u_l \cdot h_l(\theta)$. By Remark 3.11, $u_1, \ldots, u_l \in \mathbb{Z}$. Hence, $h_1(\theta), \ldots, h_n(\theta) \in \mathbb{R} \Rightarrow \theta \in \mathbb{R}$. □

If Z is of dimension zero then all the common roots of the input system are isolated, and thus, $\overline{Z}' \supseteq Z$. Hence, the set

$$\overline{Z}'_{\mathbb{R}} = \{(h_1(\theta), \ldots, h_n(\theta)) \mid \theta \in \mathbb{R} \text{ with } h(\theta) = 0\} \subseteq \mathbb{R}^n$$

contains all the real roots of the input system. Therefore, our algorithm can be used for real solving of zero dimensional square systems.

On the other hand, if Z is of positive dimension then there may be some real roots of the input system which are not contained in $\overline{Z}'_{\mathbb{R}}$. Algorithm **RUR_square** picks up least one point from each of the positive dimensional components of Z. These positive dimensional components of Z contain finitely or infinitely many real points. However, there is no guarantee that the points picked up by the algorithm are real, even if there are only finitely many real roots on some positive dimensional components. Note that Proposition 3.15 still holds, though it is not useful in this case.

3.2 Examples

In this section, we give examples that illustrate the results and shortcomings of our algorithms. Due to the limited space to display results, all the examples listed are of low dimension.

Example F_1:

Consider a system F_1 of 2 polynomials in 2 variables with integer coefficients:

$$\begin{aligned} f_1 &= 1 + 2X - 2X^2Y - 5XY + X^2 + 3X^3Y, \\ f_2 &= 2 + 6X - 6X^2Y - 11XY + 4X^2 + 5X^3Y \end{aligned} \tag{3.28}$$

The zero set of F_1 consists of 2 isolated points $(1, 1)$, $\left(\frac{1}{7}, \frac{7}{4}\right)$ and 1 irreducible component $X = -1$ of dimension 1.

The RUR for the zero set of F_1 is computed as follows:

$$\begin{aligned} h\,(T) &= 84T^4 + 306T^3 - 574T^2 - 1545T + 1989, \\ h_1\,(T) &= -T - \tfrac{r_{1,1}(T)}{r_{1,0}(T)}, \\ h_2\,(T) &= -T - \tfrac{r_{2,1}(T)}{r_{2,0}(T)} \end{aligned} \tag{3.29}$$

where

$$\begin{aligned}
r_{1,1}(T) = &- 1382279494376841984T^3 - 5914280670220800T^2 \\
&+ 9458729449441411392T - 8992070973148449600, \\
r_{1,0}(T) = &- 3963147148947893376T^3 - 1262397960878976T^2 \\
&+ 2717758211316680448T - 2585831836226329728, \\
r_{2,1}(T) = &- \frac{5412043022435784482792945336 32}{3176523}T^3 \\
&- \frac{31203742997590921280552 96}{27}T^2 \\
&+ \frac{1069598558052109058795873435648}{1058841}T \\
&- \frac{262531016085535220116679598080}{352947}, \\
r_{2,0}(T) = &- \frac{98740402375172913283239116 80}{151263}T^3 \\
&+ \frac{1904111379091981805699072}{27}T^2 \\
&+ \frac{901607402890309727171393781 76}{151263}T \\
&- \frac{3610984918268547057817776947 2}{50421}.
\end{aligned}$$

The univariate polynomial h has 4 roots θ and the values of the real and imaginary parts of $h_1(\theta)$ and $h_2(\theta)$ are approximated as follows:

(	$\Re h_1(\theta)$	,	$\Im h_1(\theta)$	), (	$\Re h_2(\theta)$	,	$\Im h_2(\theta)$	)
(	-1	,	-1.0561×10^{-46}	), (	-0.32019	,	-1.6753×10^{-47}	)
(	1	,	1.6658×10^{-47}	), (	1	,	8.0503×10^{-48}	)
(	0.14286	,	1.0519×10^{-51}	), (	1.75	,	1.0512×10^{-51}	)
(	-1	,	-5.2409×10^{-43}	), (	0.19519	,	-3.5186×10^{-40}	)

The table above suggests that, for the RUR computed as (3.29), we find 2 isolated roots along with 2 points on the positive dimensional component.

Example F_2:

Consider a system F_2 of 3 polynomials in 3 variables with integer coefficients:

$$\begin{aligned} f_1 &= X^2+Y^2+Z-1, \\ f_2 &= X^2+Y^2-Z+1, \\ f_3 &= Z-1. \end{aligned} \tag{3.30}$$

It is easy to see that F_2 has one and only one real root $(0,0,1)$ which actually lies on the intersection of 2 complex positive dimensional components $\{(-\sqrt{-1}Y, Y, 1)\}$ and $\{(\sqrt{-1}Y, Y, 1)\}$ of the zero set of the input system.

The univariate polynomial h in the RUR for the zero set of F_2 is computed as follows:

$$h = -T^4-4T^3-6T^2-4T-5. \tag{3.31}$$

Since the degree of h is 4, the RUR for the set determines 4 points lying on the positive dimensional components. By using Sturm's method, we can easily see that h has no real roots. Thus, by Proposition 3.15, none of those 4 points are real.

In general, if the zero set of the input system has some positive dimensional components on which there are only finitely many real points then our algorithm often will not pick up (some or all of) these real points. See Section 3.1.3.

Example F_3:

Let L_3 be a system of 3 linear polynomials in 3 variables with integer coefficients:

$$\begin{aligned} l_1 &= 3X-Y-1, \\ l_2 &= X-Y+1, \\ l_3 &= X+Y-3. \end{aligned} \tag{3.32}$$

The zero set of L_3 consists of a single point $(1,2)$.

Now, consider a system F_3 of 3 polynomials 2 in variables with integer coefficients:

$$\begin{aligned} f_1 &= l_2 \cdot l_3 = X^2 - 2X - Y^2 + 4Y - 3, \\ f_2 &= l_1 \cdot l_3 = 3X^2 + 2XY - 10X - Y^2 + 2Y + 3, \\ f_3 &= l_1 \cdot l_2 = 3X^2 - 4XY + 2X + Y^2 - 1. \end{aligned} \tag{3.33}$$

By construction, we immediately see that the zero set of F_3 consists of a single point $(1, 2)$. Note, however, the zero sets of any subsystem consisting of 2 polynomials has a positive dimensional component. (The subsystems (f_1, f_2), (f_1, f_3) and (f_2, f_3) have positive dimensional components l_3, l_2 and l_1, respectively.) Thus, an approach such as finding solutions for one pair of equations and "checking" them the third equation would not be sufficient.

Since F_3 is an overdetermined system, **Algorithm_RUR_overconstrained** constructs a square system G_3 of 3 polynomials in 3 variables with rational coefficients:

$$\begin{aligned} g_1 &= f_1 \cdot (Z-1) = (X^2 - 2X - Y^2 + 4Y - 3)(Z-1), \\ g_2 &= f_2 \cdot (Z-1) = (3X^2 + 2XY - 10X - Y^2 + 2Y + 3)(Z-1), \\ g_3 &= f_3 \cdot (Z-1) = (3X^2 - 4XY + 2X + Y^2 - 1)(Z-1). \end{aligned} \tag{3.34}$$

The univariate polynomial h in the RUR for the zero set of G_3 is computed as follows:

$$\begin{aligned} h = {} & -5505024000T^{10} - 391643136000T^9 \\ & -95667879936000T^8 - 434913253785600T^7 \\ & +247516851339264 0T^6 + 58123559884554240T^5 \\ & +571184791525785600T^4 + 2276891395149004800T^3 \\ & -4405394155933532160T^2 - 7041166238998855 6800T \\ & -17633030377020850 1760. \end{aligned} \tag{3.35}$$

Likewise, but not shown here, we compute h_1, h_2, and h_3. The RUR for the zero set of F_3 is obtained from the RUR of the zero set of G_3 by ignoring the last coordinate (i.e. ignoring h_3). Evaluating h_1 and h_2 at roots of h gives us multiple points at the single location $(1, 2)$.

3.3 Complexity Analysis

In this section, we give a worst-case asymptotic complexity analysis of Algorithm **RUR_toric_square** described in Section 3.1.1.

The computational model used here is either the Turing machine or the BSS machine [6]. If $\mathbb{K}$ is $\mathbb{Q}$ or some finite field then the algorithm can be implemented on Turing machines (or existing computers). On the other hand, if $\mathbb{K}$ is $\mathbb{R}$ or $\mathbb{C}$ then the algorithm cannot be implemented (exactly) on Turing machines. In this case, the BSS machine over $\mathbb{R}$ or $\mathbb{C}$ is used. On the BSS machine over a field $\mathbb{K}$, an arithmetic operation over $\mathbb{K}$ is done in constant time, and thus, roughly speaking, the time complexity of a given algorithm matches the number of arithmetic operations over $\mathbb{K}$. In order to make a valid argument on either of those computational models, in this section, we only consider the arithmetic complexity (the number of arithmetic operations) of the algorithm. The bit-length of the quantities appearing in the algorithm is not discussed here, but some discussion of the practical performance can be found in Section 6.1.1.2.

The following notations are used:

Let $\boldsymbol{O}^*(\ )$ denote a big oh notation in which a polylog factor is ignored: $\boldsymbol{O}^*(n) = \boldsymbol{O}(n \log^r n)$ for some $r \geq 0$. Also, let ω be the constant so that the matrix multiplication of two square matrices of dimension l takes $\boldsymbol{O}(l^\omega)$ arithmetic operations. It is well-known that $\omega < 2.376$.

3.3.1 Arithmetic Complexity Analysis

Consider a square system of polynomials $f_1, \ldots, f_n$ in n variables with coefficients in $\mathbb{K}$. Let A_i be the support of f_i for $i = 1, \ldots, n$. Suppose Algorithm **RUR_toric_square** is called on the input $f_1, \ldots, f_n$ and $A_1, \ldots, A_n$ and returns the RUR for some finite subset Z' of the zero set Z of the input system. The algorithm sets the support A_0 of f_0 so that f_0 is a linear polynomial.

We introduce two quantities $\mathcal{M}$ and $\mathcal{N}$.

As before, let MV_{-i} denote the mixed-volume of the convex hulls of $A_0, A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n$. (See Section 2.1.1.1.) Define $\mathcal{M} = \sum_{i=0}^{n} \mathrm{MV}_{-i}$. Thus, $\mathcal{M}$ is the degree of the toric resultant.

Step **7** of the algorithm constructs the toric resultant matrix N whose determinant is some non-trivial multiple of the toric resultant. Let $\mathcal{N} = \dim N$.

Step **18** or step **26** of the algorithm determines the cardinality M of Z' which matches the degree of the univariate polynomial h in the RUR.

Recall the following facts [97]:

- Given $l + 1$ distinct values in $\mathbb{K}$, a unique univariate polynomial of degree at most l with coefficients in $\mathbb{K}$ that takes those values at $l + 1$ distinct points in $\mathbb{K}$ can be computed via interpolation using $\boldsymbol{O}^*(l)$ arithmetic operations over $\mathbb{K}$.

- Given two univariate polynomials with coefficients in $\mathbb{K}$ of degree at most l, their GCD is computed using $\boldsymbol{O}^*(l)$ arithmetic operations over $\mathbb{K}$. Thus, given a univariate polynomial with coefficients in $\mathbb{K}$ of degree at most l, its square free part is computed using $\boldsymbol{O}^*(l)$ arithmetic operations over $\mathbb{K}$.

The value of the toric resultant for a system of $n + 1$ polynomials in n variables with supports $A_0, A_1, \ldots, A_n$ and with arbitrary but fixed coefficients in $\mathbb{K}$ is calculated using $\boldsymbol{O}^*\left(n\mathcal{M}\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$ [32] [14].

The arithmetic complexity of the algorithm is governed by the loop from step **13** through step **25** or the loop from step **26** through **36**.

By (2.5), $\deg_s \mathrm{TGCP}\left(s, \boldsymbol{u}\right) = \sum_{i=1}^{n} \mathrm{MV}_{-i} = \mathcal{M} - \mathrm{MV}_{-0}$, and thus, the coefficients of $\mathrm{TGCP}\left(s, \boldsymbol{u}\right)$ at fixed $\boldsymbol{u}$ (regarded as a univariate polynomial in s) are computed via interpolation using $\boldsymbol{O}^*\left(n\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$.

By (2.4), $\deg_T \mathrm{TPert}\left(T, u_1, \ldots, u_n\right) = \mathrm{MV}_{-0}$. Thus, at step **15** or step **28**, $\mathrm{TPert}\left(T, u_1, \ldots, u_n\right)$ at fixed $\left(u_1, \ldots, u_n\right)$ is computed via interpolation using $\boldsymbol{O}^*\left(n\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$, and at step **16** or step **29**, the square-free part of $\mathrm{TPert}\left(T, u_1, \ldots, u_n\right)$ is computed using $\boldsymbol{O}^*\left(\mathrm{MV}_{-0}\right)$ arithmetic operations over $\mathbb{K}$.

By the same argument as the previous paragraph, for $i = 1, \ldots, n$, at step **33**, $\mathrm{TPert}\left(t, u_1, \ldots, u_{i-1}, u_i \pm 1, u_{i+1}, \ldots, u_n\right)$ at fixed $\left(u_1, \ldots, u_n\right)$ are computed using $\boldsymbol{O}^*\left(n\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$, and at step **34**, $q_i^{\pm}\left(t\right)$ are computed using $\boldsymbol{O}^*\left(\mathrm{MV}_{-0}\right)$ arithmetic operations over $\mathbb{K}$. Hence, the **for** loop from step **33** through step **36** is executed using $\boldsymbol{O}^*\left(n^2\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$.

By Proposition 3.9, the loop from step **13** through step **25** is repeated $\boldsymbol{O}\left(n\mathrm{MV}_{-0}^2\right)$ times, and each iteration uses $\boldsymbol{O}^*\left(n\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$. Thus, in total, the number of arithmetic operations over $\mathbb{K}$ needed to process this loop is $\boldsymbol{O}^*\left(n^2\mathrm{MV}_{-0}^3\mathcal{M}^2\mathcal{N}^{\omega}\right)$.

By Proposition 3.10, the loop from step **26** through step **36** is repeated $\boldsymbol{O}\left(n^2M^2\right)$ times, and each iteration uses $\boldsymbol{O}^*\left(n^2\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$. Thus, in total, the number of arithmetic operations over $\mathbb{K}$ needed to process this loop is $\boldsymbol{O}^*\left(n^4M^2\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$.

Putting all together, the univariate polynomials h and $h_1, \ldots, h_n$ forming the

RUR are computed using $\boldsymbol{O}^*\left(n^2\left(\mathrm{MV}_{-0}^2+n^2M^2\right)\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$ arithmetic operations over $\mathbb{K}$.

By (3.4), $\mathrm{MV}_{-0} \geq M$. The equality holds if Z' does not contain any multiple root of the input system. In this case, the loop from step **26** through step **36** governs the complexity of the algorithms. On the other hand, if Z' contains some multiple roots of the input system then $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$ is not square-free and

$$\deg_T \mathrm{TPert}\,(T, u_1, \ldots, u_n) = \mathrm{MV}_{-0} > M = \deg h. \tag{3.36}$$

In this case, there is a slight chance that the loop from step **13** through **25** takes more arithmetic operations over $\mathbb{K}$ than the loop from step **26** through step **36**. When the loop from step **13** through **25** is executed, M has not yet been determined. On the other hand, the loop from step **26** through step **36** is executed after M is correctly determined. Thus, the loop from step **26** through step **36** does not have to be repeated unnecessarily.

3.3.1.1 $\mathcal{M}$ and $\mathcal{N}$

We have seen that the arithmetic complexity of the algorithm is expressed in terms of $\mathcal{M}$ and $\mathcal{N}$. In this section, we consider how the relation of those two quantities affect the complexity of the algorithm.

When the characteristic of $\mathbb{K}$ is 0, at step **15** or step **28** or step **33**, $\mathrm{TPert}\,(T, u_1, \ldots, u_n)$ at fixed $(u_1, \ldots, u_n)$ (regarded as a univariate polynomial in T) is computed via interpolation through the values of the coefficient of the term s^d in some non-trivial multiple of $\mathrm{TGCP}\,(s, \boldsymbol{u})$ instead of the values of the coefficient of the term s^d in $\mathrm{TGCP}\,(s, \boldsymbol{u})$ without any extraneous factor. Recall that the value of any coefficient of some non-trivial multiple of $\mathrm{TGCP}\,(s, \boldsymbol{u})$ is interpolated from the values of $\det N$ while the value of any coefficient of $\mathrm{TGCP}\,(s, \boldsymbol{u})$ without any extraneous factor is interpolated from the values of $\mathrm{TRes}\left(f_0, f_1 - sf_1^*, \ldots, f_n - sf_n^*\right)$. Recall that calculation of the explicit value of the toric resultant requires additional steps to eliminate the contribution of the extraneous factor from the value of $\det N$. That is, any coefficient of some non-trivial multiple of $\mathrm{TGCP}\left(s, \boldsymbol{u}\right)$ is computed without executing these additional steps. (See Section 3.1.1.) In such a case, $\mathrm{TPert}\left(T, u_1, \ldots .u_n\right)$ at fixed $(u_1, \ldots, u_n)$ is computed using $\boldsymbol{O}^*\left(\mathrm{MV}_{-0}\mathcal{N}^{1+\omega}\right)$ arithmetic operations over $\mathbb{K}$, and thus, the number of arithmetic operations over $\mathbb{K}$ needed to compute univariate polynomials h and $h_1, \ldots, h_n$ forming the RUR becomes $\boldsymbol{O}^*\left(n\left(\mathrm{MV}_{-0}^2 + n^2M^2\right)\mathrm{MV}_{-0}\mathcal{N}^{1+\omega}\right)$.

The quantity $\mathcal{N}$ actually depends on the algorithm used to construct the resultant matrix N. From (2.2) and (2.3), $\mathcal{N} \geq \mathcal{M}$. However, no algorithm that constructs an optimal N (i.e., N satisfying $\mathcal{N} = \mathcal{M}$) has been found except for very small n [62] [63]. Even if the best algorithm currently known is used, there is a risk that $\mathcal{N}$ becomes exponentially bigger than $\mathcal{M}$ [32] [14]: $\mathcal{N} = \boldsymbol{O}\left(\frac{e^n}{\sqrt{n}}\mathcal{M}\right)$. Thus, asymptotically, the cost for additional steps to eliminate the extraneous factor will be negligible compared to the cost for interpolations through the values of the determinant of a bigger matrix. Hence, even if the characteristic of $\mathbb{K}$ is 0, the "best" worst-case arithmetic complexity of Algorithm **RUR_toric_square** remains $\boldsymbol{O}^*\left(n^2\left(\mathrm{MV}_{-0}^2 + n^2M^2\right)\mathrm{MV}_{-0}\mathcal{M}^2\mathcal{N}^{\omega}\right)$. In practice, $\mathcal{N}$ rarely becomes exponentially bigger than $\mathcal{M}$. This matter is discussed more later in Section 6.1. Nevertheless, developing an algorithm for computing a resultant matrix of smaller (or the smallest) size is still an active area of research.

CHAPTER IV

EXACT COMPUTATION FOR ALGEBRAIC POINTS AND CURVES

This chapter explains a method for exact manipulation of algebraic points and curves. In Section 4.1, I describe the root bound approach to exact determination of the sign of the real and imaginary parts of a given algebraic number. Together with the Rational Univariate Reduction (RUR) introduced in Chapter III, exact sign determination of the real and imaginary parts of algebraic numbers enables me to develop some exact geometric predicates for algebraic points and curves (Section 4.2). As an application of these predicates, in Section 4.3, I discuss how to detect degeneracies appearing in exact boundary evaluation of solid objects.

4.1 Exact Computation for Complex Algebraic Numbers

This section describes the root bound approach to exact determination of the sign of the real and imaginary parts of a given algebraic number.

For a complex algebraic number ζ, write $\Re\zeta$ and $\Im\zeta$ for the real and imaginary parts of ζ:

$$\zeta = \Re\zeta + \sqrt{-1}\Im\zeta. \tag{4.1}$$

Note that both $\Re\zeta$ and $\Im\zeta$ are real algebraic numbers.

In Section 2.1.2.2, I explain the root bound approach to exact sign determination of a real algebraic number of the form $e(\xi_1, \dots, \xi_n)$ where ξ_i is a real root of a univariate polynomial with rational coefficients and e is an algebraic expression involving $+, -, *, /$ and $\sqrt[k]{\ }$. In this section, this approach will be extended to exact sign determination of the real and imaginary parts of a complex algebraic number. Thus, the goal of this section is stated as follows:

Let e be an expression in m variables with rational coefficients. Furthermore, let $\zeta_1, \ldots, \zeta_m$ be complex algebraic numbers. We would like to determine the sign of the real and imaginary parts of the complex algebraic number $e(\zeta_1, \ldots, \zeta_m)$ exactly.

Set e_R and e_I to be rational functions in $2m$ variables with rational coefficients that satisfy

$$\begin{aligned} e(\zeta_1, \ldots, \zeta_m) &= e_R(\Re\zeta_1, \ldots, \Re\zeta_m, \Im\zeta_1, \ldots, \Im\zeta_m) \\ &\quad + \sqrt{-1}e_I(\Re\zeta_1, \ldots, \Re\zeta_m, \Im\zeta_1, \ldots, \Im\zeta_m). \end{aligned} \tag{4.2}$$

For example, if $e(\zeta) = \zeta^2$ then $e_R(\Re\zeta, \Im\zeta) = (\Re\zeta)^2 - (\Im\zeta)^2$ and $e_I(\Re\zeta, \Im\zeta) = 2\Re\zeta\Im\zeta$.

We will apply the root bound approach to exact sign determination of the real algebraic numbers (introduced in Section 2.1.2.2) to these two real algebraic numbers

$$e_R = e_R(\Re\zeta_1, \ldots, \Re\zeta_m, \Im\zeta_1, \ldots, \Im\zeta_m) \quad \text{and} \quad e_I = e_I(\Re\zeta_1, \ldots, \Re\zeta_m, \Im\zeta_1, \ldots, \Im\zeta_m).$$

The root bounds for e_R and e_I and approximations $\widetilde{e}_R$ and $\widetilde{e}_I$ for e_R and e_I, respectively, to any prescribed precision are computed by using recursive rules (introduced in Section 2.1.2.2). In order to complete the adaption, the base cases of recursion must be treated. Thus, the remaining task is stated as follows:

Let ζ be a complex algebraic number specified as a root of a univariate polynomial e with rational coefficients. For real numbers $\Re\zeta$ and $\Im\zeta$, we would like to compute

(1) "constructive" bounds for the degree and Mahler measure of $\Re\zeta$ and $\Im\zeta$, and

(2) approximations for the real and imaginary parts of ζ to any prescribed precision.

For (2), Aberth's method [1] is used to compute approximations for the (real and imaginary parts of) roots of univariate polynomials with rational coefficients. The method is implemented with floating point numbers with arbitrary precision mantissa in order to obtain an approximation to any given precision.

For (1), we first find univariate polynomials with integer coefficients R_e and I_e such that $R_e(\Re\zeta) = I_e(\Im\zeta) = 0$ (Proposition 4.16 below). We then calculate bounds on the degrees and measures of $\Re\zeta$ and $\Im\zeta$ from the degrees and coefficients of R_e and I_e (Proposition 4.17 below).

In the rest of this section, we assume, for simplicity, that all the polynomials have integer coefficients. The results are still valid for polynomials with rational coefficients.

Proposition 4.16. *Let ζ be an algebraic number specified as a root of a polynomial $e(T) \in \mathbb{Z}[T]$. Write $\mathrm{SRes}_U(f,g)$ for the Sylvester resultant of univariate polynomials f and g w.r.t. variable U. Then*

(1) $\Re\zeta$ is a real algebraic number and a root of

$$R_e(T) = \sum_{i=0}^{m} 2^i s_i T^i \quad \in \quad \mathbb{Z}[T]$$

where $\sum_{i=0}^{m} s_i T^i = \mathrm{SRes}_U(e(T-U), e(U))$.

(2) $\Im\zeta$ is a real algebraic number and a root of

$$I_e(T) = \sum_{j=0}^{\lfloor \frac{m}{2} \rfloor} 2^{2j} (-1)^j s_{2j} T^{2j} \quad \in \quad \mathbb{Z}[T]$$

where $\sum_{i=0}^{m} s_i T^i = \mathrm{SRes}_U(e(T+U), e(U))$.

Proof. Recall that, for f and $g \in \mathbb{Z}[T]$, if α and β are roots of f and g, respectively, then $\alpha \pm \beta$ is a root of $\mathrm{SRes}_U(f(T \mp U), g(U))$ [97].

If ζ is a root of e then its complex conjugate $\overline{\zeta}$ is also a root of e. Thus, the sum $\zeta + \overline{\zeta} = 2\Re\zeta$ of two roots of e is a root of $\mathrm{SRes}_U(e(T-U), e(U))$.

Similarly, the difference $\zeta - \overline{\zeta} = 2\sqrt{-1}\Im\zeta$ of two roots of e is a root of $\mathrm{SRes}_U(e(T+U), e(U))$.

If 2ξ is a root of $\sum_{i=0}^{m} s_i T^i$ then ξ is a root of $\sum_{i=0}^{m} 2^i s_i T^i$.

If, for $\xi \in \mathbb{R}$, $\sqrt{-1}\xi$ is a root of $\sum_{i=0}^{m} s_i T^i$ then ξ is a root of $\sum_{j=0}^{\lfloor \frac{m}{2} \rfloor} (-1)^j s_{2j} T^{2j}$.

Putting these together, the statements in the proposition hold. □

By Gauß's lemma, if α is a root of a polynomial $e(T) = \sum_{i=0}^{n} e_i T^i \in \mathbb{Z}[T]$ with $e_n e_0 \neq 0$ then $\deg \alpha \leq \deg e$ and $M(\alpha) \leq M(e)$. By Landau's theorem [70], for $e(T) \in \mathbb{Z}[T]$, $M(e) \leq ||e||_2 = \sqrt{\sum_{i=0}^{n} |e_i|^2}$. Thus, we could use $\deg e$ and $||e||_2$ as "constructive" upper bounds on $\deg \alpha$ and $M(\alpha)$.

Proposition 4.17. *Following the notation above*

(1) $\deg \Re\zeta \leq \deg R_e \leq \deg^2 e$,

$M(\Re\zeta) \leq M(R_e) \leq ||R_e||_2 \leq 2^{2n^2+n} ||e||_2^{2n}$,

(2) $\deg \Im\zeta \leq \deg I_e \leq \deg^2 e$ *and*

$M(\Im\zeta) \leq M(I_e) \leq ||I_e||_2 \leq 2^{2n^2+n} ||e||_2^{2n}$.

Proof. (1) It can be shown [101] that, for f and $g \in \mathbb{Z}[T]$,

$$\deg \mathrm{SRes}_T(f(T), g(T)) \leq \deg f \deg g \quad \text{and}$$
$$||\mathrm{SRes}_U(f(T-U), g(U))||_2 \leq \left(2^{\deg f+1} ||f||_2\right)^{\deg g} ||g||_2^{\deg f}.$$

Thus

$$m = \deg \sum_{j=0}^{m} s_j T^j = \deg \mathrm{SRes}_U(e(T-U), e(U)) \leq \deg^2 e = n^2$$

and

$$\begin{aligned} \left|\left|\sum_{j=0}^{m} s_j T^j\right|\right|_2 &= ||\mathrm{SRes}_U(e(T-U), e(U))||_2 \\ &\leq \left(2^{\deg e+1} ||e||_2\right)^{\deg e} ||e||_2^{\deg e} \\ &= 2^{n^2+n} ||e||_2^{2n}. \end{aligned}$$

Hence

$$\begin{aligned}
\left|\left|\sum_{j=0}^{m} 2^j s_j T^j\right|\right|_2 &= \sqrt{\sum_{j=0}^{m} 2^{2j} \left|s_j\right|^2} \\
&\leq \sqrt{\sum_{j=0}^{m} 2^{2m} \left|s_j\right|^2} \\
&= 2^m \sqrt{\sum_{j=0}^{m} \left|s_j\right|^2} \\
&= 2^m \left|\left|\sum_{j=0}^{m} s_j T^j\right|\right|_2 \\
&\leq 2^{n^2} \left(2^{n^2+n} \left|\left|e\right|\right|_2^{2n}\right).
\end{aligned}$$

(2) The proof is similar to (1).

□

The argument in this section is summarized to the following proposition:

Proposition 4.18. *Let $e(X_1, \ldots, X_m)$ be a rational function with rational coefficients. Also, let $\zeta_1, \ldots, \zeta_m$ be algebraic numbers, each of which is specified as a root of some univariate polynomial with rational coefficients. Assume that we are able to compute approximations for the real and imaginary parts of each of $\zeta_1, \ldots, \zeta_m$ to any prescribed precision. Then, the sign of the real and imaginary parts of the algebraic number $e(\zeta_1, \ldots, \zeta_m)$ can be determined exactly.*

4.2 Exact Computation for Algebraic Points and Curves

The goal of the discussion in this section is to develop some exact geometric predicates for algebraic points and curves.

When an algebraic point is specified as a common root of a system of polynomials, using the RUR (introduced in Chapter III), every coordinate of such an algebraic point is expressed as $(h_1(\zeta), \ldots, h_n(\zeta))$ where $h_1, \ldots, h_n$ are univariate polynomials (or rational functions) with rational coefficients and ζ is some root of some other univariate polynomial with rational coefficients. Thus, in general, every coordinate of an algebraic points is expressed as $e(\zeta_1, \ldots, \zeta_m)$ where e is a rational function with rational coefficients and each ζ_i is some root of some univariate polynomial with rational coefficients. Together with the root bound approach to exact sign determination of the real and imaginary parts of algebraic numbers (introduced in Section 4.1), the sign of the real and imaginary parts of every coordinate of an algebraic point can be exactly determined.

Section 4.2.1 describes a method for determining the sign of the real and imaginary parts of the coordinates of an algebraic point expressed in the RUR. Section 4.2.2 discusses the development of some exact geometric predicates for algebraic points and curves.

4.2.1 Exact Computation for RUR

In this section, first, I describe how to determine the exact sign of the real and imaginary parts of the coordinates of an algebraic point expressed in the RUR (Section 4.2.1.1). Then, I present several algorithms for supporting exact computation for algebraic points expressed in the RUR (Section 4.2.1.2).

4.2.1.1 Exact Sign for RUR

Let e be a rational function in n variables $X_1, \ldots, X_n$ with rational coefficients. Also, let Z' be a finite set of algebraic points in $\mathbb{C}^n$. Furthermore, assume that there exist univariate polynomials $h, h_1, \ldots, h_n$ with integer coefficients such that for every point $(\zeta_1, \ldots, \zeta_n) \in Z'$, $(\zeta_1, \ldots, \zeta_n) = (h_1(\theta), \ldots, h_n(\theta))$ for some root $\theta \in \mathbb{C}$ of h. Algorithm **Exact_sign** below determines whether or not $e(\zeta_1, \ldots, \zeta_n) = 0$ exactly.

Algorithm Exact_sign

Input: $e \in \mathbb{Q}(X_1, \ldots, X_n)$ and $h, h_1, \ldots, h_n \in \mathbb{Z}[T]$.

```
Output: The exact sign of the real and imaginary parts of $e(h_1(\theta), \dots, h_n(\theta))$ for
    every root $\theta \in \mathbb{C}$ of $h$.
1: Set $r_R$ and $r_I$ to be bivariate rational functions with integer coefficients such that
       $e(h_1(\theta), \dots, h_n(\theta)) = r_R(\Re\theta, \Im\theta) + \sqrt{-1} r_I(\Re\theta, \Im\theta)$
2: Recursively compute bounds for the degree and measure of algebraic numbers
   $r_R = r_R(\Re\theta, \Im\theta)$ and $r_I = r_I \Re\theta, \Im\theta$ using Proposition 2.5. The base cases are
   given in Proposition 4.17.
3: for every root $\theta \in \mathbb{C}$ of $h$ do:
4:     Recursively compute approximations $\widetilde{r}_R$ and $\widetilde{r}_I$ for $r_R = r_R(\Re\theta, \Im\theta)$
       and $r_I = r_I(\Re\theta, \Im\theta)$, respectively, to a certain precision such that the
       root bounds allow us to determine their signs by (2.7) or (2.8). The
       base case, i.e., computing approximations for $\Re\theta$ and $\Im\theta$ to a certain
       precision, are done by Aberth's method.
```

Note that Algorithm **Exact_sign** is an irregular application of the root bound approach. The root bound approach is powerful when it is used to test whether a given number IS NOT zero. If it is used to test whether a given number IS zero then approximations we must compute approximations for the real and imaginary parts of the number up to the precision specified by their root bounds, which usually costs a lot.

4.2.1.2 Exact Computation for RUR

In Chapter III, I present Algorithm **RUR** that, given a system of m polynomials $f_1, \dots, f_m$ in n variables with rational coefficients, computes the RUR for some set $\overline{Z}'$

that contains all the isolated common roots of the input system in $\overline{\mathbb{C}}^n$ as well as at least one point from every irreducible component of the zero set of the input system. Note the set $\overline{Z}'$ may contain some points that are not common roots of the input system. Sometimes, we would like to remove this redundancy. Also, note that, if the input system is not zero-dimensional, i.e., the input system has infinitely many common roots then the finite set $\overline{Z}'$ cannot contain all the common roots of the input system. We would like to know whether or not the input system is zero-dimensional. The following algorithms solve these problems.

Algorithm **Exact_RUR** removes the points that are not common roots of the input system from $\overline{Z'}$, and thus, gives the exact RUR for some finite subset Z' of the zero set of the input system that contains all the isolated common roots of the input system as well as at least one point from every irreducible component of the zero set of the input system.

Algorithm: Exact_RUR

Input: $f_1, \ldots, f_m \in \mathbb{Q}[X_1, \ldots, X_n]$.

Output: $h, h_1, \ldots, h_n \in \mathbb{Q}[T]$ and $\Theta \subseteq \mathbb{C}$ such that $Z' = \{h_1(\theta), \ldots, h_n(\theta) \mid \theta \in \Theta\}$ contains all the isolated common roots of the input system as well as at least one point from every irreducible component of the zero set of the input system.

```
1: call algorithm RUR to compute $h, h_1, \ldots, h_n \in \mathbb{Q}[T]$ forming the RUR for some finite set $\overline{Z}'$ that contains all the isolated common roots of the input system as well as at least one point from every irreducible component of the zero set of the input system
2: set $\Theta$ to be the set of all the roots of $h$
```

```
3: for $i := 1, \ldots, n$ do:
4:     for $\theta \in \Theta$ do:
5:         call Algorithm **Exact_sign** to compute the exact sign $\sigma_R$ and
           $\sigma_I$ of the real and imaginary parts of $f_i\left(h_1\left(\theta\right), \ldots, h_n\left(\theta\right)\right)$, re-
           spectively
6:         if $\sigma_R \times \sigma_I \neq 0$ then
7:             $\Theta \leftarrow \Theta \setminus \{\theta\}$
```

The correctness and exactness of Algorithm **Exact_RUR** immediately follow from the correctness and exactness of Algorithm **RUR** and Algorithm **Exact_sign**.

Next, I present a generic algorithm to determine whether or not the zero set of a given system of polynomials with rational coefficients has positive dimensional components.

Recall that Algorithm **Exact_RUR** finds the RUR for some finite subset Z' of the zero set Z of the input system that contains at least one point from every irreducible component of Z. If Z is infinite (i.e., Z has positive-dimensional components) then Z' depends on the polynomials $f_1^*, \ldots, f_n^*$ used to perturb the input system. (See step **9** in Algorithm **RUR_toric_square** in Section 3.1.1.1).

Suppose two distinct executions of Algorithm **Exact_RUR** find two finite subsets Z_1' and Z_2' of Z and their exact RUR's:

$$Z_k' = \left\{ \left(h_1^{(k)}\left(\theta_k\right), \ldots, h_n^{(k)}\left(\theta_k\right) \right) \mid h^{(k)}\left(\theta_k\right) = 0 \right\}, \qquad k = 1, 2. \tag{4.3}$$

If ζ is an isolated common root of the input system then $\zeta \in Z_1' \cap Z_2'$, and thus, $\exists \theta_1$

and $\theta_2 \in \mathbb{C}$ such that

$$\zeta = \left(h_1^{(1)}(\theta_1), \ldots, h_n^{(1)}(\theta_1)\right) = \left(h_1^{(2)}(\theta_2), \ldots, h_n^{(2)}(\theta_2)\right). \tag{4.4}$$

Hence, $Z_1' \setminus Z_2' \neq \emptyset$ implies that Z has some positive dimensional components. We can compare Z_1' and Z_2' pointwise using Algorithm **Exact_sign**.

Algorithm Positive_Dimensional_Components

Input: $f_1, \ldots, f_m \in \mathbb{Q}[X_1, \ldots, X_n]$ and a small positive integer Max_Trials.

Output: `True` or `Probably_False`

```
1: $Has\_Pos\_Dim\_Compo \leftarrow$ Probably_False
2: $Trial \leftarrow 0$
3: while $Trail < Max\_Trials$ and $Has\_Pos\_Dim\_Compo =$ Probably_False do:
4:         call Algorithm Exact_RUR to compute the exact RUR for some finite
           subsets $Z_1'$ and $Z_2'$ of the zero set of the input system
5:         call Algorithm Exact_sign to compare $Z_1'$ and $Z_2'$ pointwise
6:         if $Z_1' \neq Z_2'$ then
7:             $Has\_Pos\_Dim\_Compo \leftarrow$ True
8:         else
9:             increment $Trial$
```

If the zero set Z of the input system has a positive dimensional component and polynomials f_i^* are chosen generically, then almost always $Z_1' \setminus Z_2' \neq \emptyset$. Thus, Max_Counts is usually set to be 2.

Now, I present an algorithm to compute the real roots of a given square zero-dimensional system of polynomials with rational coefficients.

Consider a system of n polynomials $f_1, \ldots, f_n$ in n variables with rational coefficients. Let Z be the zero set of the input system (in $\mathbb{C}^n$). Suppose that Algorithm **RUR_square** in Section 3.1.2.1 is called on the input $f_1, \ldots, f_n$ and returns the univariate polynomials h and $h_1, \ldots, h_n$ with rational coefficients forming the RUR for some finite set $\overline{Z}'$. The set $\overline{Z}'$ contains all the isolated common roots of the input system. If Z is zero-dimensional, i.e., Z is finite then all the common roots of the input system are isolated, and thus, $\overline{Z}' \supseteq Z$. Hence, the set

$$\overline{Z}'_{\mathbb{R}} = \{(h_1(\theta), \ldots, h_n(\theta)) \mid \theta \in \mathbb{R} \text{ with } h(\theta) = 0\} \subseteq \mathbb{R}^n$$

contains all the real roots of the input system. Therefore, our algorithm can be used for real solving of zero dimensional square systems.

Algorithm Exact_RUR_real

Input: A zero-dimensional system of polynomials $f_1, \ldots, f_n \in \mathbb{Q}[X_1, \ldots, X_n]$

Output: $h, h_1, \ldots, h_n \in \mathbb{Q}[T]$ and $\Theta \subseteq \mathbb{R}$ such that $Z = \{h_1(\theta), \ldots, h_n(\theta) \mid \theta \in \Theta\}$ is the set of all the real common roots of the input system.

```
1: call Algorithm RUR_square to compute $h, h_1, \ldots, h_n \in \mathbb{Q}[T]$ forming the RUR
   for the zero set of the input system
2: use Sturm's method to compute the set $\Theta$ of all the real roots of $h$
3: for $i := 1, \ldots, n$ do:
4:         for $\theta \in \Theta$ do:
```

```
5:        use the root bound approach to determine the exact sign $\sigma$ of
          $f_i\left(h_1\left(\theta\right), \ldots, h_n\left(\theta\right)\right)$
6:        if $\sigma \neq 0$ then
7:            $\Theta \leftarrow \Theta \setminus \{\theta\}$
```

It is important to note the algorithm **Exact_RUR_real** works correctly only if the system has finitely many common roots. In particular, the algorithm may not be able to find real points lying on some (complex) positive dimensional components.

4.2.2 Exact Geometric Computation

The objects dealt with in computational geometry are sets of points in a vector space over the field $\mathbb{K}$. We assume that a coordinate system is introduced to the space so that every point can be represented as a tuple of numbers belonging to $\mathbb{K}$, a certain type of metric is defined, and the topology induced from the metric is endowed to the space. For simplicity, in the rest of this chapter, we assume that $\mathbb{K} = \mathbb{R}$.

These sets of points are not necessarily finite, but they must be finitely specifiable so that they are encoded as a string of finite length in algorithms. Thus, in addition to sets of finitely many individual points, computational geometry also deals with curves, surfaces, portions of curves, portions of surfaces, and solid objects.

An algebraic curve is implicitly described as the zero set of some polynomial. An algebraic curve of a certain type may be represented parametrically as a tuple of rational functions. For simplicity, we only deal with algebraic curves defined by polynomials with rational coefficients.

An algebraic point is defined to be an intersection of algebraic curves. The coordinates of an algebraic point are algebraic numbers. In particular, the coordinates of real algebraic points are real algebraic numbers. Note that they are possibly irrational. A real algebraic number is specified as the *unique* root of some polynomial with rational coefficients in some interval on $\mathbb{R}$. The endpoints of this interval can be chosen to be rational numbers. Likewise, a real algebraic point can be represented as an n-dimensional hypercube. The corners of this hypercube can be set to have rational coordinates.

Let an algebraic point x be represented by some hypercube. Any point contained in the hypercube can be used as an approximation of x. The size of the hypercube corresponds to an upper bound for the (absolute) error of these approximations. The smaller the hypercube is, the more precise the approximations are. Usually, the hypercube can shrink into any size so that it stores approximations to arbitrary precision.

The drawback of this hypercube representation of algebraic points is that we may not able to tell whether or not

(1) two given points x and y are identical, or

(2) a given point x lies on a given curve C.

Since (1) reduces to (2), we only consider (2). The query is answered by testing whether or not C intersects with the hypercube representing x. If C does not intersect with the hypercube representing x then we know that x does not lie on C. On the other hand, if C intersects with the hypercube representing x then we cannot distinguish the case where x really does lie on C from the case where x actually does not lie on C but the hypercube has not yet shrunk enough.

This situation is resolved if we somehow can predict how small the hypercube must be (or how precise an approximation contained in the hypercube must be) in order to make a correct decision. I show that, provided that the exact RUR is used for representing algebraic points, such precision can be calculated via the root bound approach to exact sign determination of algebraic numbers.

Proposition 4.19. *Assume a real algebraic point x is specified as either*

(1) a tuple of algebraic numbers, or

(2) an intersection of algebraic curves defined as a common root of some polynomials with rational coefficients, or

(3) the result of a vector space addition/subtraction applied to other algebraic points, or

(4) the product of a rational number and another algebraic point.

In any case, there exist rational functions $e_1, \ldots, e_n$ with rational coefficients and algebraic numbers $\zeta_1, \ldots, \zeta_m$ each of which is specified as some root of some univariate polynomial with rational coefficients such that

$$x = (e_1 (\zeta_1, \ldots, \zeta_m), \ldots, e_n (\zeta_1, \ldots, \zeta_m)) . \tag{4.5}$$

Proof. The first case where x is specified as a tuple of algebraic numbers is straightforward.

For the case (2), the exact RUR provides this representation.

The case (3) and (4) are obtained recursively in an obvious way. □

I present here some algorithms to support exact manipulation of algebraic points in some space. These algorithms are referred to later in Section 4.3.

Proposition 4.20. *Let f be a polynomial in n variables with rational coefficients and let x be an n-dimensional algebraic point x specified as in Proposition 4.19. Then, whether or not $f(x) = 0$ is tested exactly.*

Proof. By Proposition 4.19, x is expressed as

$$x = \left(e_1\left(\zeta_1, \ldots, \zeta_m\right), \ldots, e_n\left(\zeta_1, \ldots, \zeta_m\right)\right). \tag{4.6}$$

where $e_1, \ldots, e_m$ are rational functions with rational coefficients and each of $\zeta_1, \ldots, \zeta_m$ is an algebraic number specified as some root of some univariate polynomial with rational coefficients. Let g be a composition of f and $e_1, \ldots, e_n$:

$$g\left(X_1, \ldots, X_m\right) = f\left(e_1\left(X_1, \ldots, X_m\right), \ldots, e_n\left(X_1, \ldots, X_m\right)\right). \tag{4.7}$$

Then, g is a rational function with rational coefficients. Thus, by Proposition 4.18, the sign of the real algebraic number

$$f\left(x\right) = f\left(e_1\left(\zeta_1, \ldots, \zeta_m\right), \ldots, e_n\left(\zeta_1, \ldots, \zeta_m\right)\right) = g\left(\zeta_1, \ldots, \zeta_m\right) \tag{4.8}$$

is exactly determined via the root bound approach. □

Corollary 4.21. *Let x and y be algebraic points. Then, the query whether or not x and y are identical is determined exactly.*

Proof. Define

$$f_i\left(X_1, \ldots, X_n\right) = X_i, \qquad i = 1, \ldots, n. \tag{4.9}$$

That is, f_i just extracts the i-th coordinate of a point. Then, x and y are identical iff

$$f_1\left(x - y\right) = \cdots = f_n\left(x - y\right) = 0. \tag{4.10}$$

The last equalities are exactly examined by Proposition 4.20. □

4.3 Degeneracy Detection in Geometric Solid Modeling

This section describes how the techniques introduced in the previous section is applied to an actual geometric problem, using degeneracy detection appearing in boundary evaluation of solid objects as an example.

4.3.1 Boundary Evaluation

Consider a solid modeling system that computes the boundary representation (b-rep) of a solid object in $\mathbb{R}^3$ given as a Constructive Solid Geometry (CSG) tree of solid objects. Leaves of a CSG tree correspond to "primitive" solid objects such as cubes, spheres, cylinders, etc., and each internal node is marked by one Boolean operation (union, intersection, or difference). We traverse a CSG tree from the bottom up and, at each internal node, perform *boundary evaluation*, meaning we find a representation of the boundary of the solid object formed by applying the Boolean operation on the solid objects represented by the two child subtrees. For every pair of surface patches of these two solid objects, we compute their intersection to form the surface patches of the resulting solid object. In order to make a geometric modeling system robust,

(1) exact computation is used to eliminate numerical errors, and

(2) degenerate configurations of the input solid objects must be handled.

Towards goal (2), first degeneracies must be detected. This degeneracy detection appearing in boundary evaluation is our focus here. Degeneracies are found by checking for irregular interaction of surface patches.

Any surface of the resulting solid object is a part of some surface of the input solid objects. Surfaces of the input solid objects are described parametrically by (triples of) rational functions with real coefficients. They are also described implicitly by (the zero sets of) trivariate polynomials with real coefficients. Implicit representations are either given or computed on demand.

In this section, we assume that surfaces of the input solid objects are given implicitly by polynomials with rational coefficients. Then, trimming curves and intersection curves are represented implicitly by bivariate polynomials with rational coefficients in the surface patch domains. Thus, an algebraic point we are interested in is described as a common root of two bivariate polynomials with rational coefficients.

This assumption may sound too strict or unrealistic. One may say that the input data potentially have some errors. This is true, but we choose to assume *some* interpretation of the data on which to compute. To represent all error in the input would result in a vague or fuzzy representation that, under some circumstances, will not have a well-defined answer. We choose to have a well-defined operation, at the possible cost of making a bad initial interpretation.

A second objection is that in order to describe some solid objects exactly, one might need irrational coefficients. For example, suppose that the designer intends to rotate a cube by 30 degrees. Then, even if all the corners of the cube at the initial position are given by rational numbers, after rotation, the corners will have irrational coordinates, and the surface representations will also have irrational coefficients. This is a real issue, requiring other means to address, but we point out that in many systems this is not the way rotations are specified. Instead, a transformation matrix is given, the entries of which are usually given as rational numbers (often floating-point data that has already been rounded from the irrational number). Thus, our representations using rational coefficients will directly reflect the input. Also, we point out that for any given rotation, we can find a rotation matrix with rational entries that is arbitrarily close to the given rotation.

In the rest of this section, we first identify irregular interactions of surface patches, restate the problems in terms of basic geometric operations, and then describe how to detect them.

4.3.2 Degeneracy Detection

Degeneracies are detected during the boundary evaluation process by checking for irregular interactions. In Table I in Section 2.1.3.4, possible degeneracies appearing in boundary evaluation are enumerated according to how surfaces, curves and points interact. I now describe, for each type of degeneracy, how it is detected.

In this section, by $F_2|_{F_1}$, we denote the intersection curve of surfaces F_1 and F_2 within the surface patch domain for F_1.

(1) Two surfaces meet but not along a curve

  (a) Two surfaces overlap:

  The degeneracy may be detected by checking whether or not their implicit forms share some common factors of positive degree. Alternatively and more efficiently, this degeneracy is detected when substitution of the parametric form of one surface into the implicit form of the other surface causes the implicit form to vanish.

  (b) Two surfaces are tangent along a curve:

  There are two subcases.

    i. When the tangency is along the entire intersection curve of two surfaces (i.e. they never cross), the degeneracy cannot be seen within either of these surface patch domains. Thus, this case (and only this case) cannot be detected locally in the patch domains. It *can* be detected by a more global view examining the relative positions of the patches within each solid object.

ii. When the tangency is at only an isolated point (or points), the point of tangency becomes a cusp or a self-intersection of the intersection curve in either of the surface patch domains.

(c) Two surfaces are tangent at a point:

In this case, their intersection curve shrinks to an isolated point within either of the surface patch domains. (e.g., $S^2 + T^2 = 0$)

(d) A curve is tangent to a surface at a point:

Suppose that a curve C_1 of a solid object S_1 is tangent to a surface F_2 of a solid object S_2. Let C_1 be the border between surfaces F_{11} and F_{12} of S_1. Within the surface patch domain for F_{11}, where C_1 is one of the trimming curves, the intersection curve $F_2|_{F_{11}}$ intersects with C_1 tangentially.

Conversely, within some surface patch domain, if some intersection curve shrinks to an isolated point, or has some cusps or self-intersections, or intersects with some trimming curve tangentially, then a degeneracy of one of the above three cases must occur.

(2) Three surfaces meet at a curve but not a point

(a) A curve lies on a surface:

Suppose that a curve C_1 of a solid object S_1 lies on a surface F_2 of a solid object S_2. Let C_1 be the border between surfaces F_{11} and F_{12} of S_1. Within the surface patch domain for F_{11}, where C_1 is one of the trimming curves, the intersection curve $F_2|_{F_{11}}$ and C_1 overlap. Whenever the degeneracy occurs, some trimming curve and some intersection curve have some non-trivial common factor.

(b) Curves overlap:

Suppose that a curve C_1 of a solid object S_1 and a curve C_2 of a solid object S_2 overlap. Let C_1 be the border between surfaces F_{11} and F_{12} of S_1. Also, let C_2 be the border between surfaces F_{21} and F_{22} of S_2. Within the surface patch domain for F_{11}, where C_1 is one of the trimming curves, C_1 and the intersection curve $F_{21}|_{F_{11}}$ overlap (and C_1 and $F_{22}|_{F_{11}}$ overlap). As in the previous case, whenever the degeneracy occurs, some trimming curve and some intersection curve have some non-trivial common factor.

Conversely, within some surface patch domain, if some trimming curve and some intersection curve have some non-trivial common factor, then a degeneracy of either of the above two cases must occur.

(3) Four or more surfaces meet at a point.

(a) A point lies on a surface:

Suppose that a point P_1 of a solid object S_1 lies on a surface F_2 of a solid object S_2. There exist surfaces F_{11} and F_{12} of S_1 such that P_1 is one of the endpoints of the border C_1 between F_{11} and F_{12}. Within the surface patch domain for F_{11}, C_1 meets the intersection curve $F_2|_{F_{11}}$ at P_1. Thus, whenever the degeneracy occurs, one of the endpoints of some trimming curve lies on some intersection curves.

(b) Curves intersect tangentially:

Suppose that a curve C_1 of a solid object S_1 and a curve C_2 of a solid object S_2 intersect tangentially. Let C_1 be the border between surfaces F_{11} and F_{12}

of S_1. Also, let C_2 be the border between surfaces F_{21} and F_{22} of S_2. Within the surface patch domain for F_{11}, where C_1 is one of the trimming curves, the point, where C_1 and the intersection curve $F_{21}|_{F_{11}}$ meet, and the point, where C_1 and the intersection $F_{22}|_{F_{11}}$ meet, are identical.

Conversely, within some surface patch domain, if some trimming curve meets two (or more) intersection curves at the same point, then the degeneracy of this type must occur.

(c) A point lies on a curve:

Suppose that a point P_1 of a solid object S_1 lies on a curve C_2 of a solid object S_2. There exist surfaces F_{11} and F_{12} of S_1 such that P_1 is one of the endpoints of the border C_1 between F_{11} and F_{12}. Let C_2 be the border between surfaces F_{21} and F_{22} of S_2. Within the surface patch domain for F_{11}, C_1 meets the intersection curve $F_{21}|_{F_{11}}$ at P_1. Thus, whenever the degeneracy occurs, one of the endpoints of some trimming curve lies on some intersection curves.

(d) Points coincide:

Suppose that a point P_1 of a solid object S_1 lies on a point P_2 of a solid object S_2. There exist surfaces F_{11} and F_{12} of S_1 such that P_1 is one of the endpoints of the border C_1 between F_{11} and F_{12}. Within the surface patch domain for F_{11}, C_1 meets the intersection curve $F_{21}|_{F_{11}}$ at P_1. Thus, whenever the degeneracy occurs, one of the endpoints of some trimming curve lies on some intersection curves.

Conversely, within some surface patch domain, if some trimming curve meets some intersection curve at one of the endpoints of the trimming curve, then the degeneracy of one of the above three cases must occur.

To summarize, except the case "Surfaces overlap," that is detected in $\mathbb{R}^3$, and the first realization of the case "Surfaces are tangent along a curve," that must be handled at higher level, degeneracies are detected locally within each surface patch domain.

Proposition 4.22. *Degeneracies listed in Table I in Section 2.1.3.4 except "Surfaces overlap" and "Surfaces are tangent along a curve" are detected by checking, within each surface patch domain, whether or not*

(1) some intersection curve shrinks to an isolated point, or

(2) some intersection curve has some cusps or self-intersections, or

(3) some intersection curve intersects with some trimming curve tangentially, or

(4) some trimming curve and some intersection curve have some non-trivial common factor, or

(5) some trimming curve meets two (or more) intersection curves at the same point, or

(6) some trimming curve meets some intersection curve at one of the endpoints of the trimming curve.

4.3.3 Degeneracy Detection (Algebra)

In this section, the occurrences of degeneracies found in Proposition 4.22 are restated in terms of algebra.

Recall that, within each surface patch domain, a curve (a trimming curve or an intersection curve) is described implicitly by bivariate polynomials with rational coefficients, and a point is described as an intersection of curves, and thus, a common root of a pair of bivariate polynomials with rational coefficients.

The cases (1) and (2) in Proposition 4.22 can be detected by resolving the topology of an algebraic planar curve. Note that some curve might shrink to a single point. Several algorithms for resolving the topology of an algebraic planar curve are known. We used the algorithm introduced in [59], but with the new algorithm for finding common roots of a pair of bivariate polynomials with rational coefficients, so that the algorithm does not fail at cusps or self-intersections, or tangential intersections which are the points we are particularly interested in.

The case (3) actually fits more in the cases (5) and (6), with considering the topology also. First, find the intersections and then, determine if each of them are tangential by looking at topology.

The case (4) can be detected by testing whether or not a pair of algebraic curves in $\mathbb{R}^2$ share some non-trivial components. The test, in general, reduces to computing the GCD of a pair of bivariate polynomials with rational coefficients. It can be shown that a pair of algebraic curves in $\mathbb{C}^2$ share some non-trivial components iff their defining polynomials have a non-constant GCD. In $\mathbb{R}^2$, the statement is no longer valid since curves may share some positive dimensional components in $\mathbb{C}^2$ on which there are only finitely many real points. In the worst case, we use a 2-dimensional Sturm query [71] which tells us the number of real zeros in a given region. Fortunately, for the bivariate case, an efficient algorithm for computing the GCD of a pair of polynomials [97] is known. Thus, testing “non-degeneracy” is relatively easy. Furthermore, for a certain type of curves, we can immediately tell whether or not they have finitely many zeros (e.g. planes). Only those instances which remain to be suspicious after all these tests are applied would be passed to the costly test.

The remaing cases ((5) and (6)) reduce to the primitive geometric computations within the 2-dimensional surface patch domain, namely, testing

(a) whether or not a pair of 2-dimensional algebraic points are identical, and

(b) whether or not a 2-dimensional algebraic point lies on an algebraic planar curve.

The trimming curves and the intersection curves are algebraic planar curves described implicitly by polynomials with rational coefficients. Every algebraic point is (initially) specified as an intersection of a pair of algebraic curves, that is, a common root of a pair of bivariate polynomials with rational coefficients. Thus, our task can be restated as follows:

Given a pair of algebraic planar curves C_1 and C_2 specified implicitly by bivariate polynomials $f_1(S,T)$ and $f_2(S,T)$ with rational coefficients and some region $[L_S, H_S] \times [L_T, H_T]$ with rational corners, we would like to compute all the intersections of C_1 and C_2 in the region.

Without loss of generality, we may assume that the zero set of the square system of bivariate polynomials f_1 and f_2 is zero-dimensional (otherwise, it must be treated as the above). Thus, the RUR for the zero set of (f_1, f_2) in $(\mathbb{R}^*)^n$ will generate all the points in $(\mathbb{R}^*)^n$.

In order to see if there are any points with 0-coordinate, we test whether univariate polynomials $f_1(0,T)$ and $f_2(0,T)$ have some common roots and whether univariate polynomials $f_1(S,0)$ and $f_2(S,0)$ have some common roots.

Thus, all intersections are represented as in Proposition 4.19.

Next, using a root bound approach, we can exclude those roots outside of the region $[L_S, H_S] \times [L_T, H_T]$. Note that this is done after all roots are found, i.e. we cannot limit the roots found to the domain at an earlier point in the computation (as we could with prior, less general, methods).

By Proposition 4.20 and its corollary, we can test whether or not a pair of 2-dimensional algebraic points are identical and whether or not a 2-dimensional algebraic point lies on an algebraic planar curve. This has been described in section 4.2.

4.3.4 Example

Suppose we would like to intersect a cube and a torus shown.This configuration suffers from degeneracy *Two Surfaces are Tangent at a Point.* One surface of the cube and one surface of the torus touch each other at a single point. On the surface of the cube, the intersection curve is described by bivariate polynomial

$$\begin{aligned} f = \quad & 36X^4 - 72X^3 + 72X^2Y^2 \\ & - 72X^2Y + 64X^2 - 72XY^2 + 72XY \\ & - 28X + 36Y^4 - 72Y^3 + 80Y^2 \\ & - 44Y + 9 \end{aligned} \tag{4.11}$$

(with some coordinate system). In order to resolve the topology of the curve, the system tries to find points where f and its partial derivative f_X both vanish.

$$\begin{aligned} f_X = \quad & 144X^3 - 216X^2 + 144XY^2 - 144XY \\ & + 128X - 72Y^2 + 72Y - 28. \end{aligned} \tag{4.12}$$

One of the solutions is $(2, 2)$ which is the self intersection of the curve, which is not found by some systems, e.g. MAPC [59], but it is found using the RUR. See Section 6.1.3.

CHAPTER V

NUMERICAL PERTURBATION

This chapter describes the exact numerical perturbation scheme for resolving degeneracies appearing in geometric computation. In Section 5.1, a formal description of the exact numerical perturbation scheme is presented. In Section 5.2, examples of numerical perturbation is shown.

5.1 Exact Numerical Perturbation

In this section, I present a formal description of exact numerical perturbations analogous to that for symbolic perturbations discussed by Seidel [90] [91], and I discuss issues related to implementation of exact numerical perturbation.

Throughout this chapter, write $\mathbb{R}^*$ for $\mathbb{R} \setminus \{0\}$.

5.1.1 Degeneracies

I first give a formal definition of degeneracies.

Consider an algorithm A and let F be a function from some input space $\mathcal{I}$ to some output space $\mathcal{O}$ that is computed by A. We assume that some topology is introduced to both $\mathcal{I}$ and $\mathcal{O}$. For simplicity, I will consider only the case $\mathcal{I}$ is $\mathbb{R}^M$ with the Euclidean topology.

For example, consider the function CHS that computes the sequence of indices of vertices of the Convex Hull of a given sequence of m points $(x_1, y_1), \ldots, (x_m, y_m)$ in $\mathbb{R}^2$. The input space for CHS is the Euclidean space $\mathbb{R}^{2m}$ where the sequence of m points is encoded to $2m$-tuple $(x_1, y_1, \ldots, x_m, y_m)$. The output space for CHS is the set of all the permutations of n distinct integers from $\{1, \ldots, m\}$ where $1 \leq n \leq m$

and the discrete topology is introduced. On the other hand, for the function CHA that computes the area of the Convex Hull of a given set of $(x_1, y_1), \ldots, (x_m, y_m)$ in $\mathbb{R}^2$, $\mathcal{I} = \mathbb{R}^{2m}$ and $\mathcal{O} = \mathbb{R}$ both with the Euclidean topology.

A computation of an algorithm for a function $F : \mathbb{R}^M \to \mathcal{O}$ is modeled by a ternary tree T called an *extended algebraic decision tree* [78]. Each interior node v of T is associated with the predicate $f_v : \mathbb{R}^M \to \mathbb{R}$ and its three branches are labeled by -1, 0 and 1, respectively. Each leaf v of T is associated with the result function $g_v : \mathbb{R}^M \to \mathcal{O}$. For simplicity, I assume that, for every internal node (or leaf) v of T, the predicate f_v is continuous at every input x that reaches v.

In this model, $F(x)$ is computed by a traversal of the tree T from the root to one of the leaves. At each internal node v of T, $s_v(x) = \operatorname{sgn}(f_v(x))$ is evaluated and the branch labeled by $s_v(x)$ is taken. When a leaf v of T is reached, $g_v(x)$ is evaluated and returned.

An input $x \in \mathbb{R}^M$ is said to be *degenerate* (for F) if there exists an internal node v of T such that $s_v(x) = \operatorname{sgn}(f_v(x))$ becomes 0, that is, there exists some predicate that evaluates to 0 at x.

Let $x = (x_1, y_1, \ldots, x_m, y_m) \in \mathbb{R}^{2m}$ where, among all m points encoded to x, 3 or more of them are collinear. Then, x is degenerate for both CHS and CHA. Note that CHA is a continuous function while CHS is not. In fact, CHA is continuous even at x while CHS is discontinuous at x. Note that CHS is continuous at x if x is not degenerate.

5.1.2 Perturbations

In this section, I give a formal definition of symbolic and numeric perturbations.

First, I define (linear) symbolic and numerical perturbations.

For every input $x = (x_1, \ldots, x_M) \in \mathbb{R}^M$, define a *(linear) symbolic perturbation*

π_x of x to be the ray starting at x and going to some direction $d_x = (d_{x1}, \ldots, d_{xM}) \in (\mathbb{R}^*)^M$. More formally,

$$\begin{aligned} \pi_x : \quad [0, \infty) \ni \epsilon \quad & \mapsto \\ & x + \epsilon d_x = (x_1 + \epsilon d_{x1}, \ldots, x_M + \epsilon d_{xM}) \in \mathbb{R}^M. \end{aligned} \tag{5.1}$$

For every input $x = (x_1, \ldots, x_M) \in \mathbb{R}^M$, define a *(linear) numerical perturbation* ψ_x of x to be the point obtained by perturbing x by d_x for some *deviation* $d_x = (d_{x1}, \ldots, d_{xM}) \in (\mathbb{R}^*)^M$. More formally,

$$\psi_x = x + d_x = (x_1 + d_{x1}, \ldots, x_M + d_{xM}) \in \mathbb{R}^M. \tag{5.2}$$

Next, I define (linear) symbolic and numerical perturbation schemes.

For every function $F : \mathbb{R}^M \to \mathcal{O}$, a *symbolic perturbation scheme* Π assigns a symbolic perturbation π_x to every input $x \in \mathbb{R}^M$.

Similarly, for every function $F : \mathbb{R}^M \to \mathcal{O}$, a *numerical perturbation scheme* Ψ assigns a numerical perturbation ψ_x to every input x.

Immediately from the definition (5.2) above, it is clear that a (linear) numerical perturbation of x is totally specified by giving a deviation d_x. Thus, assigning a (linear) numerical perturbation ψ_x to an input x means associating a deviation d_x with x. Note that a (linear) symbolic perturbation of x is totally specified just by giving a direction d_x. Of course, if d'_x is a direction and $d_x = cd'_x$ for some positive real number c, then d'_x specifies the same (linear) symbolic perturbation as d_x does.

Now, I define and consider (linear) symbolic and numerical perturbed functions.

Given a function $F : \mathbb{R}^M \to \mathcal{O}$ and a symbolic perturbation scheme Π, define a *symbolically perturbed function* of F to be the function $\overline{F}^{\Pi} : \mathbb{R}^M \to \mathcal{O}$ such that

$$\overline{F}^{\Pi}(x) = \lim_{\epsilon \to 0+} F(\pi_x(\epsilon)), \qquad \forall x \in \mathbb{R}^M. \tag{5.3}$$

We assume that a perturbed function is well-defined, i.e., the limit appearing in the RHS of (5.3) exists.

Given a function $F : \mathbb{R}^M \to \mathcal{O}$ and a numerical perturbation scheme Ψ, define a *numerically perturbed function* of F to be the function $\widetilde{F}^{\Psi} : \mathbb{R}^M \to \mathcal{O}$ such that

$$\widetilde{F}^{\Psi}(x) = F(\psi_x), \qquad \forall x \in \mathbb{R}^M. \tag{5.4}$$

We write $\overline{F}$ or $\widetilde{F}$ for the perturbed function whenever the perturbation scheme is clear from the context.

Note that the perturbed function *does* depend on the perturbation scheme. That is, for two different symbolic perturbation schemes Π and Π', the symbolically perturbed functions $\overline{F}^{\Pi}$ and $\overline{F}^{\Pi'}$ are not always the same. Also, for two different numerical perturbation schemes Ψ and Ψ', the numerically perturbed functions $\widetilde{F}^{\Psi}$ and $\widetilde{F}^{\Psi'}$ are usually different. However, the way symbolically pertubed functions depend on symbolic perturbation schemes and the way numerically perturbed functions depened on numerical pertubation schemes are very different.

How does F and $\overline{F}$ or $\widetilde{F}$ relate? There is a simple, obvious but useful lemma:

Lemma 5.23. Seidel [90] [91] *For any symbolic perturbation scheme Π, if F is continuous at $x \in \mathbb{R}^M$ then $\overline{F}^{\Pi}(x) = F(x)$.*

Note that the statement in the lemma holds even if there is a degeneracy at x.

If F is not continuous at x, then there is not much we can say about the relationship between $\overline{F}^{\Pi}(x)$ and $F(x)$.

One immediate consequence of Lemma 5.23 is another fact about the difference between symbolic and numerical perturbations regarding the dependency of perturbed functions on perturbation schemes. When F is continuous at x, $\overline{F}^{\Pi}(x) = \overline{F}^{\Pi'}(x)$ for any pair of symbolic perturbation schemes Π and Π'. On the other hand, for two different numerical perturbations schemes Ψ and Ψ', $\widetilde{F}^{\Psi}(x)$ and $\widetilde{F}^{\Psi'}(x)$ are usually different.

For example, for an arbitrary but fixed input x, $CHA(x) = \overline{CHA}(x)$. If, among all m points encoded to x, 3 or more of them on the convex hull are collinear then CHS is discontinuous at x and $CHS(x) \neq \overline{CHS}(x)$. If x is not degenerate for CHS then CHS is continuous at x and $CHS(x) = \overline{CHS}(x)$. On the other hand, for an arbitrary but fixed input x, $CHA(x) \neq \widetilde{CHA}(x)$. If x is degenerate for CHS then $CHS(x) \neq \widetilde{CHS}(x)$. If x is not degenerate for CHS then CHS is continuous at x and $CHS(x) = \widetilde{CHS}(x)$.

Note, in this case, $CHS(x)$ happens to be a subsequnce of $\overline{CHS}(x)$, and with relatively simple post-processing, $CHS(x)$ is recovered from $\overline{CHS}(x)$. However, post-processing is usually not easy, or even worse, impossible. The relation between $F(x)$ and $\overline{F}(x)$ is not always clear.

We have seen (Section 5.1.1) that, for a function $F : \mathbb{R}^M \to \mathcal{O}$ and an input $x \in \mathbb{R}^M$, the computation of $F(x)$ is modeled by a traversal of some algebraic decision tree T from the root to a leaf.

The symbolically perturbed function $\overline{F}(x)$ is computed by a traversal of the same tree T from the root to one of its leaves. At each internal node v of T, instead of $s_v(x) = \operatorname{sgn}(f_v(x))$, $\overline{s}_v(x) = \lim_{\epsilon\to 0+} \operatorname{sgn}(f_v(\pi_x(\epsilon)))$ is evaluated and used to take the branch. When a leaf v of T is reached, instead of $g_v(x)$, $\overline{g}_v(x) = \lim_{\epsilon\to 0+} \operatorname{sgn}(g_v(\pi_x(\epsilon)))$ is evaluated and returned.

Similarly, the numerically perturbed computation $\widetilde{F}(x)$ is modeled by a traversal of T. At each internal node v of T, instead of $s_v(x) = \operatorname{sgn}(f_v(x))$, $\widetilde{s}_v(x) = \operatorname{sgn}(f_v(\psi_x))$ is evaluated and used to take the branch to the appropriate subtree. When a leaf v of T is reached, instead of $g_v(x)$, $\widetilde{g}_v(x) = \operatorname{sgn}(g_v(\psi_x))$ is evaluated and returned.

Let $f : \mathbb{R}^M \to \mathbb{R}$ be a continuous function, and let $x \in \mathbb{R}^M$. A symbolic perturbation π_x is said to be *valid* for f iff $\lim_{\epsilon \to 0+} \operatorname{sgn}(f(\pi_x(\epsilon)))$ exists and is not zero. Similarly, a numeric perturbation ψ_x is *valid* for f iff $f(\psi_x) \neq 0$. A (symbolic or numerical) perturbation scheme is said to be *valid* for f iff, for all $x \in \mathbb{R}^M$, the (symbolic or numerical) perturbation of x is valid.

Let T be the extended algebraic decision tree for an algorithm for computing a function $F : \mathbb{R}^M \to \mathcal{O}$. Furthermore, let $\mathcal{F}$ be the set of all the predicates appearing in the tree. Recall that we have assumed that the predicates are all continuous. Suppose Π is a symbolic perturbation scheme that is valid for all the predicates in $\mathcal{F}$. Then, during the computation of $\overline{F}^{\Pi}(x)$, no branch labeled 0 will be taken. Thus, in order to compute $\overline{F}^{\Pi}$ for every input x, we do not need to implement some (or all) of those 0-branches, i.e., the program does not have to deal with degenerate cases. Similarly, if Ψ is a numerical perturbation scheme that is valid for all the predicates in $\mathcal{F}$ then we do not need to implement those 0-branches.

Thus, the implementation of an algorithm for computing $\overline{F}$ or $\widetilde{F}$ is simpler, in terms of a traversal of the algebraic decision tree, than the implementation of an algorithm for computing F because the branches for handling "degenerate inputs" can be omitted. Of course, other aspects of the computation may be more complicated.

The above arguments are summarized into the following two theorems; Theorem 5.24 is for symbolic perturbation and is proven in [90] [91] and Theorem 5.25 is its numerical counterpart.

Theorem 5.24. Seidel [90] [91]

Let T be an extended algebraic decision tree that computes a function $F : \mathbb{R}^M \to \mathbb{R}^N$. Furthermore, let Π be a symbolically valid perturbation scheme for all the predicates appearing at internal nodes of T. Then

(1) A symbolically perturbed traversal of T computes the symbolically perturbed function $\overline{F}^{\Pi}$.

(2) If F is continuous at $x \in \mathbb{R}^M$ then $\overline{F}^{\Pi}(x) = F(x)$.

(3) The above statements remain true even if some (or all) of the 0-branches of T are removed.

Theorem 5.25. *Let T be an extended algebraic decision tree that computes a function $F : \mathbb{R}^M \to \mathcal{O}$. Furthermore, let Ψ be a numerically valid perturbation scheme for all the predicates appearing in T. Then*

(1) A numerically perturbed traversal of T computes the numerically perturbed function $\widetilde{F}^{\Psi}$.

(2) If F is continuous at $x \in \mathbb{R}^M$, then, for any $\delta > 0$, there exists a numerical perturbation $\psi_x = x + d_x$ such that $\left|\widetilde{F}^{\Psi}(x) - F(x)\right| < \delta$.

(3) The above statements remain true even if some of (or all) the 0-branches of T are removed.

Proof. (1) Described above.

(2) Fix any $\delta > 0$. Since F is continuous at $x \in \mathbb{R}^M$, $\exists y \in \mathbb{R}^M$ with $y \neq x$ s.t. $|F(y) - F(x)| < \delta$. Set $d_x = y - x$. Then, $d_x \in (\mathbb{R}^*)^M$. Now, define the numerical perturbation ψ_x at x to be the point $x + d_x = y$ so that $\widetilde{F}^{\Psi}(x) = F(\psi_x) = F(x + d_x) = F(y)$.

(3) Described above.

□

The statement (2) in Theorem 5.24 says that, provided F is continuous at x, the output of the original problem is obtained by the perturbed computation without post-processing.

On the other hand, the statement (2) in Theorem 5.25 will be understood as follows:

The result of the numerically perturbed function is truly different from the result of the original function. However, the measure of the difference can be as small as desired.

5.1.3 Directions and Amount of Perturbation

We would like to construct a valid (symbolic or numerical) perturbation. Recall that we have seen that a symbolic or numerical perturbation is assigned to an input by giving a direction or deviation vector. In a symbolic perturbation, the amount of perturbation is symbolic and the limit when this symbolic amount approaches to 0 will be taken. In a numerical perturbation, the deviation vector specifies both the direction and the amount of perturbation. In this section, I show that if the direction of a (symbolic or numerical) linear perturbation is randomly chosen, the perturbation is almost always valid. I also show that determining the validity of a random numerical perturbation is much more efficient than for symbolic.

Let $f : \mathbb{R}^M \to \mathbb{R}$ be a continuous function. Furthermore, let $\pi_x = x + \epsilon d_x$ be a linear symbolic perturbation of an input $x \in \mathbb{R}^M$. Recall that π_x is the ray starting at x and going in the direction d_x. Then, π_x is invalid for f if any open initial segment of the ray π_x intersects with the set $f^{-1}(0) \subseteq \mathbb{R}^M$. This occurs when either

(1) a linear ray π_x entirely lies on the set $f^{-1}(0)$, or

(2) f is "amorphous, " meaning that, in any open initial segment of the ray π_x, the values of f become both zero and non-zero infinitely many times.

Since (2) is not the case for almost all test functions appearing in geometric computation, we will ignore this case hereafter.

A linear symbolic perturbation scheme is totally specified by giving its direction at every input. The following theorems tell us the probability that a symbolic perturbation fails to be valid when the direction at every input is chosen "uniformly at random."

Theorem 5.26. Seidel [91]

Let f be a multivariate polynomial of total degree at most D, and let B_f be a black box algorithm computing f. Let π_x be a linear symbolic perturbation of x that is valid for f. Then, $\lim_{\epsilon \to 0+} \operatorname{sgn}(f(\pi_x(\epsilon)))$ can be determined using at most $D+1$ calls to B_f plus some small overhead.

Theorem 5.27. Seidel [91]

Let T be an extended algebraic decision tree with a set $\mathcal{F}$ of predicates, each a multivariate polynomial of total degree at most D, and let $x \in \mathbb{R}^M$ be an arbitrary but fixed input to T. Furthermore, let R be a finite set of real numbers. If the direction d_x is chosen uniformly at random from $R^M \backslash \{0\}$, then the linear perturbation $\pi_x = x + \epsilon d_x$ fails to be valid with probability at most $\frac{D|\mathcal{F}|}{|R|}$.

Suppose that the direction of a linear perturbation of every input is chosen at random. By Theorem 5.27, we are very unlikely to choose an invalid perturbation direction. Thus, we will use the following strategy: In the unlikely event that a chosen perturbation direction d_x turns out to be bad, we can simply restart the computation using a new direction for d_x.

Next, I show analogous results for numerical perturbations.

A linear numerical perturbation scheme is totally specified by its deviation at every input. The question is what is the probability that a numerical perturbation fails to be valid when the deviation at every input is chosen "uniformly at random?"

Theorem 5.28. *Let f be a multivariate polynomial, and let B_f be an algorithm that evaluates f. Let ψ_x be a linear numercial perturbation of x that is numerically valid for f. Then, we can determine whether or not $f(\psi_x) = 0$ using just* one *call to B_f together with some small overhead.*

Proof. The query whether or not $f(\psi_x) = 0$ is determined by evaluating f at $\psi_x = (x_1 + d_{x1}, \ldots, x_M + d_{xM})$ once. $\square$

Notice that, comparing to the result for symbolic perturbations in Theorem 5.26, we find that numerical perturbation can simplify computation drastically.

Theorem 5.29. *Let T be an extended algebraic decision tree with a set $\mathcal{F}$ of predicates, each is a multivariate polynomial of total degree at most D, and let $x \in \mathcal{I}$ be an arbitrary but fixed input to T. Furthermore, let R be a finite set of real numbers. If the deviation d_x is chosen uniformly at random from R^M then the linear numerical perturbation*

$$\psi_x = (x_1 + d_{x1}, \ldots, x_M + d_{xM})$$

fails to be valid with probability at most $\frac{D|\mathcal{F}|}{|R|}$.

Proof. It suffices to show that, for every $f \in \mathcal{F}$, when the coordinates $d_{x1}, \ldots, d_{xM}$ of d_x are randomly chosen from R^M, the probability that ψ_x is invalid is $\frac{D}{|R|}$.

The perturbation ψ_x is invalid whenever $f(\psi_x) = 0$. Now, $h(x_1, \ldots, x_M, d_{x1}, \ldots, d_{xM}) = f(\psi_x)$ is a multivariate polynomial in variables x_i and d_{xi} of total degree at most D. By the Schwartz-Zippel lemma [89], when d_x are randomly chosen from R^M, the probability $h(x_1 \ldots, x_M, d_{x1}, \ldots, d_{xM})$ vanishes is at most $\frac{D}{|R|}$. $\square$

By Theorem 5.29, we see that there are not many invalid numerical perturbations. In the unlikely event that chosen deviation d_x turns out to be bad, we can simply abort and restart the computation with a new deviation d_x and, almost always, the corresponding perturbation becomes valid.

5.1.4 Issues of Implementation

There are several issues that must be dealt with in order to move from the theory of exact numerical perturbation to a practical implementation.

5.1.4.1 Directions and Amount of Perturbation

Recall that assigning a (linear) symbolic perturbation ψ_x to an input x means associating a direction d_x with x while assigning a (linear) numerical perturbation of x means associating is the direction and the amount of a deviation d_x.

By Theorem 5.29, a random perturbation almost always works, i.e., if the deviation is a (linear) numerical perturbation is chosen at random, with the probability 1, it will be valid. However, not all deviations are considered "good." We would like to choose the deviation of a perturbation that is likely to capture the intent of the designer. The choice of a "good" deviation will depend heavily on the problem itself. For example, earlier work on boundary evaluation [96] [36] [75] uses an expansion and contraction operation on basic primitives to achieve a "good" perturbation. Since an appropriate choice of deviation is problem-dependent, I cannot propose a single solution, but rather describe below (Section 5.2), a couple of general methods for choosing deviations.

It is possible that the deviation chosen in order to capture designer's intent might not remove degeneracies. An example is seen in Figure 9, where the degeneracy remains if both objects are perturbed through expansion and contraction.

The output of a numerically perturbed computation may have some extra "very small but positive measure" structures. Strictly speaking, the intent of the designer

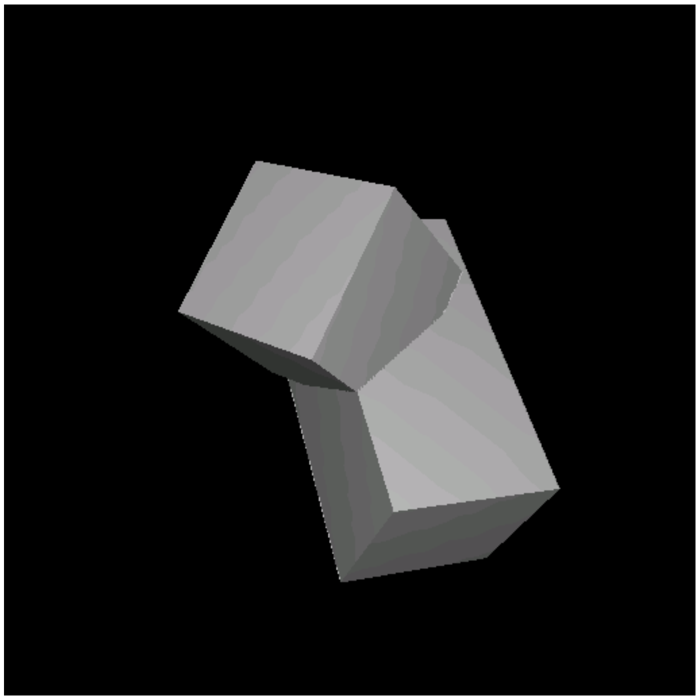


Fig. 9. Example where numerical expansion and contraction lead to undesirable results. Perturbing surfaces inward and outward does not remove the degeneracy.

is lost when numerical perturbation is applied because of these small structures. In many cases, though, those small structures will be no worse than the measure zero structures obtained in symbolic perturbation.

Another issue with the amount of perturbation comes from meeting the validity constraints of the above theorems. We have seen (Section 5.1.3) that a numerical perturbation is valid if the amount of perturbation is so small that any smaller perturbation would not change the sign of any predicate. However, giving a bound on the amount of valid perturbation is not straightforward.

One way of getting around this difficulty is to rely on an idea of a *global tolerance*. That is, we assume that the input geometric data is correct within some amount, τ. A global tolerance is often used to take in to account the inexact nature of real-world data. Any perturbation smaller than the input τ is allowed. As the tolerance value approaches zero, the perturbed surfaces still approach the original surfaces, and thus we can achieve valid exact numerical perturbation. This matches the notion of a *backward-stable* operation: we compute the output of a perturbed version of the original input.

5.1.4.2 Exact Computation

It is important to note that exact computation is necessary for implementing numerical perturbations. Like symbolic perturbations, a perturbed computation relies on exact sign determination of predicates. Also, for numerical perturbations, exact computation is needed because we may have to reduce our perturbation amount to any level desired. Note that the use of exact computation entails a significant cost in efficiency.

5.1.4.3 Symbolic and Numerical Perturbation

Implementing symbolic perturbations is much harder than implementing numerical perturbations.

In order to implement symbolic perturbation, an infinitesimal amount ϵ has to be dealt with. The computation proceeds treating the perturbation amounts as a symbolic variable, and the limit of the perturbation is taken at the end. Several efforts have been made to reduce the amount of computation. For example, the simulation of simplicity scheme [25] applied to the sidedness test reduced to the sign of a polynomial in the infinitesimal amount ϵ as ϵ approaches to zero. This can be found by the sign of the lowest degree non-zero term. Nevertheless, in the implementation of a symbolic perturbation, ϵ must be treated more or less symbolically.

On the other hand, the implementation of a numerical perturbation is straightforward. Only an input must be preprocessed, but the implementation of the algorithm remains the same.

5.2 Examples of Numerical Perturbations

In this section, we describe two simple instantiations of numerical perturbation, "random translation" and "contraction/expansion."

5.2.1 Random Translation

Random translation is a numerical perturbation where input instances are translated by random deviations. The random translation approach fits most easily into the description of exact numerical perturbation outlined above. By Theorem 5.29, with a truly random choice of deviation, with probability 1, any degeneracy will be eliminated.

As an example, consider the problem $\mathcal{P}_1$ to locate four parabolas

$$\begin{cases} f_1 & : & 2X^2 - Y = 0, \\ f_2 & : & X^2 - Y = 0, \\ f_3 & : & -X^2 - Y = 0, \\ f_4 & : & -2X^2 - Y = 0. \end{cases}$$

They are in degenerate configuration; namely, they intersect with each other at a single point, the origin. See Figure 10 a. Suppose a random translation is applied to them as follows:

$$\begin{cases} \widetilde{f_1} & : & \text{translate } f_1 \text{ along } Y\text{-axis by } -\frac{1}{2}, \\ \widetilde{f_2} & : & \text{translate } f_2 \text{ along } X\text{-axis by } -\frac{1}{4}, \\ \widetilde{f_3} & : & \text{translate } f_3 \text{ along } X\text{-axis by } \frac{1}{2}, \\ \widetilde{f_4} & : & \text{translate } f_4 \text{ along } Y\text{-axis by } \frac{1}{4}. \end{cases}$$

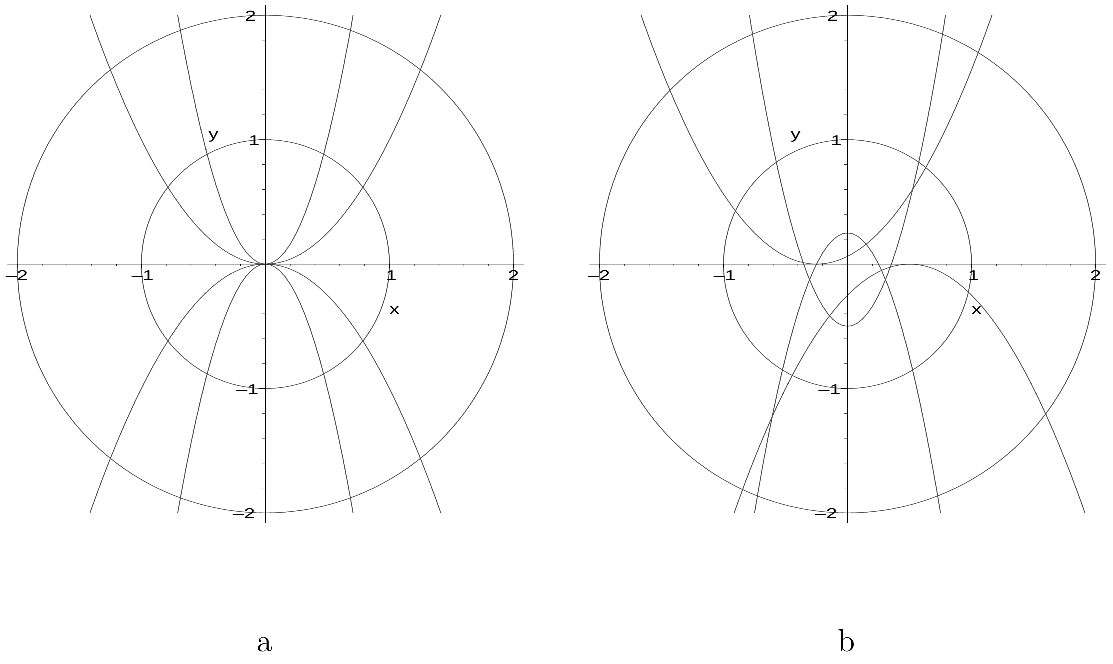


Fig. 10. Four parabolas degenerate and non-degenerate.
a. Four parabolas in degenerate configuration. They intersect with each other at a single point, the origin.
b. Four parabolas randomly translated.

That is, we obtain

$$\begin{cases} \widetilde{f_1} & : \quad 2X^2 - \frac{1}{2} - Y = 0, \\ \widetilde{f_2} & : \quad X^2 + \frac{1}{2}X + \frac{1}{16} - Y = 0, \\ \widetilde{f_3} & : \quad -X^2 + X - \frac{1}{4} - Y = 0, \\ \widetilde{f_4} & : \quad -2X^2 + \frac{1}{4} - Y = 0. \end{cases}$$

(Note, this is not actually a very random perturbation, but it illustrates the idea well.) Then, no three or four of the perturbed parabolas intersect simultaneously at a single point. See Figure 10 b. They are no longer in degenerate configurations. Thus, this random translation is valid.

Now, consider another problem $\mathcal{P}_2$ to locate four parabolas $f_1, \ldots, f_4$ with the circle

$$g_1 : X^2 + Y^2 - 1 = 0.$$

If we travel on g_1 clockwise from the point $(0, 1)$ to the point $(0, -1)$, then we meet those parabolas in order of f_1, f_2, f_3, f_4. On the other hand, we meet the perturbed parabolas in order of $\widetilde{f}_2, \widetilde{f}_1, \widetilde{f}_3, \widetilde{f}_4$. Thus, even though this random translation is valid, it fails to maintain the designer's intent.

If we consider the problem $\mathcal{P}_3$ to locate four parabolas $f_1, \ldots, f_4$ and the circle

$$g_2 : X^2 + Y^2 - 4 = 0$$

then the random translation removed degeneracies and still keeps the order of intersections between parabolas and the circle g_2. Thus, this random translation is valid and also maintains the designer's intent.

In general, it is neither trivial nor easy to maintain the designer's intent with application of random translations. The difference between problems $\mathcal{P}_2$ and $\mathcal{P}_3$ has to do with the size of the deviation of the perturbation relative to the size of the circle being tested (i.e. the larger circle was equivalent to having a smaller perturbation). For a given circle size, the perturbation amount could be reduced to be small enough to guarantee correct ordering of the intersections. Similarly, a given numerical perturbation can always be made small enough (since exact computation is used) to be topologically equivalent to any symbolic perturbation. Determining that amount ahead of time, though, is usually a difficult problem.

5.2.2 Expansion / Contraction

Another type of numerical perturbation described here is expansion / contraction. This method has been shown to be quite useful in the handling of boundary evaluation operations [75].

With expansion / contraction, an input is effectively scaled outward or inward, from the centroid of an object in the input. For simple geometric objects, this is usually straightforward. By adjusting which objects are perturbed inward (contracted) and which outward (expanded), we can capture designer's intent relatively well.

There are a few shortcomings to the expansion and contraction as applied to boundary evaluation:

(1) First, there are a limited number of cases for which expansion or contraction is not sufficient. See Figure 9 in section 5.1.4.1.

(2) Second, it is possible with perturbation schemes to create small, unintended objects. While these will indeed be small (see the statement (2) in Theorem 5.24 in Section 5.1.2), and thus, should not affect the overall topology of the output, they can be annoying to deal with in subsequent computation.

I will describe an implementation of expansion / contraction to the standard CSG primitives in Section 6.2 in Chapter VI.

CHAPTER VI

IMPLEMENTATION

The algorithms described in the previous chapters have been implemented. This chapter explains the details of the implementation and shows some experimental results. In Section 6.1, I describe the implementation of the algorithm for computing the Rational Univariate Reduction of a given system of polynomials with rational coefficients. In Section 6.2, I describe the implementation of a numerical perturbation scheme in order to handle degeneracies appearing in boundary evaluation of solid objects.

6.1 Rational Univariate Reduction (RUR)

This section describes the implementation of the algorithm for computing the Rational Univariate Reduction (RUR) for a given system of polynomials with rational coefficients, i.e., the case $\mathbb{K} = \mathbb{Q}$ of the algorithm introduced in Chapter III.

Since all the algorithms reduce to Algorithm **RUR_toric_square** (see Figure 8, in Section 3.1), here, I discuss the implementation of Algorithm **RUR_toric_square** introduced in Section 3.1.1.1.

The goal is to develop a library for computing the RUR for the zero set of a square system of n polynomials in n variables with rational coefficients such that

(1) the univariate polynomials forming the RUR are derived from the toric resultant for the input system, in particular, the implementation is Groëbner-free,

(2) the RUR is computed *exactly*, meaning that all the rational coefficients of the univariate polynomials $h, h_1, \ldots, h_n$ forming the RUR will be computed to full precision, and

(3) for small n, the library runs in an acceptable amount of time in practice although the main concern here is to guarantee the exactness.

6.1.1 Implementation of **RUR_toric_square**

This section describes an implementation of Algorithm **RUR_toric_square** for computing the RUR for the zero set of a square system of n polynomials $f_1, \ldots, f_n$ in n variables with rational coefficients. The pseudo code and the step-by-step description of the algorithm are given in Section 3.1.1.1 and 3.1.1.2, respectively.

Let A_i be the support of f_i for $i = 1, \ldots, n$. The algorithm sets, at step **1**, the support A_0 of f_0 so that f_0 is a linear polynomial.

Step **7** constructs the toric resultant matrix N for a system of $n+1$ polynomials in n variables with supports $A_0, A_1, \ldots, A_n$. Emiris's incremental algorithm [32] is implemented. This algorithm computes, as byproducts, the convex hulls Q_i of A_i and the quantities MV_{-i} for $i = 0, 1, \ldots, n$ where MV_{-i} is the mixed-volume of $Q_0, Q_1, \ldots, Q_{i-1}, Q_{i+1}, \ldots, Q_n$. The computation of Q_i and the computation of MV_{-i} both reduce to some linear programming problems [32] [14] where all the linear constraints have rational coefficients. These linear programming problems are solved via a standard two-phase simplex method that is implemented with multi-precision rational number arithmetic in order to help deal with instability issues.

Note that, for the resultant matrix constructed by Emiris's incremental algorithm, the equality (2.2) holds [32]. Thus, the equality (2.4) also holds, on which the correctness of Algorithm **RUR_toric_square** relies.

Recall that, when the characteristic of $\mathbb{K}$ is 0, in particular, $\mathbb{K} = \mathbb{Q}$, there are two options for determining d at step **10** and computing TPert $(T, u_1, \ldots, u_n)$ at fixed $(u_1, \ldots, u_n)$ at step **15**, step **28** and step **33**. I implement a version in which d at step **10** is determined by scanning the coefficients of some non-trivial multiples of

TGCP $(s, 1, 0, \ldots, 0)$ instead of the coefficients of TGCP $(s, 1, 0, \ldots, 0)$ without any extraneous factor. Also, TPert $(T, u_1, \ldots, u_n)$ is computed via interpolation through the values of the coefficient of the term s^d in some non-trivial multiple of TGCP $(s, \boldsymbol{u})$ instead of the values of the coefficient of the term s^d in TGCP $(s, \boldsymbol{u})$ without any extraneous factor. That is, we do not eliminate the contribution of the extraneous factor from det N before interpolating some non-trivial multiple of TGCP $(s, \boldsymbol{u})$. (See Section 3.1.1.) In Section 3.3.1.1, we have seen that, asymptotically, the cost for eliminating the extraneous factor is negligible compared to the cost for interpolations through the values of the determinant of a bigger resultant matrix. However, this is not true in practice. The resultant matrix constructed by Emiris's incremental algorithm is generically not too big [32]. In particular, for small n, we usually gain significant speed up by avoiding the costly process of elimination of the extraneous factors, even though the cost for interpolations slightly increases.

Whenever det N is evaluated, the non-zero entries of N are specialized to the coefficients of the linear u-polynomial $f_0 = u_0 + u_1X_1 + \cdots + u_nX_n$ and polynomials in the perturbed system $f_1 - sf_1^*, \ldots, f_n - sf_n^*$. By Proposition 3.7 and Remark 3.11, parameters $u_0, u_1, \ldots, u_n$ are always specialized to some integers. The coefficients of the input polynomials $f_1, \ldots, f_n$ are rational numbers. Since the characteristic of $\mathbb{Q}$ is 0, the coefficients of auxiliary polynomials $f_1^*, \ldots, f_n^*$ can be chosen from rational numbers at step **9**, and at any interpolation, we can assign rational values to s. Thus, the entries of N are always specialized to rational numbers. Hence, all the coefficients of some non-trivial multiple of TGCP $(s, \boldsymbol{u})$ are rational numbers. It immediately follows that all the coefficients of TPert $(T, u_1, \ldots, u_n)$ are rational numbers and they can be computed to full-precision.

The rest of the algorithm involves arithmetic operations and the Euclidean algorithm over the ring of univariate polynomials with rational coefficients, and the computation of the first subresultant of two univariate polynomials. Therefore, by the use of multi-precision rational number arithmetic, all the steps of Algorithm **RUR_toric_square** can be implemented exactly and the exact RUR will be computed.

6.1.1.1 Toric Resultants Algorithms

In order to have a practically efficient implementation of the algorithm, it is important to choose a fast algorithm that constructs resultant matrices of reasonable size. There are several algorithms for computing the toric resultant for a square system of polynomials [14] [32] [18] [30] [62]. While I implement Emiris's incremental algorithm, the other algorithms can be used if the prerequisite conditions are met.

Emiris's algorithms [14] [32] for computing the toric resultant for a system of $n+1$ polynomials in n variables constructs the resultant matrix whose determinant is some non-trivial multiple of the toric resultant for the system. The toric resultant without the extraneous factor is computed via interpolation through the values of the determinant of the resultant matrix whose entries are specialized in several ways [14].

There are two versions: the mixed-subdivision based algorithm and the incremental algorithm. The mixed-subdivision based algorithm [14] constructs a single resultant matrix that works, but the size of the resultant matrix constructed is often much larger than the optimal one. In fact, the difference might become exponential in n [14] [32]. On the other hand, the incremental algorithm [32] tries several matrices. Starting at a matrix of the smallest possible size, the algorithm keeps enlarging matrices until one that works is found. If none of these trials are successful, the incremental algorithm constructs the same resultant matrix as the mixed-subdivision based algorithm does. Thus, in the worst case, the incremental algorithm requires much more computation and still ends up returning a big matrix. However, it is observed that the incremental algorithm usually constructs a resultant matrix of reasonable size within only a few iterations.

In terms of the arithmetic complexity, the argument above is rephrased as follows: letting $\mathcal{N}_S$ and $\mathcal{N}_I$ be the size of the resultant matrices constructed by the mixed-subdivision based algorithm and the incremental algorithm, respectively, the arithmetic complexity for these algorithms is $\boldsymbol{O}^*\left(\mathcal{N}_S^{\omega}\right)$ and $\boldsymbol{O}^*\left(\mathcal{N}_I^{1+\omega}\right)$, respectively. In the worst case, $\mathcal{N}_I = \mathcal{N}_S$, however, usually, $\mathcal{N}_I$ is much smaller than $\mathcal{N}_S$.

Note that, for both versions, the number of rows of the resultant matrix whose entries are specialized to the coefficients of f_0 is fixed to MV_{-0}. Thus, the equality (2.2) holds [14] [32].

There are some algorithms that explicitly compute the toric resultant without any extraneous factor.

D'Andrea's formula [18] computes the toric resultant as a quotient of two determinants. The matrix whose determinant becomes the numerator is as big as the resultant matrix constructed by Emiris's algorithms. Thus, evaluating the toric resultant using this formula costs at least as much as evaluating the determinant of the toric resultant matrix (with some extraneous factor) constructed by Emiris's algorithms. While D'Andrea's formula has the added benefit of removing the extraneous factor, we eliminate this extraneous factor via interpolation.

Khetan's formula [62] [63] computes the toric resultant as the determinant of a single matrix. Formulas have been found for unmixed systems of 3 polynomials in 2 variables [62] and 4 polynomials in 3 variables [63], but it is probably impossible to find such formulas for general systems of $n + 1$ polynomials in n variables. If we would apply the formula to a mixed system with supports $A_0, A_1, \ldots, A_n$ then we must treat the input system as unmixed by pretending all the input polynomials have the identical support $\bigcup_{i=0}^{n} A_i$. The degree of the toric resultant for this "fake" unmixed system could be much larger than that of the original system. Also, the resultant matrix contains a block whose entries themselves are the determinants of some other matrices. Thus, the cost for evaluating the resultant matrix constructed using Khetan's formula is more than the cost for evaluating the optimal resultant matrix.

6.1.1.2 Expression Swell

The algorithms suffer from expression swell, thus slowing performance. The swell could be caused by large input coefficients. However, even if the coefficients of the input polynomials are small, the intermediate and final quantities could grow quite large.

Algorithm **RUR_toric_square** consists of exact evaluation of the determinant of a given square matrix with rational entries, polynomial interpolations over rational numbers and operations over the ring of univariate polynomials with rational coefficients.

Modular arithmetic is used in order to avoid expression swell occurring in exact evaluation of determinants. The value of the determinant is obtained from the values of the determinant evaluated on several moduli by using the Chinese remainder algorithm.

At step **10**, step **15**, step **28** and step **33**, we evaluate the determinant of the toric resultant matrix N whose entries are specialized in several ways, while, at step **39**, we evaluate the determinant of the first subresultant matrices of size $2M - 1$ where $M = \deg h$. We have seen that $\dim N > \mathrm{MV}_{-0} \geq M$, but $\dim N$ is not necessarily larger than $2M - 1$. The size of the entries of N depends on the input and could be large or small, while the entries of the first subresultant matrices are usually large because of expression swell of intermediate quantities. Thus, we always use modular arithmetic to compute the first subresultants, while modular arithmetic is used to evaluate $\det N$ only when $\dim N$ is large and/or the size of the entries of N is large.

For more about exact evaluation of determinants, see [28] and [56].

Modular arithmetic can also be used for interpolations and operations over the ring of univariate polynomials with rational coefficients. That is, we use the Chinese remainder theorem for $\mathbb{Q}[T]/I$ where I is some appropriately chosen ideal of $\mathbb{Q}[T]$. In this way, we are able to limit the degree of polynomials involved in the computation. For more details, see fast interpolation and Chinese remaindering algorithms described in [97].

Recall that, at step **40**, $h_i(T) = -T - r_{i,1}(T) \cdot r_{i,0}(T)^{-1} \bmod h(T)$ where $r_{i,0}(T) + r_{i,1}(T)\,t$ is the first subresultant of $q_i^-(t)$ and $q_i^+(2T - t)$. Given $r_{i,0}(T)$, its inverse modulo h is computed using the extended Euclidean algorithm, which usually causes significant expression swell. In order to avoid this problem, we instead could compute $h_i(T)$ as a rational representation: $h_i(T) = -T - \frac{r_{i,1}(T)}{r_{i,0}(T)} \bmod h(T)$. In most applications, rational representations with significantly smaller coefficients are preferable to polynomials with large coefficients.

6.1.2 Experiments

Algorithm **RUR_toric_square** and the other algorithms in Figure 8 are implemented exactly. This section shows some experimental results of our implementation. The implementation is compiled with GNU C++. The GNU Multi Precision (GMP) arithmetic library is used to support multi-precision rational number arithmetic. All the experiments shown in this section are performed on a 3 GHz Intel Pentium CPU with 6 GB memory using Linux Kernel 2.6. In Table II, I show timing breakdowns for the application of the exact RUR to a few sample systems. I give a brief discussion of each case, and summarize the results.

Systems F_1 through F_3 are all drawn from examples described in Section 3.2, while systems F_4 through F_6 are all drawn from cases encountered in an actual geometric boundary evaluation computation. The source data is real-world data provided from the BRL-CAD [24] solid modeling system.

System F_4 consists of a line with an ellipse. There are 2 intersections and both are real.

System F_5 consists of two ellipses. Rather than real intersections, these ellipses have 2 complex intersections.

System F_6 consists of two ellipses with supports

$$(2,0),(1,0),(0,2),(0,1),(0,0)$$

and

$$(2,0),(1,1),(1,0),(0,2),(0,1),(0,0),$$

respectively. System F_6 has 4 roots and all of them are real.

For this example, we spent the most time computing polynomials h_1 and h_2. This was because the coefficients of the polynomial h, h_1 and h_2 become huge.

6.1.2.1 Summary of Timing Breakdowns

While the examples shown above are not comprehensive, from these and other cases examined, the following conclusions can be drawn:

- The performance of our algorithm is reasonable for lower dimension/degree

Table II. Timing breakdown for several examples $F_1, \dots, F_6$.
Rows 2 through 6 characterize the input systems. Rows 7 through 10 characterize the complexity of the toric resultant algorithm (See Section 3.3). Rows 11 through 13 characterize the outputs. Rows 14 through 18 show the timing. Row 15 shows the percentage for computing the resultant matrix (Steps from **1** through **7** in Algorithm **RUR_toric_square**). Row 16 shows the percentage for computing h (Steps from **8** through **31** in Algorithm **RUR_toric_square**). Row 17 shows the percentage for computing $q_i^{\pm}$ (Steps from **32** through **36** in Algorithm **RUR_toric_square**). Row 18 shows the percentage for computing h_i (Steps from **38** through **40** in Algorithm **RUR_toric_square**).

Input System	F_1	F_2	F_3	F_4	F_5	F_6
# of polynomials	2	3	3	2	2	2
# of variables	2	3	2	2	2	2
max. degree of monomials	4	2	2	2	2	2
max bit-length of coefficients	4	1	4	307	51	95
# of roots of system	∞	∞	∞	2	∞	4
MV_{-0}	4	4	12	2	4	4
$\mathcal{M} = \sum_{i=0}^{n} \mathrm{MV}_{-i}$	12	12	36	5	8	8
$\mathcal{N} = \dim N$	12	17	42	6	10	10
$M =$ # of roots of h	4	4	10	2	4	4
max bit-length of coefficients of h	18	20	68	844	159	292
max bit-length of coefficients of $r_{i,j}$	173	96	1549	1686	2061	2820
max bit-length of coefficients of h_i	44	11	273	358	8563	416
total time (sec)	.333	1.87	111	.0662	.672	.104
computing resultant matrix (%)	27	41	0	70	10	11
computing h (%)	14	13	9	2	4	1
computing $q_i^{\pm}$ (%)	48	44	51	6	13	3
computing h_i (%)	11	2	40	22	73	86

systems. However, for higher dimension/degree systems, the implementation tends not to be practical.

- Constructing the toric resultant matrix takes up an insignificant portion of the time. Evidently, the use of Emiris's incremental algorithm rather than the mixed-subdivision based algorithm is justified.

- For lower dimension/degree systems, the most time consuming part of the algorithm is repeated evaluation of the determinant of the toric resultant matrix. By the use of the incremental algorithm, I am able to construct resultant matrices of reasonably small size and sometimes even the optimal one. e.g. F_1. However, the size of the toric resultant matrix grows quite rapidly with respect to the dimension/degree of the input system.

- For positive dimensional systems, the resultant evaluation contributes a certain amount to the total time, while for the zero dimensional systems, the resultant evaluation is insignificant.

- For higher dimension/degree systems, the most time consuming part is computing univariate polynomials forming the exact RUR, mainly because of their huge coefficients. Further optimization such as the use of modular arithmetic should be a target of future speedup efforts.

- For these examples, except for F_5, we did not find any benefit in using rational representations for h_i instead of univariate polynomials. For higher dimension/degree systems, though, significant improvement should be seen.

6.1.3 RUR v.s. MAPC

This section compares my exact implementation of the RUR with the library MAPC [59], introduced in Section 2.1.3.5, that supports exact manipulation of algebraic points and curves in the 2-dimensional space.

In MAPC, a 2-dimensional algebraic point is specified as an intersection of a pair of algebraic curves, and is represented by a rectangle containing one and only one intersection of these curves. A 2-dimensional algebraic curve is specified by bivariate polynomials with rational coefficients. Thus, an algebraic point is represented as a rectangle containing one and only one common root of those bivariate polynomials with rational coefficients. Such a rectangle is generated as follows: first, enumerate disjoint rectangles where intersections are possibly located, and then, for each such rectangle, determine whether or not there really is an intersection by checking how curves hit the boundary edges of the rectangle. Thus, some intersections at singularities cannot be found. (See the description and Figure 7 in Section 2.1.3.5).

The rectangle representing an algebraic point can be shrunk into any size on demand. This feature allows us to manipulate algebraic points exactly. However, MAPC may not be able to test whether or not a point lies on a curve, or whether or not two given points are identical. We have already seen (Section 4.2) that, provided algebraic points are represented in the RUR, those queries are correctly answered.

MAPC is able to find only intersections in a certain region; simply consider only rectangles in the region. On the other hand, the exactly implemented RUR finds all the intersections first, then, if necessary, pick up those in a given region by using some other means (e.g., using Algorithm **Exact_sign**).

We are particularly interested in the comparison of the exactly implemented RUR with the part of MAPC that computes intersections of two algebraic planar

Table III. Timings for solve systems F_4, F_6, F_7, F_8.
Row 2 shows the number of real roots of the input system computed by the exactly implemented RUR. Row 3 shows the degree of the univariate polynomial h in the RUR. When the input system is zero-dimensional, this quantities generally matches to the number of all the roots of the input system. Row 4 shows the number of real roots of the input system computed by MAPC. MAPC may not be able to find some roots. Rows 5 and 6 show the timing by the exactly implemented RUR and MAPC, respectively, in seconds.

Input System	F_4	F_6	F_7	F_8
# of real roots by RUR	2	2	0	4
# of roots of h	2	2	2	4
# of real roots by MAPC	2	2	0	2
total time by RUR (sec)	0.0662	0.104	0.183	0.687
total time by MAPC (sec)	0.017	0.017	0.024	0.017

curves, or equivalently finds common roots of two bivariate polynomials with rational coefficients. We find intersections of a pair of algebraic planar curves via the exactly implemented RUR and MAPC. Timings for a comparison of the exactly implemented RUR with that of MAPC are shown in Table III.

Systems F_4 and F_6 are from Section 6.1.2.

The other systems in Section 6.1.2 have some intersections that MAPC fails to find, and thus, they are not listed here.

Systems F_7 and F_8 are drawn from the same source as systems F_4 and F_6.

System F_7 consists of a line and an ellipse. There are 2 intersections and both are real.

System F_8 consists of two ellipses. Rather than real intersections, they have 2 complex intersections. MAPC finds only those 2 real intersections.

6.1.3.1 Summary of Timing Breakdowns

In Table III, I show timings for a comparison of the exactly implemented RUR with that of MAPC.

Recall that MAPC cannot finds some intersections at singularity while the exactly implemented RUR finds all the intersections. This describes the difference in the number of roots found in F_7 and F_8.

The exactly implemented RUR suffers from expression swell independent of the actual geometry and topology of the intersections. This is observed in F_8.

From these cases, it is clear that for generic cases which a method such as MAPC can handle, the exactly implemented RUR has an unacceptably high performance cost. For this reason, it will be best to use the excatly implemented RUR only in a hybrid fashion, when the other methods will fail. An important caveat should be considered, in that the exactly implemented RUR has not been fully optimized. On the the hand, MAPC has been significantly optimized through the use of various speedup techniques, such as floating-point filters and lazy evaluation approaches. Such optimizations should increase the performance of the excatly implemented RUR significantly, although there will still be fundamental limitations that will make the excatly implemented RUR less efficient in most generic cases encountered in practice.

Finally, note that performance could be improved by isolating only real roots of the minimal polynomial, h, which would allow us to isolate the real roots of the system. In fact, this is the approach taken by the GBRS system [87]. However, doing so would not allow us to determine the positive dimensional components, where the points isolated on the component could be complex.

6.2 Numerical Pertubations

This section describes the implementation of one of the numerical perturbation schemes, "expansion / contraction," in order to remove degeneracies appearing in boundary evaluation of solid objects.

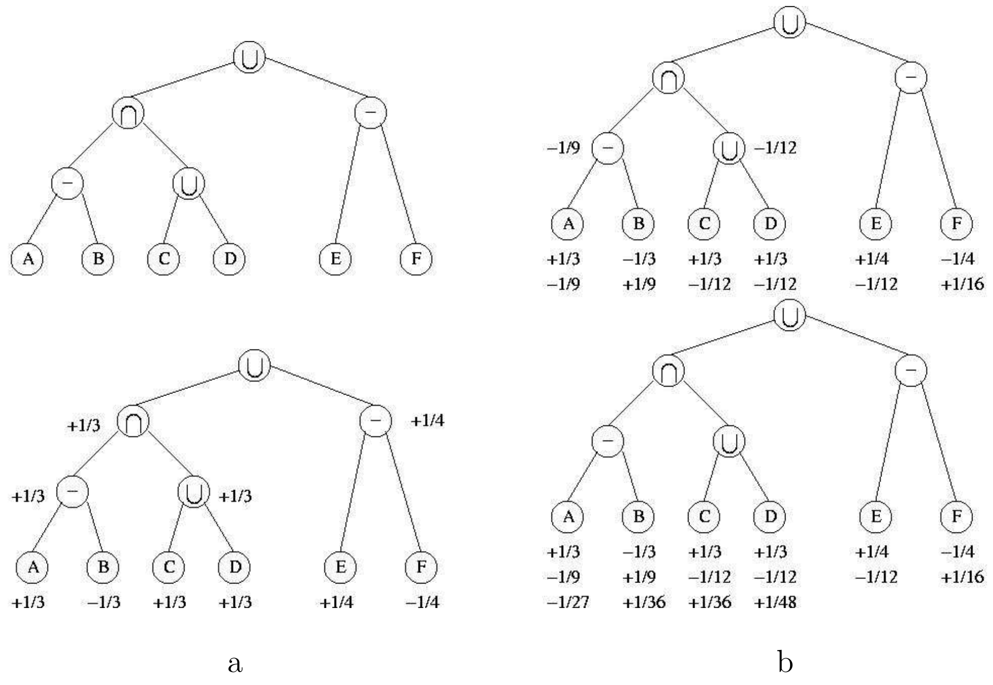


Fig. 11. Expansion / contraction.

a. A perturbation information of a certain node becomes a set of expansion / contraction operations at the leaves [61]. If $A \cup B$ or $A \cap B$ need to be expanded, both A and B would be expanded. With a difference operation $A - B$, however, A would be expanded, and B would be contracted.

b. Since different degeneracies might require a primitive solid object to be expanded in different directions, for the different operations, different amounts of perturbations are allowed.

ESOLID, introduced in Section 2.1.3.5, is a geometric solid modeling system that performs exact boundary evaluation for a given CSG model of solid objects[58]. Throughout the system, exact computation is used in order to eliminate the robustness problem due to numerical errors. Unfortunately, ESOLID assumes that every object is in general position. On a degenerate input, ESOLID may return a wrong result, or even worse, crash.

ESOLID uses the library MAPC [59] that supports exact manipulation of algebraic points and curves in a 2-dimensional space. For efficiency, MAPC assumes curves are non-singular and ignores some singular intersections such as tangential intersections.

We would like to overcome this situation.

In Section 4.3 in Chapter IV, I show that the use of RUR helps us to detect degeneracies. I will add another component that performs one of the numerical perturbation schemes, “expansion / contraction” over the solid objects input to ESOLID. See Figure 1 in Chapter I.

A numerical perturbation approach requires very little modification to ESOLID, yet eliminates degeneracies.

I emphasize that the use of exact computation enables us to achieve greater robustness and consistency. The goal is not necessarily to have an exact solution to the input problem, but without accounting for numerical error and degeneracies, programs are subject to crashes or inconsistent output, neither of which is a desirable condition. We would like to have a program that can reliably produce a set of valid output, for a wide variety of input.

6.2.1 Expansion / Contraction for CSG Trees

Applying expansion / contraction to general curved solid objects is difficult, and often suffers from the complex topological issues. Fortunately, however, it is relatively easy to apply expansion / contraction to the standard CSG primitive solid objects: boxes, cylinders (generalized cones), spheres (generalized ellipsoids) and tori. Perturbations at one level of a CSG tree can always be propagated to the primitive solid objects at leaves.

In my implementation, each primitive object can be scaled by a certain amount (bounded from above by the global tolerance τ), relatively easily. By fixing the centroid of a solid, any point in the solid object is scaled outward or inward by the same amount in any direction. For example, an ellipsoid can be expanded just by increasing its radius.

Perturbation is applied, during boundary evaluation, whenever a degeneracy is detected. Since this may occur at a high level node in a CSG-tree, the perturbation information must be propagated from the node down to the leaves (i.e. the input primitive solid objects). There is a rule [61] that decides which object(s) is expanded and which contracted. Assume we wish to propagate an expansion information down the tree. Each union or intersection node would also propagate an expansion downward. That is, if $A \cup B$ or $A \cap B$ need to be expanded, both A and B would be expanded. With a difference operation $A - B$, however, A would be expanded, and B would be contracted. In this way, a perturbation information of a certain node becomes a set of expansion / contraction operations at the leaves. See an example in Figure 11 a.

After a perturbation is applied, the following holds:

Proposition 6.30. *1. Any surface of a primitive solid is expanded/contracted so that it is parallel to that of the original.*

2. Any surface of any solid represented as some subtree rooted at some internal node of a CSG-tree is expanded/contracted so that either 1) it is parallel to that *of the original or 2) there is no counterpart in the original.*

3. *Any edge of a primitive solid is expanded/contracted so that it is parallel to that of the original.*

4. *Any edge of any solid represented as some subtree rooted at some internal node of a CSG-tree is expanded/contracted so that either 1) it is parallel to that of the original or 2) there is no counterpart in the original.*

5. *The centroid of any primitive solid does not move.*

Expansion / contraction of primitive solid objects offers the opportunity to capture design intent. Expansion / contraction follows the principles proposed first by Sugihara and Iri [96] and Fortune [36], where the surfaces of input solid objects are symbolically perturbed inward or outward. In the implementation described in this dissertation, however, for simplicity, perturbations are applied only to primitive solid objects. The approach taken here is different from these previous approaches in that mine can be applied to non-polyhedral solid objects, and in that the perturbation information is propagated through the CSG tree only to resolve specific degeneracies.

It should be noted that if there are multiple degeneracies in a CSG tree, it is possible that different degeneracies might require a primitive solid object to be expanded in different directions. In this case, by allowing perturbations of different amounts for the different operations, both operations can be satisfied. Since one degeneracy must precede the other in the tree, the perturbation from the higher level node affects degeneracies at the lower-level nodes. As long as the solid objects are perturbed by an amount enough to resolve their own degeneracy, but not so much as to change the perturbation direction required by the higher degeneracy, the perturbation will work. See Figure 11 b. This is likely to lead to much higher bit complexity, however.

6.2.1.1 Implementation

The perturbation scheme described in Section 6.2 is implemented. ESOLID is used to convert parts from CSG format to an exact B-rep format. Whenever a degeneracy is detected, ESOLID either aborts or loops. If ESOLID aborts then numerical perturbation is applied. Perturbation is propagated through the entire CSG tree and eventually all the input primitive solids are perturbed. Then, the computation restarts.

The experiments are performed on three parts shown in Figure 12 taken from a Bradley Fighting Vehicle that is developed by the Army Research Lab with their BRL-CAD solid modeler [24], using a 3.0 GHz Intel Pentium CPU with 6 GB of memory. Each model contained degenerate data that could crash a standard modeler. The numerical perturbation described in this dissertation scheme allows us to compute results with a general-position algorithm, while maintaining designer's intent.

Tables IV, V and VI show experiments on the examples in Figure 12. Rows 3 and 4 give the number of times the basic bivariate and univariate root-finding routines were invoked while computing. MAPC performs root isolation and sign evaluation (of a polynomial at a given value) by using floating point filters. When the filter fails, methods involving multi-precision arithmeticoperations are invoked. The number of such root isolation and sign evaluations is shown, along with the percentage of the time that the floating point filter fails (and thus exact methods must be used). Column 1 shows the result when ESOLID (i.e. without perturbation) is used for operations on solids not in degenerate configuration, while column 2 shows the performance of ESOLID on a perturbed version (by $\frac{1}{1024}$) of those same solids. Column 3 gives the results for the entire part, including the perturbed solids that have removed the degeneracies.

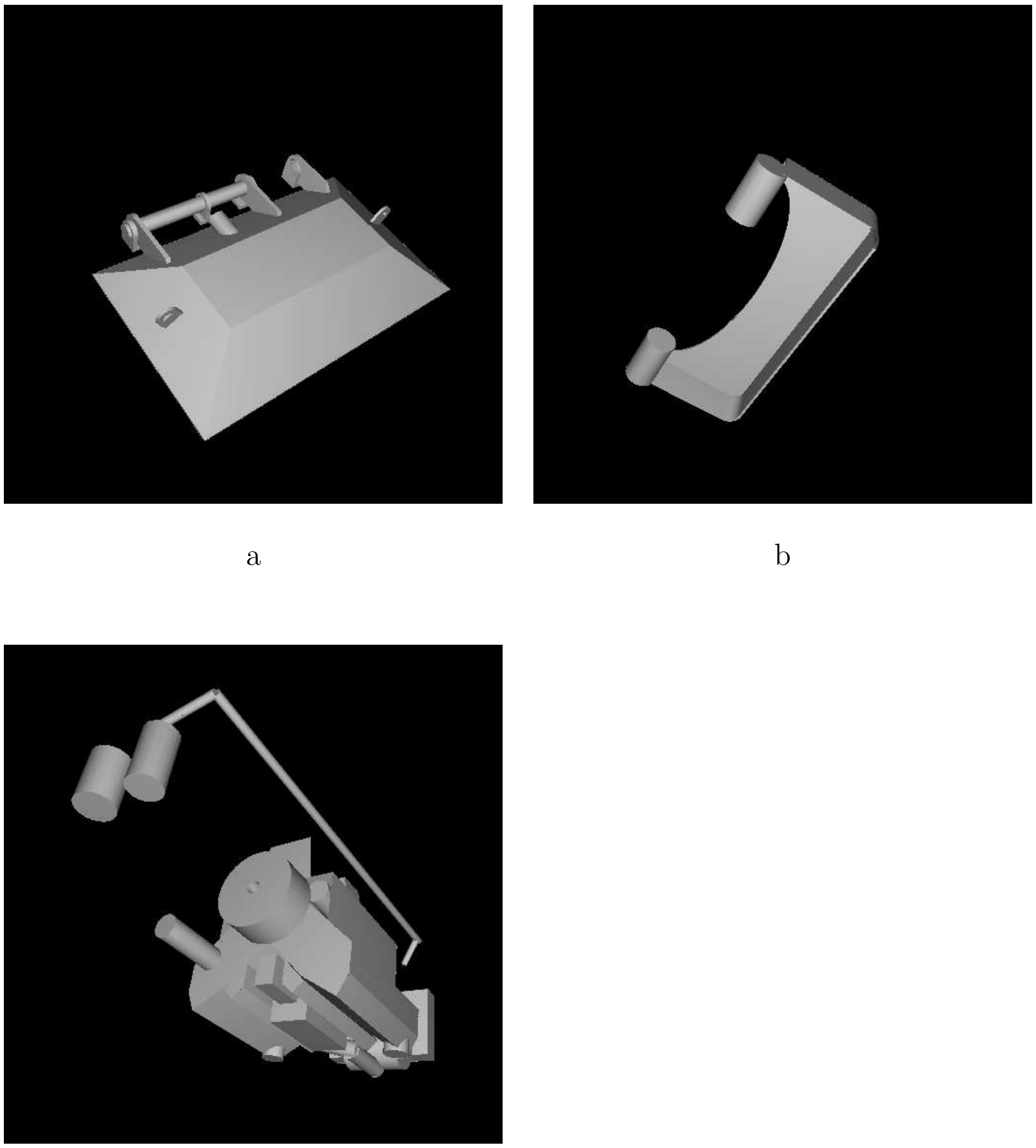
The part *cargo hatch* is obtained by joining 11 solids. ESOLID fails to model 4 of them because of degeneracies. The part *commander hatch* is obtained by joining 5 solids. ESOLID fails to model 1 of them because of a degeneracy. In both examples, all the degeneracies can be eliminated by numerical perturbation. The part *engine* is obtained by joining 14 solids. One of them has a degeneracy that makes ESOLID loop. Thus, this part is ignored. Among the other 13 solids, ESOLID fails on 3 of them because of the degeneracies.

The perturbed versions ran slower than the unperturbed versions (on the portions that both could compute), due to higher bit-length. Filters failed slightly more often. For the parts *cargo hatch* and *engine*, the increase in time was very modest, however, the part *commander hatch* took significantly more time to compute. This exceedingly longer time was due to secondary effects of the perturbation. Specifically, the perturbation created a situation (having to separate two very close curves that were once only one curve) that the particular evaluation algorithm used here was not adept at handling.

In summary, on these real-world cases, note first that numerical perturbation allows us to compute the result while maintaining designer's intent. The perturbed versions do, indeed, run slower than the unperturbed versions and require a higher percentage of sign evaluations of polynomials using exact arithmetic, rather than floating-point filters (due to increases in bit-length).

Table IV. Experiments on cargo hatch.
Column 2 show statistics without perturbation until ESOLD fails because of degeneracies. Column 3 show statistics with a numerical perturbation but stops at the point when no perturbation version (shown in column 2) fails. Column 4 show statistics when entire part is build (with a numerical perturbation). Bivariate root-finding and univariate root-finding are two major heavy calls to MAPC. These root-finding operations internally call root isolation and sign evaluation. FP-filter is used for root isolation and sign evaluation if possible.

	w/o pert.	w/ pert.	all solids
time (msec)	36570	50487	141474
# of CSG boolean op's	17	17	34
# of bivariate root-finding	25137	25764	50902
# of univariate root-finding	23809	26344	56800
# of root isolation	32616	36879	82024
% of exact computation	6.30	3.88	7.33
# of sign evaluation	486139	573022	1288838
% of exact computation	6.37	7.37	11.33



a b

c

Fig. 12. Real-world tank examples. a. Cargo hatch, b. Commander hatch, c. Engine. Each of these contained degenerate data that could crash a standard modeler. The numerical perturbation allowed us to compute results with a general-position algorithm, while maintaining designer's intent.

Table V. Experiments on commander hatch.
Column 2 show statistics without perturbation until ESOLD fails because of degeneracies. Column 3 show statistics with a numerical perturbation but stops at the point when no perturbation version (shown in column 2) fails. Column 4 show statistics when entire part is build (with a numerical perturbation). Bivariate root-finding and univariate root-finding are two major heavy calls to MAPC. These root-finding operations internally call root isolation and sign evaluation. FP-filter is used for root isolation and sign evaluation if possible.

	w/o pert.	w/ pert.	all solids
time (msec)	95658	701096	722385
# of CSG boolean op's	6	6	12
# of bivariate root-finding	5331	4103	6636
# of univariate root-finding	12354	4772	8635
# of root isolation	19889	8761	13032
% of exact computation	18.87	17.62	11.85
# of sign evaluation	359768	136517	208196
% of exact computation	3.66	7.85	6.15

Table VI. Experiments on engine.
Column 2 show statistics without perturbation until ESOLD fails because of degeneracies. Column 3 show statistics with a numerical perturbation but stops at the point when no perturbation version (shown in column 2) fails. Column 4 show statistics when entire part is build (with a numerical perturbation). Bivariate root-finding and univariate root-finding are two major heavy calls to MAPC. These root-finding operations internally call root isolation and sign evaluation. FP-filter is used for root isolation and sign evaluation if possible.

	w/o pert.	w/ pert.	all solids
time (msec)	224803	279560	1233390
# of CSG boolean op's	15	15	33
# of bivariate root-finding	18252	17827	36456
# of univariate root-finding	30222	29154	51079
# of root isolation	49370	48878	86664
% of exact computation	14.46	16.42	20.38
# of sign evaluation	826664	821022	1501026
% of failure of FP-filter	3.59	4.49	7.95

CHAPTER VII

CONCLUSION

This dissertation describes several operations for supporting exact geometric computation, in particular, the Rational Univariate Representation (RUR) for the zero set of a system of polynomials. This chapter summarizes my research, reviews how the objective of my dissertation is demonstrated and discusses future work to be done.

7.1 Objective

The objective of this dissertation is the following:

> The Rational Univariate Representation (RUR) effectively supports exact computation over algebraic points and curves. This enables robust geometric computation, in particular, degeneracy handling.

The objective has been proved by:

(1) developing an exact representation of an algebraic point based on the Rational Univariate Representation RUR (Chapter III and Section 6.1 in Chapter VI),

(2) developing methods to support exact computation over algebraic numbers, points and curves (Section 4.1 and Section 4.2 in Chapter IV),

(3) applying these methods in order to detect degeneracies appearing in boundary evaluation of solid objects (Section 4.3 in Chapter IV), and

(4) developing an exact numeral perturbation scheme for handling degeneracies appearing in boundary evaluation of solid objects (Chapter V and Section 6.2 in Chapter VI).

7.1.1 Rational Univariate Reduction

This section summarizes the results in Chapter III and Section 6.1 in Chapter VI.

New algorithms for computing the Rational Univariate Representation (RUR) for the zero set of a given system of multivariate polynomials are presented (Chapter III).

An algorithm for computing the RUR for the zero set of a square system refines the algorithm originally presented in [81]. In both algorithms, the univariate polynomials forming the RUR are derived from the toric perturbation, which is a generalization of the toric u-resultant, by specializing indeterminate u's to some appropriate values. A deterministic way to specialize those indeterminates appropriately is presented.

A new algorithm for computing the RUR for the zero set of an overdetermined system has been presented. The algorithm constructs a square system of higher dimension so that the projection of the RUR for the zero set of the square system will become the RUR for the zero set of the input system. In contrast to the algorithm for an overdetermined system described in [83], where a square system of the same dimension is constructed from the input system with some random choices, the new algorithm is deterministic.

As a consequence of derandomization, a new algorithm for computing real roots of a given system of multivariate polynomials becomes much simpler.

The arithmetic complexity of the new algorithm is analyzed. The analysis tells us that the size of the resultant matrices governs the complexity of the algorithm.

The implementation of the new algorithms for the case when all the coefficients of the input polynomials are rational numbers is described in detail. The implementation is exact; all the rational coefficients of the univariate polynomials forming the RUR will be computed to full precision.

7.1.2 Exact Manipulation of Algebraic Points and Curves

This section summarizes the results in Chapter IV.

The root bound approach to exact sign determination of the real and imaginary parts of algebraic numbers is explained.

I propose an exact representation for an algebraic point. In particular, I show that algebraic points specified as intersections of algebraic curves can be exactly expressed in the Rational Univariate Representation. Together with the root bound approach to exact sign determination of the real and imaginary parts of algebraic numbers, I show some fundamental predicates in computational geometry can be computed exactly. This enables us to perform exact manipulation of algebraic points and curves.

The proposed technique for exact manipulation of algebraic points and curves is applied to the degeneracy detection problem appearing in boundary evaluation of solid objects. Degeneracy detection is done by checking irregular interaction between surfaces, edges and points. Since the RUR can be used even for points at a singularity, and exact comparisons over algebraic points in the RUR is straightforward, the RUR is a powerful method for problems dealing with degeneracies.

7.1.3 Numerical Perturbations

This section summarizes the results in Chapter V and Section 6.2 in Chapter VI.

A formal description of exact numerical perturbation schemes for removing degeneracies is given analogous to the formal description of symbolic perturbation schemes [90] [91]. Several aspects of numerical perturbation schemes are derived. In particular, it is shown that random numerical perturbations work almost always.

Several problems regarding the implementation of numerical perturbation schemes are discussed. Some examples are shown. It turns out that an arbitrary numerical perturbation is unlikely to meet any criteria of capturing designer's intent. Even though the computation might complete, the output can be different from the what the designer intended - some guidance for how to perturb is needed.

One numerical perturbation scheme, "expansion / contraction of primitive solid objects" is implemented in order to remove degeneracies appearing in boundary evaluation of solid objects. It is successfully used to model several solid objects from real-world examples.

7.2 Future Work

There are several avenues of future work open, some of which I list here.

The algorithms for computing the RUR for a given system of polynomials form the foundations in this research. More work on the RUR is planed.

I have developed Algorithm **Positive_dimensional_Components** to detect whether or not the zero set of the input system has some positive dimensional components (Section 4.2.1.2). While it would also be nice to develop a means of actually finding a representation for the positive dimensional components of the zero set of the input system, this would be a much harder problem.

I have not analyzed asymptotic arithmetic complexity of Algorithm **RUR_overconstrained** (Section 3.1.2.2). I strongly believe that the arithmetic complexity of the new algorithm is worse compared to the algorithm described in [83], although the new algorithm behaves better for small n. A bit-complexity analysis of these algorithms would also be useful.

Although some efficiency improvements have been installed, the implementation of the algorithms can be further optimized. One avenue in particular would be to use better algorithms for constructing the resultant matrices or evaluating the toric resultant. Any improvement here would be helpful, since the resultant computation governs the performance of the algorithms both in theory and practice. Also, faster subroutines for linear algebra operations and polynomial ring operations would be useful. For instance, I should take advantage of the sparse structure of the resultant matrix when its determinant is evaluated.

Also, it would be nice to extend the implementation described in this dissertation to include coefficients belonging to a field other than the field of rational numbers. Over a finite field, an implementation must look different since we cannot use techniques that work over a field of characteristic 0. Also, an implementation for coefficients that are real or complex algebraic numbers would be interesting. Even more generally, there are practical reasons to consider polynomials whose coefficients are not given exactly. In contrast to Gröbner basis approaches to determining the RUR, the toric approach is continuous over perturbations in the coefficients, thus there is some hope that it would offer a method for dealing with such input. Developing theory and implementation to support such polynomials would be very valuable.

The goal of this research is to support robust geometric computation. In this dissertation, the exact numerical perturbation scheme is proposed and applied to boundary evaluation of solid objects (Section 4.3.2 and Section 6.2). More geometric problems should be explored. In the future, I would like to further develop the various types of implementations of exact numerical perturbation schemes, and test them.

REFERENCES

[1] O. Aberth, “Iteration methods for finding all zeros of a polynomial simultaneously,” *Mathematics of Computation*, vol. 27, no. 122, pp. 339 – 344, 1973.

[2] P. Aubry, F. Rouillier, and M. S. E. Din, “Real solving for positive dimensional systems,” *Journal of Symbolic Computation*, vol. 34, no. 6, pp. 543 – 560, 2002.

[3] S. Basu, R. Pollack, and M.-F. Roy, *Algorithms in Real Algebraic Geometry.* Berlin: Springer, 2003.

[4] M. O. Benouamer, D. Michelucci, and B. Peroche, “Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation,” *Computer-Aided Design*, vol. 26, no. 6, pp. 403 – 416, 1994.

[5] D. N. Bernstein, “The number of roots of a system of equations,” *Functional Analysis and its Applications*, vol. 9, no. 2, pp. 183 – 185, 1975.

[6] L. Blum, F. Cucker, M. Shub, and S. Smale, *Complexity and Real Computation.* New York: Springer, 1997.

[7] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra, “A strong and easily computable separation bound for arithmetic expressions involving square roots,” in *Proc. of 8th Annual ACM-SIAM Symposium on Discrete Algorithms.* ACM, 1997, pp. 702 – 709.

[8] ——, “Exact efficient computational geometry made easy,” in *Proc. of 15th Annual Symposium on Computational Geometry.* ACM, 1999, pp. 341 – 350.

[9] ——, “A strong and easily computable separation bound for arithmetic expressions involving radicals,” *Algorithmica*, vol. 27, no. 1, pp. 87 – 99, 2000.

[10] C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt, "A separation bound for real algebraic expressions," in *Proc. of 9th Annual European Symposium on Algorithms*, ser. LNCS 2161. Springer, 2001, pp. 254 – 265.

[11] C. Burnikel, K. Mehlhorn, and S. Schirra, "The LEDA class **real** number," Max-Planck-Institut fur Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, Tech. Rep. MPI-I-96-1-001, 1996.

[12] J. F. Canny, "Generalized characteristic polynomials," *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 241 – 250, 1990.

[13] J. F. Canny and I. Z. Emiris, "An efficient algorithm for the sparse mixed resultant," in *Proc. of 10th AAECC*, ser. LNCS 673. Springer, 1993, pp. 89 – 104.

[14] ——, "A subdivision-based algorithm for the sparse resultant," *Journal of ACM*, vol. 47, no. 3, pp. 417 – 451, 2000.

[15] D. Cox, "What is a toric variety?" in *Topics in Algebraic Geometry and Geometric Modeling*, ser. Contemporary Mathematics, Vol. 334, R. Goldman and R. Krasauskas, Eds. Providence: AMS, 2003, pp. 203 – 223.

[16] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms*, 2nd ed. New York: Springer, 1996.

[17] ——, *Using Algebraic Geometry.* New York: Springer, 1998.

[18] C. D'Andrea, "Macaulay style formulas for sparse resultants," *Transactions of the AMS*, vol. 354, no. 7, pp. 2595 – 2629, 2002.

[19] C. D'Andrea and I. Z. Emiris, "Computing sparse projection operators," in *Symbolic Computation: Solving equations in Algebra, Geometry and Engineering*, ser. Contemporary Mathematics, Vol. 286, E. L. G. et al., Ed. Providence: AMS, 2001, pp. 121 – 139.

[20] ——, "Sparse resultant perturbation," in *Algebra, Geometry, and Software*, M. Joswing and N. Takayama, Eds. Berlin: Springer, 2003, pp. 93 – 107.

[21] O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud, "Algebraic methods and arithmetic filtering for exact predicates on circle arcs," *Computational Geometry*, vol. 22, no. 1 - 3, pp. 119 – 142, 2002.

[22] M. S. E. Din and E. Shost, "Properness defects of projections and computation of at least one point in each connected component of a real algebraic set," *Discrete and Computational Geometry*, vol. 32, no. 3, pp. 417 – 430, 2004.

[23] T. Dubé and C. Yap, "The exact computation paradigm," in *Computing in Euclidean Geometry*, 2nd ed., ser. Lecture Notes on Computing, D. Du and F. Hwang, Eds. Singapore: World Scientific, 1995, pp. 452 – 492.

[24] P. C. Dykstra and M. J. Muuss, "The BRL-CAD package an overview," Advanced Computer Systesms Team, Ballistics Research Laboratory, Aberdeen Proving Ground, MD, Tech. Rep., 1989, http://ftp.arl.mil/brlcad/.

[25] H. Edelsbrunner and E. Mücke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *ACM Transactions on Graphics*, vol. 9, no. 1, pp. 66 – 104, 1990.

[26] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert, "Complete, exact, and efficient computations with cubic curves," in *Proc. of 20th Annual Symposium*

on Computational Geometry. ACM, 2004, pp. 409 – 418.

[27] I. Z. Emiris, "On the complexity of sparse elimination," *Journal of Complexity*, vol. 12, no. 2, pp. 134 – 166, 1996.

[28] ——, "A complete implementation for computing general dimensional convex hulls," *International Journal of Computational Geometry and Applications*, vol. 8, no. 2, pp. 223 – 253, 1998.

[29] ——, "Enumerating a subset of the integer points inside a Minkowski sum," *Computational Geometry*, vol. 22, no. 1 - 3, pp. 143 – 166, 2002.

[30] ——, "Discrete geometry for algebraic elimination," in *Algebra, Geometry, and Software*, M. Joswing and N. Takayama, Eds. Berlin: Springer, 2003, pp. 77 – 91.

[31] I. Z. Emiris and J. F. Canny, "A general approach to removing degeneracies," in *Proc. of 32nd IEEE Symposium on the Foundations of Computer Science.* IEEE, 1991, pp. 405 – 413.

[32] ——, "Efficient incremental algorithm for the sparse resultant and the mixed volume," *Journal of Symbolic Computation*, vol. 20, no. 2, pp. 117 – 149, 1995.

[33] I. Z. Emiris, J. F. Canny, and R. Seidel, "Efficient perturbations for handling geometric degeneracies," *Algorithmica*, vol. 19, no. 1 - 2, pp. 219 – 242, 1997.

[34] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr, "On the design of cgal a computational geometry algorithms library," *Software – Practice & Experience*, vol. 30, no. 11, pp. 1167 – 1202, 2000.

[35] R. T. Farouki, C. A. Neff, and M. A. O'Connor, "Automatic parsing of degenerate quadric-surface intersections," *ACM Transactions on Graphics*, vol. 8, no. 3, pp. 174 – 208, 1993.

[36] S. Fortune, "Polyhedral modeling with multiprecision integer arithmetic," *Computer-Aided Design*, vol. 29, no. 2, pp. 123 – 133, 1997.

[37] ——, "Vertex-rounding a three-dimensional polyhedral subdivision," in *Proc. of 14th Ann. Symp. on Computational Geometry.* ACM, 1998, pp. 116 – 125.

[38] S. Funke, C. Klein, K. Mehlhorn, and S. Schmitt, "Controlled perturbation for Delaunay triangulations," in *Proc. of 16th Ann. ACM-SIAM Symp. on Discrete Algorithms '05.* ACM, 2005, pp. 1047 – 1056.

[39] N. Geismann, M. Hemmer, and E. Schömer, "Computing a 3-dimensional cell in an arrangement of quadrics: Exactly and actually!" in *Proc. of 17th Annual Symposium on Computational Geometry.* ACM, 2001, pp. 264 – 273.

[40] M. Giusti, G. Lecerf, and B. Salvy, "A Gröbner free alternative for polynomial system solving," *Journal of Complexity*, vol. 17, no. 1, pp. 154 – 211, 2001.

[41] L. Gonzáez-Vega, "A subresultant theory for multivariate polynomials," in *Proc. of ISSAC '91.* ACM, 1991, pp. 79 – 85.

[42] L. Gonzalez-Vega, "Implicitization of parametric curves and surfaces by using multidimensional newton formulae," *Journal of Symbolic Computation*, vol. 23, no. 2 - 3, pp. 137 – 151, 1997.

[43] L. Gonzalez-Vega and I. Necula, "Efficient topology determination of implicitly defined algebraic plane curves," *Computer Aided Geometric Design*, vol. 19, no. 9, pp. 719 – 743, 2002.

[44] L. Gonzalez-Vega, F. Rouillier, and M.-F. Roy, “Symbolic recipes for polynomial system solving,” in *Some Tapas of Computer Algebra*, ser. Algorithms and computation in mathematics, Vol. 4, A. M. Cohen, H. Cuypers, and H. Sterk, Eds. Berlin: Springer, 1999, pp. 34 – 65.

[45] M. T. Goodrich, L. J. Guibas, J. Hershberger, and P. J. Tanenbaum, “Snap rounding line segments efficiently in two and three dimensions,” in *Proc. of 13th Ann. Symp. on Computational Geometry.* ACM, 1997, pp. 284 – 293.

[46] D. Halperin and E. Leiserowitz, “Controlled perturbation for arrangements of circles,” in *Proc. of 19th Ann. Symp. on Computational Geometry.* ACM, 2003, pp. 264 – 273.

[47] D. Halperin and S. Raab, “Controlled perturbation for arrangements of polyhedral surfaces,” in *Proc. of 15th Ann. Symp. on Computational Geometry.* ACM, 1999, pp. 163 – 172.

[48] D. Halperin and C. R. Shelton, “A perturbation scheme for spherical arrangements with application to molecular modeling,” *Computational Geometry*, vol. 10, no. 4, pp. 273 – 287, 1998.

[49] J. D. Hobby, “Practical segment intersection with finite precision output,” *Computational Geometry*, vol. 13, no. 4, pp. 199 – 214, 1999.

[50] C. M. Hoffman, “The problems of accuracy and robustness in geometric computation,” *IEEE Computer*, vol. 22, no. 3, pp. 31 – 41, 1989.

[51] ——, “Robustness in geometric computations,” *Journal of Computing and Information Science in Engineering*, vol. 1, no. 2, pp. 143 – 155, 2001.

[52] C. M. Hoffman, J. E. Hopcroft, and M. S. Karasick, “Robust set operations on polyhedral solids,” *IEEE Computer Graphics and Applications*, vol. 9, no. 6, pp. 50 – 59, 1989.

[53] H. Hong, “An efficient method for analyzing the topology of plane real algebraic curves,” *Mathematics and Computers in Simulation*, vol. 42, no. 4 - 6, pp. 571 – 582, 1996.

[54] H. Hong and M. Minimair, “Sparse resultant of composed polynomials I mixed-unmixed case,” *Journal of Symbolic Computation*, vol. 33, no. 4, pp. 447 – 465, 2002.

[55] G. Jeronimo, T. Krick, J. Sabia, and M. Sombra, “The computational complexity of the chow form,” *Foundations of Computational Mathematics*, vol. 4, no. 1, pp. 41 – 117, 2004.

[56] E. Kaltofen and G. Villard, “Computing the sign or the value of the determinant of an integer matrix, a complexity survey,” *Journal of Computational and Applied Mathematics*, vol. 162, no. 1, pp. 133 – 146, 2004.

[57] V. Karamcheti, C. Li, and C. Yap, “A Core library for robust numerical and geometric computation,” in *Proc. of 15th Annual Symposium on Computational Geometry*. ACM, 1999, pp. 351 – 359.

[58] J. Keyser, T. Culver, M. Foskey, S. Krishnan, and D. Manocha, “ESOLID - a system for exact boundary evaluation,” *Computer-Aided Design*, vol. 36, no. 2, pp. 175 – 193, 2004.

[59] J. Keyser, T. Culver, D. Manocha, and S. Krishnan, “Efficient and exact manipulation of algebraic points and curves,” *Computer-Aided Design*, vol. 32,

no. 11, pp. 649 – 662, 2000.

[60] J. Keyser, K. Ouchi, and J. M. Rojas, "The exact rational univariate representation and its application," in *Geometric and Algorithmic Aspects of Computer-Aided Design and Manufacturing*, ser. AMS/DIMACS volume 67, R. Janardan, M. Smid, and D. Dutta, Eds. Providence: AMS, 2005, pp. 299 – 328.

[61] J. C. Keyser, "Exact boundary evaluation for curved solids," Ph.D. Dissertation, Department of Computer Science, University of North Carolina, Chapel Hill, NC, 2000.

[62] A. Khetan, "The resultant of an unmixed bivariate system," *Journal of Symbolic Computation*, vol. 36, no. 3 - 4, pp. 425 – 442, 2003.

[63] ——, "Exact matrix formula for the unmixed resultant in three variables," *Journal of Pure and Applied Algebra*, vol. 198, no. 1 - 3, pp. 237 – 256, 2005, to appear.

[64] L. Kronecker, *Leopold Kronecker's Werke.* Leipzig: Teubner, 1895 - 1931.

[65] G. Lecerf, "Quadratic newton iteration for systems with multiplicity," *Journal of Foundations of Computational Mathematics*, vol. 2, no. 3, pp. 247 – 293, 2002.

[66] C. Li and C. Yap, "A new constructive root bound for algebraic expressions," in *Proc. of 12th Annual ACM-SIAM Symposium on Discrete Algorithms '01.* ACM, 2001, pp. 476 – 505.

[67] T. Y. Li and X. Wang, "The BKK root count in C^n," *Mathematics of Computation*, vol. 65, no. 216, pp. 1477 – 1484, 1996.

[68] K. Mahler, "An application of Jensen's formula to polynomials," *Mathematika*, vol. 7, pp. 98 – 100, 1960.

[69] S. Mehlhorn and M. Näher, *LEDA - A Platform for Combinatorial and Geometric Computing.* Cambridge: Cambridge University Press, 1999.

[70] N. Mignotte and D. Ştefănescu, *Polynomials: An Algebraic Approach.* Singapore: Springer, 1999.

[71] P. S. Milne, "On the solutions of a set of polynomial equations," in *Symbolic and Numerical Computation for Artificial Intelligence*, B. R. Donald, D. Kapur, and J. L. Mundy, Eds. London: Academic Press, 1992, pp. 88 – 102.

[72] M. Minimair, "Sparse resultant of composed polynomials II mixed-unmixed case," *Journal of Symbolic Computation*, vol. 33, no. 4, pp. 467 – 478, 2002.

[73] B. Mourrain, "A new criterion for normal form algorithms," in *Proc. of 13th International Symposium, AAECC 13 '99*, ser. LNCS 1719. Springer, 1999, pp. 430 – 443.

[74] B. Mourrain, M. N. Vrahatis, and J. C. Yakoubsohn, "On the complexity of isolating real roots and computing with certainty the topological degree," *Journal of Complexity*, vol. 18, no. 2, pp. 612 – 640, 2002.

[75] K. Ouchi and J. Keyser, "Handling degeneracies in exact boundary evaluation," in *Proc. of ACM Symposium on Solid Modeling and Applications.* ACM, 2004, pp. 321 – 326.

[76] P. Pedersen and B. Sturmfels, "Product formulas for resultants and chow forms," *Mathematische Zeitschrift*, vol. 214, no. 3, pp. 377 – 396, 1993.

[77] S. Pion and C. Yap, "Constructive root bound for k-ary rational input numbers," in *Proc. of 19th Annual Symposium on Computational Geometry.* ACM, 2003, pp. 256 – 263.

[78] F. P. Preparata and M. Shamos, *Computational Geometry: An Introduction.* New York: Springer, 1985.

[79] S. Raab, "Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes," in *Proc. of 15th Ann. Symp. on Computational Geometry.* ACM, 1999, pp. 163 – 172.

[80] J. M. Rojas, "Toric laminations, sparse generalized characteristic polynomials, and a refinement of Hilbert's tenth problem," in *Foundations of Computational Mathematics*, F. Cucker and M. Shub, Eds. Berlin: Springer, 1997, pp. 369 – 381.

[81] ——, "Solving degenerate sparse polynomial systems faster," *Journal of Symbolic Computation*, vol. 28, no. 1 - 2, pp. 155 – 186, 1999.

[82] ——, "Toric intersection theory for affine root counting," *Journal of Pure and Applied Algebra*, vol. 136, no. 1, pp. 67 – 100, 1999.

[83] ——, "Algebraic geometry over four rings and the frontier to tractability," in *Hilbert's tenth problem : relations with arithmetic and algebraic geometry*, ser. Contemporary Mathematics, Vol. 270, J. D. et al., Ed. Providence: AMS, 2000, pp. 275 – 321.

[84] ——, "Why polyhedra matter in non-linear equation solving," in *Topics in Algebraic Geometry and Geometric Modeling*, ser. Contemporary Mathematics,

Vol. 334, R. Goldman and R. Krasauskas, Eds. Providence: AMS, 2003, pp. 293 – 320.

[85] J. M. Rojas and X. Wang, "Counting affine roots of polynomial systems via pointed Newton poly topes," *Journal of Complexity*, vol. 12, no. 2, pp. 116 – 133, 1996.

[86] F. Rouillier, "Solving zero-dimensional systems through the rational univariate representation," *Applicable Algebra in Engineering, Communication and Computing*, vol. 9, no. 5, pp. 433 – 461, 1999.

[87] F. Rouillier and P. Zimmermann, "Efficient isolation of polynomial's real roots," *Journal of Computational and Applied Mathematics*, vol. 162, no. 1, pp. 33 – 50, 2004.

[88] T. Sakkalis, "The topological configuration of a real algebraic curve," *The Bulletin of the Australian Mathematical Society*, vol. 43, no. 1, pp. 37 – 50, 1991.

[89] J. Schwartz, "Fast probabilistic algorithms for verification of polynomial identities," *Journal of ACM*, vol. 27, no. 4, pp. 701 – 717, 1980.

[90] R. Seidel, "The nature and meaning of perturbations in geometric computing," in *Proc. of 11th Annual Symposium on Theoretical Aspects of Computer Science*, ser. LNCS 775. Springer, 1994, pp. 3 – 17.

[91] ——, "The nature and meaning of perturbation in geometric computing," *Discrete and Computational Geometry*, vol. 19, no. 1, pp. 1 – 17, 1998.

[92] H. Sekigawa, "Using interval arithmetic with the mahler measure for zero determination of algebraic numbers," *Josai Information Sciences Researches*, vol. 9, no. 1, pp. 83 – 99, 1998.

[93] X. Song, T. W. Sederberg, J. Zheng, R. T. Farouki, and J. Hass, "Linear perturbation methods for topologically consistent representations of free-form surface intersections," *Computer Aided Geometric Design*, vol. 21, no. 3, pp. 303 – 319, 2004.

[94] B. Sturmfels, "On the Newton polytope of the resultant," *Journal of Algebraic Combinatorics*, vol. 3, no. 2, pp. 207 – 236, 1994.

[95] ——, *Solving Systems of Polynomial Equations.* Providence: AMS, 2002.

[96] K. Sugihara and M. Iri, "A solid modelling system free from topological inconsistency," *Journal of Information Processing*, vol. 12, no. 4, pp. 380 – 393, 1989.

[97] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, 2nd ed. New York: Cambridge University Press, 2003.

[98] N. Wolpert, "Jacobi curves: Computing the exact topology of arrangements of non-singular algebraic curves," in *Proc. of 11th Annual European Symposium on Algorithms*, ser. LNCS 2832. Springer, 2003, pp. 532 – 543.

[99] C. Yap, "Symbolic treatment of geometric degeneracies," *Journal of Symbolic Computation*, vol. 10, no. 3 - 4, pp. 349 – 370, 1990.

[100] ——, "Towards exact geometric computation," *Computational Geometry*, vol. 7, no. 1, pp. 3 – 23, 1997.

[101] ——, *Fundamental Problems in Algorithmic Algebra.* New York: Oxford University Press, 2000.

[102] J. Yu, "Exact arithmetic solid modeling," Ph.D. Dissertation, Department of Computer Science, Purdue University, West Lafayette, IN, 1991.

VITA

Koji Ouchi

Education

Ph. D. in Computer Science Texas A&M University, College Station, TX, December 2006

Master of Science, Computer Science Major, New York University, New York, NY, January 1997

Bachelor of Science, Mathematics Major, University of Tokyo, Tokyo, JAPAN, March 1992

Work Experience

Wolfram Research, Champaign, IL, October 2006 -

Graduate Assistant, Texas A&M University, College Station, TX, January 1998 - September 2006

Publications

J. Keyser, K. Ouchi and J. M. Rojas, "The Exact Rational Univariate Representation for Detecting Degeneracies," in *Geometric and Algorithmic Aspects of Computer-Aided Design and Manufacturing*, ser. AMS/DIMACS volume 67, R. Janardan, M. Smid, and D. Dutta, Eds. Providence: AMS, 2005, pp. 299 - 328.

K. Ouchi with J. Keyser, "Handling Degeneracies in Exact Boundary Evaluation," *Proc. of ACM Symposium on Solid Modeling and Applications*, ACM, 2004, pp. 321 - 326.